Documentation - V1.4

About Documantation: All source file of this documentation will be contained in **/RF-Board/DocumentationSource/**. I use **Markdown** to write documentation, and use software **Maxiang (Link:Maxiang)** to generate PDF file.

Update Log

21/12/2016 - V1.4: Add Chapter IV RF, update other chapters from V1.3.

07/12/2016 - V1.3: Update Chapter II. Compared with V1.0, rewrite all chapters. This is a complete new version documentation.

06/12/2016 - V1.2.2: Update **O.4 Setup the Environment of Kernel Module Development** and chapter I.

06/12/2016 - V1.2.1: Update Chapter III.

05/12/2016 - V1.2: Rewrite parts of the Chapter O. Update the **O.1, O.2(V1.2), O.3(V1.2)**, except the **O.4 For kernel module development (O.3 in V1.1 and V1.0)**.

05/12/2016 - V1.1: Rewrite the Chapter III.

Documentation - V1.4
- Update Log
- O. Preparation
  - O.1. Software and Driver Installation
  - O.2. Board Configuration
  - O.3. Loading FPGA from U-Boot (Optional)
  - O.4. Setup the Environment of Kernel Module Development
- I. Some Information Related to Project Development (Optional)
  - I.1. Boot Linux from SD Card
  - I.2. Communicating with ARM-Linux by PuTTY
  - I.3. Compiling and downloading user C program
- II. Communicating between FPGA and HPS (with Interrupt)
  - II.1. Hardware Part
    - II.1.1. Build New Project
    - II.1.2. Qsys Setup
    - II.1.3. Quartus Part
  - II.2. Software Part
  - II.3. Project Demonstration Procedure
  - II.4. Information about Handling FPGA Interrupt
  - II.5. Reference
- III. Simple Project of Read/Write HPS SDRAM by FPGA
  - III.0. Introduction of mSGDMA
  - III.1. Hardware/Qsys Part
    - QUICK IMPLEMENTATION
    - QSYS PART
  - III.2. Software Part
  - III.3. Divide the HPS SDRAM into Two Parts
  - III.4. Project Demonstration Procedure
  - III.5. Reference
- IV. RF Board
  - IV.1. Documentations of Myriad-RF Board and Zipper
    - IV.1.1. Documentations
    - IV.1.2. Important Information about RF Board
    - IV.1.3. Important Information about Zipper Board
  - IV.2. Theory Part
  - IV.3. RF Board Configuration Procedure
  - IV.3. Hardware Part
    - QUICK IMPLEMENTATION
  - IV.4. Software Part

IV.5. Project Demonstration
IV.5.1. Procedure and Structure
IV.5.2. Possible Reasons If Procedure Failed

O. Preparation

O.1. Software and Driver Installation

Follow the instruction in the PDF of **1-SoCKit_Getting_Started_Guide.pdf**(File location: **/RF-Board/ImportantDocuments/**), *Chapter 2 and 4*. In Chapter 4, only follow the instructions of installing driver of Altera USB-Blaster II (4.2 Installing the USB-Blaster II Driver). Don't follow the instructions of 4.3, because the object of this part is only installing the needed driver.

O.2. Board Configuration

For **gerneral usage**, following the instructions in **1-SoCKit_Getting_Started_Guide.pdf**, *Chapter 3*.

Gerneral Usage: Boot Linux from SD Card, and program FPGA by Quartus Programmer or Quartus Signal-Tap.

The board configuration switches' and jumpers' positions and function charts are introduced in **4-SoCKit_User_manual.pdf** (File location: **/RF-Board/ImportantDocuments/**), *Chapter 3 - 3.1 Board Setup Components*. Following is the list:

> 3.1.1 JTAG Chain and Setp Switches: SW4.1 and SW4.2. (Page 13)
> 3.1.2 FPGA Configuration Mode Switch: MSEL. (Page 14 ~ Page 15)
> 3.1.3 HPS BOOTSEL and CLKSEL Setting Headers: BOOTSEL, CLKSEL. (Page 15 ~ Page 16)
> 3.1.4 HSMC VCCIO Voltage Level Setting Header: JP2. (Page 17)

O.3. Loading FPGA from U-Boot (Optional)

I recommand to **load FPGA from U-Boot**. FPGA can be programmed at the same time of booting Linux. When you focus on debugging software, this will save a lot of time.
REF: Loading FPGA from u-boot.

Reason: Firstly, this method will make debugging the software much more easily. By loading fpga from U-Boot, you can just boot board and debug the software without programming FPGA by Quartus tools. Secondly, this change does not have impacts on programming FPGA by Quartus tools. After deleting the rbf file in FAT part of SD Card, the U-Boot will jump the procedure of programming FPGA and then you can program the FPGA by Quartus tools.

Steps of Loading FPGA from U-Boot

1. Board setting: BOOTSEL[2:0]=101, ***MSEL[4:0]=00000***, SW4.1=off, SW4.2=on.
2. Convert the sof file to rbf file. After project compilation in Quartus, a sof file can be found at **[Project Folder]/output_files/[Project Name].sof**. This sof file can be used to program FPGA by **Quartus tools**, such as **Programmer** and **Signal-Tap**. Transfer this sof file to a rbf file. Then copy this rbf file to the root path of the **FAT part of SD Card** (this part can be accessed by Windows). There are two ways to convert the file on Windows.

Using convert tool: Click the **Convert Programming File** of **File** menu in Quartus.
Using PowerShell: Open Windows PowerShell (search it at the bottom Start menu). Following is an example command. About the correct path of **quartus_cpf.exe**, you can search the installation folder of Quartus.

```
C:\altera\15.0\quartus\bin64\quartus_cpf.exe -c .\[Project Name].sof .\[Proj
ect Name].rbf
```

(**NOTE:** In original Documantation, I write some thing about the ‘same name’, ‘fpga.rbf’. It produces some confusion. So I delete it. You can forget about it.)

3. Configure U-Boot Settings: Power on board, stop autoboot at the stage of countdown. Then input the following command.

```
set fpgaload "fatload mmc 0:1 0x2000000 [Project Name].rbf;fpga load 0 $""{f
pgadata} $""{filesize}"
```

Note: If the name of rbf file changed, then only change the **[Project Name].rbf** to the new name.

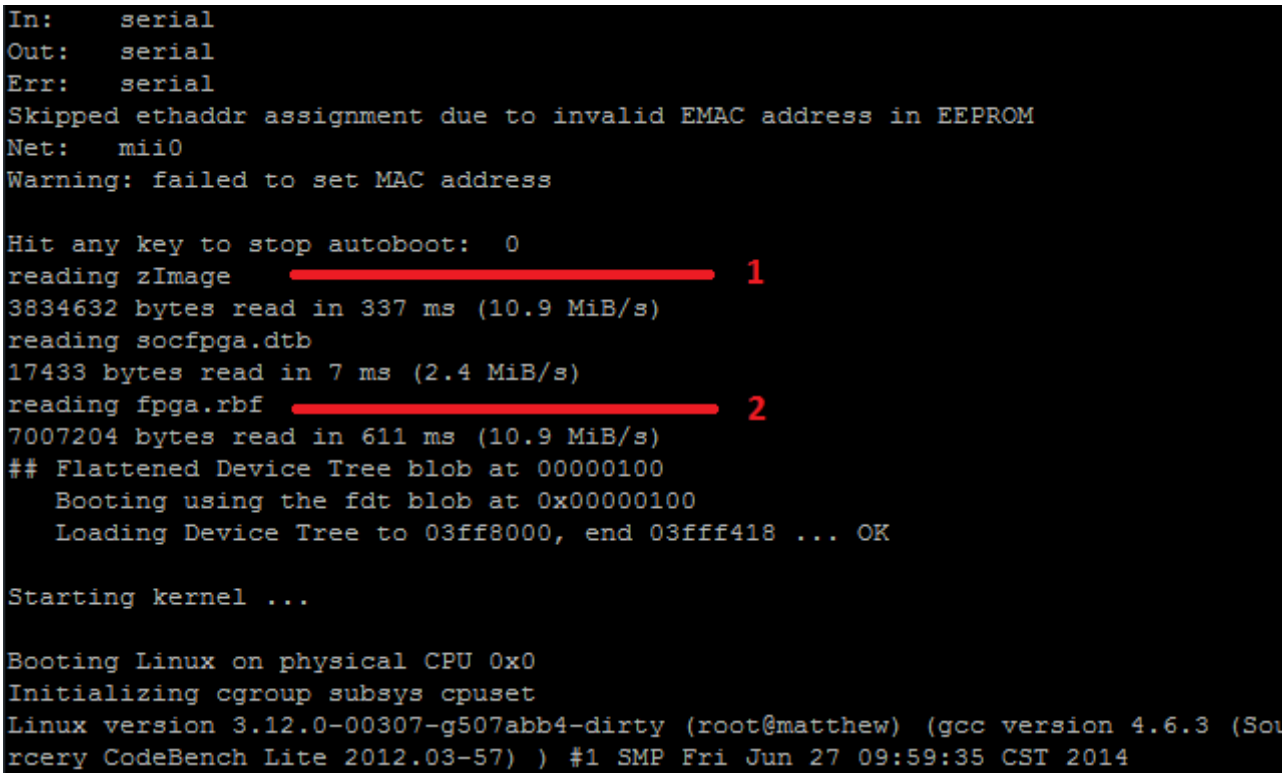
Continue

```
set bootcmd "run mmcload;run fpgaload;run mmcboot"
env save
run bootcmd
```

Then whenever power on the board, FPGA part will be programed by U-Boot automatically with the file of **[Project Name].rbf**.

Programming FPGA Success

At 1, start booting Linux. At 2, start programming FPGA by **fpga.rbf**. There is no error message, means programming FPGA success.



Programming FPGA Failed

At 1, the console output

> ** Unable to read file fpga.rbf **
> altera_load: Failed with error code -4

The **error code -4** means programming FPGA failed.

```
In:    serial
Out:   serial
Err:   serial
Skipped ethaddr assignment due to invalid EMAC address in EEPROM
Net:   mii0
Warning: failed to set MAC address

Hit any key to stop autoboot:  0
reading zImage
3834632 bytes read in 337 ms (10.9 MiB/s)
reading socfpga.dtb
17433 bytes read in 7 ms (2.4 MiB/s)
reading fpga.rbf
** Unable to read file fpga.rbf **
altera_load: Failed with error code -4 ----------1
## Flattened Device Tree blob at 00000100
   Booting using the fdt blob at 0x00000100
   Loading Device Tree to 03ff8000, end 03fff418 ... OK

Starting kernel ...

Booting Linux on physical CPU 0x0
Initializing cgroup subsys cpuset
Linux version 3.12.0-00307-g507abb4-dirty (root@matthew) (gcc version 4.6.3 (Sou
```

O.4. Setup the Environment of Kernel Module Development

According to the information on website, in Linux system, only the kernel modules can handle the hardware interrupts (such as FPGA interrupts to HPS). It means that the FPGA interrupts cannot be handled by user programs. In order to handle the interrupts from FPGA, we must develop kernel modules.

Before developing kernel modules, we firstly setup the development environment.

INSTRUCTIONS:

1. Firstly, check out the Linux version of the SoCKit board. Power on the board and login Linux. Then input following command and the console will show the version of kernel.

```
uname -r
```

My case:

```
root@socfpga:~# uname -r
3.12.0-00307-g507abb4-dirty
root@socfpga:~#
```

> **NOTE:** In my case, **3.12.0-00307-g507abb4-dirty**, the first part **3.12.0** is **pure kernel version**, and the second part **-00307-g507abb4-dirty** is **extra version**.

2. According to the **pure kernel version**, download needed kernel packet from this website: Linux Kernel Download. In my case, I need to download **linux-3.12.tar.xz** (File loaction:**/RF-Board/ImportantSource/**).
3. Get the kernel configuration from SoCKit Linux system. Login Linux of the board and input following commands.

```
cd /proc/
modprobe configs
ls
```

Then you will see these files.

```
root@socfpga:/proc# ls
1               2               50              filesystems     partitions
10              20              51              fs              self
11              21              6               interrupts      slabinfo
12              22              7               iomem           softirqs
127             23              8               ioports         stat
13              24              9               irq             swaps
14              25              buddyinfo       kallsyms        sys
146             26              bus             key-users       sysrq-trigger
15              27              cgroups         kmsg            sysvipc
150             3               cmdline         kpagecount      timer_list
153             32              config.gz       kpageflags      tty
157             37              consoles        loadavg         uptime
16              4               cpu             locks           version
162             40              cpuinfo         meminfo         vmallocinfo
17              45              crypto          misc            vmstat
178             46              device-tree     modules         zoneinfo
179             47              devices         mounts
18              48              diskstats       mtd
185             49              driver          net
19              5               execdomains     pagetypeinfo
root@socfpga:/proc#
```

Copy the **config.gz** to Ubuntu. Extract the **config.gz** (gzip -d config.gz), then you will get **config** file. This file will be used in **STEP 7**. **Note:** In some cases, you need to copy it to /home/root/ folder, and then copy it to Ubuntu.

4. Install Ubuntu in Virtual Machine. Copy downloaded kernel packet to Ubuntu.

NOTE: Excute Step 5 ~ Step 7 on Ubuntu

5. Install arm-linux-gnueabihf version 4.7. **Note: DO NOT use version higher than 4.7. Because the kernel version is too old. Higher version cannot compile the kernel modules with kernel 3.12.0.** An important lib also need to be installed.

```
$ sudo apt-get install gcc-4.7-arm-linux-gnueabihf
$ sudo apt-get install libncurses5-dev
```

At now we only have **arm-linux-gnueabihf-gcc-4.7** tool. The version tail need to be deleted. Use following command.

```
$ sudo cp /usr/bin/arm-linux-gnueabihf-gcc-4.7 /usr/bin/arm-linux-gnueabihf-
gcc
```

Then we have the correct **arm-linux-gnueabihf-gcc** tool.

6. Extract the kernel packet and change the top **Makefile**. Use following command (my case):

```
$ tar -xf linux-3.12.tar.xz
$ cd linux-3.12
$ vim Makefile
```

Changing list:

> **EXTRAVERSION=** Change to **EXTRAVERSION=-00307-g507abb4-dirty NOTE:**Just the extra version in **STEP 1**.
> **ARCH ?= $(SUBARCH)** Change to **ARCH = arm (NOTE: THERE SHOULD NOT BE AN EXTRA SPACE AFTER ARCH=arm)**
> **CROSS_COMPILE ?=** Change to **CROSS_COMPILE = arm-linux-gnueabihf-**

7. Copy the **config** (STEP 3) to the Linux kernel folder (in my case is **linux-3.12**). Then open the console and change the present path to the Linux kernel folder. Input following commands.

```
$ cp config .config
$ make oldconfig
$ make
```

Then the console will show the process of compalition. After several seconds (**DO NOT** wait all finish. Do not wait until the end, because the kernel will not be built, only a module for it, as will be described later), ctrl+c to stop make, then the kernel folder can be used for kernel module generating.

> **NOTE:**
> 1.The development of kernel module must be excuted on Linux (Ubuntu) system. I tried to use SoC EDS Command Shell to compile the kernel module, but it was failed.
> 2.If you want to check whether the environment is correct, please following the instructions of **Chapter II**. If you can excute the instructions of Chapter II without error, then the environment is correctly setup.

I. Some Information Related to Project Development (Optional)

In this chapter, I will give some information related to project development, such as how to boot linux from SD Card, how to compile program, and other related information.

These information are not important. You can **jump this chapter** if you are familiar with these information.

I.1. Boot Linux from SD Card

1. Arrow SoCKit Evaluation Board V14.0 - How to Boot Linux
2. Or follow the instruction in the PDF of **1-SoCKit_Getting_Started_Guide.pdf**, *Chapter 5* (File loacation: **/RF-Board/ImportantDocuments/**).

I.2. Communicating with ARM-Linux by PuTTY

Follow the instruction in the PDF of **1-SoCKit_Getting_Started_Guide.pdf**, *Chapter 5*.

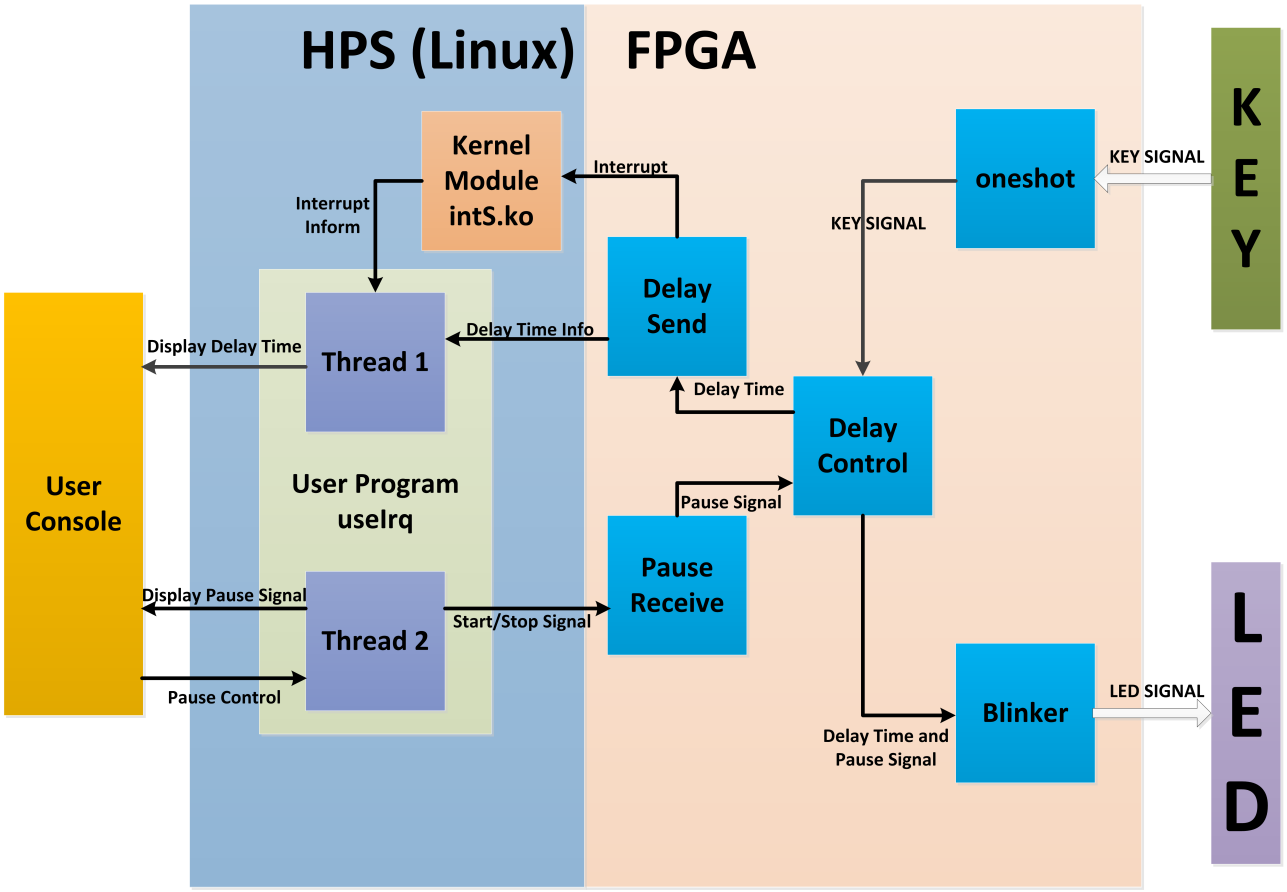
I.3. Compiling and downloading user C program

1. **Compiling program:** Using Altera SoC EDS command shell.
2. **Downloading program:** Now using SD Card Reader. Transfer files under linux system (Ubuntu).
3. **Debug:** Using gdb on board.

II. Communicating between FPGA and HPS (with Interrupt)

Chapter II will introduce a simple project of using FPGA interrupt and communicating between FPGA and HPS via LWHPS2FPGA bridge. Linux program (HPS) can control the LED (FPGA part) start/stop flashing, and FPGA can send the delay time, which is controlled by keys on board, to Linux program.

Project Diagram



II.1. Hardware Part

In this project, new components, which were writen by myself, will be added into Qsys. In **/RF-Board/Comm FPGA and HPS with IRQ/HWFile/** there are several files will be used in the following instructions.

> **blinker.vhd:** Control the LED blinking.
> **delay_ctrl.vhd:** Control the blinking delay time. When user press the keys, **delay_ctrl** will change the delay time and send it **blinker** and **delay_send**.
> **delay_send.vhd:** Will be used in Qsys. Receive the delay time from **delay_ctrl** and send it to Linux program (HPS) via LWHPS2FPGA bridge. It will also send the interrupt to HPS when the delay time has changed.
> **oneshot.vhd:** Detect the change of keys. Make sure that only one time key pressing will be counted when you press the key. Send the key pressing information to **delay_ctrl** to control the delay time.
> **pause_rec.vhd:** Will be used in Qsys. Receive the start/stop blinking signal from Linux program (HPS) and send it to **blinker** to control the running of LED blinking.
> **irquse.vhd:** Top level file of this project.
> **irquse.sof:** Quartus programming file of this project. This file can be used directly to program FPGA via JTAG. **irquse.sof** will be re-generated after all the steps in **II.1** excuted.
> **fpga.rbf:** The rbf file converted from **irquse.sof** directly, without having to do all the steps of **II.1**.

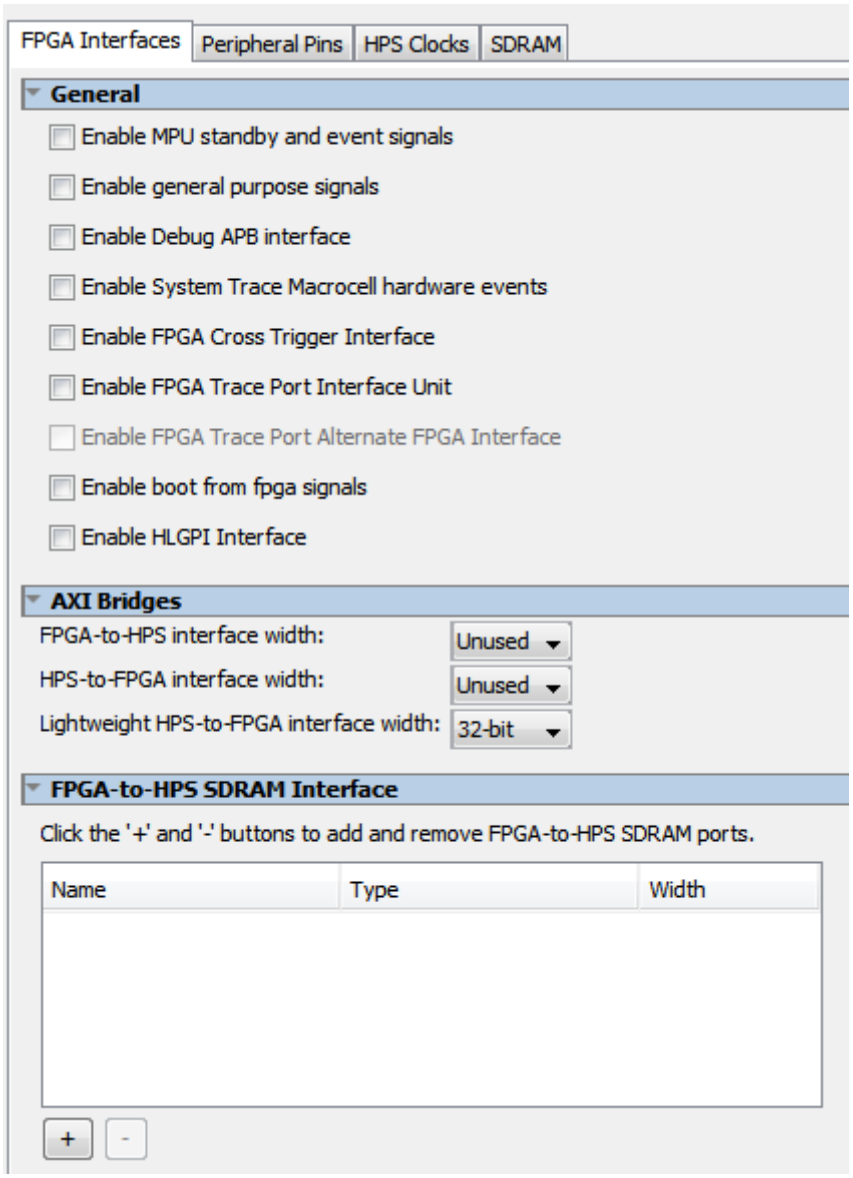
/RF-Board/Comm FPGA and HPS with IRQ/HW/ is my project development folder.

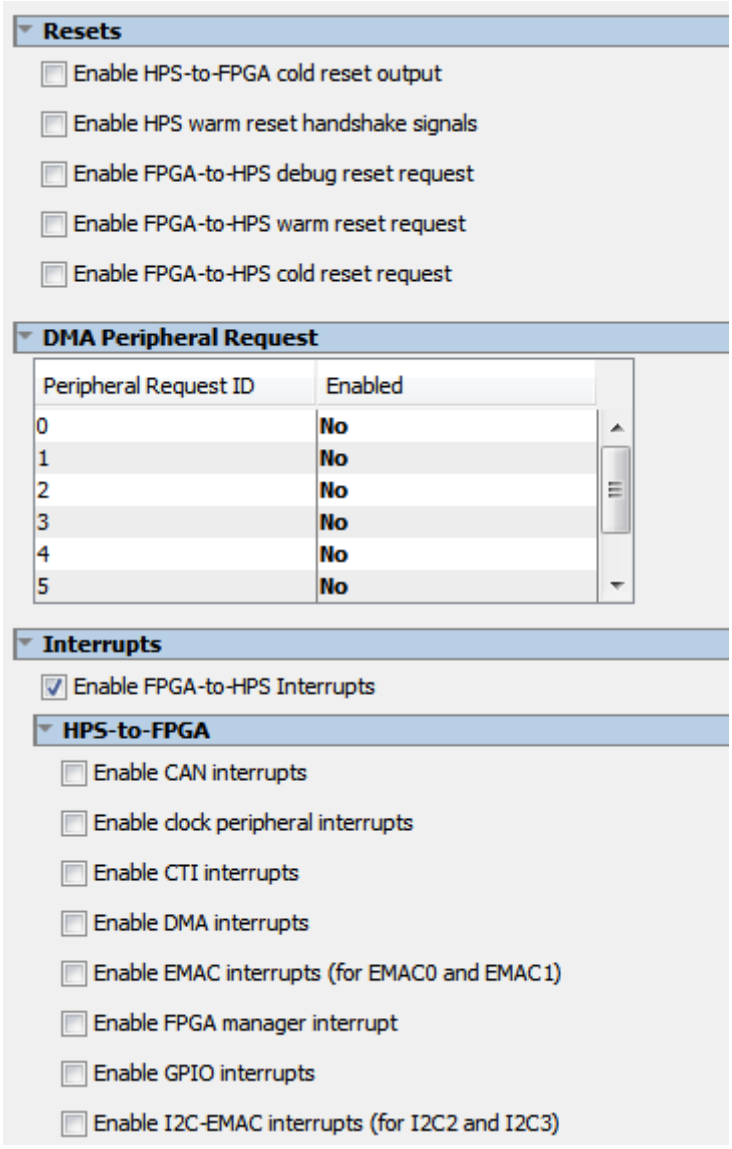
II.1.1. Build New Project

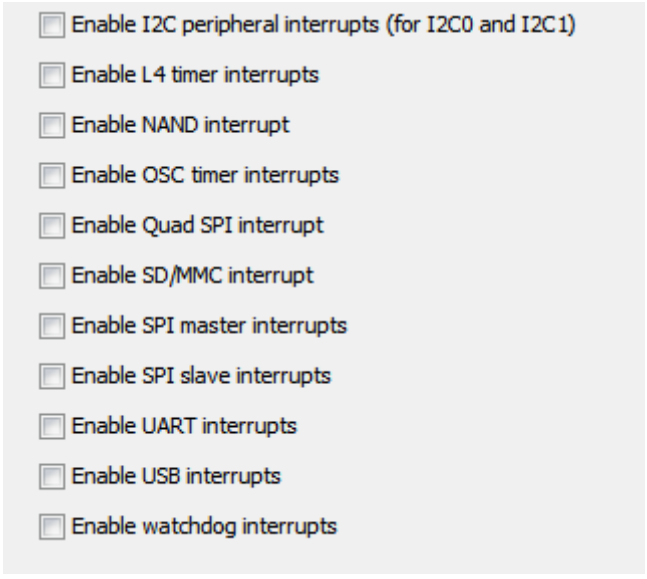
Build a new project, named **irquse**. Device choose **5CSXFC6D6F31C6**.

II.1.2. Qsys Setup

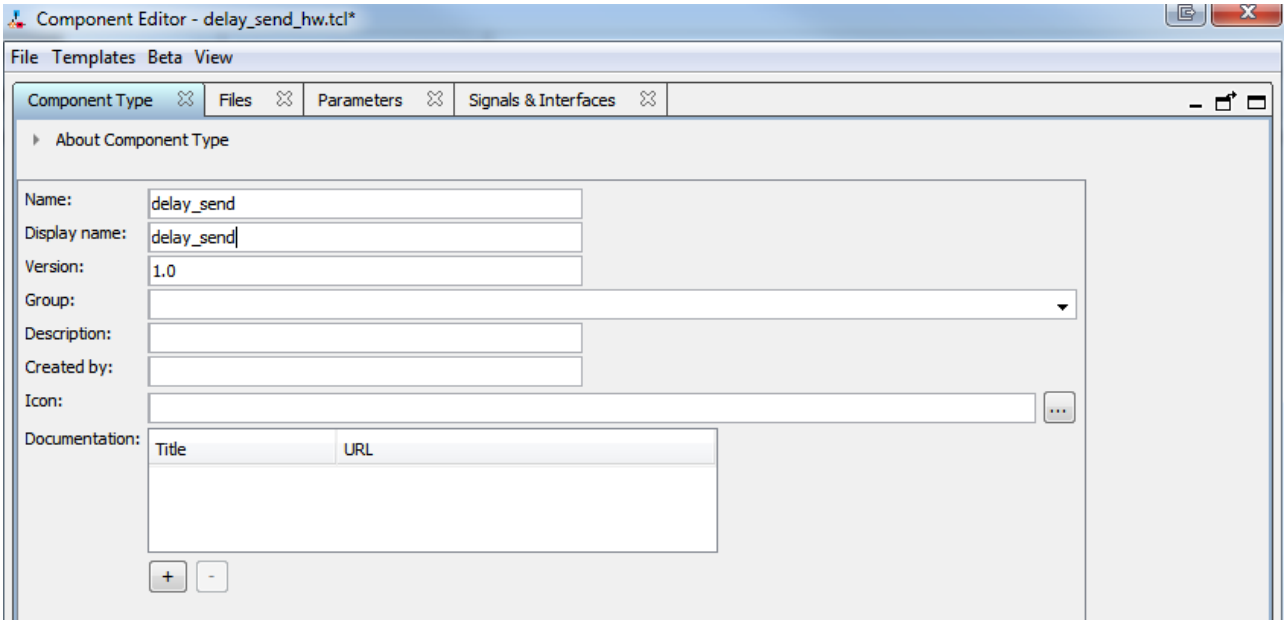
1. Copy **delay_send.vhd** and **pause_rec.vhd** to current project folder. Open Qsys tool.
2. Firstly add HPS to Qsys. Search and add **Arria V/Cyclone V Hard Processor System** to Qsys. Configure HPS as following figure. **NOTE:** Only **FPGA Interface** tag will be modified, others keep default.


FPGA Interfaces
Peripheral Pins
HPS Clocks
SDRAM
General
Enable MPU standby and event signals
Enable general purpose signals
Enable Debug APB interface
Enable System Trace Macrocell hardware events
Enable FPGA Cross Trigger Interface
Enable FPGA Trace Port Interface Unit
Enable FPGA Trace Port Alternate FPGA Interface
Enable boot from fpga signals
Enable HLGPI Interface
AXI Bridges
FPGA-to-HPS interface width:
Unused
HPS-to-FPGA interface width:
Unused
Lightweight HPS-to-FPGA interface width:
32-bit
FPGA-to-HPS SDRAM Interface
Click the '+' and '-' buttons to add and remove FPGA-to-HPS SDRAM ports.
Name
Type
Width
+
-

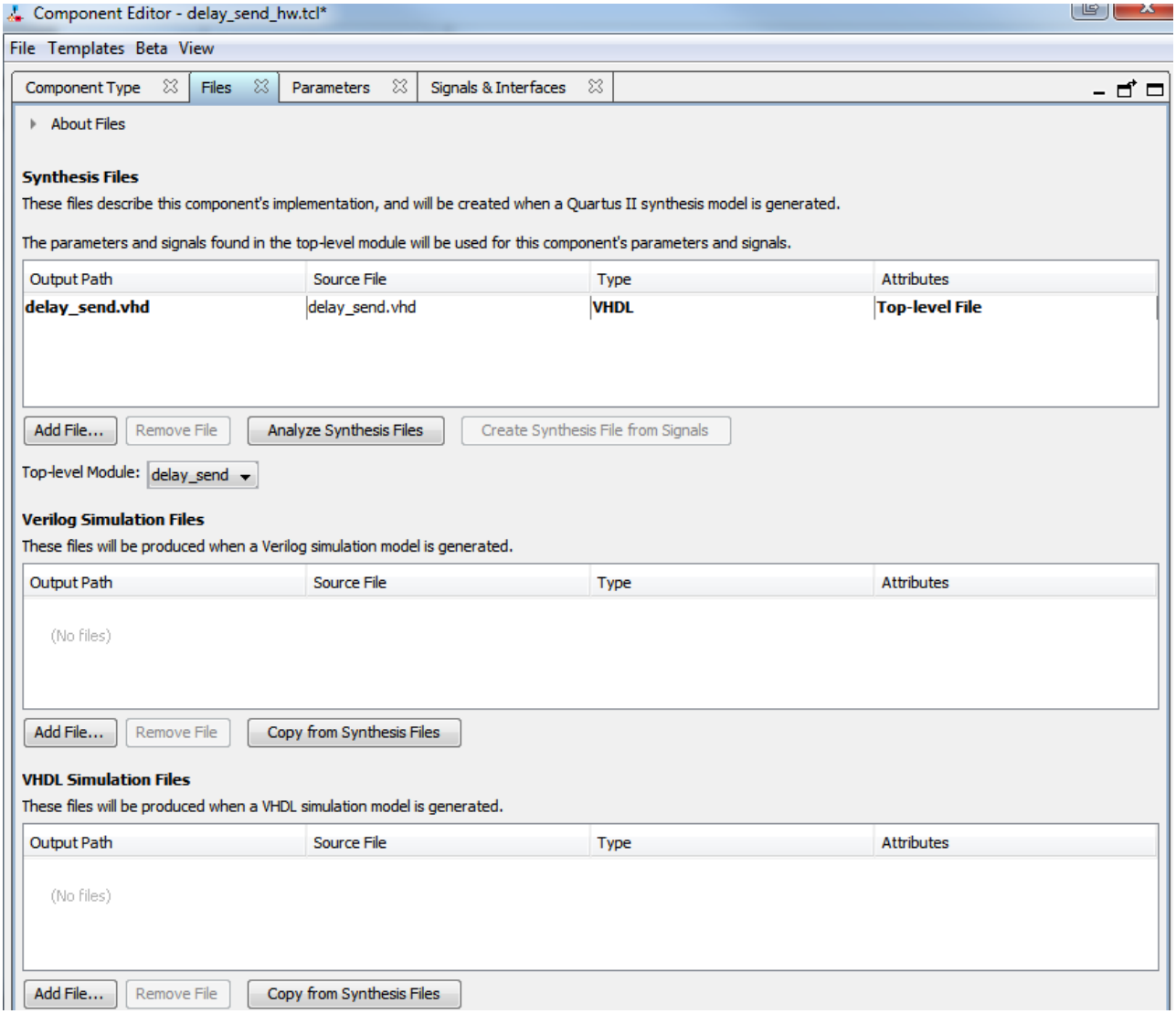




3. Add **delay_send** component. At the **IP Catalog** window, press **New…** and configure it as following figures.
   Component Type


Component Editor - delay_send_hw.tcl*
File Templates Beta View
Component Type
Files
Parameters
Signals & Interfaces
About Component Type
Name:
delay_send
Display name:
delay_send
Version:
1.0
Group:
Description:
Created by:
Icon:
Documentation:
Title
URL
+
-

Files

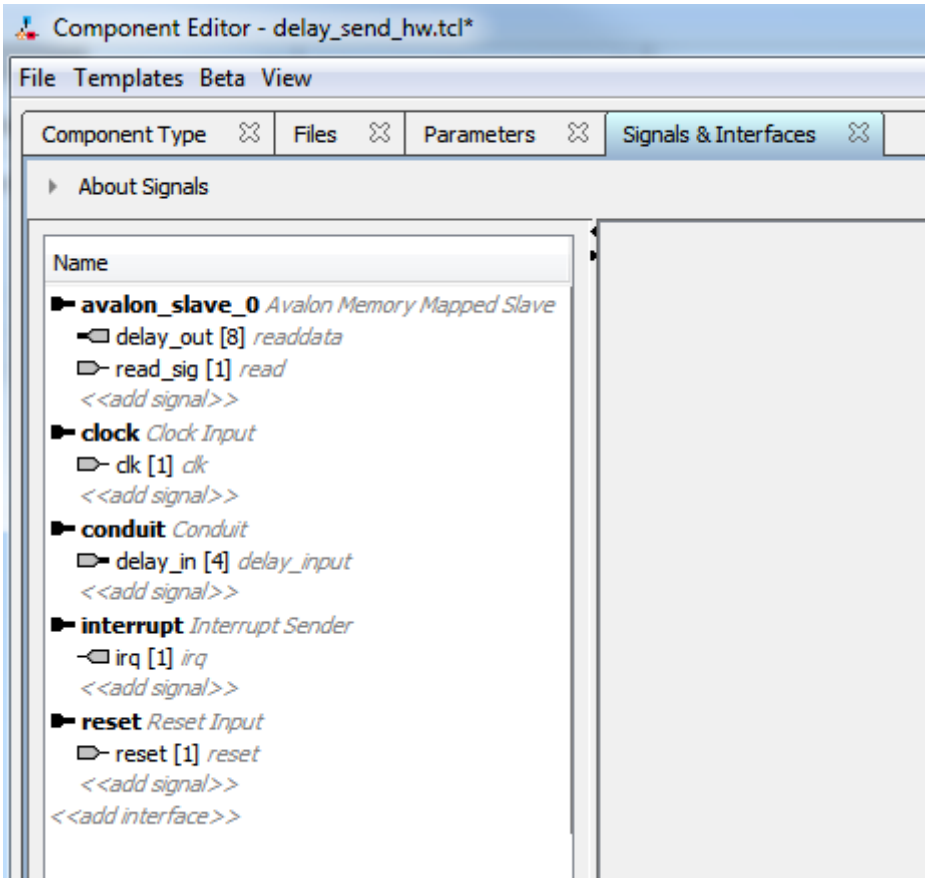


NOTE: After adding synthesis file (delay_send.vhd), you need to press **Analyze Synthesis File** to get all pins of this component, and configure them in **Signals & Interfaces** tag.

Parameters Keep default.

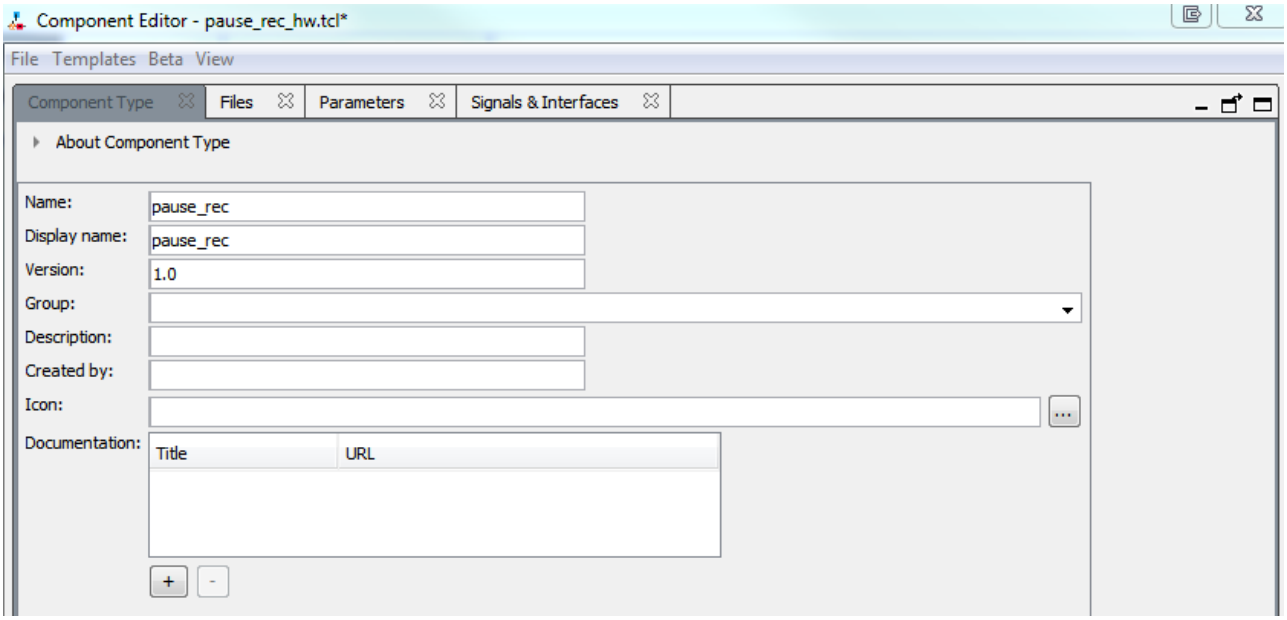
Signals & Interfaces

NOTE: ***DO NOT*** forget to add the associated reset for the avalon_slave_0 interfaces. This means click on the interface and select reset from the list. Note that an error will be also shown.

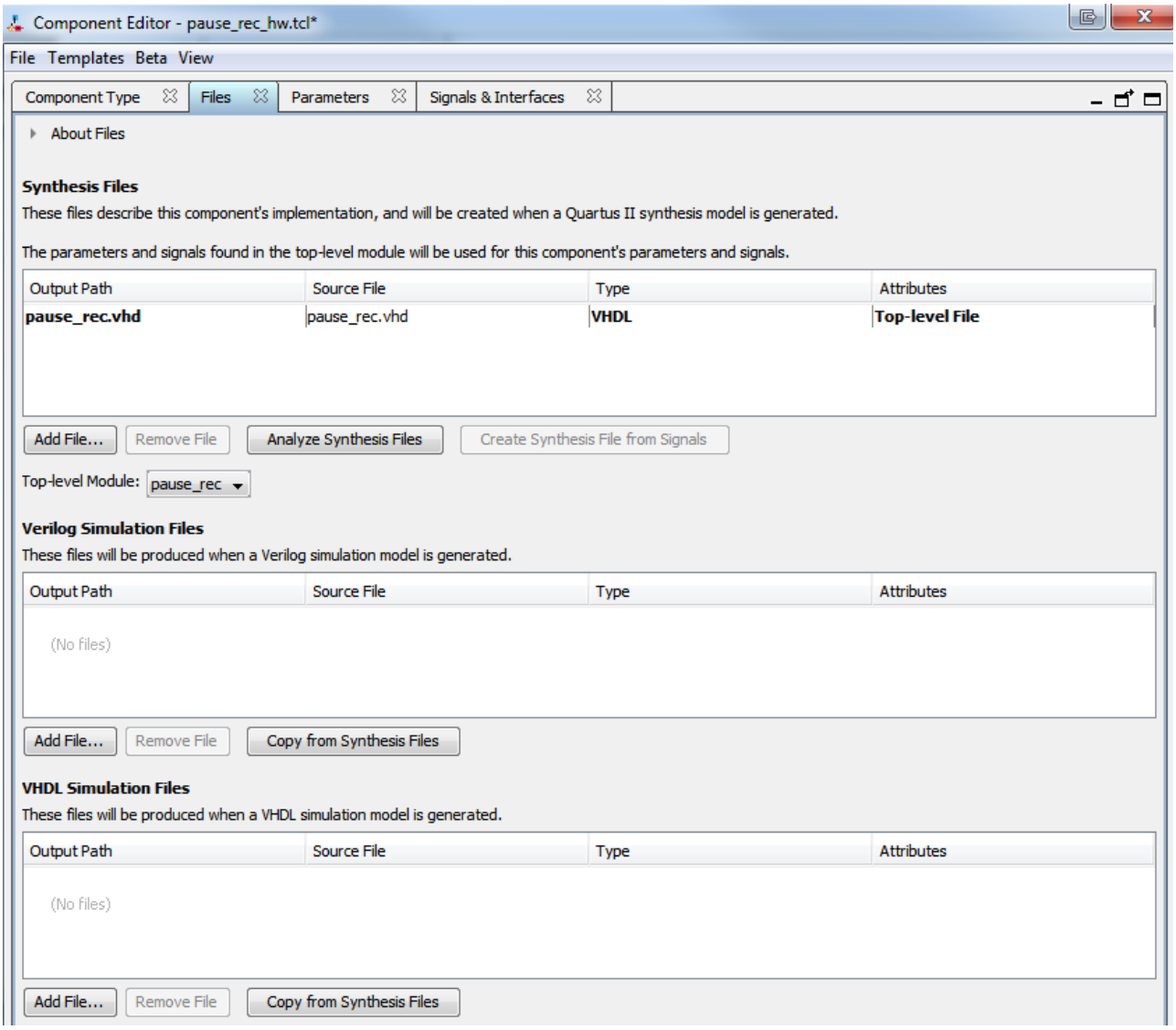


Then press **Finish** and save this component. Then you will see a new component at the **IP Catalog** window, under the **Project** root. Double click it and add it to Qsys.

4. Add **pause_rec** component. Similar to step 3. Configure it as following figure.
   Component Type



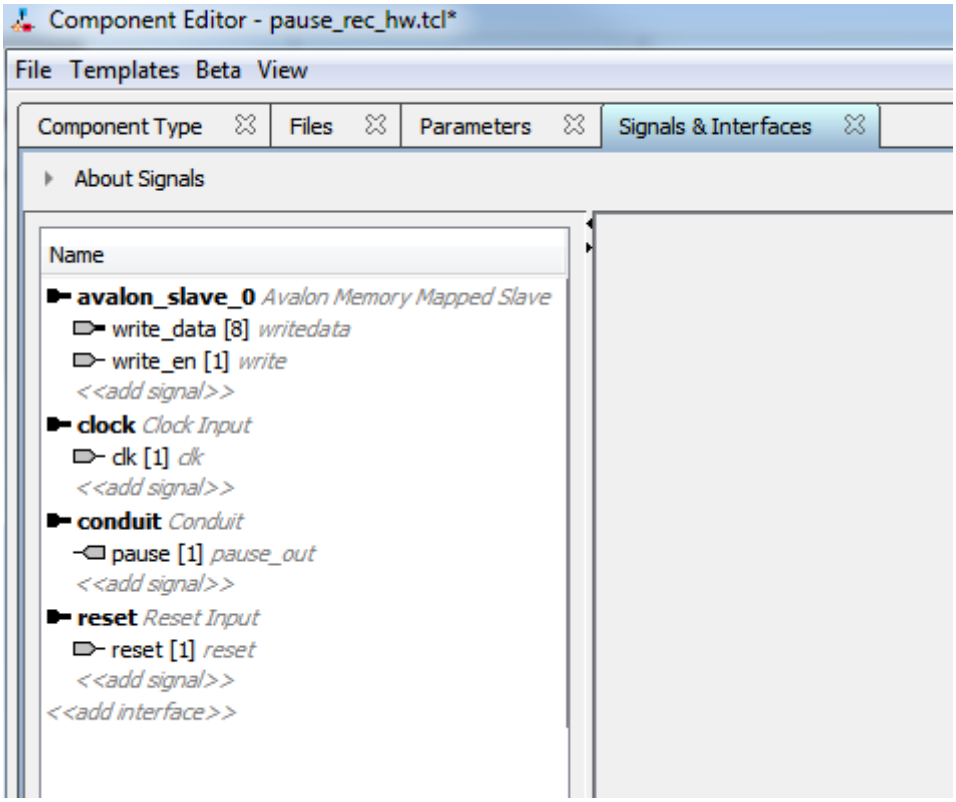
Files



Parameters Keep default.

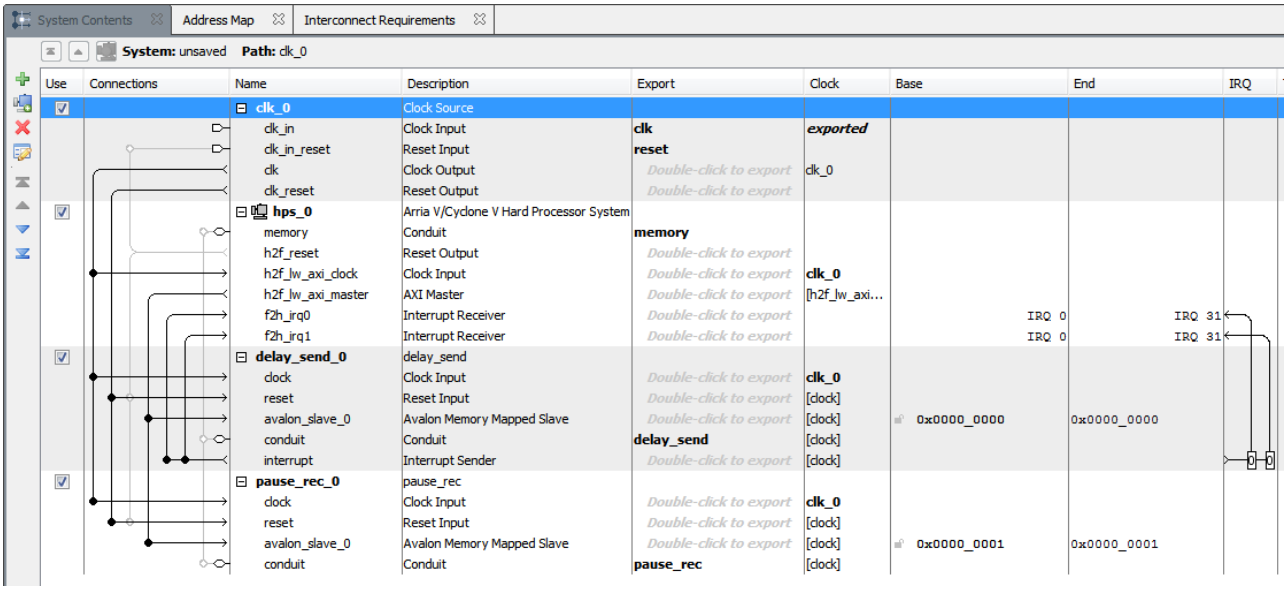
Signals & Interfaces

NOTE: ***DO NOT*** forget to add the associated reset for the avalon_slave_0 interfaces. This means click on the interface and select reset from the list. Note that an error will be also shown.



Then press **Finish** and save this component. Double click **pause_rec** component and add it to Qsys.

5. Setup connection and address, export conduit, setup IRQ number. Configure it as following figure.



6. Save as **soc.qsys**. Press **Generate HDL** and choose VHDL type. Gernerate. Qsys part end.
7. **NOTE:** The signals from LWHPS2FPGA bridge cannot be used directly. You need add signals in components to connect the port and then use these signals. Following is the explaination.
   EXPLAINATION
   For example, if a Avalon MM Slave port will be used in a IP core which is written by self. You write like the following example:

```
entity oneIPCore is
    port(
        avalon_mm_slave_writedata : in std_logic_vector(7 downto 0);
        avalon_mm_slave_write : in std_logic
    );
end oneIPCore;

architecture behave of oneIPCore is
    signal a,b,c,d,e : std_logic;
begin
    a <= avalon_mm_slave_writedata(0);
    b <= avalon_mm_slave_writedata(1);
    c <= avalon_mm_slave_writedata(2);
    d <= avalon_mm_slave_writedata(3);
    e <= a and b and c and d;
end behave;
```

Sometime this will create unknown errors when you program FPGA. So, you need to change it as following:

```
entity oneIPCore is
    port(
        avalon_mm_slave_writedata : in std_logic_vector(7 downto 0);
        avalon_mm_slave_write : in std_logic
    );
end oneIPCore;

architecture behave of oneIPCore is
    signal a,b,c,d,e : std_logic;
    signal avalon_mm_slave_writedata_in : std_logic_vector(7 downto 0);
begin
    avalon_mm_slave_writedata_in <= avalon_mm_slave_writedata;
    a <= avalon_mm_slave_writedata_in(0);
    b <= avalon_mm_slave_writedata_in(1);
    c <= avalon_mm_slave_writedata_in(2);
    d <= avalon_mm_slave_writedata_in(3);
    e <= a and b and c and d;
end behave;
```

Then every thing will go well.

II.1.3. Quartus Part

1. At the **Files** tag of **Project Navigator** window, add **[Project Folder]/soc/synthesis/soc.qip** file into this project, which is generated in **II.1.2**. Extend the **soc/synthesis/soc.qip**. Open the

first file **soc/synthseis/soc.vhd**, which will be used in following steps.

2. Copy **blinker.vhd delay_ctrl.vhd irquse.vhd oneshot.vhd** to current project folder. And these files to current project. Check component **soc** port defination. If the port names of component **soc** (of top level file) are the same to the soc.vhd, then **Start Analysis & Synthesis** (**NOTE:** do not **Start Compilation**).
   NOTE: Components are used instead of entities, which means that if you forget to add any of the files needed by the project, **an error will NOT be thrown**. Just a cryptic warning is given
3. After finishing the analysis and synthesis, pin assignment need to be done. Click the **Tcl Scripts** of **Tools** menu. In the opened window, choose **/Project/soc/synthesis/submodules/hps_sdram_p0_assignments.tcl**, then click **Run**. Close the window after finishing. Then click **Pin Planner** of **Assignments** menu. Then assign pins as following list.

   > These pins have to be manually set.
   > CLOCK_50 — K14
   > KEY[3] — AD11
   > KEY[2] — AD9
   > KEY[1] — AE12
   > KEY[0] — AE9
   > LED[3] — AD7
   > LED[2] — AE11
   > LED[1] — AD10
   > LED[0] — AF10

4. Now **Start Compilation**. After finishing compilation, a sof file, **irquse.sof**, will be generated. Location: **[Project Folder]/output_files/irquse.sof**. Program FPGA by **irquse.sof** directly, or convert it to rbf file (introduced in section **O.2. Program FPGA from U-Boot**) and program FPGA from U-Boot.

IMPORTANT NOTE

> By following these steps the VHDL code worked only if it was compiled with Quartus 15.0.0 Build 145 04/22/2015 SJ Web Edition and it did not work with Quartus II 64 bit version 15.0.1 Build 150 06/03/2015 SJ Full Version.
>
> Any other version was quartus was not tested so it is unclear whether it will work or not.

II.2. Software Part

This project need one kernel module and one userspace program. The kernel module will handle the interrupt from FPGA. FPGA will send the interrupt to HPS when the delay time change. User program will send start/stop signal to FPGA, and read present delay time from FPGA when an interrupt happen.

There are 3 files in **/RF-Board/Comm FPGA and HPS with IRQ/SWFile**.

> **intS.ko:** Kernel module. Insert this kernel module before run user program. This kernel module can handle the interrupt from FPGA.
> **uselrq:** User program. Send start/stop signal to FPGA, and read present delay time from FPGA when an interrupt happen. This program use two threads. One thread is used to send start/stop signal, and another one is used to detect interrupt and read delay time from FPGA.
> **en_bridge.sh:** This bash script will enable all three FPGA-HPS bridges (HPS2FPGA, FPGA2HPS, LWHPS2FPGA). When you first boot up the board, the bridges are disabled by default. You must enable the bridge before sending commands to FPGA.
> **NOTE:** Open **en_bridge.sh** in Linux, maybe you will see some ^M. It is caused by different defination of **return** under DOS and Unix/Linux. **Solution**

/RF-Board/Comm FPGA and HPS with IRQ/SW is my project software development folder.

NOTE: The kernel module can only develop in Linux (Ubuntu) system. I tried to use SoC EDS Command Shell to compile the kernel module, but it was failed. And make sure the **Makefile** of kernel module has writen **correct kernel loaction**, the first line in the following example.

Kernel Loaction: The location where the Linux-3.12 was intstalled in **O.4. Step 6**

```
KERNEL_LOCATION=/home/zjw/linux-3.12
ARMMAKE=make ARCH=arm SUBARCH=arm CROSS_COMPILE=arm-linux-gnueabihf-

obj-m := intS.o

intS.ko: intS.c
    $(ARMMAKE) -C $(KERNEL_LOCATION) M=$(PWD) modules

clean:
    rm -f *.ko *.o *.mod.c *.symvers *.order
```

II.3. Project Demonstration Procedure

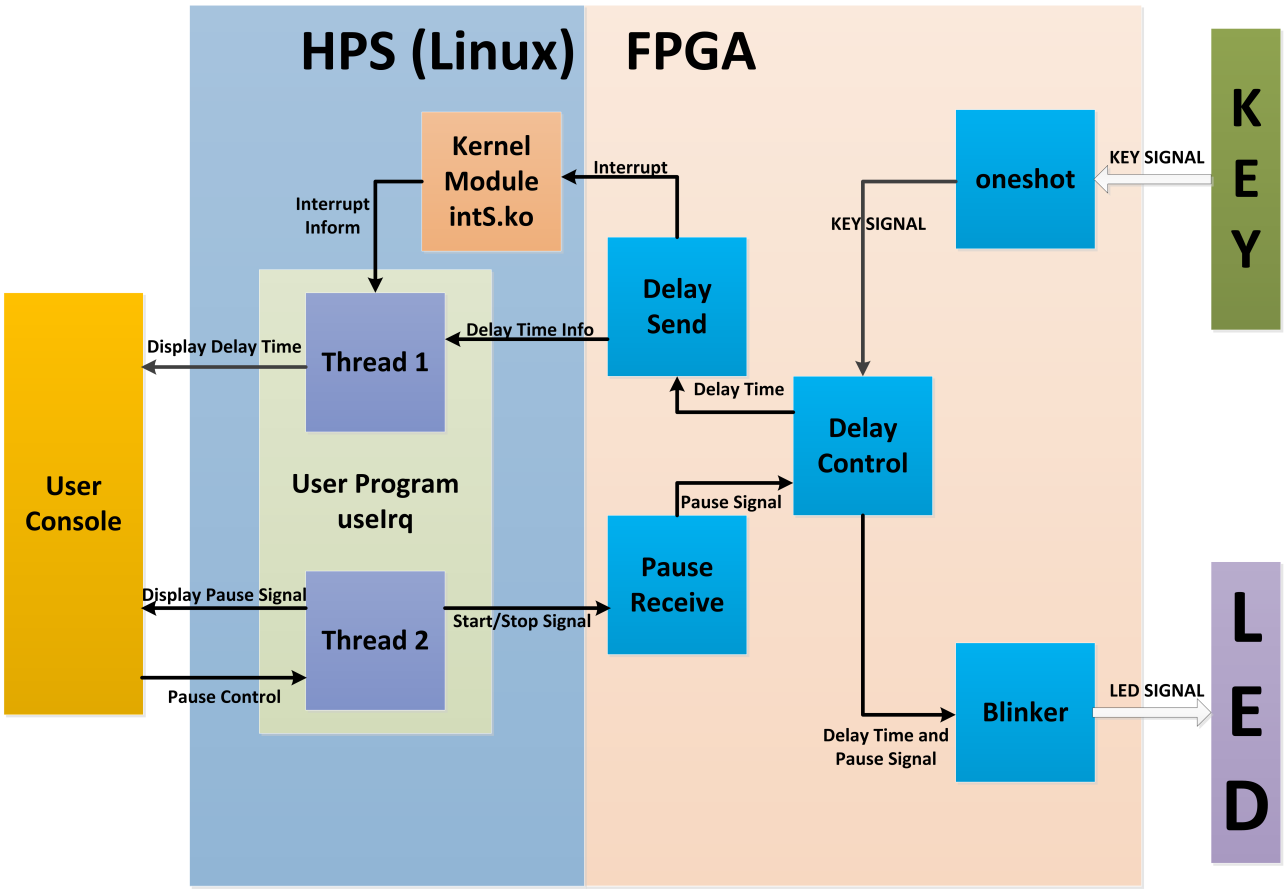
1. Power on board. Login Linux system. Run **en_bridge.sh** to open bridge.
2. Insert the kernel module, **intS.ko**, into kernel.

```
insmod intS.ko
```

3. Run user program, **uselrq**. Input number 9 will quit this program. Input number 1 will start/stop the sparkling LED. When running this program, press the FPGA key (close to JP2 at the right corner of board, **key0 to speed up and key1 to slow down**), the change of speed will display on the concle at same time you press the key.
4. Quit user program. Remove kernel module.

```
rmmod intS.ko
```

Procedure Diagram



II.4. Information about Handling FPGA Interrupt

1. Interrupt number: In Qsys, I arrange the Interrupt 0 for usage. In kernel module the interrupt number is 72. **FPGA to HPS interrupt number start from 72.**
2. Interrupt can only be handled by kernel, a kernel module is necessary.
3. After insert the module, kernel will creat a sysfs file. **Situation 1:** There was no interrupt happened before user program access the sysfs file. After user program access this file, the program (or thread) will be stucked until an interrupt happen. After get rid of stucked, program will continue to run next line. **Situation 2:** There was one interrupt happened before user

program access the sysfs file. After user program access this file, the program will not be stucked and run next line. **Situation 3:** There was several interrupts happened before user program access the sysfs file. After user program access this file, the program will not be stucked and run next line, but only the last interrupt will be handled by kernel.

4. About sending signal to user space when a interrupt happen: Kernel can send signal to a certain program. But this method required the program ID when inserting module. So I chose another method: using one **thread** to deal with the interrupt.

II.5. Reference

1. FPGA program reference: Howard Mao - Blinking LEDs
2. About writing Linux kernel: Howard Mao - Writing a Linux Device Driver
3. About handling FPGA interrupts: Howard Mao - Sending and Handling Interrupts
4. Pin assignment: **/RF-Board/ImportantDocuments/4-SoCKit_User_manual.pdf**, *3.6.1 User Push-buttons, Switches and LED on FPGA, Page 24*

III. Simple Project of Read/Write HPS SDRAM by FPGA

This project focused on the transferring data between FPGA FIFO and HPS memory. This project shows the methods of **transfer data from the FIFO in FPGA to HPS memory** by using mSGDMA **ST-MM** mode, and **reading data from HPS memory to FPGA** by using mSGDMA **MM-ST** mode.

III.0. Introduction of mSGDMA

In this project, mSGDMA Sub-core is used. It includes three sub-module: **Dispatcher** (control module), **Read Master** and **Write Master**. I put the user guides in **/RF-Board/ImportantDocuments**. There are three PDFs, **3-Modular_SGDMA_Dispatcher_Core_UG.pdf, 3-Modular_SGDMA_Read_Master_Core_UG.pdf, 3-Modular_SGDMA_Write_Master_Core_UG.pdf**. The official info website is Modular SGDMA.

Actually, in the original source file, **/RF-Board/ImportantSource/Modular_SGDMA_DE.zip**, there are some head files can be used. But these files are used in Nios II, so please write the function head file by self.

NOTE: One set of mSGDMA can only achieve one function. There are three kinds of sets: **MM-ST, ST-MM, MM-MM**.
a.MM-ST: Combined by **Dispatcher** and **Read Master**. This set can achieve the function of reading HPS memory data (via Avalon MM interface) and write the data to FIFO in FPGA (via Avalon ST interface).
b.ST-MM: Combined by **Dispatcher** and **Write Master**. This set can achieve the function of fetching data from FIFO in FPGA (via Avalon ST interface) and write the data to HPS memory (via Avalon MM interface).
c.MM-MM: Combined by **Dispatcher**, **Read Master** and **Write Master**. This mode is not used in this project. This set can achieve the function of transfer data within HPS memory.

III.1. Hardware/Qsys Part

In the section of **QUICK IMPLEMENTATION**, the steps of FPGA implementation will be introduced, except the procedure of generating Qsys file of this project. The contents of building and generating Qsys file of this project will be introduced in **QSYS PART** section.

Several files will be used in **QUICK IMPLEMENTATION** are located at **/RF-Board/RW HPS SDRAM by FPGA/HWFile**.

> **soc.qsys:** The Qsys file of this project.
> **mem.vhd:** The top level file of this project.
> **mem.sof:** Quartus programming file of this project. This file can be used directly to program FPGA via JTAG. **mem.sof** will be re-generated after all the steps in QUICK IMPLEMENTATION excuted.
> **fpga.rbf:** The rbf file converted from **mem.sof** directly, without having to do all the steps in QUICK implementation.

/RF-Board/RW HPS SDRAM by FPGA/HW is my project folder.

QUICK IMPLEMENTATION

1. Open folder: **/RF-Board/RW HPS SDRAM by FPGA/HWFile**. In this folder, there 2 files will be used in this section: **soc.qsys, mem.vhd**.
2. Create a new project named **mem**. Device choose **5CSXFC6D6F31C6**.
3. Copy **soc.qsys** to current project folder. Open Qsys tool, then choose and open **soc.qsys**. Save. Then press **Generate HDL**, change **'Synthesis - Create HDL design files for synthesis'** to **VHDL** type. Press **Generate** to generate the VHDL file. Close Qsys after finishing.
4. In Quartus, click **Add/Remove Files in Project** of **Project** menu. Add file **[Project Folder]/soc/synthesis/soc.qip**, and then click **OK**. In the **Files** tag of **Project Navigator** window, extend the **soc/synthesis/soc.qip**. Open the first file **soc/synthseis/soc.vhd**, which will be used in following steps.

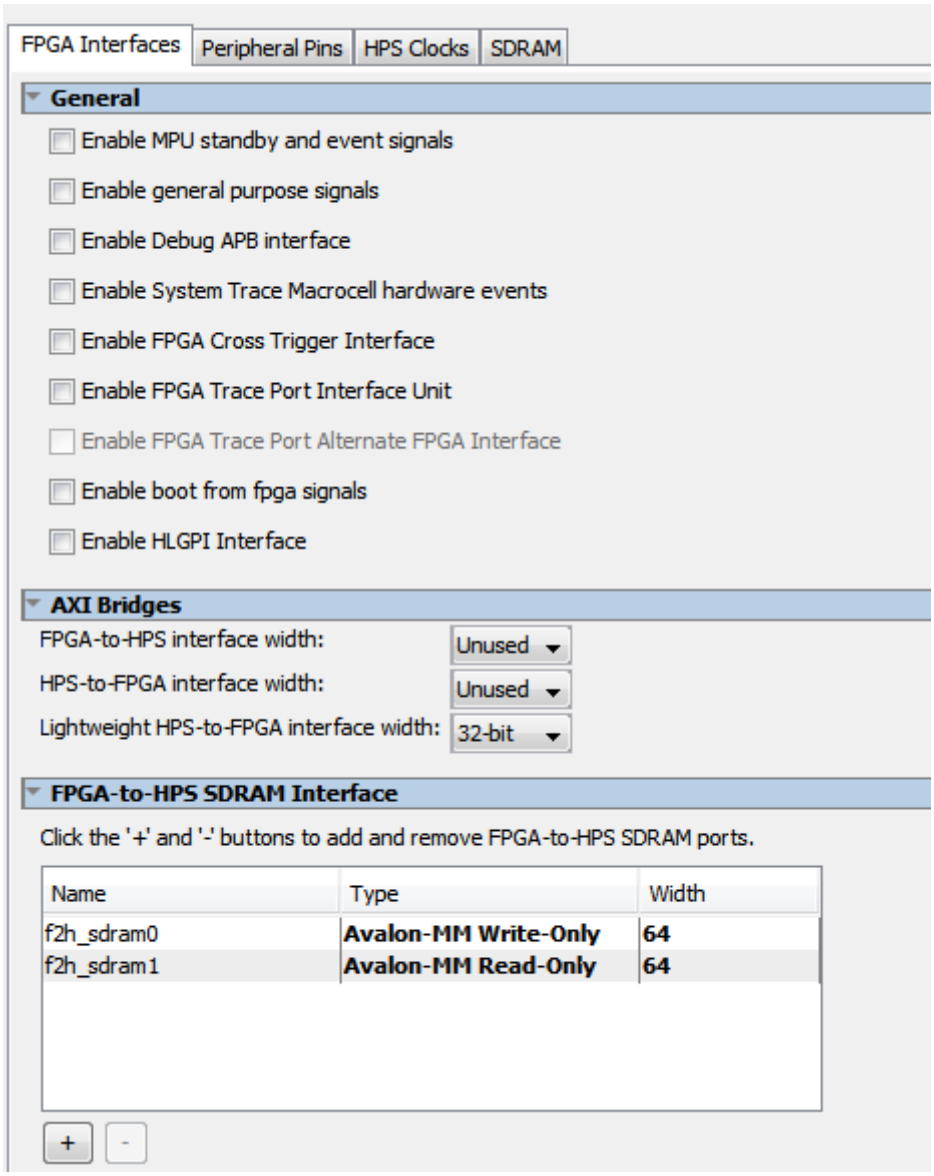
5. Copy **mem.vhd** to current project folder. It will be used as top level file. Add **mem.vhd** to current project by the method introduced in step 4. Open it and check if the port defination of component **soc** is the same with port defination in **soc.vhd** which is opened in step 4. If there is no problem, then **Start Analysis & Synthesis** (**NOTE:** do not **Start Compilation**).
6. After finishing the analysis and synthesis, pin assignment need to be done. Click the **Tcl Scripts** of **Tools** menu. In the opened window, choose **/Project/soc/synthesis/submodules/hps_sdram_p0_pin_assignments.tcl**, then click **Run**. Close the window after finishing. Then click **Pin Planner** of **Assignments** menu. Find the node **CLK_50**, input **PIN_K14** at the **Location** column. Then close **Pin Planner**.
7. Now **Start Compilation**. After finishing compilation, a sof file, **mem.sof**, will be generated. Location: **[Project Folder]/output_files/mem.sof**. Program FPGA by **mem.sof** directly, or convert it to rbf file (introduced in section **O.2. Program FPGA from U-Boot**) and program FPGA from U-Boot.
8. **NOTICE:** There was a problem during developing this project. When FPGA read HPS memory by Avalon MM, all system will collapse after rising the signal of ‘read’ from 0 to 1. **Solution:** After pushing down the power switcher, wait until we can log in Linux. Then press the **HPS_RST** key to restart the HPS and the problem will disappear.

QSYS PART

The procedure of Qsys configuring will be introduced in this section.

1. Open Qsys tool. First search **Arria V/Cyclone V Hard Processor System** and add it to current Qsys file at **IP Catalog**. Configure it as following.

FPGA Interfaces


FPGA Interfaces
Peripheral Pins
HPS Clocks
SDRAM
General
Enable MPU standby and event signals
Enable general purpose signals
Enable Debug APB interface
Enable System Trace Macrocell hardware events
Enable FPGA Cross Trigger Interface
Enable FPGA Trace Port Interface Unit
Enable FPGA Trace Port Alternate FPGA Interface
Enable boot from fpga signals
Enable HLGPI Interface
AXI Bridges
FPGA-to-HPS interface width:
Unused
HPS-to-FPGA interface width:
Unused
Lightweight HPS-to-FPGA interface width:
32-bit
FPGA-to-HPS SDRAM Interface
Click the '+' and '-' buttons to add and remove FPGA-to-HPS SDRAM ports.
Name
Type
Width
f2h_sdram0
Avalon-MM Write-Only
64
f2h_sdram1
Avalon-MM Read-Only
64
+
-

Resets

- [ ] Enable HPS-to-FPGA cold reset output
- [ ] Enable HPS warm reset handshake signals
- [ ] Enable FPGA-to-HPS debug reset request
- [ ] Enable FPGA-to-HPS warm reset request
- [ ] Enable FPGA-to-HPS cold reset request

DMA Peripheral Request

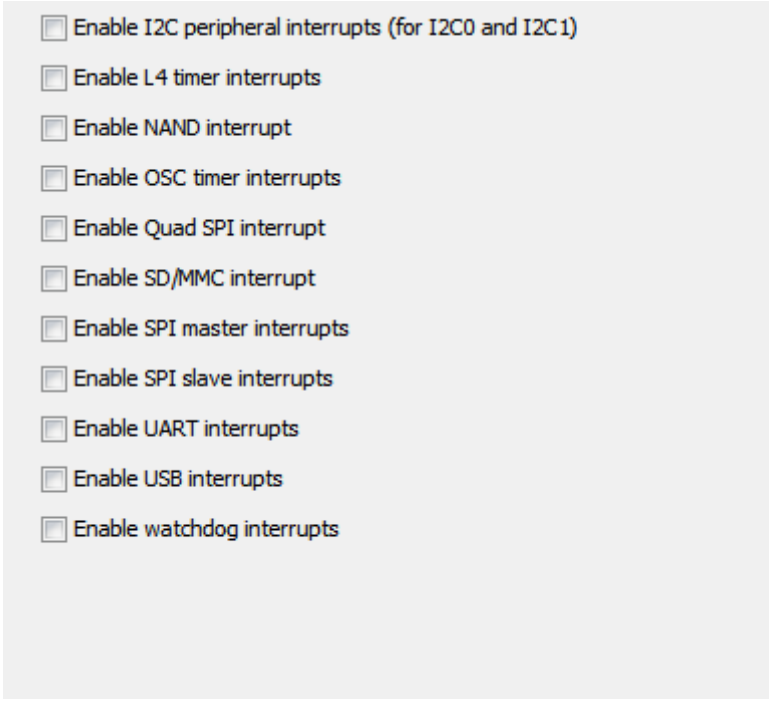
Peripheral Request ID	Enabled
0	**No**
1	**No**
2	**No**
3	**No**
4	**No**
5	**No**

Interrupts

- [ ] Enable FPGA-to-HPS Interrupts

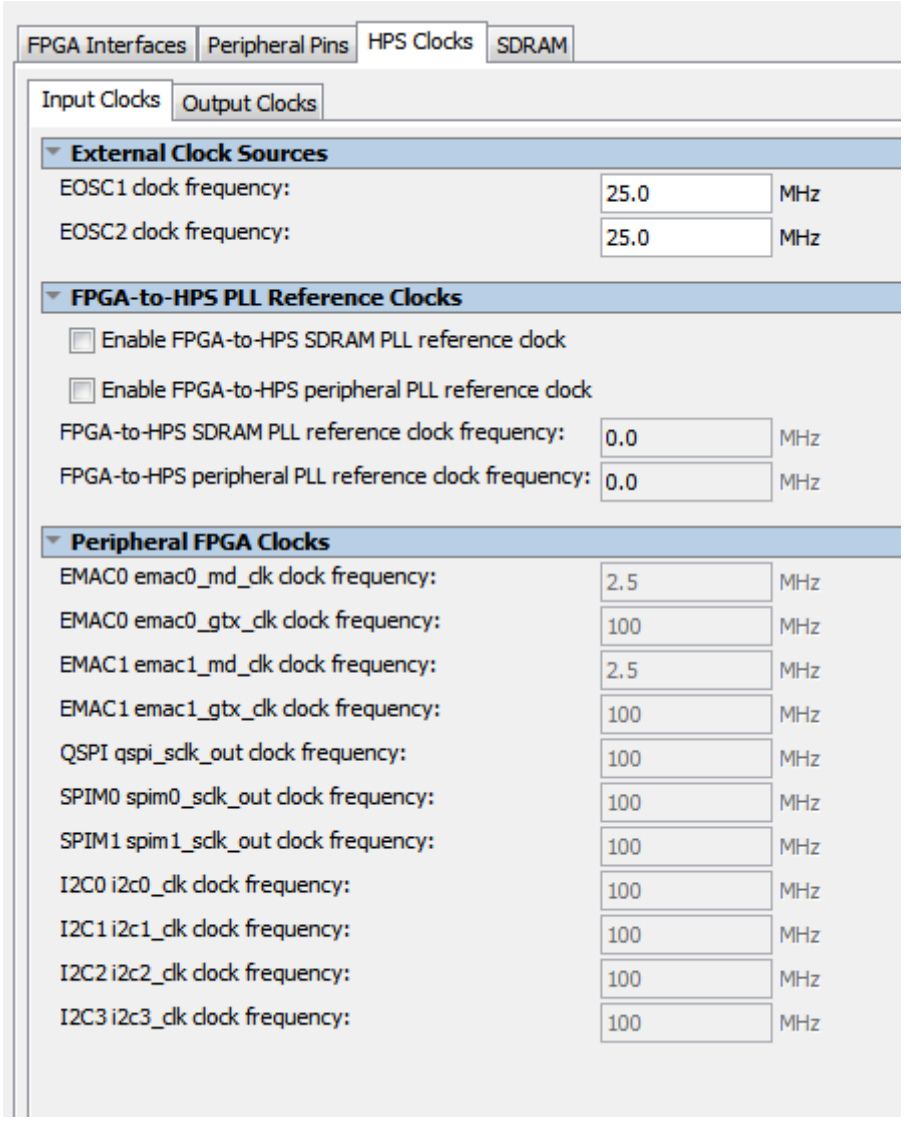
HPS-to-FPGA

- [ ] Enable CAN interrupts
- [ ] Enable clock peripheral interrupts
- [ ] Enable CTI interrupts
- [ ] Enable DMA interrupts
- [ ] Enable EMAC interrupts (for EMAC0 and EMAC1)
- [ ] Enable FPGA manager interrupt
- [ ] Enable GPIO interrupts
- [ ] Enable I2C-EMAC interrupts (for I2C2 and I2C3)

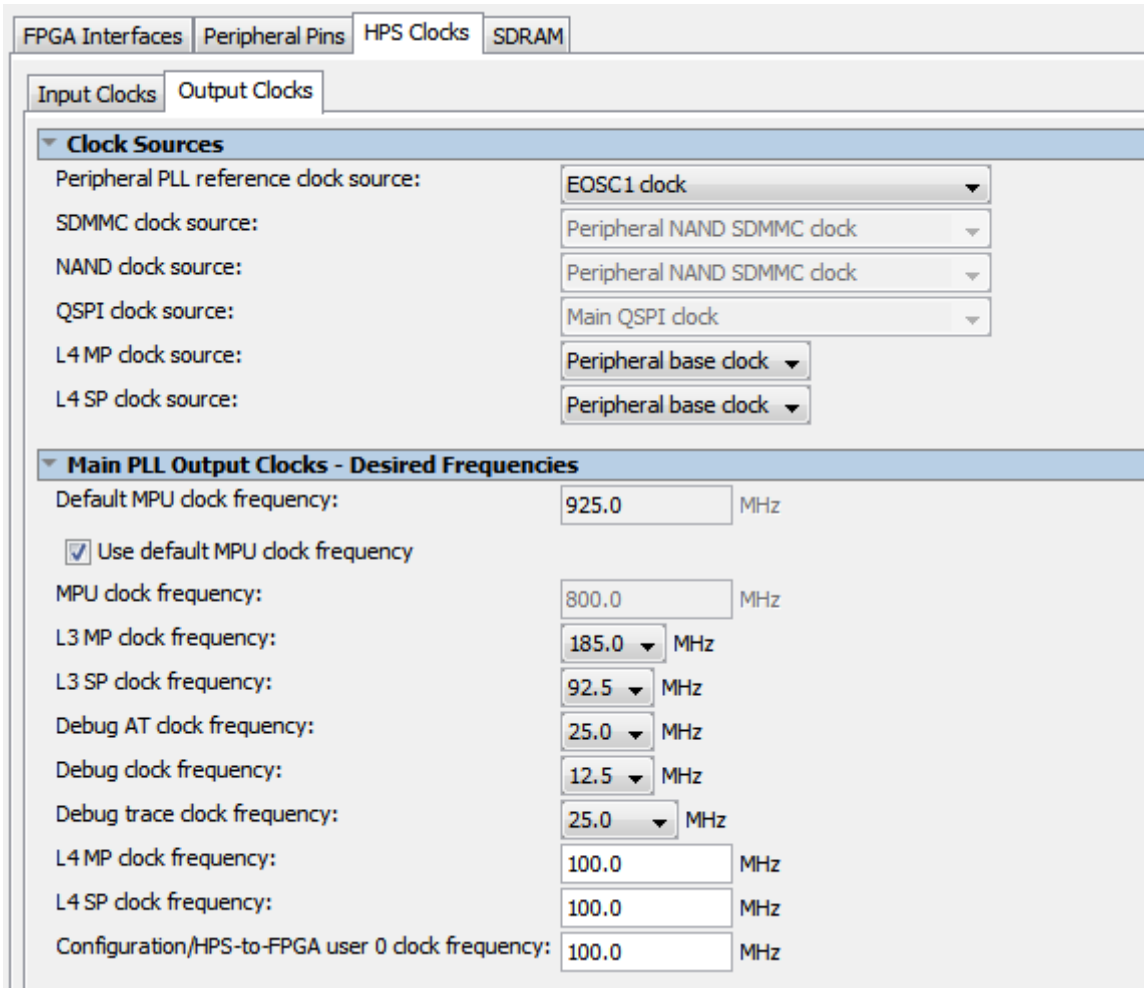


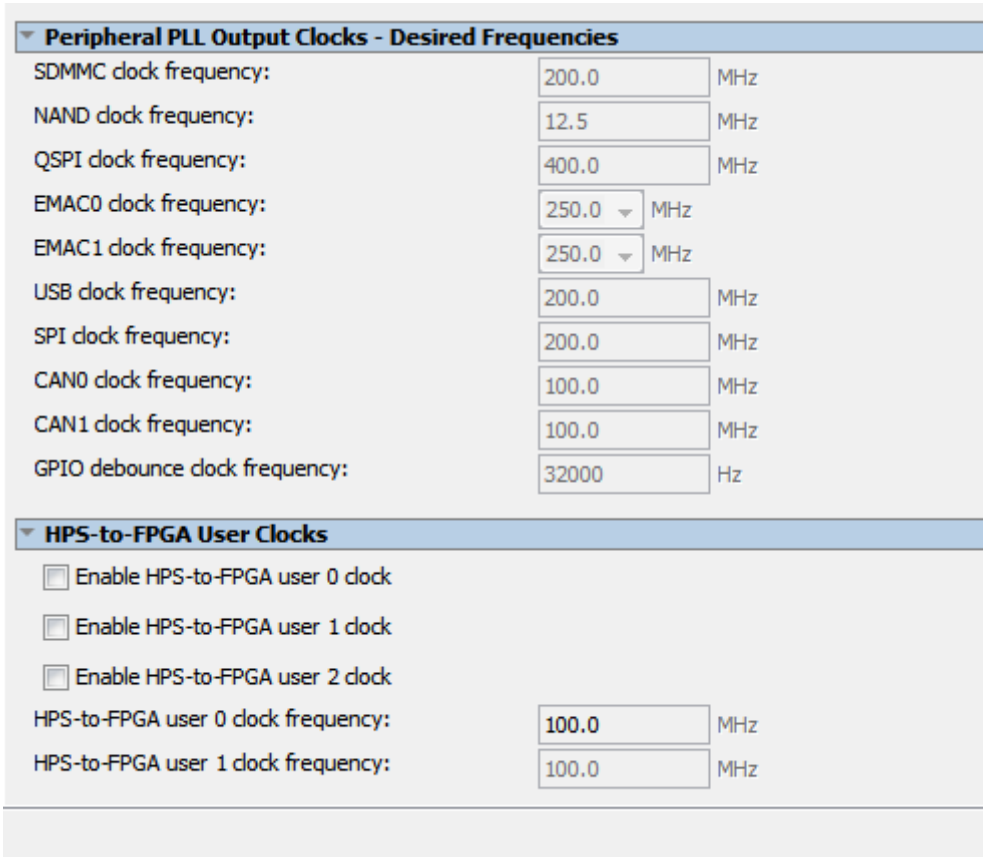
Peripheral Pins Default Settings (No change)

HPS Clocks - Input Clocks

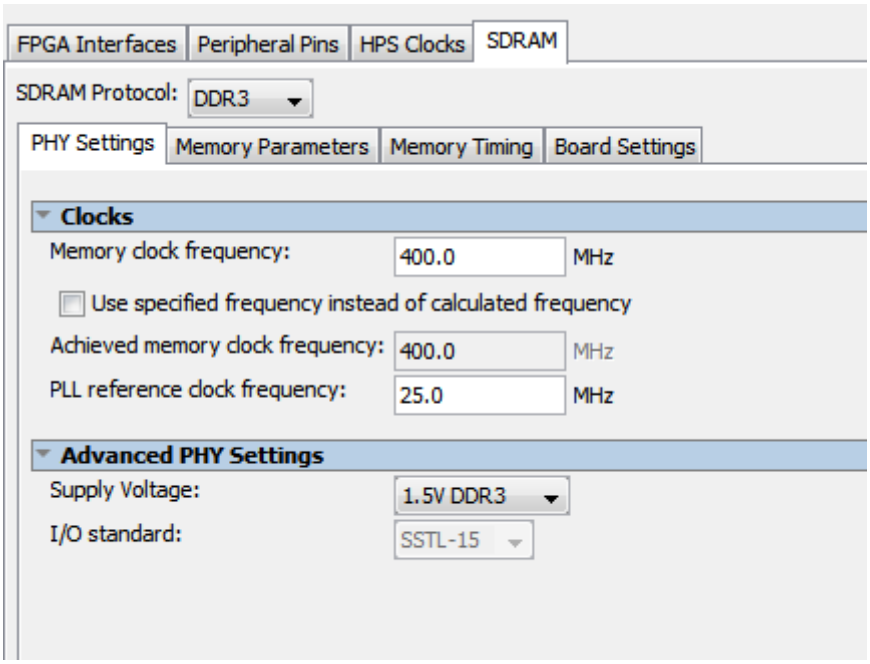


HPS Clocks - Output Clocks

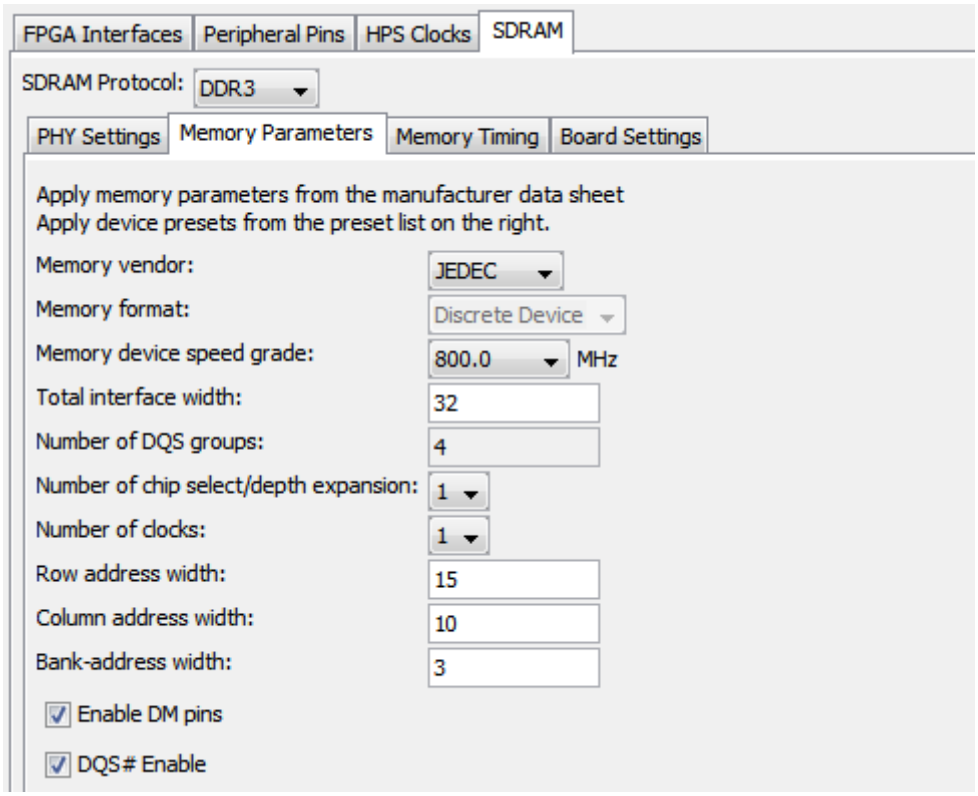


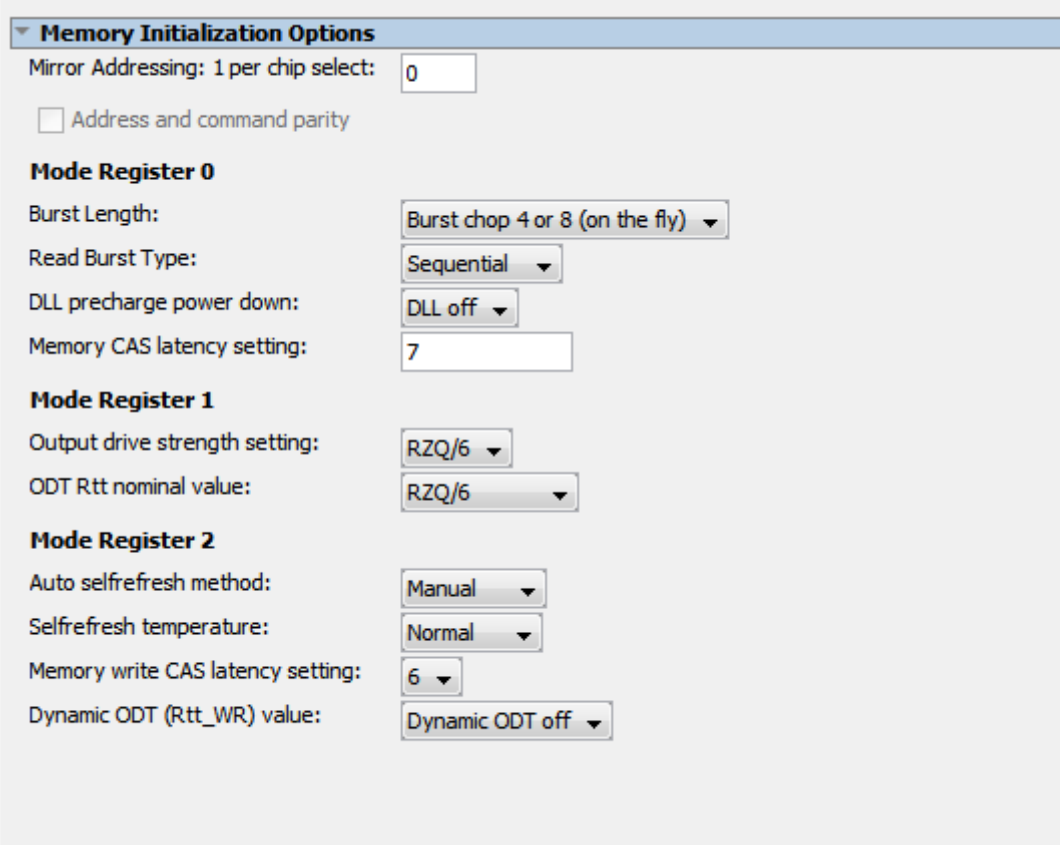


SDRAM - PHY Settings

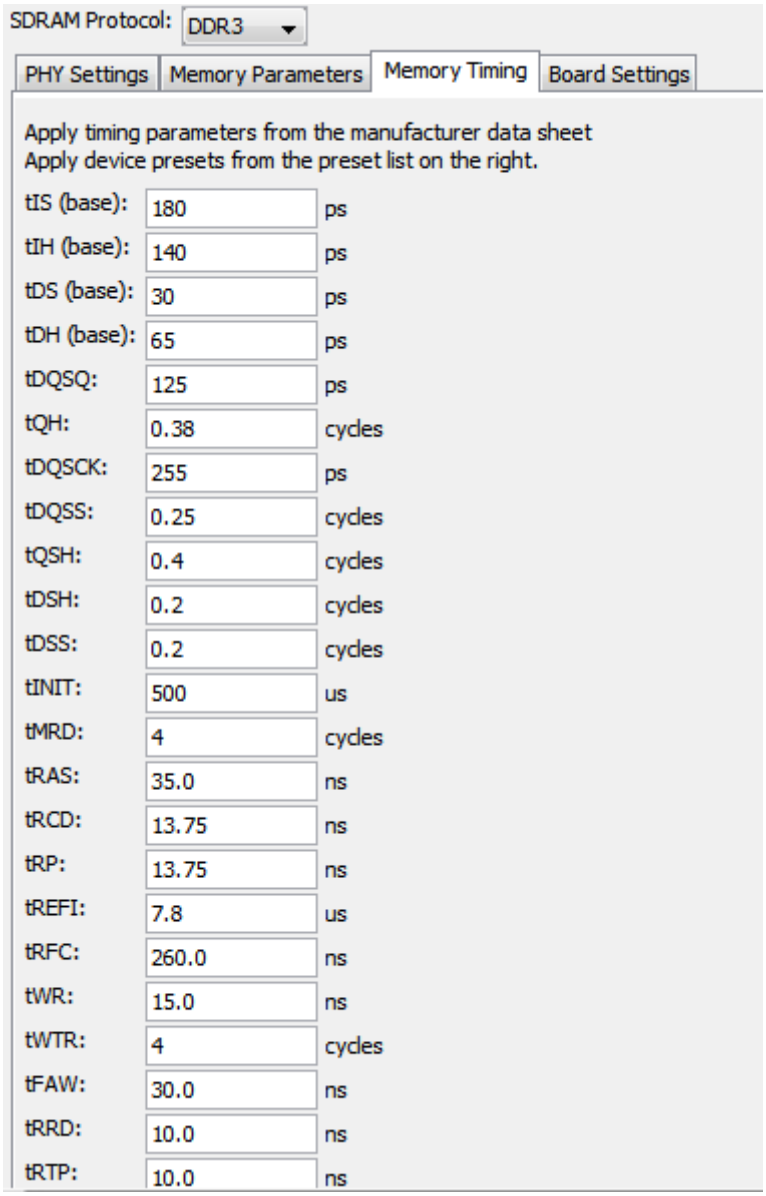


SDRAM - Memory Parameters

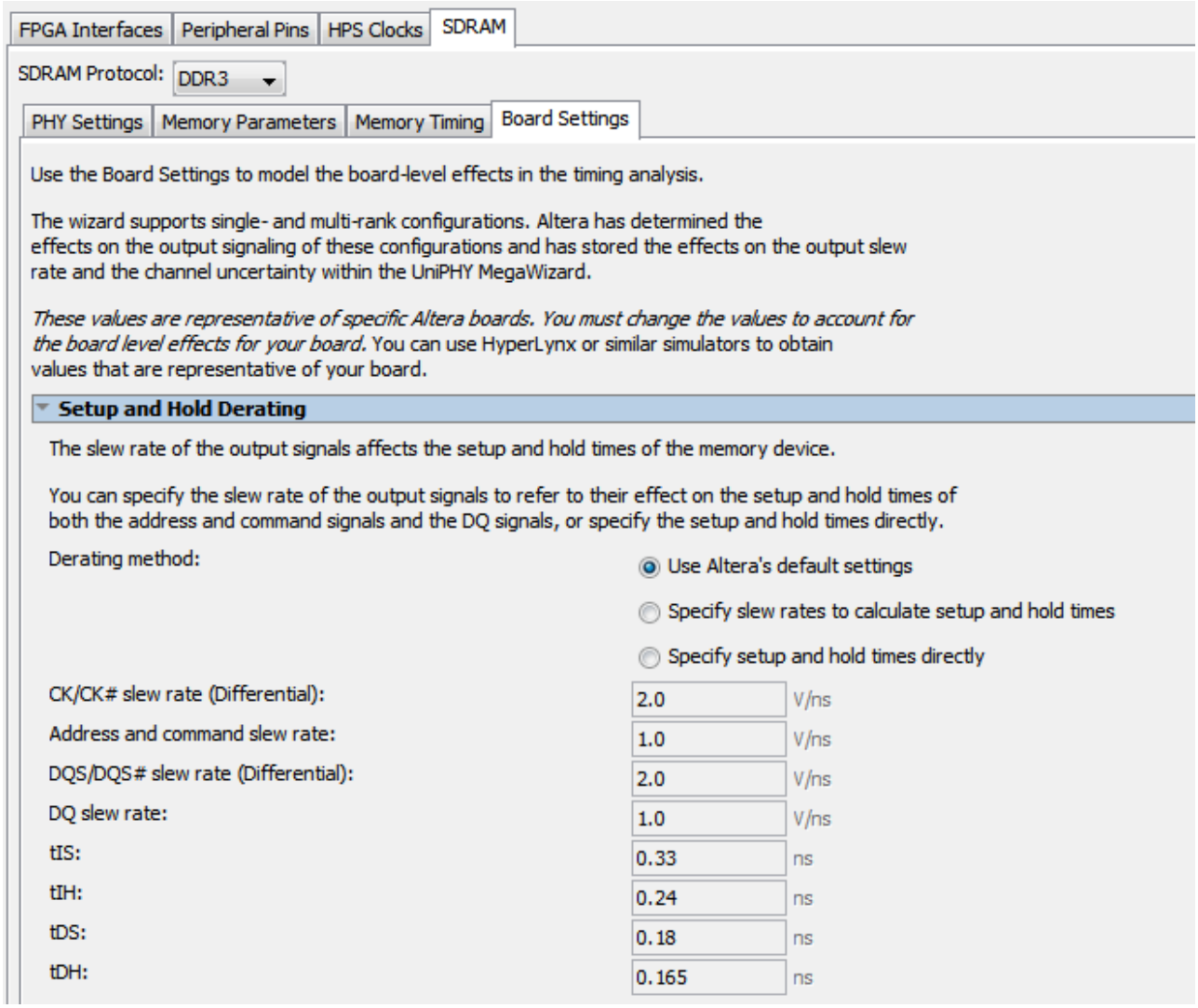


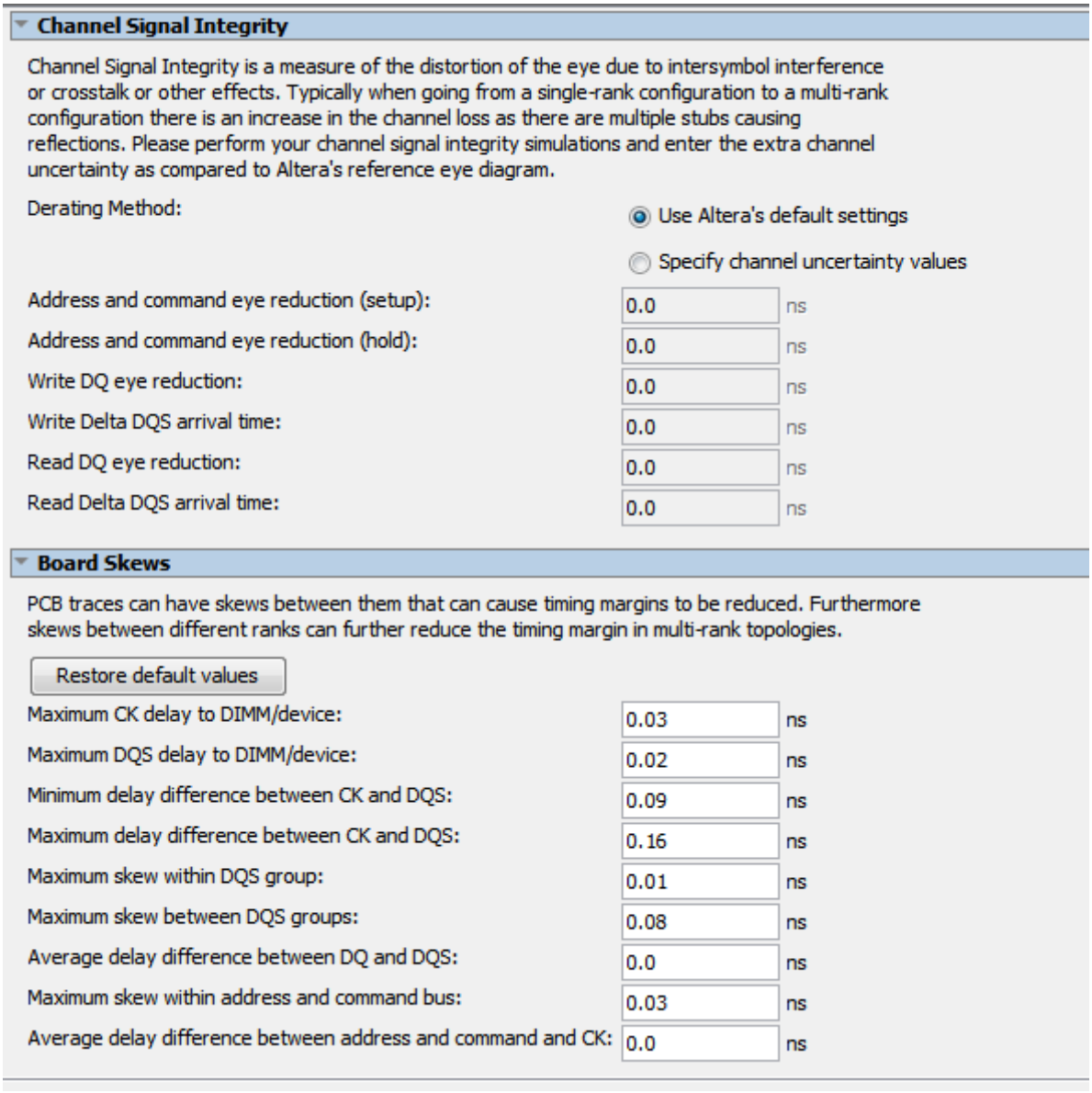


SDRAM - Memory Timing



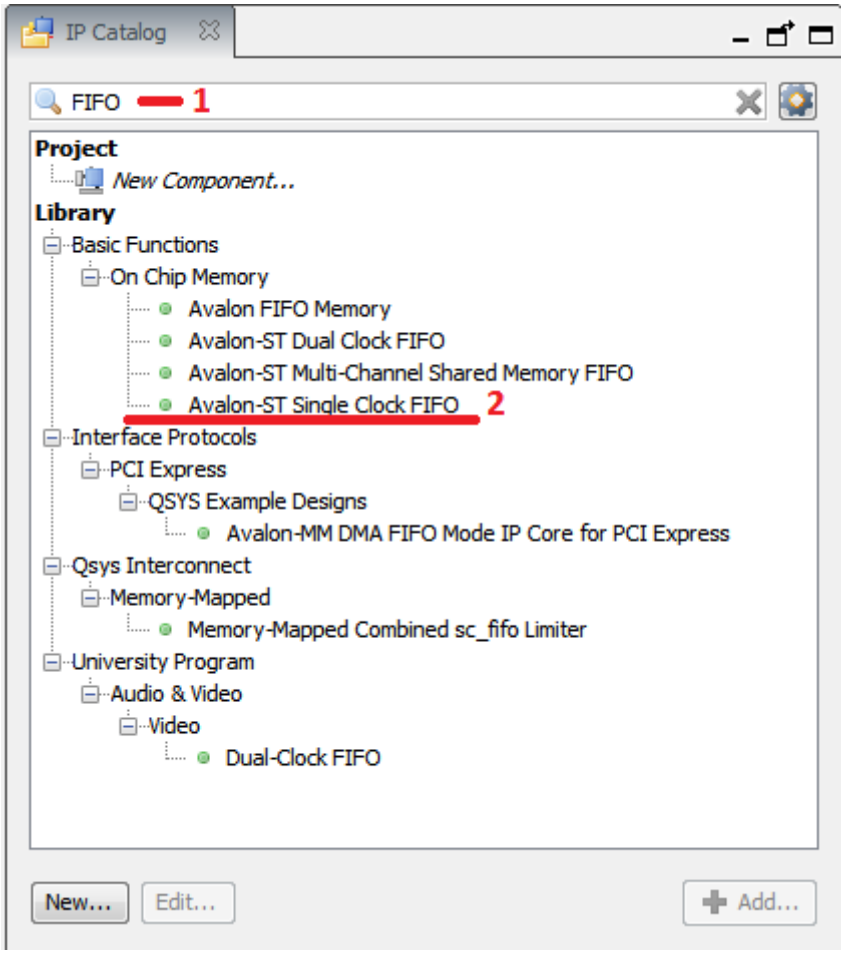
SDRAM - Board Settings


FPGA Interfaces
Peripheral Pins
HPS Clocks
SDRAM
SDRAM Protocol:
DDR3
PHY Settings
Memory Parameters
Memory Timing
Board Settings
Use the Board Settings to model the board-level effects in the timing analysis.
The wizard supports single- and multi-rank configurations. Altera has determined the effects on the output signaling of these configurations and has stored the effects on the output slew rate and the channel uncertainty within the UniPHY MegaWizard.
These values are representative of specific Altera boards. You must change the values to account for the board level effects for your board. You can use HyperLynx or similar simulators to obtain values that are representative of your board.
Setup and Hold Derating
The slew rate of the output signals affects the setup and hold times of the memory device.
You can specify the slew rate of the output signals to refer to their effect on the setup and hold times of both the address and command signals and the DQ signals, or specify the setup and hold times directly.
Derating method:
Use Altera's default settings
Specify slew rates to calculate setup and hold times
Specify setup and hold times directly
CK/CK# slew rate (Differential):
2.0
V/ns
Address and command slew rate:
1.0
V/ns
DQS/DQS# slew rate (Differential):
2.0
V/ns
DQ slew rate:
1.0
V/ns
tIS:
0.33
ns
tIH:
0.24
ns
tDS:
0.18
ns
tDH:
0.165
ns

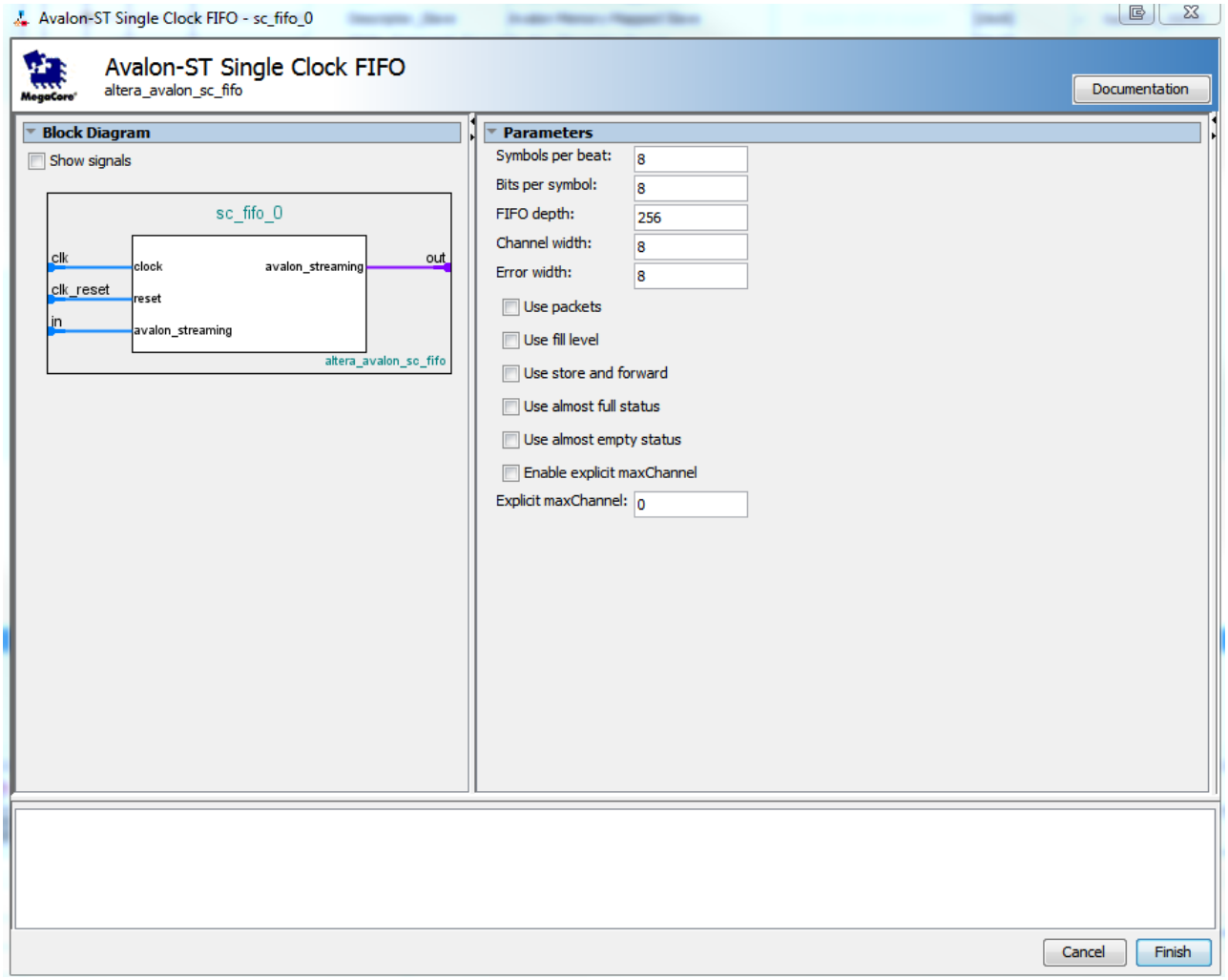


Save after configuring.

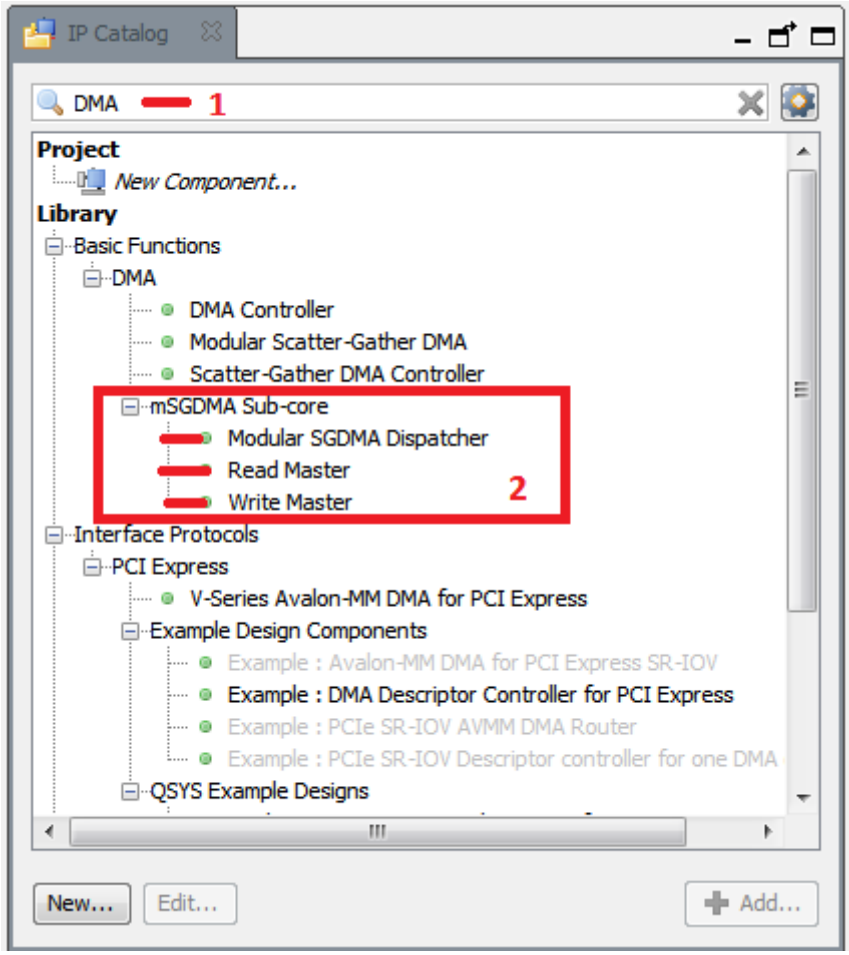
2. Add a FIFO. Search **FIFO** at **IP Catalog**, choose **Avalon-ST Single Clock FIFO** as following figure. Name it as **sc_fifo_0**.



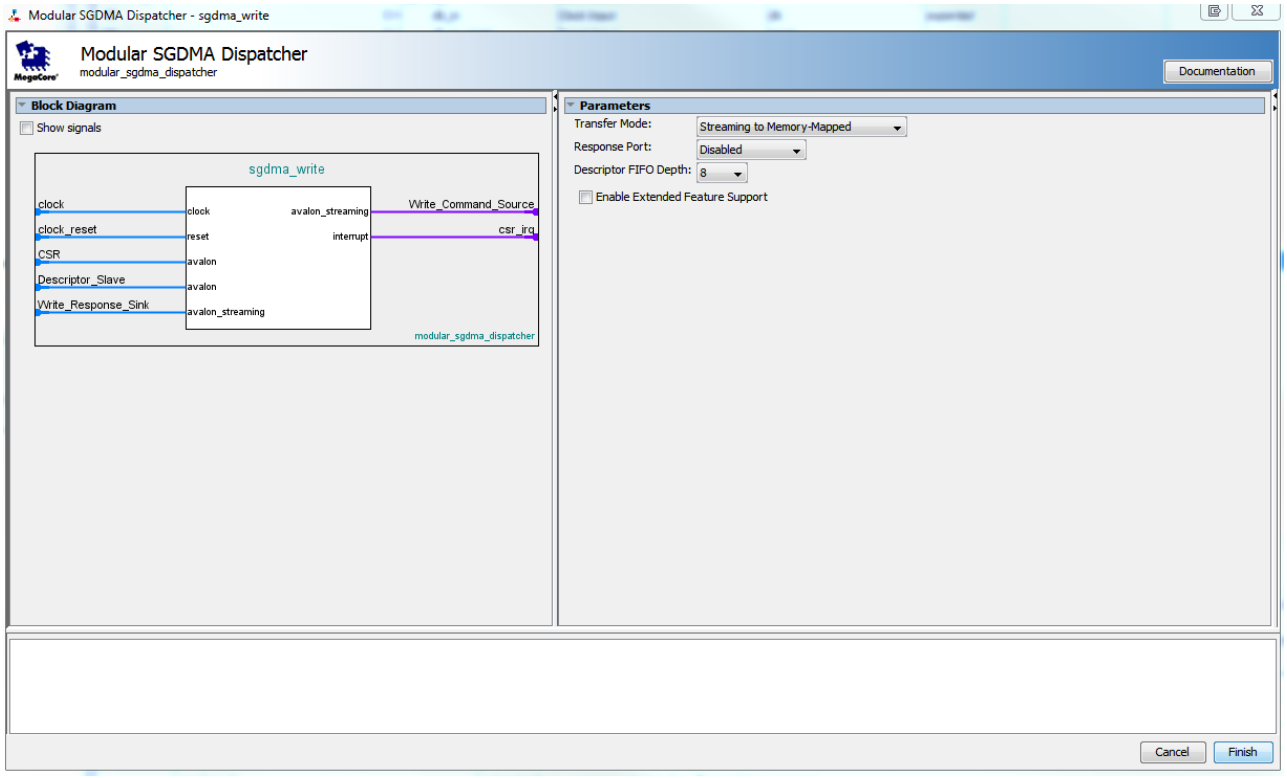
Configure FIFO as following figure.



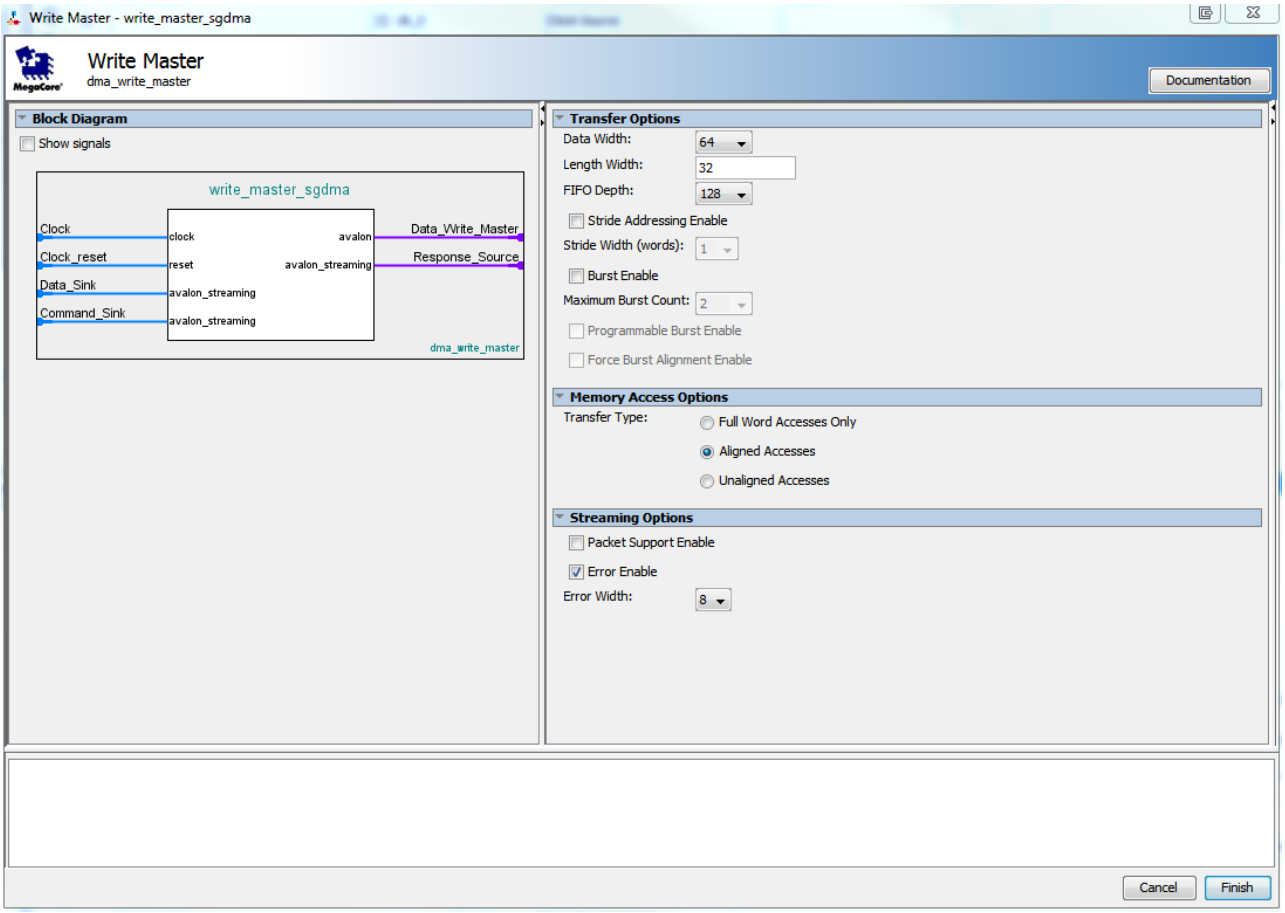
3. Adding modules to achieve the function of **transferring data from the FIFO in FPGA to HPS memory**. Section **III.0** gives an introduction of **mSGDMA**. Search **DMA** at **IP Catalog**, then you will see these three modules as following figure.



Firstly, add a **Modular SGDMA Dispatcher**, name it as **sgdma_write**, configure it as following figure.



Then add a **Write Master**, name it as **write_master_sgdma**, configure it as following figure.

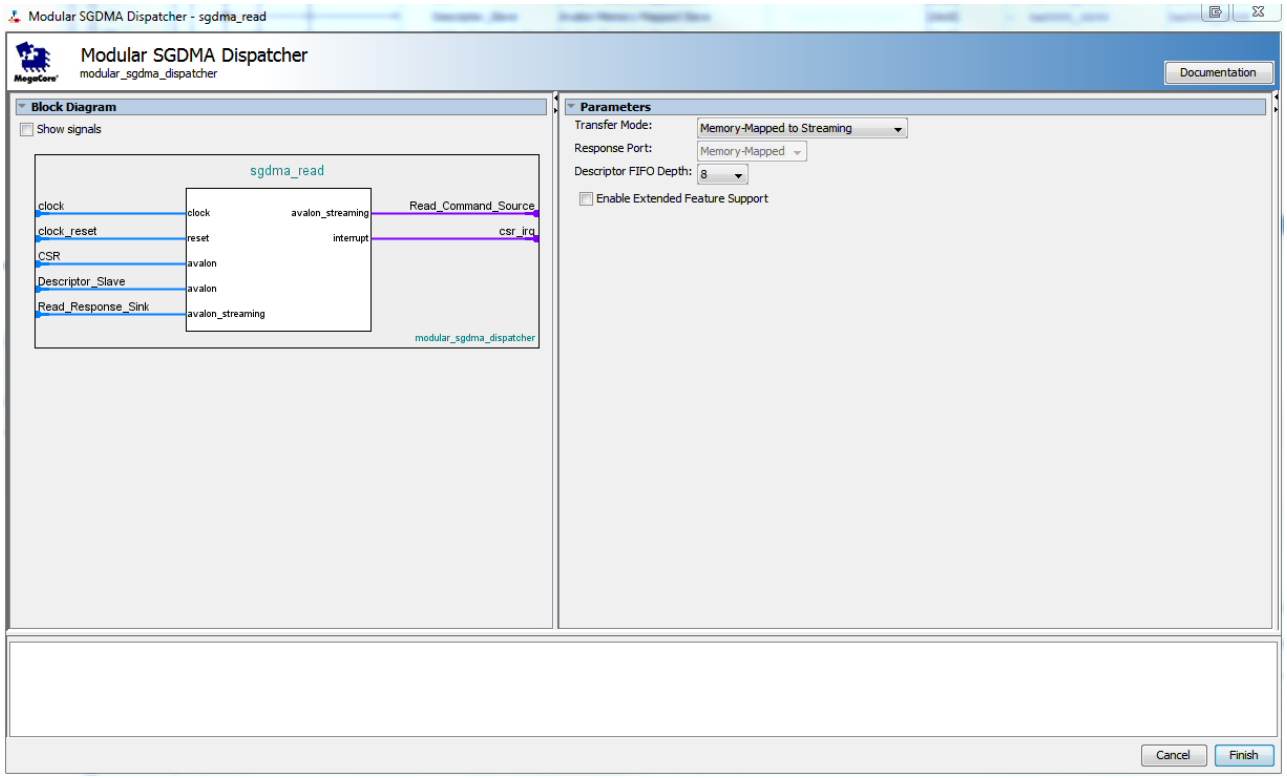


Connect **sgdma_write, write_master_sgdma, and FIFO** together. Following is the list of several important connections.

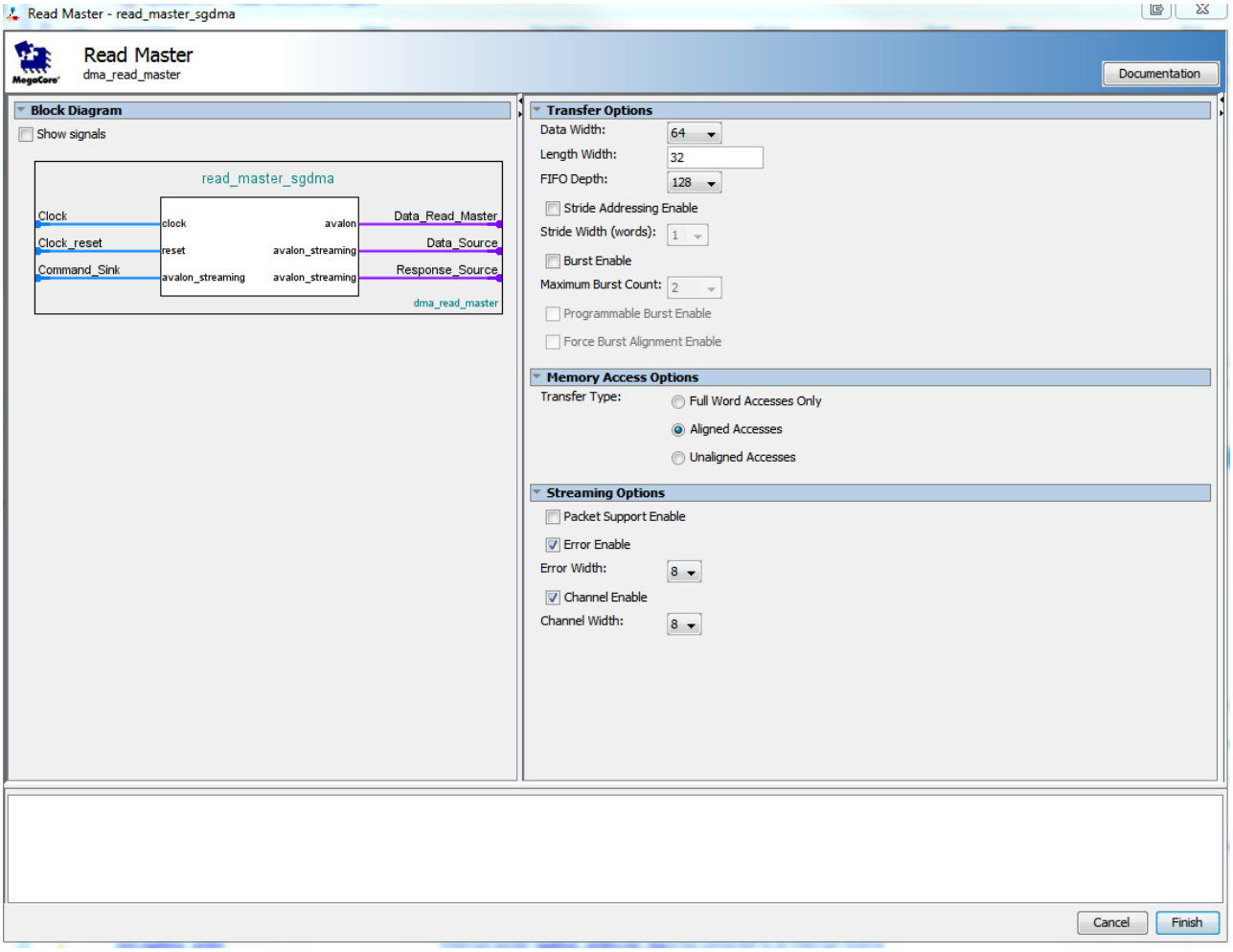
> **sgdma_write**::CSR — **hps_0**::h2f_lw_axi_master
> **sgdma_write**::Descriptor_Slave — **hps_0**::h2f_lw_axi_master
> **sgdma_write**::Write_Command_Source — **write_master_sgdma**::Command_Sink
> **sgdma_write**::Write_Response_Sink — **write_master_sgdma**::Response_Source
> **write_master_sgdma**::Data_Write_Master — **hps_0**::f2h_sdram0_data
> **write_master_sgdma**::Data_Sink — **sc_fifo_0**::out

The function of **transferring data from the FIFO in FPGA to HPS memory** has been done. The user program will control and send commands to **sgdma_write** via **Lightweight HPS-to_FPGA interface**. The **sgdma_write** then control **write_master_sgdma** to read data from FIFO (via Avalon ST interface), and send these data to HPS memory (via Avalon MM interface).

4. Adding modules to achieve the function of **reading data from HPS memory to FPGA**. Step 4 is similar to step 3.
   Firstly, add a **Modular SGDMA Dispatcher**, name it as **sgdma_read**, configure it as following figure.



Then add a **Read Master**, name it as **read_master_sgdma**, configure it as following figure.



Connect **sgdma_read, read_master_sgdma, and FIFO** together. Following is the list of several important connections.

> **sgdma_read**::CSR — **hps_0**::h2f_lw_axi_master
> **sgdma_read**::Descriptor_Slave — **hps_0**::h2f_lw_axi_master
> **sgdma_read**::Read_Command_Source — **read_master_sgdma**::Command_Sink
> **sgdma_read**::Read_Response_Sink — **read_master_sgdma**::Response_Source
> **read_master_sgdma**::Data_Read_Master — **hps_0**::f2h_sdram1_data
> **read_master_sgdma**::Data_Source — **sc_fifo_0**::in

The function of **reading data from HPS memory to FPGA** has been done. The user program will control and send commands to **sgdma_read** via **Lightweight HPS-to_FPGA interface**. The **sgdma_read** then control **read_master_sgdma** to read data from HPS memory (via Avalon MM interface), and send these data to FIFO (via Avalon ST interface). Other modules in FPGA can get the data from FIFO.

5. Adjust the other configuration, such as connecting clock and reset to every module, changing address map to avoid the conflicts. When there is no error in **Messages** window, save as **soc.qsys** and can be used in section of **QUICK IMPLEMENTATION**.

III.2. Software Part

NOTICE: There was a problem during developing this project. When FPGA read HPS memory by Avalon MM, all system will collapse after rising the signal of 'read' from 0 to 1. **Solution:** After pushing down the power switcher, wait until we can log in Linux. Then press the **HPS_RST** key to restart the HPS and the problem will disappear.

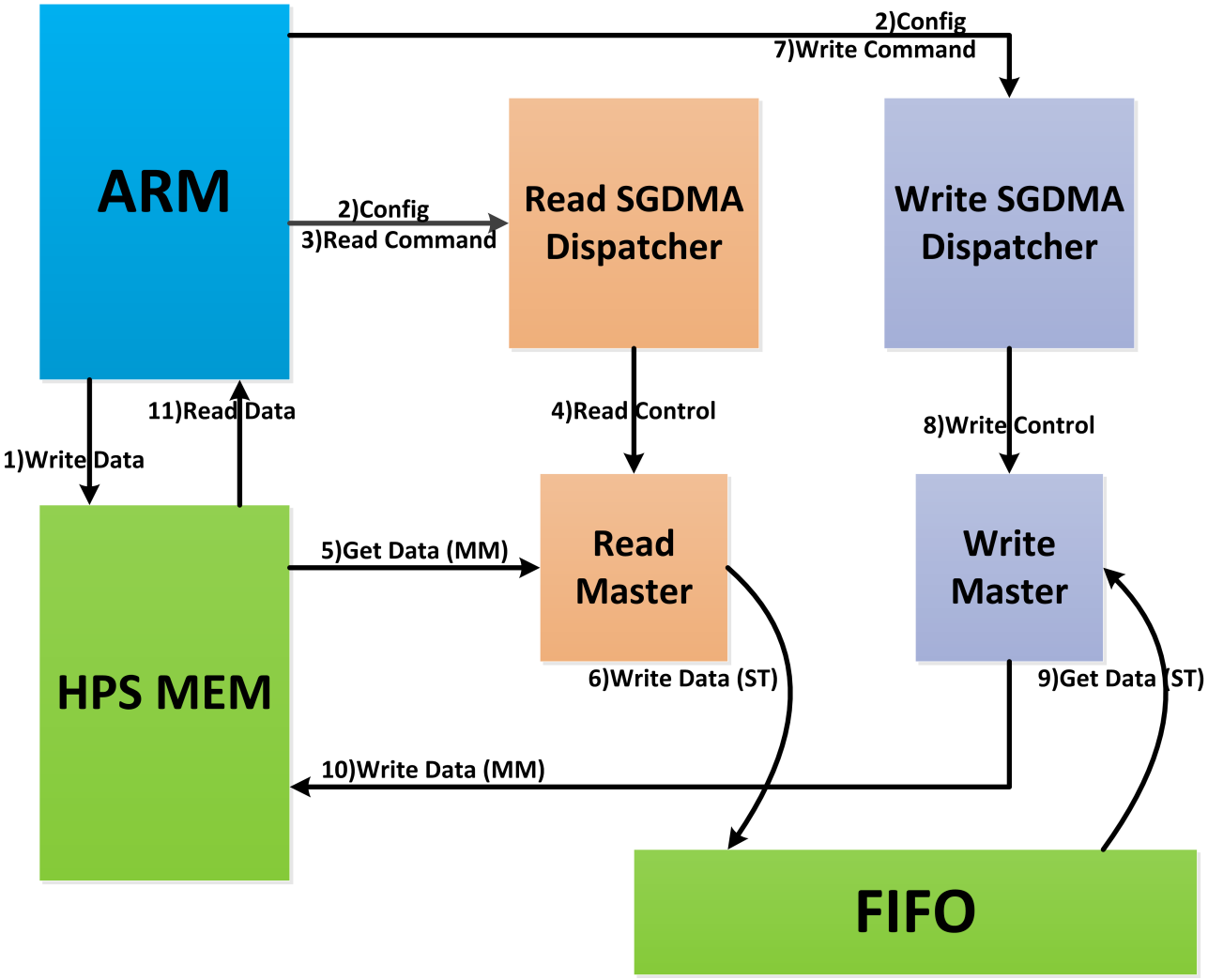
/RF-Board/RW HPS SDRAM by FPGA/SW is my software development folder.

/RF-Board/RW HPS SDRAM by FPGA/SWFile contains three ARM-Linux program, and one linux script. They can be used directly on ARM-Linux.

> **startram:** Set correct register value about fpga2sdram. **NOTICE:** input HEX number.
> Introduction to these registers: rst || cfg || default
> Recommandation settings: rst: 3fff, cfg: 0, default: 0.
> **rwmem:** This program can display the contents of certain part of memory. User also can write certain data (100+offset) into memory. Start address and data number are set by user. **Data number < 1024**.
> **memwrite:** This program will send commands to FPGA and excute function of transferring between FPGA and HPS.
> **en_bridge.sh:** This bash script will enable all three FPGA-HPS bridges (HPS2FPGA, FPGA2HPS, LWHPS2FPGA). When you first boot up the board, the bridges are disabled by default. You must enable the bridge before sending commands to FPGA.
> **NOTE:** Open **en_bridge.sh** in Linux, maybe you will see some ^M. It is caused by different defination of **return** under DOS and Unix/Linux. **Solution**

Procedure of *memwrite*:

1. ARM send configuration info and read command to **FPGA Read Function Part** (mSGDMA MM-ST mode set, built in **III.1. Hardware/Qsys Part – QSYS PART – Step 4**).
2. **FPGA Read Function Part** fetch the data from HPS memory and send the data to FIFO in FPGA. (Start address is 0x30000000 (768MB Position), number of bytes set by user.)
3. ARM send configure info and write command to **FPGA Write Function Part** (mSGDMA ST-MM mode set, built in **III.1. Hardware/Qsys Part – QSYS PART – Step 3**).
4. **FPGA Write Function Part** fetch the data from FIFO in FPGA, and write the data to HPS memory. (Start address is 0x28000000 (640MB Position), number of bytes set by user.)
5. Then ARM-Linux program can read the data.



III.3. Divide the HPS SDRAM into Two Parts

For the safe of Linux system, the SDRAM of HPS should be divided into two parts.

The first part will be used by Linux system only. All kernel modules and user programs can access this part memory as usual. FPGA should not access this part of memory.

The second part will be used by FPGA and Linux as a data exchaging space. FPGA can access this part via Avalon MM interface. Linux system cannot access this part as internal memory. It is an external memory to Linux, and its address is illegal to Linux. User programs and kernel modules on Linux need to use **mmap** to read or write this part memory, just like the procedure of controlling FPGA from Linux user programs.

The Method: (Refer: Writing to HPS Memory) (All finishen on SoCKit Board console)

1. Power on board, stop autoboot at the stage of countdown.
2. Input 'editenv mmcboot', then the console will show the content of mmcboot.

```
SOCFPGA_CYCLONE5 editenv mmcboot
edit: setenv bootargs console=ttyS0,115200 root=${mmcroot} rw rootwait;bootm
 ${loadaddr} - ${fdtaddr}
```

3. Add 'mem=512M' and press enter.

```
edit: setenv bootargs console=ttyS0,115200 root=${mmcroot} rw rootwait mem=5
12M;bootm ${loadaddr} - ${fdtaddr}
SOCFPGA_CYCLONE5
```

Note: In some case, **bootm** is replaced by **bootz**. The effect is unknown.

4. Input 'saveenv' to save environment variables.

```
SOCFPGA_CYCLONE5 saveenv
```

5. Boot system by restarting the board from the power switch.

Now the SDRAM of HPS has been diveided into two parts. The first part is 512MB, used by Linux. The second part is 512MB, used by FPGA and Linux as a data exchaging space.

III.4. Project Demonstration Procedure

Firstly, copy **startram, rwmem, memwrite** to ARM-Linux's folder: **/home/root**. Then PowerOn the board and program the FPGA by **mem.sof** or **fpga.rbf**(U-Boot). Login Linux on board and excute following steps.

1. Enable bridge, run **en_bridge_sh**.
2. Run **startram** program to set register. **[rst: 3fff, cfg: 0, default: 0]**.
3. Run **rwmem**. Write certain value to 0x30000000. [Start address: 30000000 / write(0) / Number:100]

4. Run **rwmem**. Display the first several decades memory of 0x30000000. [Start address: 30000000 / Read(1) / Number:10]
5. Run **rwmem**. Display the first several decades memory of 0x28000000. [Start address: 28000000 / Read(1) / Number:10] Then compare the contents of 0x28000000 with the contents of 0x30000000.
6. Run **memwrite**. [read number: 100 / write number: 100]
7. Run **rwmem**. Display the first several decades memory of 0x28000000. [Start address: 28000000 / Read(1) / Number:10] You will see the new contents of 0x28000000, which is same with 0x30000000.

III.5. Reference

1. Using **mmap** to access FPGA-HPS bridge and externel memory (the 2nd part memory described in III.3). Software program reference: Howard Mao - Exploring the Arrow SoCKit Part III - Controlling FPGA from Software - Setting the Delay from the HPS
2. The description of API functions used to control SGDMA Dispatcher: **/RF-Board/ImportantDocuments/3-Modular_SGDMA_Dispatcher_Core_UG.pdf**, Page 20 ~ Page 40. Related head files and source files location: **/RF-Board/RW HPS SDRAM by FPGA/SGDMA/**.

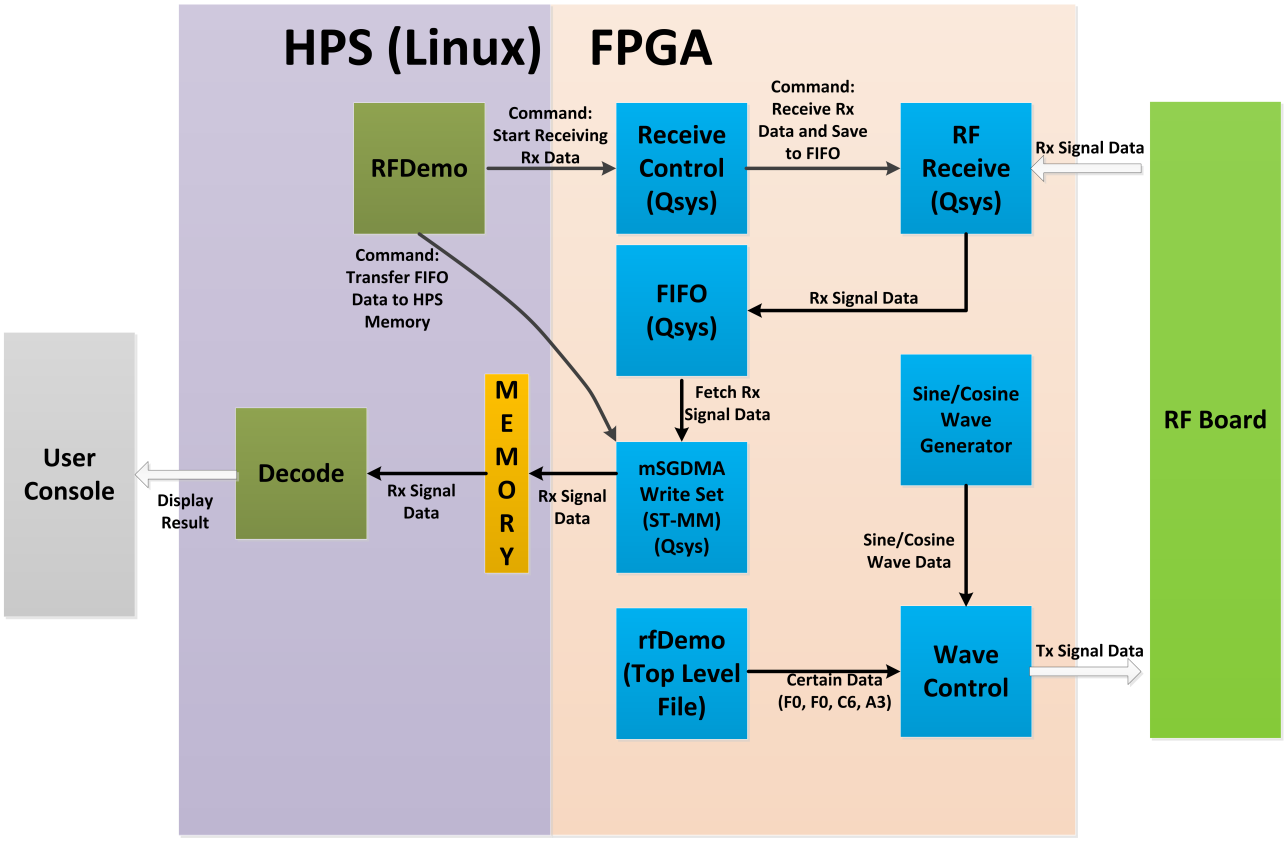
> **csr_regs.h:** Defination of CSR register, and related API functions.
> **descriptor_regs.h:** Defination of Descriptor register, and related API functions.
> **response_regs.h:** Defination of reponse register, and related API functions.
> **sgdma_dispathcer.c:** API functions used to control dispatcher.
> **NOTE:** These files cannot be used directly in software developing, because they are used in NIOS II system. You need to rewrite them.

IV. RF Board

This project will implement a simple system of sending and receiving certain data via loop of RF Board. Because of the time limitation and unexpected problem, this project is not good enough. The important information and notices of Myriad-RF Board should be read carefuly.

NOTE: This project is developed based on the MATLAB simulation program (proposed in **IV.2. Theory Part**). If you would like to understand the structure of this project, I suggest you could read this simulation program first. It is an easy simulation program.

Diagram of Project Structure



IV.1. Documentations of Myriad-RF Board and Zipper

IV.1.1. Documentations

There are so many useful documentations about the RF board. In this section, I will introduce all of them and give out the path.

1. **LMS6002D IF-RF LoopBack Options.pdf (Path:/RF-Board/ImportantDocuments/MyriadRF/):** This file give out the diagram of all the loop back options inside the RF-Board. You can debug the RF things with this diagram.
2. **LMS6002Dr2-DataSheet-1.2r0.pdf (Path:/RF-Board/ImportantDocuments/MyriadRF/):** This is the data sheet of RF board.
3. **LMS6002Dr2-Programming and Calibration Guide-1_1r4.pdf (Path:/RF-Board/ImportantDocuments/MyriadRF/):** Before start data transimitting, the RF board needs calibration. This file give all the information about this procedure.
4. **Zipper Development Kit_1 0r5.pdf (Path:/RF-Board/ImportantDocuments/Zipper/):** Introduce almost things about developing with RF-Board and Zipper board, such as configuring the RF board with a Windows software, pin assignments, confinguring Zipper board, and so on.
5. **Zipper_v.2_Schematics_r.2.pdf (Path:/RF-Board/ImportantDocuments/Zipper/):** Will be used in pin assignments.
6. **HSMC_Pin_Assignment.xlsx (Path:/RF-Board/ImportantDocuments/Zipper/):** The pin assignments of FPGA. For the pin assignment of SoCKit, **4-SoCKit_User_manual.pdf, Page 25 ~ 27, Path: /RF-Board/ImportantDocuments**

And Myriad-RF has a GitHub source (reference-development-kit). Some useful programs, documents can be found here.

IV.1.2. Important Information about RF Board

This section will introduce some important information about RF Board, which were big troubles during developing.

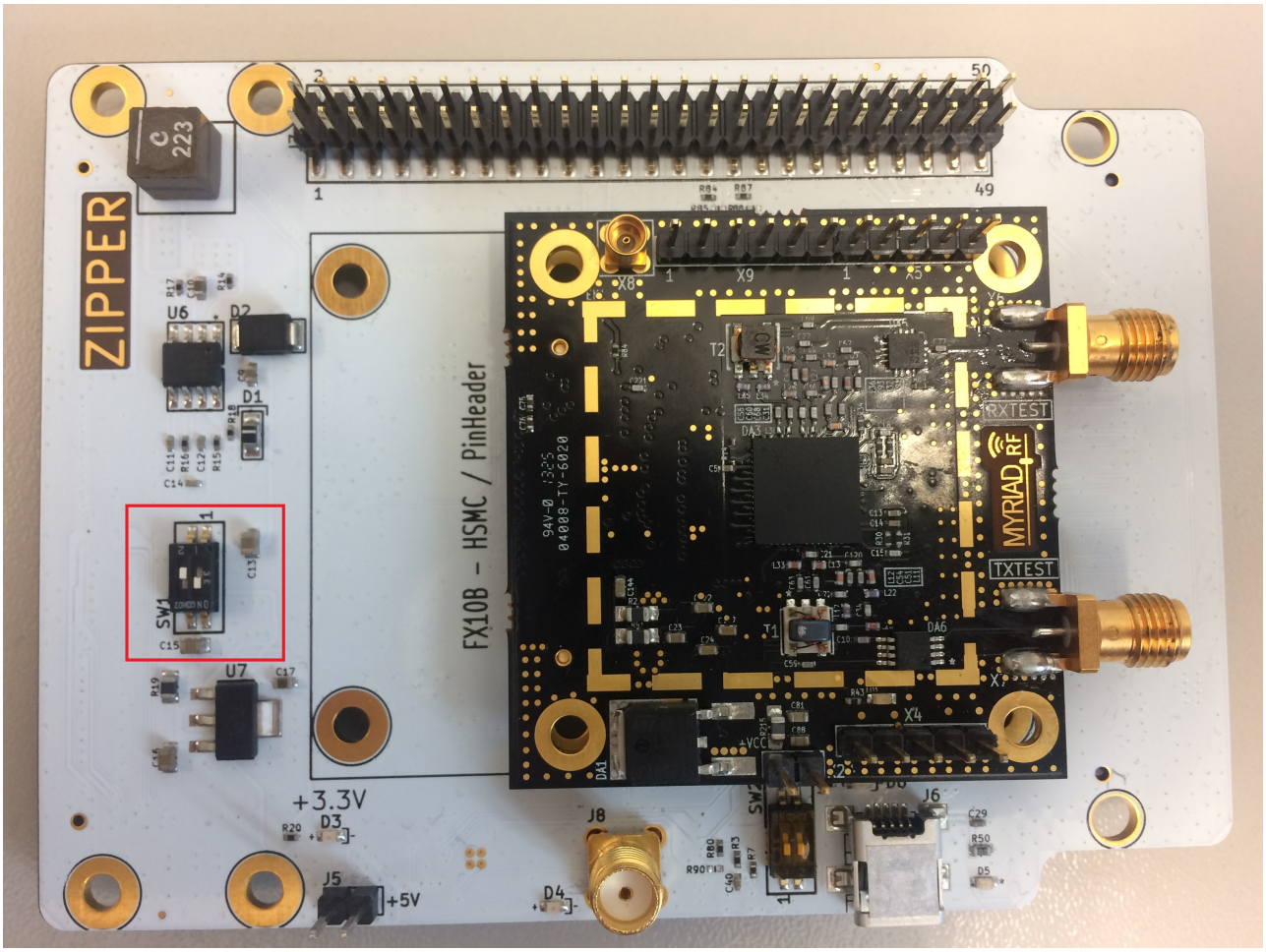
1. **ADC/DAC number format:** The ADC and DAC of RF board use signed 12-bit integer as following:

   ```
   0x7FF ~ 2047
   ……
   0x000 ~ 0
   0xFFF ~ -1
   ……
   0x801 ~ -2047
   0x800 ~ -2048
   ```

2. **LoopBack Frequency Limitation on Baseband Signal:** By experiment, the base band frequency should be higher than 1MHz. Low baseband frequency signal cannot pass the inner LoopBack path. If you want to send low baseband frequency wave, you can connect the Rx and Tx output by a RF guide line.
3. **Wave Choice:** Inphase part: DPSK with sine wave. Quadra part: DPSK with cosine wave.
4. **Sending Wave Data:** You should always sending wave data to RF Board, though there is no infomation need to be sent. Because of the response time of LPF, sending wave discontinuously will create the problem of lossing data.
5. **Enable Signal of Rx Part:** I suggest that **do not** control the **Enable Signal (RXEN)** of Rx part by users, you can always enable the Rx (i.e. **RXEN <= '1'**). Rx needs a lot of warming time to work properly. It is hard to know when the Rx data can be used.

IV.1.3. Important Information about Zipper Board

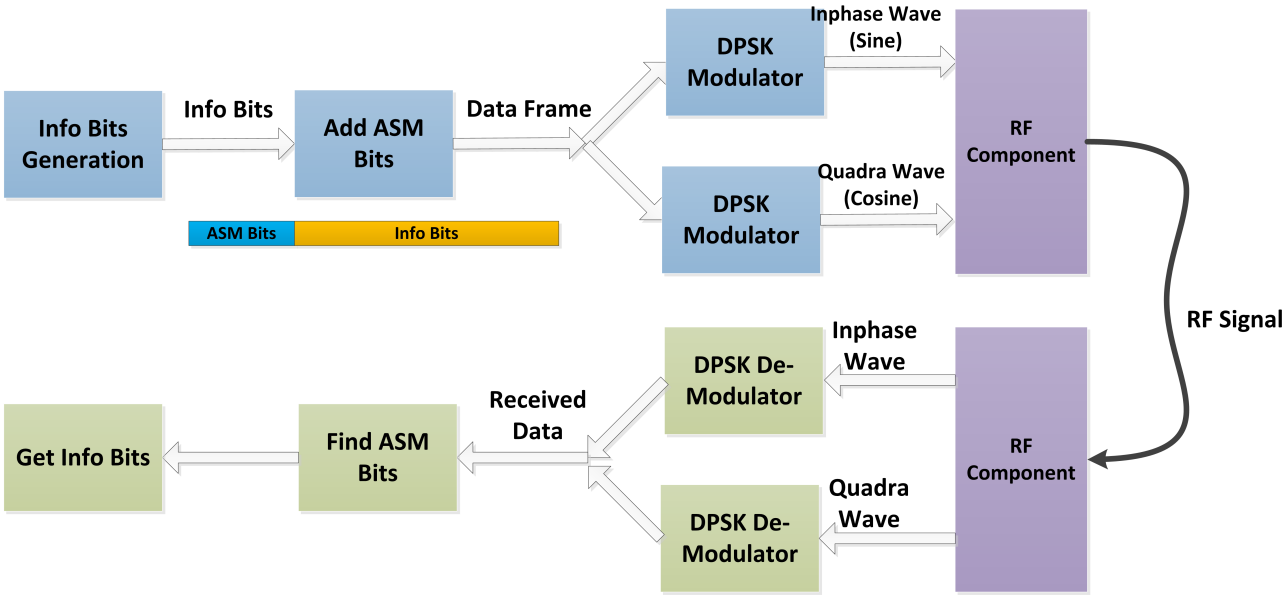
1. **Powered by FPGA:** New zipper board needs configuration to be powered by FPGA. **SW1 - 1 - ON**



2. **Calibration by FPGA:** Without configuring Zipper Board, we cannot do calibration by FPGA, can only by software on computer via USB. More information in **Zipper Development Kit_1 0r5.pdf, Page 50, 4.6 SPI Options.**

IV.2. Theory Part

In this project, I use DPSK both in Inphase part and Quadra part to transimit data. The basic procedure is shown as following diagram:



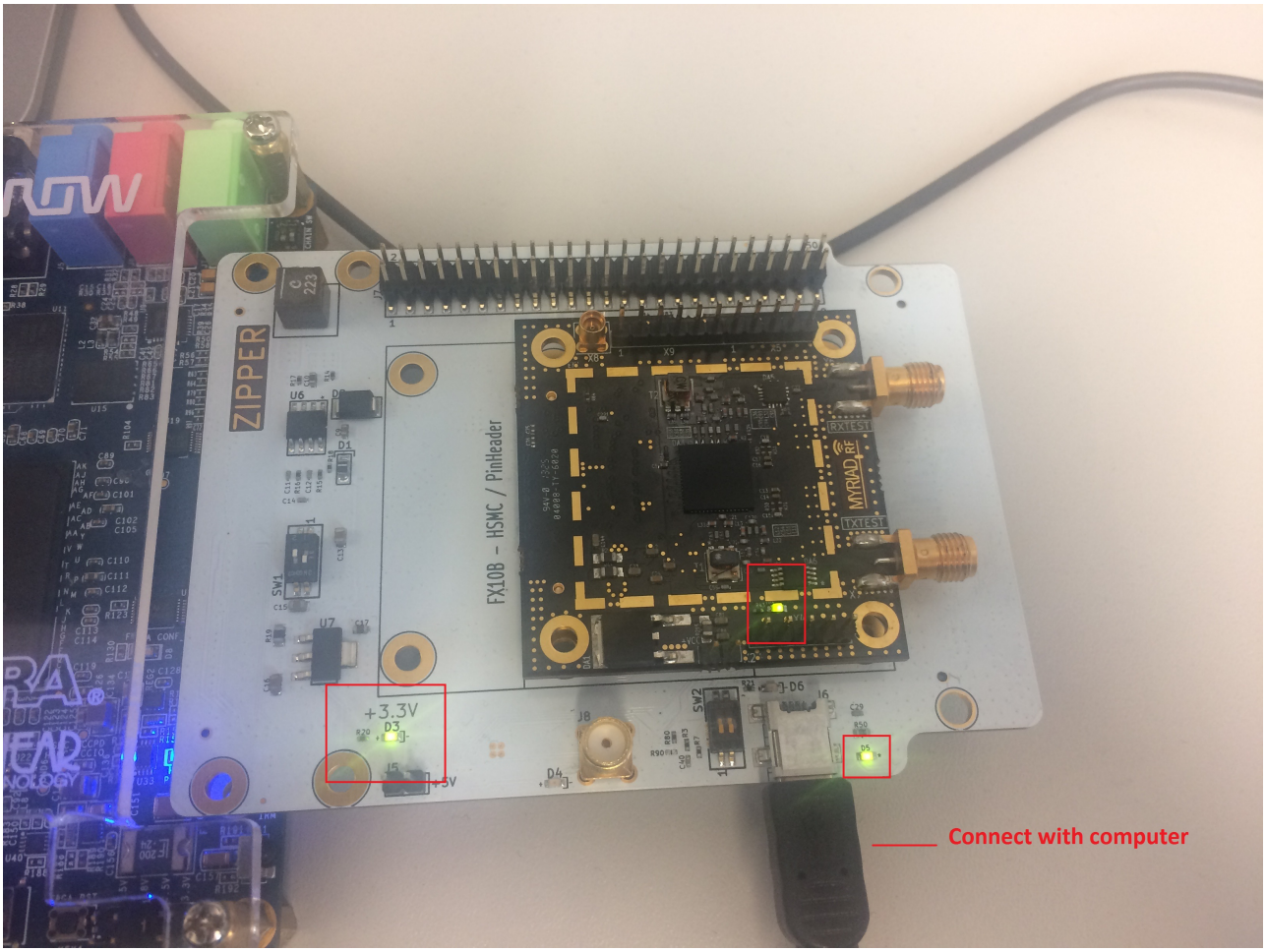
A MATLAB simulation program (**Path: /RF-Board/RF SendRec via Loop/MATLAB_RFDemo/**) can be used to understand the structure of this project. Run **RFDemo.m** to see the result, and you can read this program to understand the project structure.

NOTE: The project is developed based on this MATLAB program.

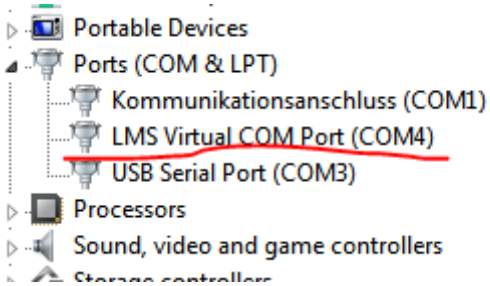
IV.3. RF Board Configuration Procedure

In this section, a software will be used: **control_LMS6002.exe (Path: /RF-Board/ImportantSource/)**. The detailed introduction about this software is presented in **Zipper Development Kit_1 0r5.pdf**. You can check the information about these setting in this file.

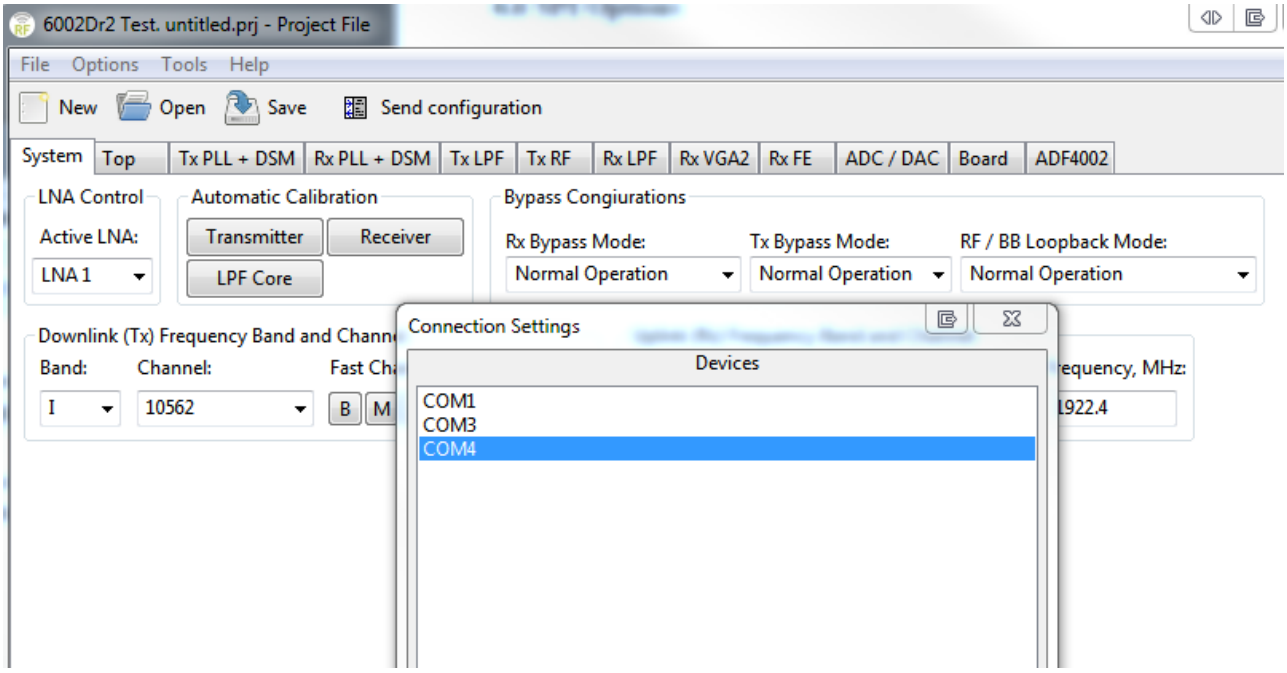
1. Connect RF board to Zipper board, and connect Zipper board to SoCKit board by HSMC. Make sure that the RF board can be powered by FPGA via HSMC. (**IV.1.3 - 1**)
2. Power on the FPGA. Then connect Zipper board with computer by USB cable as following figure. All three LEDs should be lighted as following figure.



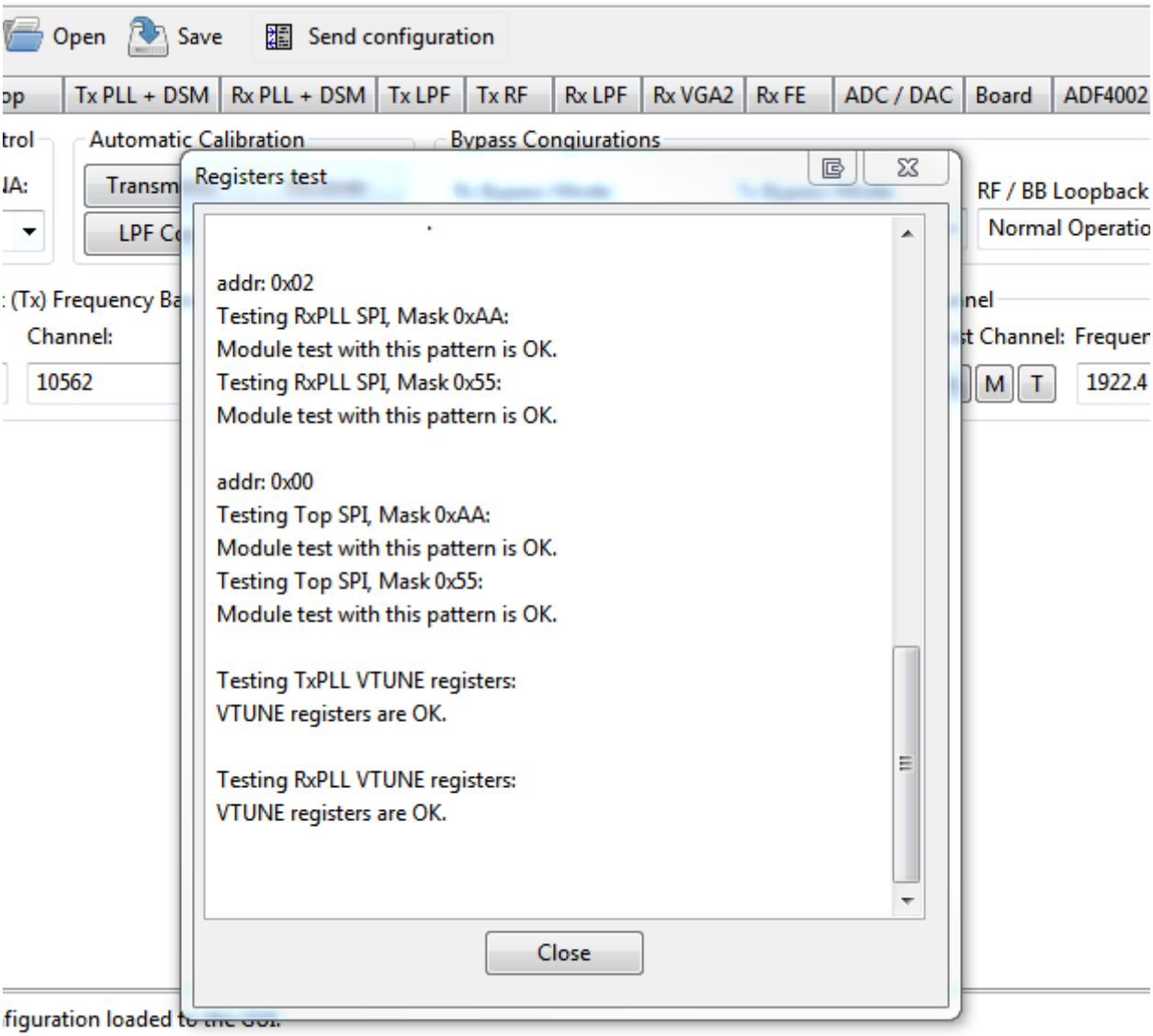
3. At the first time, the driver for zipper board should be installed. Driver location: **/RF-Board/ImportantSource/LMS VCP drivers/** After installation, check the COM number of the Zipper Board. In my case, it is COM4.



4. Firstly, disconnet the Zipper with computer, and power off the board. Secondly, power on the board, connect Zipper board with computer, open the software **control_LMS6002.exe**. Then click **Options -> Communication Settings**, choose the correct COM and click **OK**. In my case, it is COM4.

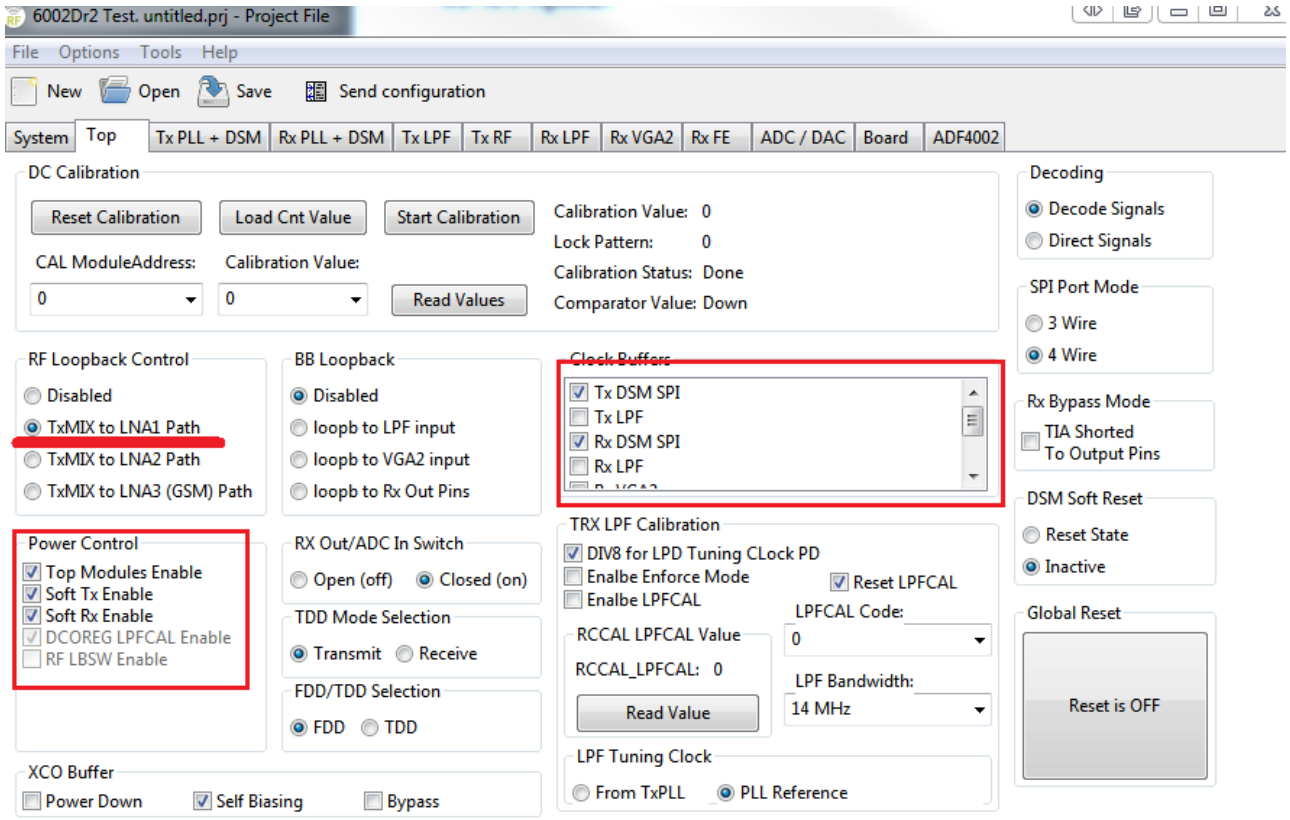


5. Click **Tools -> Registers Test**. Make sure all the test return *OK*. The there are some FAILED return, please test more times, or re-do step 4. If there are still some FAILED, it means this RF board is broken and need to change a new RF board.

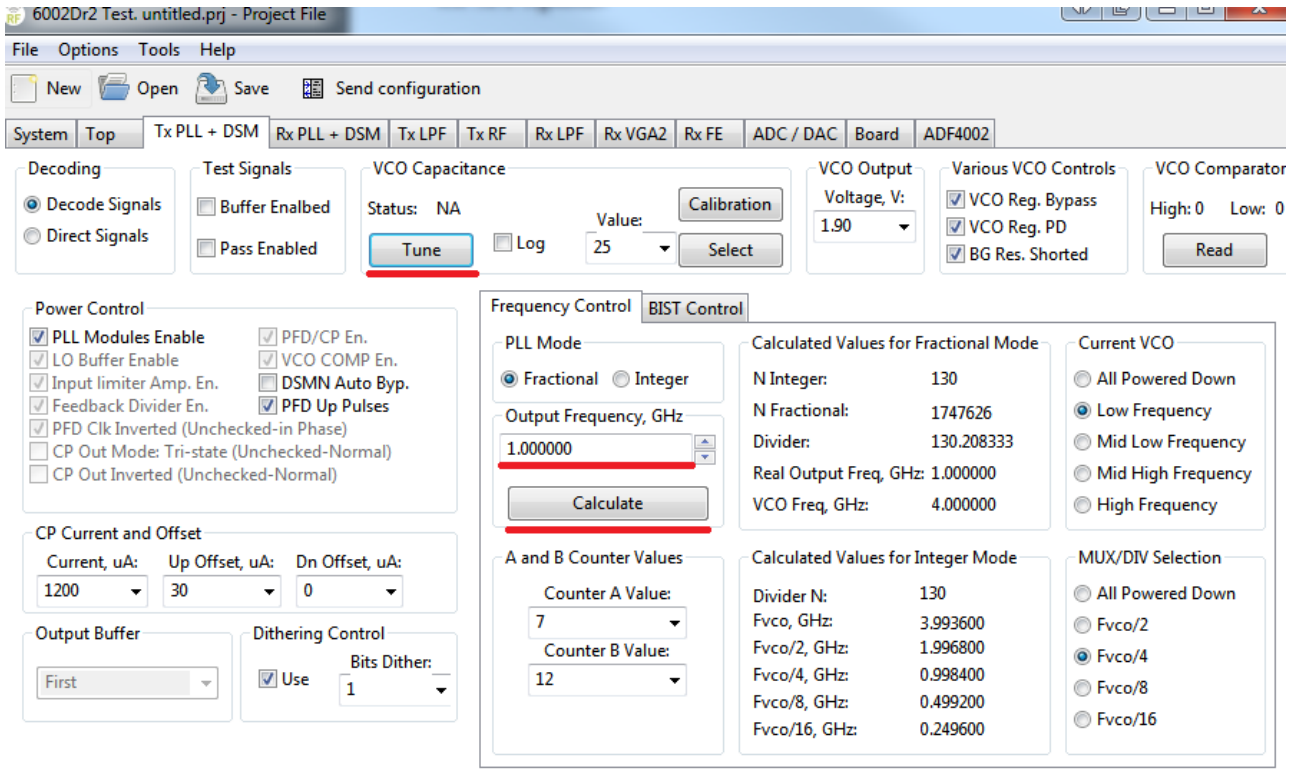


on downloaded to the hardware.

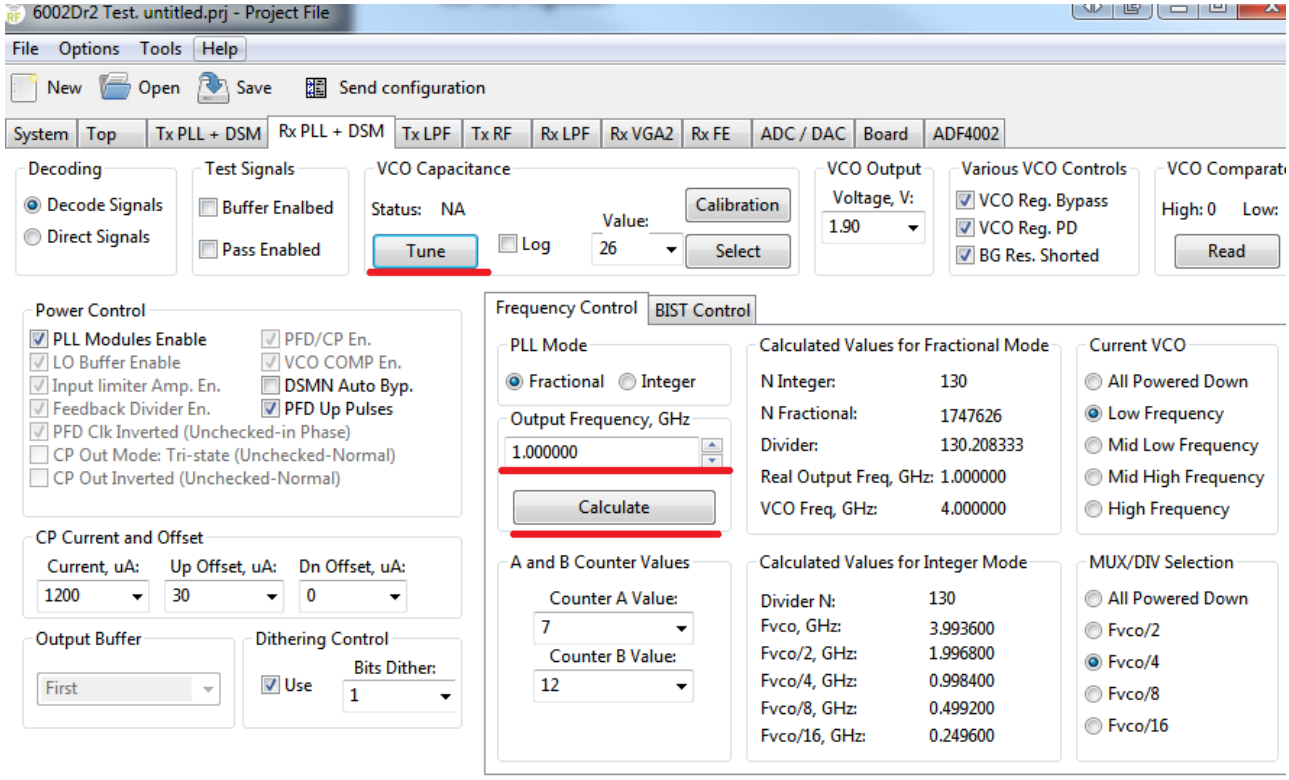
6. Then configure the RF board. Start from **Top** label.
   Top



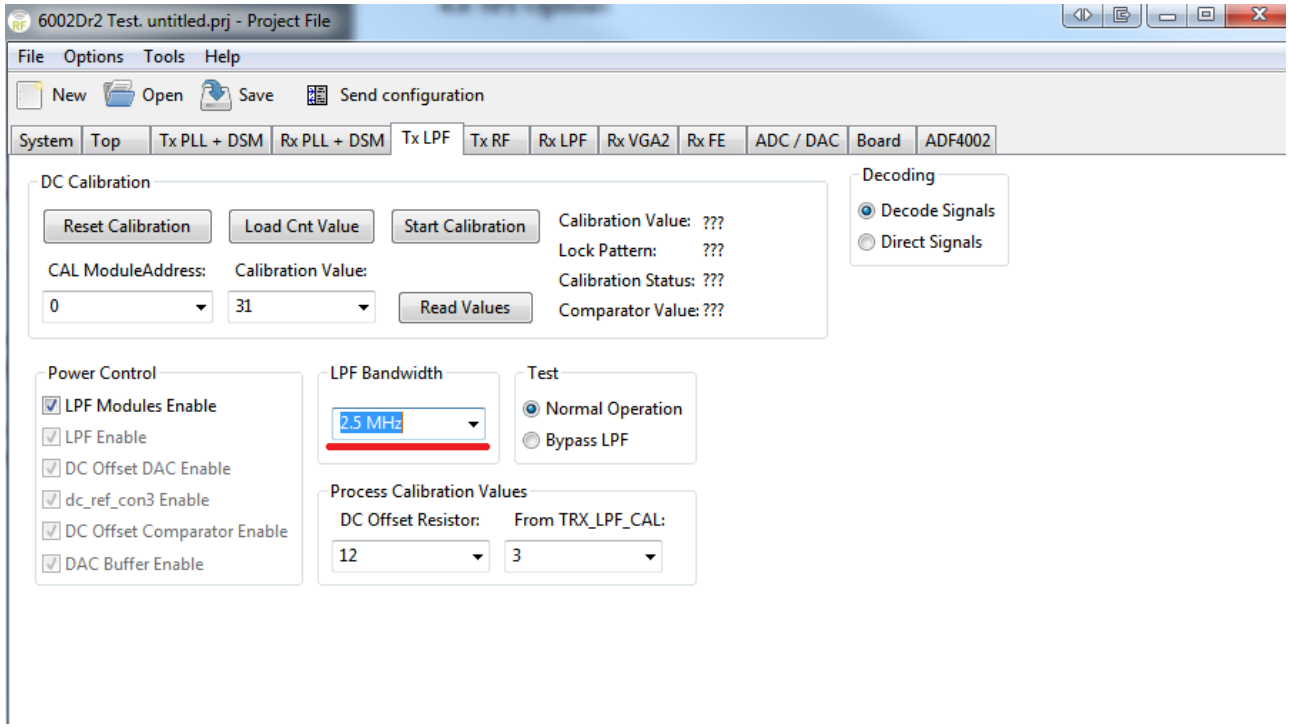
Tx PLL + DSM Firstly, change the output frequency to 1.0 GHz. Then click **Calculate**. Then click **Tune**.



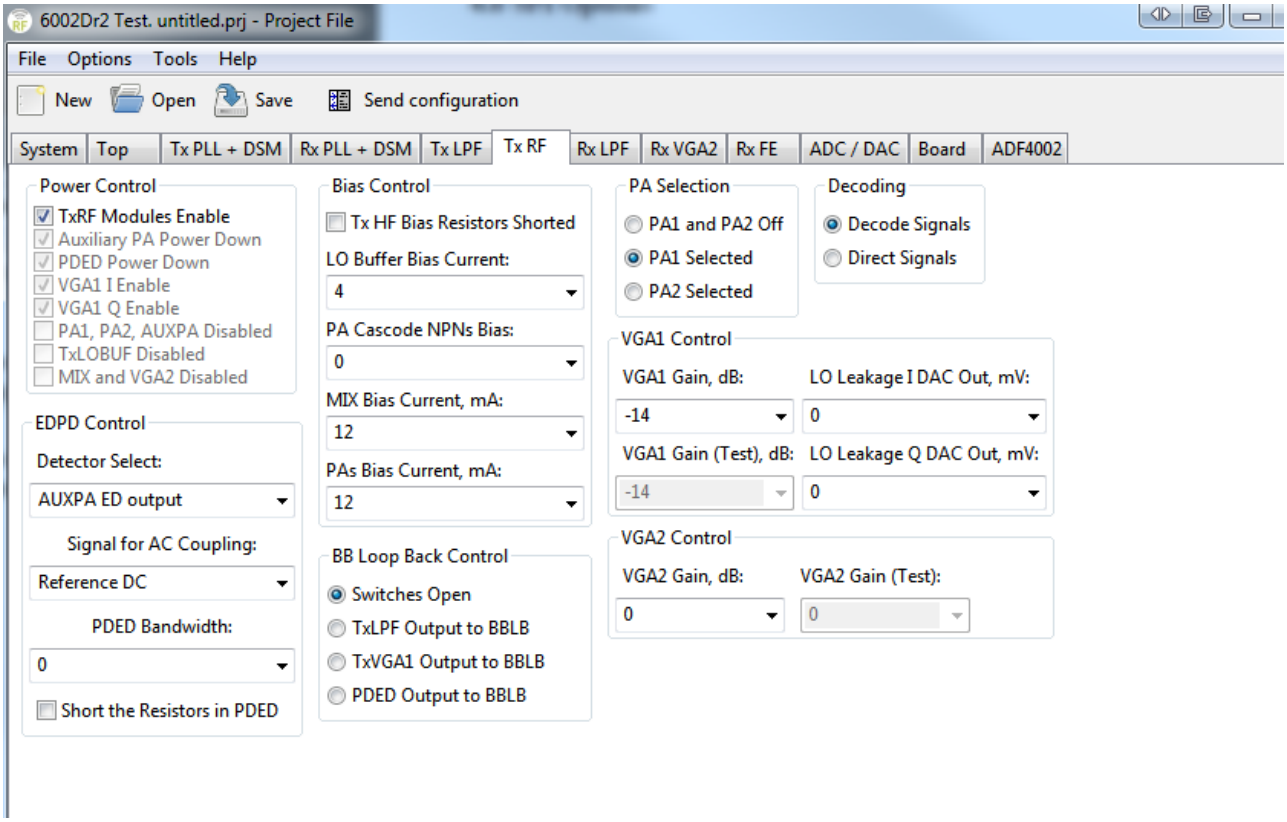
Rx PLL + DSM Firstly, change the output frequency to 1.0 GHz. Then click **Calculate**. Then click **Tune**.



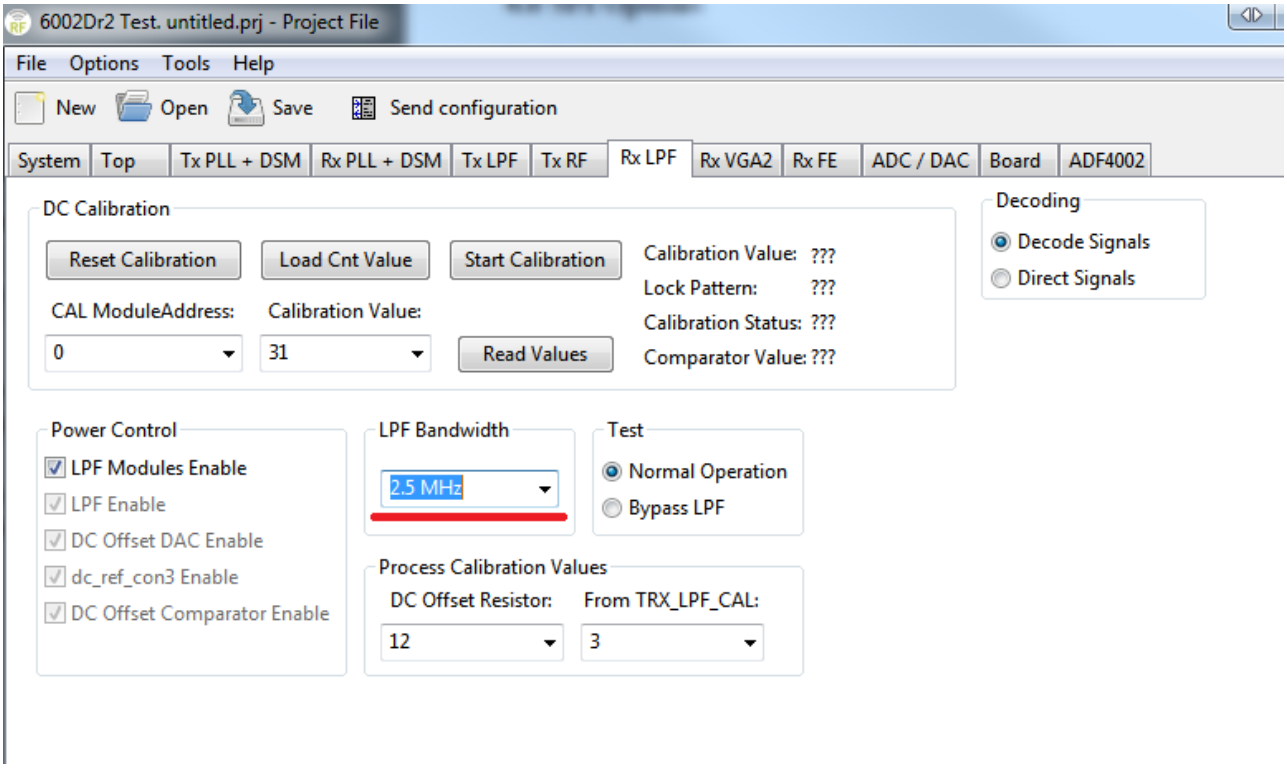
Tx LPF Change to 2.5MHz. (The bit speed is 1 MHz, and then the DPSK bandwidth is 2 MHz.)



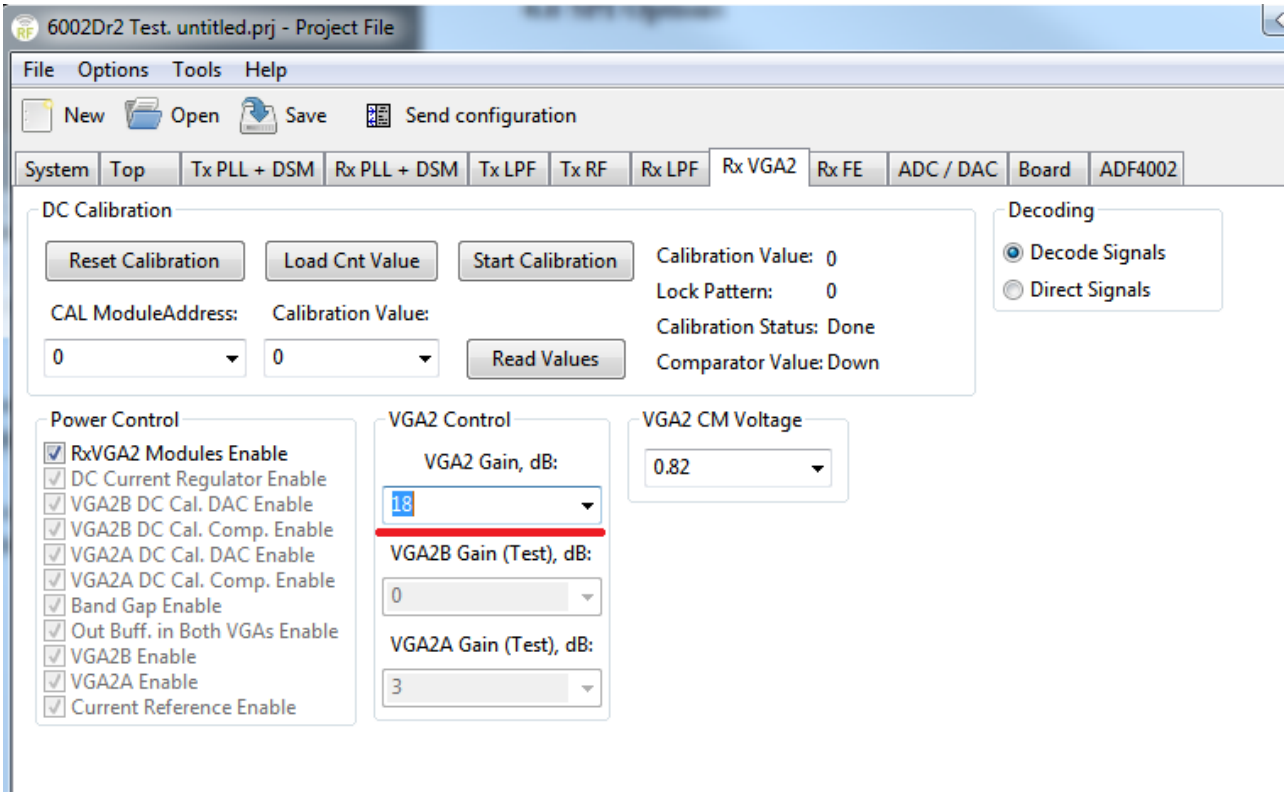
Tx RF Keep default.



Rx LPF Change to 2.5MHz. (The bit speed is 1 MHz, and then the DPSK bandwidth is 2 MHz.)



Rx VGA2 Change gain to 18 dB.



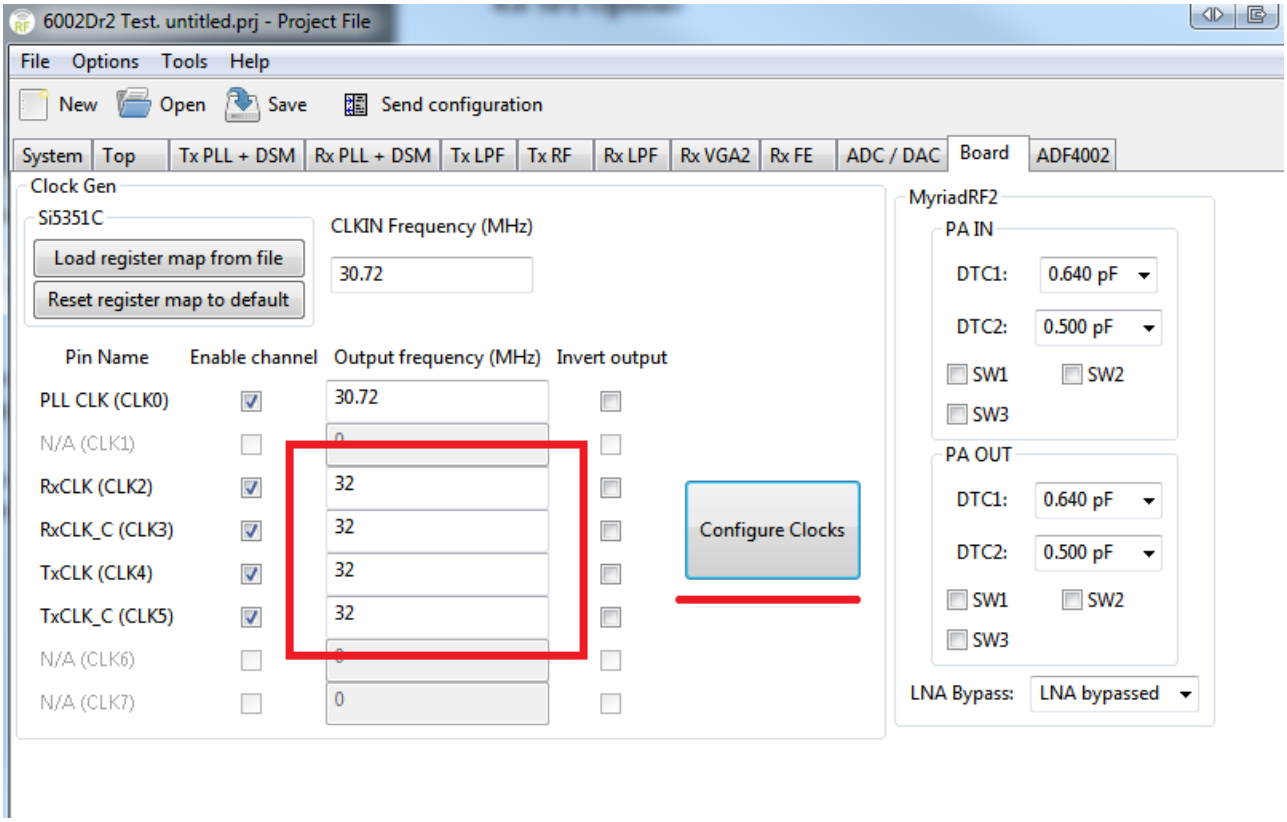
Rx FE Keep default.

ADC/DAC Keep default.

Board Configure the Rx and Tx clock to 32 MHz. One info bit will be represented by 16 sample points (ADC/DAC), and transimit one couple of Inphase/Quadra data need 2 clock period.

Remember to press *Configure Clocks*.

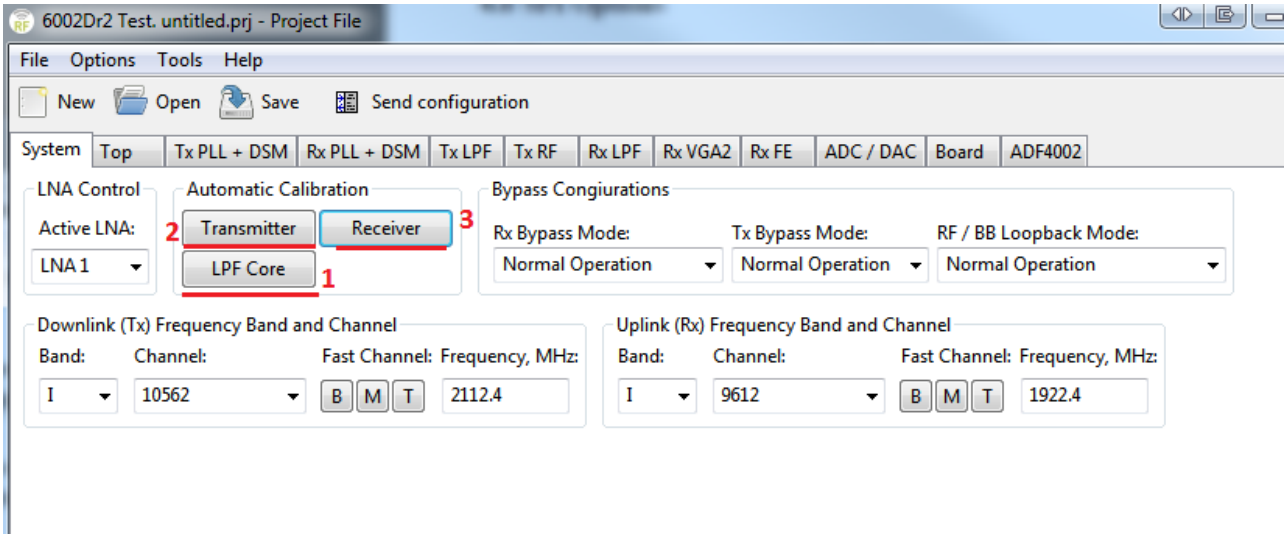
1MHz(Bit speed) X 16(Points) X 2(clock for one couple In/Qu data) = 32MHz



ADF4002 Keep default.

System Then go back to System Tab to do calibration. Press keys as following sequence: **LPF Core -> Transmitter -> Receiver**

After Calibration, remember press ***Send configuration*** to make sure all configuration will send to RF board.



NOTE: Please remember, after this first time configuration, ***do not close the software, nor disconnect the Zipper with computer***. The RF Board configuration part of Zipper Board will be powered by USB port from computer. Whenever you power on the FPGA board, i.e. power on RF board, the configuration part will configure the RF board **automaticly** and you do not need to configure the RF board again.

Now the RF board configuration finish. Then start hardware part.

IV.3. Hardware Part

In the section of **QUICK IMPLEMENTATION**, the steps of FPGA implementation will be introduced, except the procedure of generating Qsys file of this project.

Several files will be used in **QUICK IMPLEMENTATION** are located at **/RF-Board/RF SendRec via Loop/HWFile/QuickImplementation**.

> **recCtrl.vhd:** Used in Qsys. Receive the control signal from user program.
> **recCtrl_hw.tcl:** Tcl file for **recCtrl.vhd**.
> **rfDemo.vhd:** The top level file of this project.
> **RFReceive.vhd:** Used in Qsys. Receive the Rx data from RF board and save these data in FIFO.
> **RFReceive_hw.tcl:** Tcl file for **RFReceive.vhd**.
> **sinewave.vhd:** Generate sine and cosine wave for transmitting data.
> **waveCtrl.vhd:** Doing DPSK modulation. The first function is to do differential operation to information bits. The second function is to choose and send certain sine/neg-sine/cosine/neg-cosine data to RF board, according to the sending bits.
> **soc.qsys:** The qsys file of this project.
> **rfDemo_pin.tcl:** This is the pin assignment file for RF board, clock and LED.

In **/RF-Board/RF SendRec via Loop/HWFile/**, there are two other useful files:

> **rfDemo.sof:** Project's sof file.
> **fpga.rbf:** The rbf file converted from **rfDemo.sof** directly.

/RF-Board/RW HPS SDRAM by FPGA/HW is my project folder.

QUICK IMPLEMENTATION

1. Create a new project named **rfDemo**. Device choose **5CSXFC6D6F31C6**.
2. Copy all the files in **/RF-Board/RF SendRec via Loop/HWFile/QuickImplementation** to currect project folder. Open **Qsys** tool, then open the **soc.qsys**. Press **Gernerate HDL**, change the file to **VHDL**, and check the path. Then generate.
3. Add **soc/synthesis/soc.qip**, **sinewave.vhd**, **waveCtrl.vhd**, **rfDemo.vhd** to this project. Check: port defination of **soc.vhd** with the port defination of component **soc** in **rfDemo.vhd**, they should be same. After checking, **Start Analysis and Synthesis**.
4. After finishing the analysis and synthesis, pin assignment need to be done. Click the **Tcl Scripts** of **Tools** menu. In the opened window, choose **/Project/soc/synthesis/submodules/hps_sdram_p0_pin_assignments.tcl**, then click **Run**. Then choose **/Project/rfDemo_pin.tcl**, then click **Run**. Pin assignment finish.
5. Now **Start Compilation**. After finishing compilation, a sof file, **mem.sof**, will be generated. Location: **[Project Folder]/output_files/rfDemo.sof**. This file can be used to program FPGA.

IV.4. Software Part

/RF-Board/RF SendRec via Loop/SW is my software development folder.

/RF-Board/RW HPS SDRAM by FPGA/SWFile contains three ARM-Linux program, and one linux script. They can be used directly on ARM-Linux.

> **startram:** Set correct register value about fpga2sdram. **NOTICE:** input HEX number. Introduction to these registers: rst || cfg || default Recommandation settings: rst: 3fff, cfg: 0, default: 0.
> **Decode:** Read received wave data from HPS memory and demodulate, decode these data.
> **RFDemo:** This program will control Rx part to start receiving wave data from RF board and save these data to FIFO. Then this program will send commands to write part, write these receiving wave data to HPS memory, which can be used to demodulate and decode in **Demod**.
> **en_bridge.sh:** This bash script will enable all three FPGA-HPS bridges (HPS2FPGA, FPGA2HPS, LWHPS2FPGA). When you first boot up the board, the bridges are disabled by default. You must enable the bridge before sending commands to FPGA.
> **NOTE:** Open **en_bridge.sh** in Linux, maybe you will see some ^M. It is caused by different defination of **return** under DOS and Unix/Linux. **Solution**

IV.5. Project Demonstration

IV.5.1. Procedure and Structure

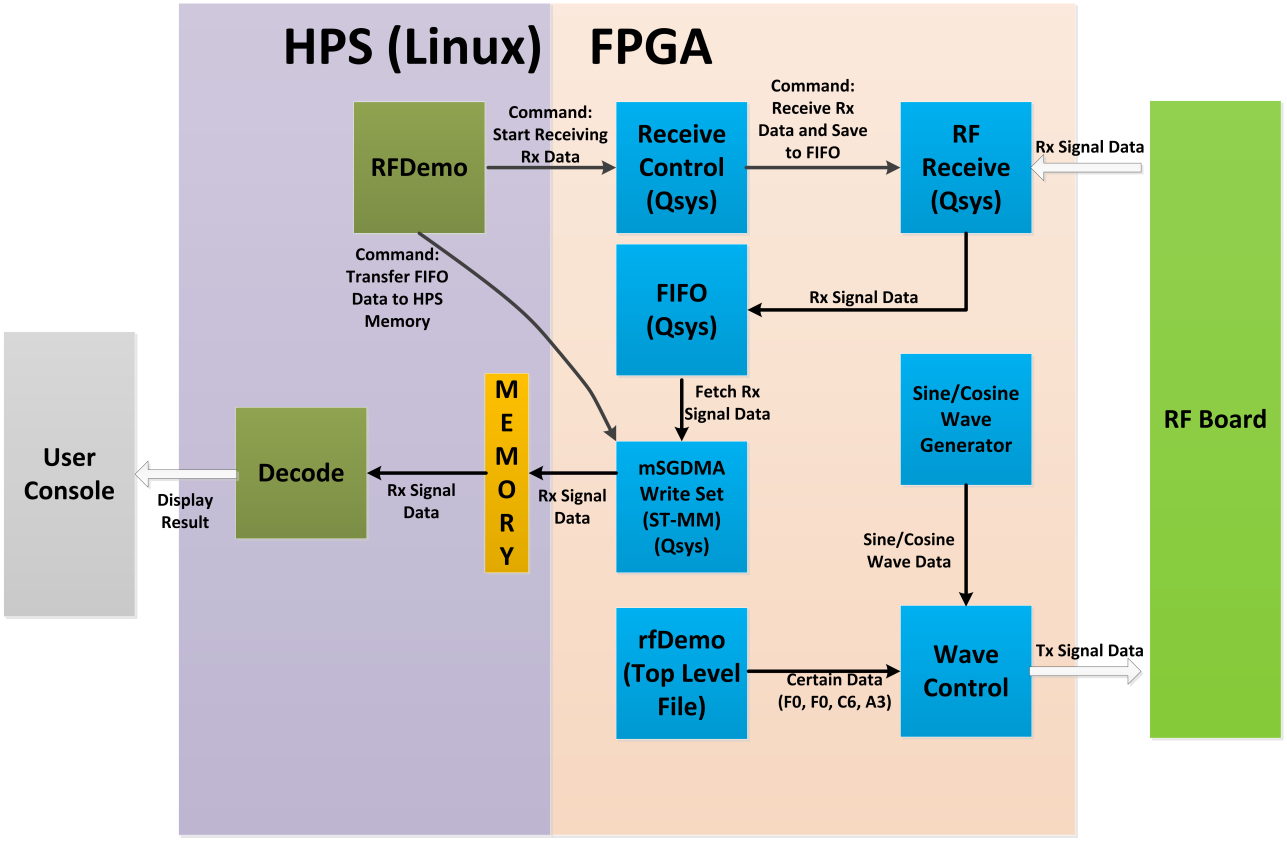
NOTE: If procedure failed, please read **IV.5.2. Possible Reasons If Procedure Failed**

Firstly, copy **startram, RFDemo, Decode** to ARM-Linux's folder: **/home/root**. Then Power on the board and program the FPGA by **rfDemo.sof** or **fpga.rbf**(U-Boot). Login Linux on board and excute following steps.

1. Enable bridge, run **en_bridge_sh**.
2. Run **startram** program to set SDRAM register. **[rst: 3fff, cfg: 0, default: 0]**.
3. Run **RFDemo** program, start receiving wave data from RF Board, and save 4096 points wave data (Inphase and Quadra each is 2048 points) to HPS memory.
4. Run **Decode** program, to demodulate and decode received wave data, get the transmitted info. If the decoded info bits match with original info bits, the console will display the information as following figure.

```
root@socfpga:~# ./Decode
Start reading memeory ... DONE
DQPSK Demodulate Start
Start Point: 5.
Start Point: 6.
Start Point: 7.
Start Point: 8.
Start Point: 9.
Start Point: 10.
Start Point: 11.
DQPSK Demodulate End
DQPSK Demodulating ... DONE
root@socfpga:~#
```

Diagram of Project Structure



Some Explaination

1. **About the output of *Decode* program:** Though decoding DPSK does not need to synchronization, it still has to choose a best start point of receiving data for successful decoding. In this project, one info bit will be represented by 16 samples. The **Decode** program will start demodulating and decoding at different point (start from the 1st point ~ start from 16th point) and figure out which point is the best starting point. In this case shown in figure, the best start point is 8th point.
2. **About the Project Structure:** You could notice that the Tx part has no relation to HPS or other part in FPGA. That because I do not have enough time to improve this project. The **rfDemo** module will always generate a 4 bytes (32 bits) data sequence (F0 F0 C6 A3 F0 F0 C6 A3 F0 …) to **Wave Control**, then **Wave Control** will differential encode these data and send correct wave signal data to Tx part of RF board. In program **Decode**, the data sequence

of (F0 F0 C6 A3 F0 F0 C6 A3 F0 …) will be detected. I think it is a good start point for future development.

IV.5.2. Possible Reasons If Procedure Failed

1. If the **Signal Tap** tool is used, there will be a possible problem: when you try to read or write the (LW)HPS2FPGA bridge, all system will crashed. **Solution:** You should change the setting of **Signal Tap**, try to find the setting which will not create this problem and will help you debug. **So remember always backup an executable version of project.**
2. If **Decode** program cannot show **DQPSK Demodulating … DONE**, you should try more times. If it is still cannot show success message after several time trying, this problem may caused by the configuration of RF Board. **Solution:** Adjust the RF board configuration in software: **Rx VGA2 -> VGA2 Gain**, **Tx RF -> VGA1 Gain**. Increase or decrease the gain of these two VGA, will change the amplitude of received signal. In most situation, **Decode** failed because of the wrong configuration of VGA Gain.