CryptoCore Manual

Thomas Pototschnig

2019-12-04

This manual is written for engineers and programmers who are involved in the development of products that use the CryptoCore.

Contents

Chapter 1

Hardware Overview

1.1 Overview

The CryptoCore is IOTA hardware designed for applications that need fast, dedicated proof of work and a secure memory.

The device consists of an IOTA CryptoCore FPGA (ICCFPGA) module and a development board that doubles as a Raspberry Pi HAT, making it perfect for stand-alone applications and/or quick prototyping.

When developing on the CryptoCore, you can use the development board without any of the security features to debug the firmware, upload new firmware to ROM, and add new bitstreams to RAM or QSPI flash.

When you've finished developing, you can take advantage of the following built-in security features:

Secure Element The secure element is a tiny chip that's efficient at cryptographic functions, and robust against physical attacks.

This chip can store up to 8 seeds, therefore all communication with it can be encrypted, using rotating keys to prevent replay attacks.

CPU Protection Embedded on the CPU is a memory protection unit, which restricts access to memory, depending on the privilige level in which the code is running.

Bitstream Encryption The bitstream can be encrypted to prevent manipulation or extraction of program code and to prevent unauthorized code from being flashed onto the ICCFPGA module.

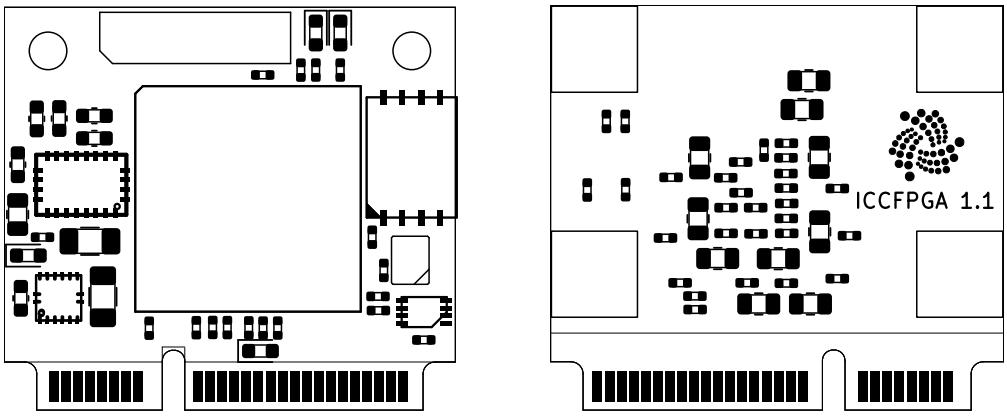
1.2 ICCFPGA Module

1.2.1 Overview

The ICCFPGA module is a generic FPGA that, when loaded with the default ICCFPGA bitstream, provides security measures and hardware acceleration for the algorithms used in IOTA.

This section describes the ICCFPGA module as though it has been flashed with the ICCFPGA bitstream. However, you could program and compile custom code for this module.

As well as these features, the ICCFPGA module has a low power consumption, offers digital inputs and outputs, and exposes peripherals such as SPI or I2C.



1.2.2 Technical Data

- XC7S50 Spartan 7 Series FPGA
- 16 MB Flash (used for configuration, and accessible from the soft CPU)
- ATECC608A secure element
- RISC-V compatible soft CPU with 128 kB ROM, 128 kB RAM, 4 kB Supervisor RAM, running at 100 MHz
- Hardware accelerators for proof of work (Curl-P81), Keccak384, Troika (all single cycle per hashing round), and ternary/binary conversions.
- JTAG for ICCFPGA
- JTAG for RISC-V (full debugging support including semihosting)
- 19 GPIOs (alternate functions: 2 SPI-Master, I2C, and UART)
- 16 spare GPIOs (not used by the ICCFPGA)
- FPGA control signals* on the card connector

*: Control signals: /INIT, /PROGRAM (from V1.2), CLK_HOLD (for stopping the internal phase-locked loop)

1.2.3 Pinout

The ICCFPGA module uses a mini-PCIe connector (and its pin numbering) with custom signals on the connector.

Module Pin-Number	Name	Function	Alt.Func	BGA-Pad
1	RV_TCK	RISC-V JTAG TCK		H2
3	RV_TDO	RISC-V JTAG TDO		J1
5	RV_TDI	RISC-V JTAG TDI		J2
8	IO0	GPIO0	MOSI0	L1
10	IO1	GPIO1	MISO0	L2
11	RV_RESET	RISC-V reset		M1
12	IO2	GPIO2	SCK0	P2
13	RV_TMS	RISC-V TMS		M2
14	IO3	GPIO3	SS0	M4
16	IO4	GPIO4	SS1	P3
17	IO7	GPIO7	UART TXD	N4
19	IO8	GPIO8	UART RXD	P4
20	FPGA_TCK	FPGA JTAG TCK		
22	FPGA_TMS	FPGA JTAG TMS		
23	FPGA_TDO	FPGA JTAG TDO		
25	FPGA_TDI	FPGA JTAG TDI		
28	IO9	GPIO9		P10
30	IO10	GPIO10		N10
31	IO11	GPIO11		M10
32	IO12	GPIO12		P11
33	IO13	GPIO13		N11
36	IO14	GPIO14		M11
38	IO5	GPIO5	SCL0	P12
42	IO6	GPIO6	SDA0	M12
44	IO15	GPIO15		P13
45	IO16	GPIO16		M13
46	IO17	GPIO17		N14
47	IO18	GPIO18		M14
49	CLK_HOLD	FPGA hold clock		L12
51	/INIT	FPGA init		
48	/PROGRAM*	FPGA program (>=V1.2)		
6, 7	NC	Not Connected		
2, 24, 29, 39, 41, 52	+3.3V			
4, 9, 15, 18, 21, 26, 27, 29, 34, 35, 37, 40, 43, 50	GND			

*: Pin 48 can be used for triggering the ICCFPGA reconfiguration. This pin is useful for rebooting the ICCFPGA module after updating the bitstream in the QSPI flash. Otherwise, a power-cycle is currently needed. This functionality also is available in software running on the RISC-V soft-cpu for triggering reboots after bitstream-updates.

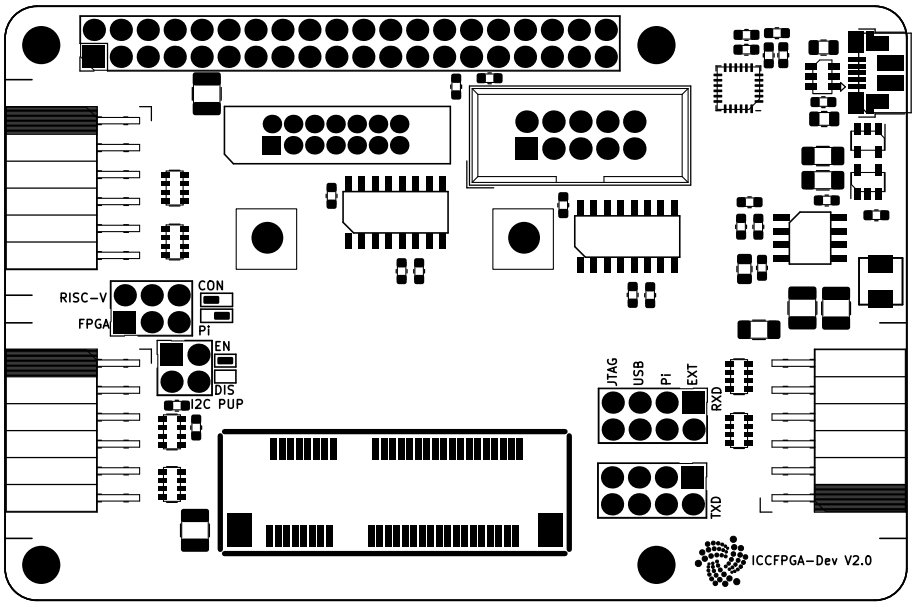
1.3 ICCFPGA Development Board

The development board includes a Pmod interface and also functions as a Rasperberry Pi HAT, making it easy to connect to a variety of peripheral devices.

1.3.1 Overview

You can use the development board as a HAT for Raspberry Pi 3 and Raspberry Pi 4 or you can run the board as a standalone device.

Both 5V inputs (USB and Raspberry Pi) are combined to a single 5V supply through two MOSFET-based "ideal diodes". This prevents shorting both power-inputs and avoids voltage drops and power loss on the diodes. Both power supplies can be applied without restrictions at the same time.



1.3.2 Technical Data

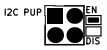
- Raspberry Pi 3/4 compatible connector
- USB with CP2103 USB-to-UART bridge
- Three Pmod-compatible pin sockets (one with SPI, and 4 GPIO pins, one with SPI, I2C, and 2 GPIO pins, one with UART, and 4 GPIO pins)
- JTAG via Raspberry GPIO or extra connectors (JTAG for FPGA and RISC-V seperate)
- UART via Raspberry GPIO, USB, RISC-V JTAG connector or pin-socket
- Power via Raspberry GPIO or USB

1.3.3 Jumpers

The development board includes four jumper blocks for configuring the circuit board.

JTAG select jumper (J9). Selects JTAG from Raspberry GPIO or J2/J6 pin-header

UART jumper (J7, J8). Selects TXD and RXD from RISC-V JTAG, USB, Raspberry Pi GPIO or pin-header J5*



I2C Pull-Up jumper (J10). Enables or disables 3.3k pullup resistor on I2C connected to J4.

*: TX is the driving output of the ICCFPGA module. Whereas all four RX jumpers can be set, only one TX jumper may be set to avoid many drivers using a single line.

1.3.4 Pinout

Pin	Name	Function	Alternate Function
1	IO3	GPIO3	SS0
2	IO0	GPIO0	MOSI0
3	IO1	GPIO1	MISO0
4	IO2	GPIO2	SCK0
7	IO9	GPIO9	
8	IO10	GPIO10	
9	IO11	GPIO11	
10	IO12	GPIO12	
6, 12	+3.3V		
5, 11	GND		

PMOD Connector J3

Pin	Name	Function	Alternate Function
1	IO4	GPIO4	SS1
2	IO0	GPIO0	MOSI0
3	IO1	GPIO1	MISO0
4	IO2	GPIO2	SCK0
7	IO13	GPIO13	
8	IO14	GPIO14	
9	IO5	GPIO5	SCL0
10	IO6	GPIO6	SDA0
6, 12	+3.3V		
5, 11	GND		

PMOD Connector J4

Pin	Name	Function	Alternate Function
2	TXD	UART TXD	*
3	RXD	UART RXD	*
7	IO15	GPIO15	
8	IO16	GPIO16	
9	IO17	GPIO17	
10	IO18	GPIO18	
1, 4	NC		
6, 12	+3.3V		
5, 11	GND		

Pmod Connector J5

*: TXD and RXD are connected to the ICCFPGA module only if jumper J7 and J8 are closed. TXD and RXD can be used as GPIO pins (TXD = IO8, RXD = IO7) only if UART is not used.

Chapter 2

Quickstart

2.1 Raspberry Pi Quickstart

This quickstart guide explains how to install the software needed to get the Raspberry Pi up and running with the Development HAT and ICCFPGA module.

1. Make sure the jumpers J9, J7 and J8 on the development board are set to ‘PI‘ according to 1.3.3.
2. Prepare an SD-card with Raspbian
3. Before booting, create the file ‘ssh‘ in the ‘/boot/‘ partition to enable ssh-access.
4. Boot Raspberry Pi and log in via SSH with username ‘pi‘ and password ‘raspberry‘
5. Change the default password

```
passwd
```

6. Open the ‘raspi-config‘ file and enable the serial interface (**Do not** enable shell access via serial).
   Reboot afterwards.
7. Install ‘git‘

```
sudo apt update && sudo apt install git
```

8. Clone the ‘iccfpga-utils‘ repository

```
cd ~
git clone --recursive https://gitlab.com/iccfpga-rv/iccfpga-utils
```

9. Enter the ‘iccfpga‘ directory and start the installer script

```
cd iccfpga-utils
./install_raspberry.sh
```

   The script will clone, compile and install all submodules

10. The Raspberry Pi will be rebooted in the previous step. After reboot install the core-file permanently or temporarily.

```
cd iccfpga-utils

# downloads the core- and firmware-files from the latest
# commit in the repository
./download_bin.sh

sudo ./flash_core.sh    # non-volatile programming to QSPI-flash
--- OR ---
sudo ./upload_core.sh   # volatile uploading to the FPGA
```

    The flashing is a bit special, because it needs a power-cycle to load the core-file from QSPI flash. In the second variant, the core gets startet instantly after the upload.

11. Start serial terminal

```
cd ~/iccfpga-utils
./start_serial.sh
```

12. Copy & Paste following into the serial terminal:

```
{"command":"version"}
```

The response should look like:

```
1 {"version":"0.07rv","command":"version","duration":0,"code":200}
```

There are three LEDs on the FPGA module. The first is a power-LED which always is lit when the board is powered. The second LED (between FPGA and mini PCIe connector) indicates if a core file was successfully loaded into the FPGA. The third LED at the top of the module is a user-LED which is enabled in software by the firmware running on the soft-cpu in the FPGA (indicating that the firmware was started successfully).

Chapter 3

Development Tools

Chapter 4

Cookbook

4.1 Cookbook for Raspberry Pi

There is a collection of bash scripts in the 'raspberry_scripts'-directory of the 'raspberry-utils'-repository that make recurring tasks easieer. They use programs that were installed during the installation from the quick-start guide.

Following a short overview. The sections after will explain, how to use them.

download_bin.sh : Downloads the latest core- and firmware files from the repository. The core-file in the repository always contains the newest firmware for the RISC-V cpu. The files will be downloaded to a sub-directory 'bin/'.

upload_core.sh : Uploads the core (volatile) to the FPGA.

flash_core.sh : Flashs the core-file into the QSPI flash (non-volatile).

flash_erase.sh : Erases the QSPI flash (completely!)

start_xvc_server.sh : Starts the Virtual Cable Server. The VCS is a software replacement for a real USB-Xilinx JTAG programmer. Vivado's hw_server can connect to the Virtual Cable Server and use it as JTAG interface to the FPGA. There are no restrictions, that means, everything a USB JTAG programmer supports also is supported using the Virtual Cable Server. This also includes security relevant settings like bitstream-encryption, eFUSE registers ,...

upload_riscv.sh : Uploads a RISC-V firmware to the soft-cpu. The core-file already comes with the latest firmware but it could be necessary to upload another firmware to the RISC-V.

start_debugger.sh : Starts the OpenOCD debugger for the RISC-V soft-cpu (the FPGA has to be configured beforehand). The guide in **??** explains, how to setup a development environment for the firmware and how to use the debugger from within Eclipse.

start_serial.sh : Starts the 'picocom' serial terminal. If jumpered correctly on the development board **??**, the serial terminal can be used to transmit and receive data to and from the FPGA-module through the '/dev/ttyS0' serial device*.
*: '/dev/ttyS0' has to be enabled via 'raspi-config'

4.1.1 Uploading the CryptoCore program

Prerequisites: Raspberry Pi quickstart 2.1, Jumper J9 (FPGA) on "PI" 1.3.3

The CryptoCore program is included in XSVF files, which contain 'recordings' of JTAG communication.
These files were generated in Vivado and can be replayed on the Raspberry Pi.

There are two XSVF files in the repository.

'iccfpga-utils/raspberry_scripts/bin/iccfpga_spi_flash.xsv' : XSVF for flashing the Core into the QSPI-flash. The flash gets automatically erased before flashing.

'iccfpga-utils/raspberry_scripts/bin/iccfpga_fpga.xsv' : XSVF for uploading the Core to the FPGA. This only is temporary until the next power-cycle.

Both XSVF files are automatically generated by the build script described in 4.2.1

For convenience there are two scripts, one for flashing the core:

```
cd ~/iccfpga-utils/raspberry
sudo ./flash_core.sh
```

the other for uploading the core to RAM:

```
cd ~/iccfpga-utils/raspberry
sudo ./upload_core.sh
```

The difference is that the second script only temporarily - until the next power-cycle - uploads the core to the FPGA without touching the QSPI flash.

4.1.2 Starting the Virtual Cable Server

Prerequisites: Raspberry Pi quickstart 2.1, Jumper J9 (FPGA) on "PI" 1.3.3

The Virtual Cable Server (VCS) is an alternative to using a USB Xilinx JTAG adapter to flash bitstreams onto the CryptoCore, using a Raspberry Pi 3/4.

```
cd ~/iccfpga-utils/raspberry
sudo ./start_xvc_server.sh
```

Output:

```
Virtual Cable Server startet

To connect vivado to the server use <some-ip>:2542
```

Now you can connect Vivado with the VCS as desribed in 4.2.3

4.1.3 Starting the OpenOCD debugger for RISC-V

Prerequisites: Raspberry Pi quickstart 2.1, Jumper J9 (FPGA) on "PI" 1.3.3

The OpenOCD debugger can be used for remote debugging. A local instance is started and an Eclipse installation on a PC can connect to the debugger for debugging the firmware.

```
cd ~/iccfpga-utils/raspberry
sudo ./start_debugger.sh
```

4.1.4 Uploading the Firmware

Prerequisites: Raspberry Pi quickstart 2.1, Jumper J9 (FPGA) on "PI" 1.3.3

Although the XSVF files already include the firmware, it is possible to temporarily upload custom firmware to the CryptoCore.

```
cd ~/iccfpga-utils/raspberry
sudo ./upload_riscv.sh
```

The firmware is started automatically.

4.1.5 Opening a Serial Terminal

Prerequisites: Raspberry Pi quickstart 2.1, Jumper J7 & J8 on "PI" 1.3.3

To test the RS232 API, you need to open a serial interface on the Raspberry Pi, using certain parameter.

To open a serial terminal with these parameters, do the following:

```
1 cd ~/iccfpga-utils/raspberry
2 ./start_serial.sh
```

This command runs a script, which opens a piocom serial terminal. You can close the terminal by pressing **CTRL+A** + **CTRL+Q**.

4.2 Cookbook for PC

4.2.1 Synthesizing the Bitstream with Embedded Firmware and Key Memory

Prerequisites: ICCFPGA-Repository ??, RISC-V Toolchain ??, Vivado ??

There is a script in the 'iccfpga-core'-Repository which conveniently does the following:

- Recompile the RISC-V firmware
- Resynthesize the Core with embedded firmware and API- and AES-keys
- Generate XSVF-files for FPGA and QSPI-flash

This is the most non-interactive way of completly rebuilding the core file without even having to open Vivado.

```
# export path to the installation directory of vivado
export VIVADOPATH=<vivado-bin-directory>

# export path to the risc-v toolchain
export PATH=$PATH:/opt/rv32im/bin


# start the build process
cd iccfpga/iccfpga-core/utils
./generate_all.sh
```

The script will try to compile the RISC-V firmware and completly resynthesize the Core with the firmware embedded.
It also generates XSVF-files which can be replayed on a Raspberry Pi (described in 4.1.1).

Note: In debugging mode, the firmware can be uploaded via RISC-V OpenOCD debugger. In production mode (locked JTAG) this is the only way to update the firmware.

4.2.2 Changing the API or AES key

Prerequisites: ICCFPGA-Repository ??, Python

The API and AES keys are, by default, in a file called 'data.coe' in the 'iccfpga/iccfpga-core/secure-data/' directory.
The file is generated by a python script that reads 'keydata.txt' from the same directory.
It can be executed with:

```
cd iccfpga/iccfpga-core/secure-data
python ./generate_coe.py
```

After regenerating the 'data.coe' file, the XSVF file has to be completely resynthesized by running the script described in 4.2.1.

4.2.3 Connecting Vivado to a Virtual Cable Server, using a Raspberry Pi

Prerequisites: Starting a Virtual Cable Server 4.1.2, Vivado ??, Jumper J9 (FPGA) on "PI"

TODO: some pictures

4.2.4 Connecting Vivado to a Xilink USB JTAG Adapter

Prerequisites: Clone the ICCFPGA-Repository ??, Install Vivado ??, Jumper J9 (FPGA) on "CON" 1.3.3

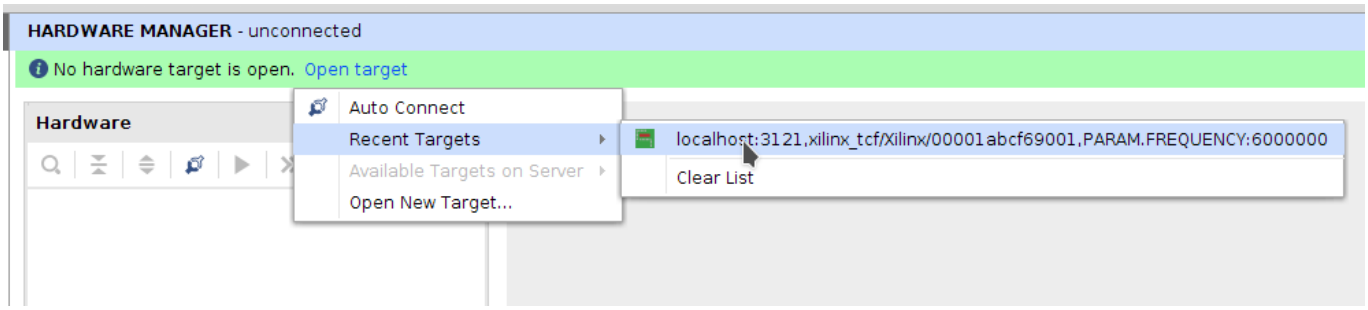
This guide explains how to flash bitstreams to QSPI flash, using a Xilink USB JTAG adapteR.

1. Start the Vivado hardware server

Listing 4.1: Starting hw_server

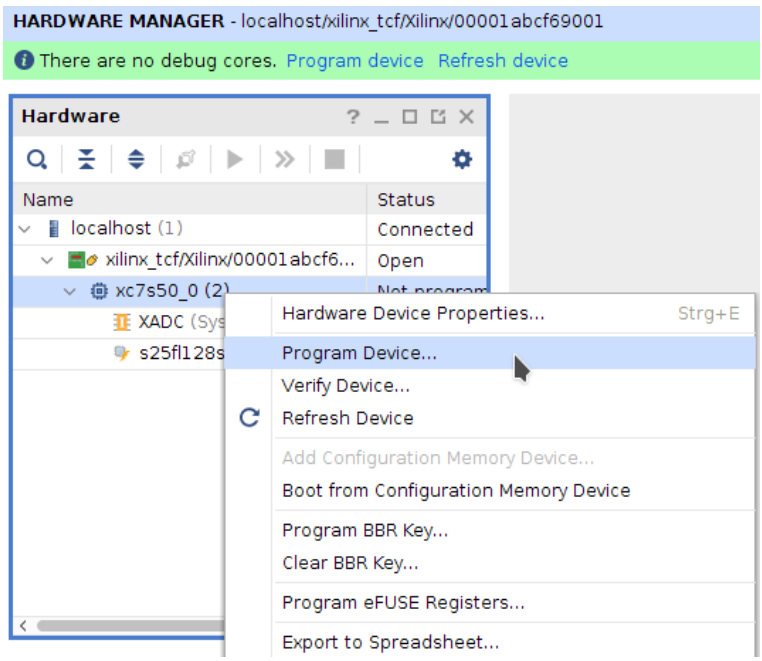
```
pc:~/xilinx/Vivado/2018.2/bin$ sudo ./hw_server

****** Xilinx hw_server v2018.2
  **** Build date : Jun 14 2018-20:18:37
    ** Copyright 1986-2018 Xilinx, Inc. All Rights Reserved.

INFO: hw_server application started
INFO: Use Ctrl-C to exit hw_server application

INFO: To connect to this hw_server instance use url: TCP:192.168.0.100:3121
```

2. Connect the hardware server to the ICCFPGA module by using the hardware manager

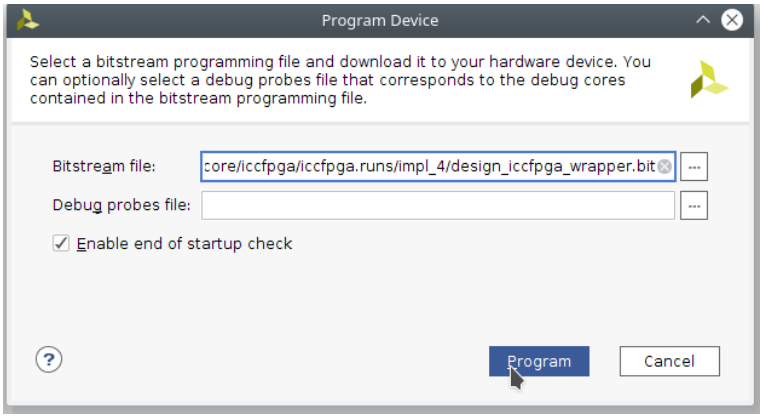


4.3 Uploading a Bitstream to the FPGA

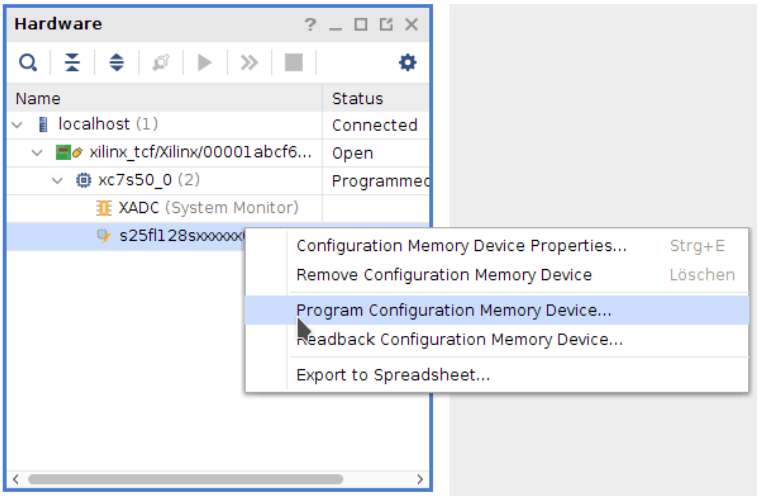
Prerequisites: Connect Vivado 4.2.3 or 4.2.4



3. Click **Program Device...** and select your bitstream file



Note: This only loads the bitstream onto the ICCFPGA but it is not stored permanently in the QSPI flash. If the bitstream has to be stored permanently on the module, the QSPI flash has to be added (it is already shown in the dialogue before) and has to be programmed.



Note: This flashing only works if the flag 'CFG_AES_ONLY' is not enabled in 3! The reason is how the QSPI flash is connected to the ICCFPGA. The indirect programming mode is used in which the FPGA is configured with a bitstream that gives Vivado access to the QSPI flash. Unfortunately there is no encrypted version of this bitstream. If the flag is enabled, the QSPI flash only can be written through the RISC-V soft CPU, which has access to the flash.

4.4 Flashing Bitstreams to QSPI Flash

Prerequisites: Connected Vivdo 4.2.3 or 4.2.4

TODO some pictures

4.5 Debugging the RISC-V firmware

This guide explains how to debug the RISC-V firmware on the ICCFPGA module.
You can use one of the following options to debug the firmware:

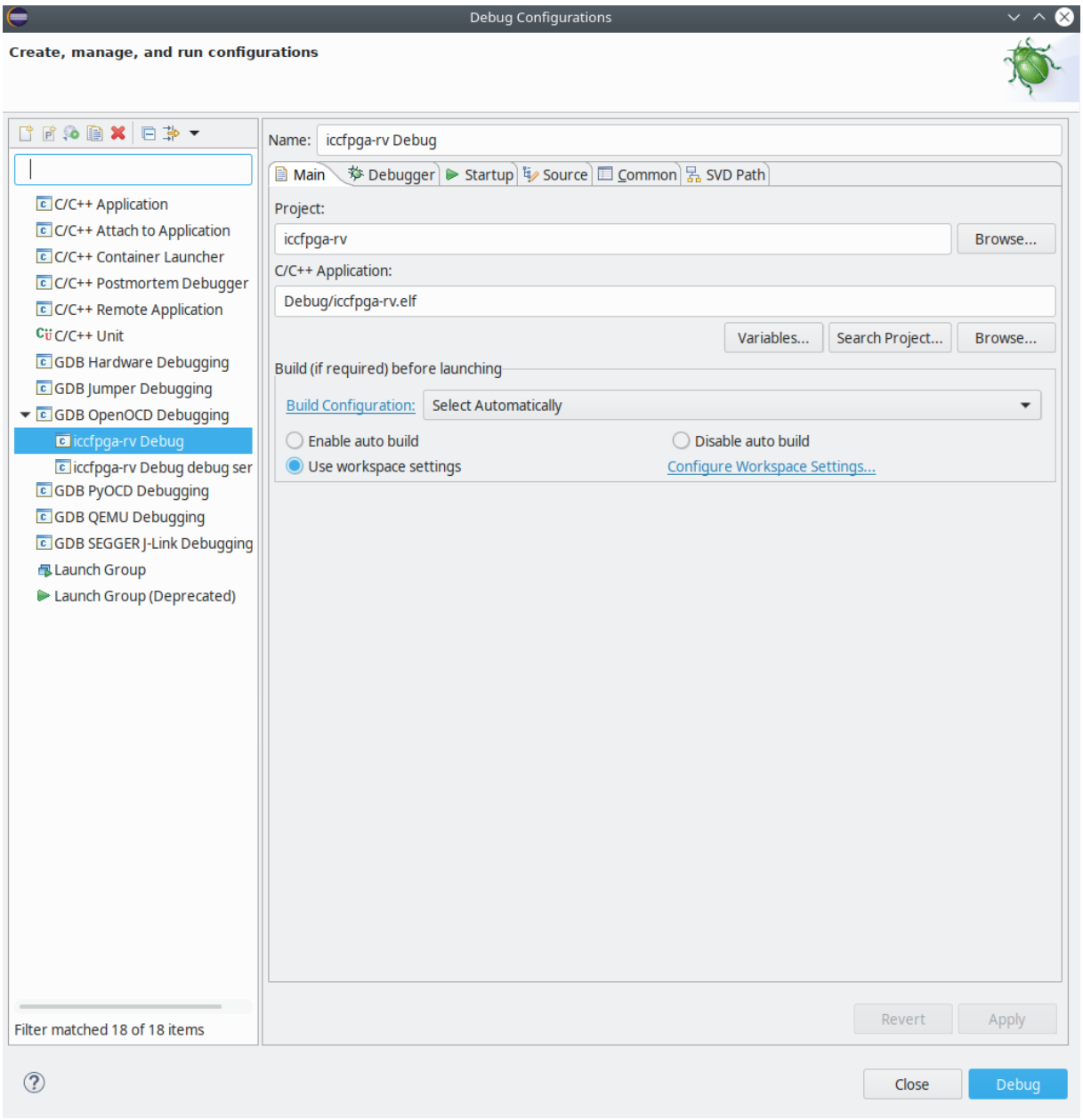
- Use a USB JTAG
- Use a Raspberry Pi as a debugging server

For both options, you need to install Eclipse for RISC-V, which is a popular development environment that we use in this manual to debug the RISC-V firmware.
You can download Eclipse for free here: `https://github.com/gnu-mcu-eclipse/org.eclipse.epp.packages/releases/`

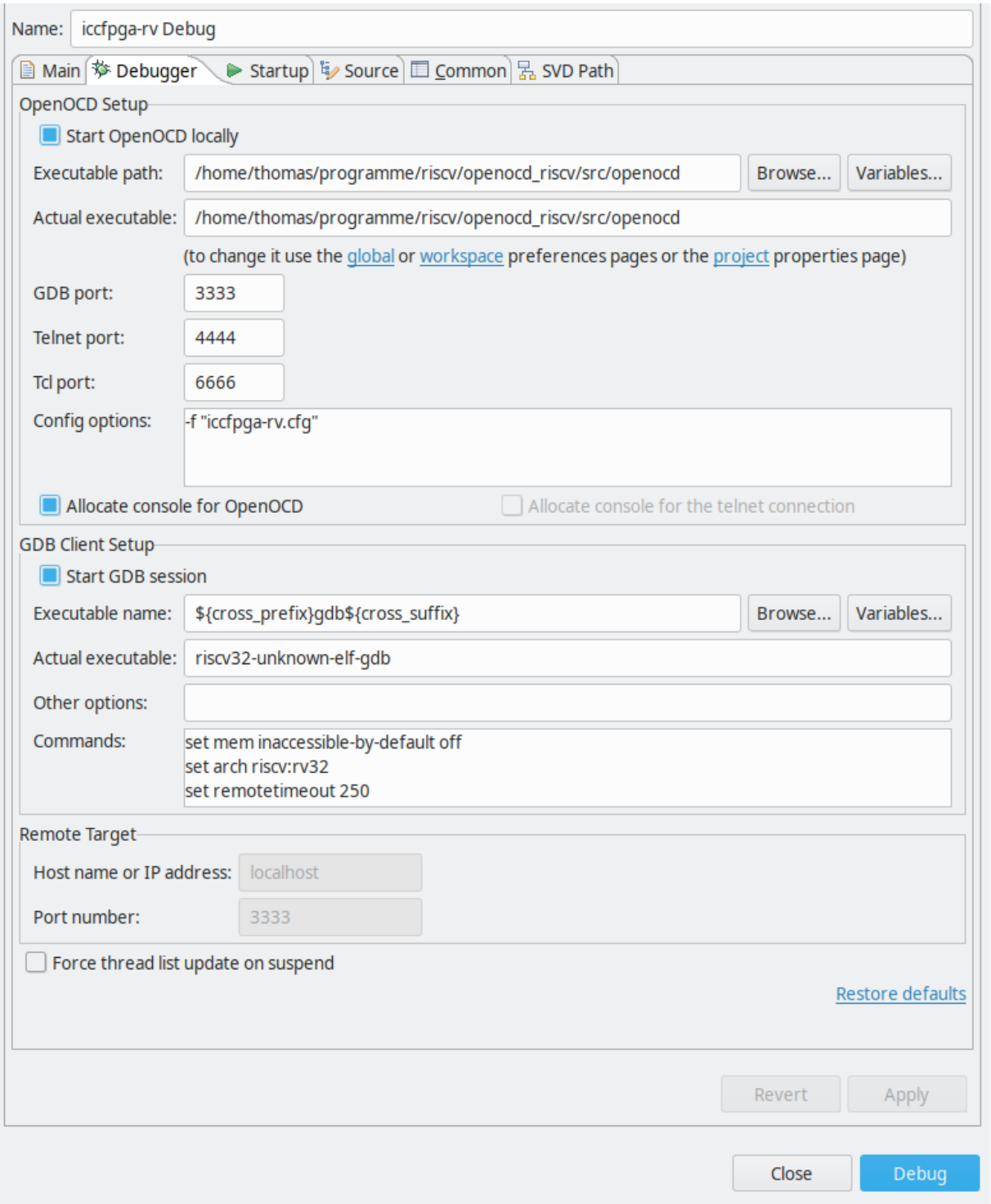
4.5.1 Debugging the RISC-V firmware with a USB JTAG

You can use any compatible variants of USB JTAG that use the FT2232 chipset. **Note:** Variants that have +5V on pin 1 and +3.3V on pin 3 cannot be used on version 1.0 of the development board version 1.0.
Here is an example of a USB JTAG with a compatible pinout:

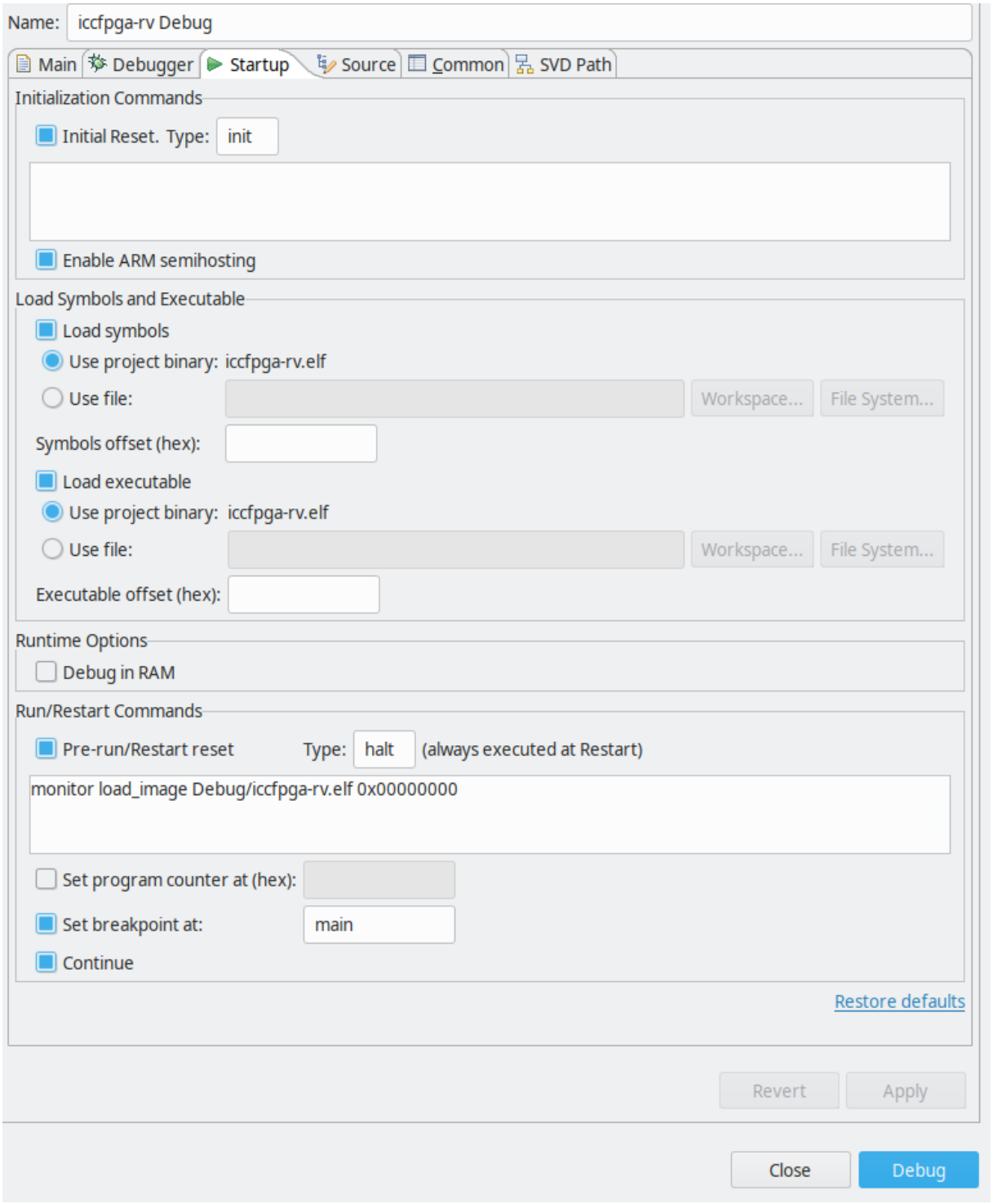
1. Set jumper J9 to "CON" for "RISC-V" to enable debugging. In this mode, Eclipse will start the OpenOCD server.
2. Create an OpenOCD debugging profile such as the following:



3. In the **Debugger** tab, complete the fields and set the OpenOCD executable path to the location where you installed OpenOCD



4. Complete the “Startup” Page



5. Click **Apply** > **Debug** to start debugging

4.5.2 Debugging the RISC-V Firmware with a Raspberry Pi

The RISC-V 'VexRiscv' uses a non-standard debugging interface that needs a particular version of OpenOCD.
To extend this version of OpenOCD with configuration files for the development board, we forked the VexRiscv repository.

1. Compile our forked version of OpenOCD for RISC-V

Listing 4.2: Compiling OpenOCD

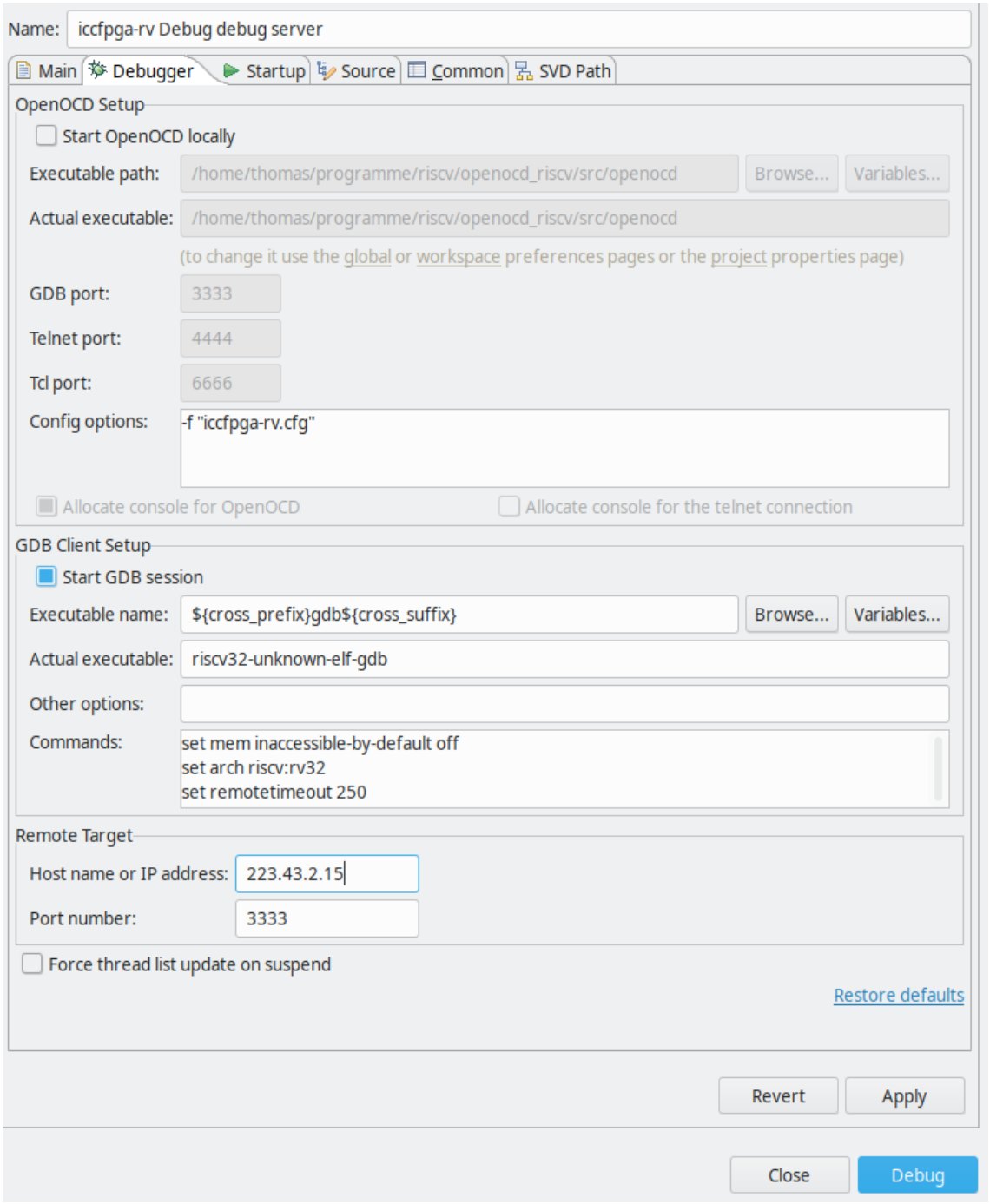
```
sudo apt-get install libtool automake libusb-1.0.0-dev texinfo libusb-dev \
    libyaml-dev pkg-config
git clone https://github.com/shufps/openocd_riscv
cd openocd_riscv
./bootstrap
./configure --enable-bcm2835gpio --enable-ftdi --enable-dummy --enable-sysfsgpio
make
sudo make install
```

2. On the development board, set jumper J9 to "PI" for "RISC-V"
3. Start an OpenOCD server on the Raspberry Pi. **Note:** If you are using Raspberry Pi 3, please use 'interface/raspberry3-native-iccfpga-vexrisc.cfg'.

Listing 4.3: Starting OpenOCD on Raspberry Pi

```
sudo openocd -c "bindto 0.0.0.0" \
    -f interface/raspberry4-native-iccfpga-vexriscv.cfg \
    -f target/iccfpga-vexriscv.cfg
```

4. Connect to the OpenOCD server from Eclipse. **Note:** Make sure the firewall does not block port 3333, and enter the IP address of your Raspberry Pi in the Remote-Target section.



5. Click **Apply** > **Debug** to start debugging

4.5.3 Uploading RISC-V Firmware to the ICCFPGA module, using OpenOCD on a Raspberry Pi

To upload RISC-V firmware to the ICCFPGA module without using Eclipse, you can use the OpenOCD server on the Raspberry Pi.

1. Upload the code through OpenOCD. **Note:** If you're using a Raspberry Pi 3, replace 'interface/raspberry4-native-iccfpga-vexriscv.cfg' with 'interface/raspberry3-native-iccfpga-vexriscv.cfg'.

Listing 4.4: Upload Code to RISC-V with OpenOCD

```
1 sudo openocd -f interface/raspberry4-native-iccfpga-vexriscv.cfg \
2     -f target/iccfpga-vexriscv.cfg -c "init" -c "reset halt" \
```

```
    -c "load_image iccfpga-rv.elf 0x00000000" -c "reset run" \
    -c "resume" -c "exit"
```

2. Take the ICCFPGA module out of reset state

Listing 4.5: Unreset RISC-V

```
#!/bin/bash
# export jtag_rv_reset, switch pin to output and deassert reset

cd /sys/class/gpio
echo 19 > export
echo "out" > gpio19/direction
echo "1" > gpio19/value
```

4.6 Securing the FPGA

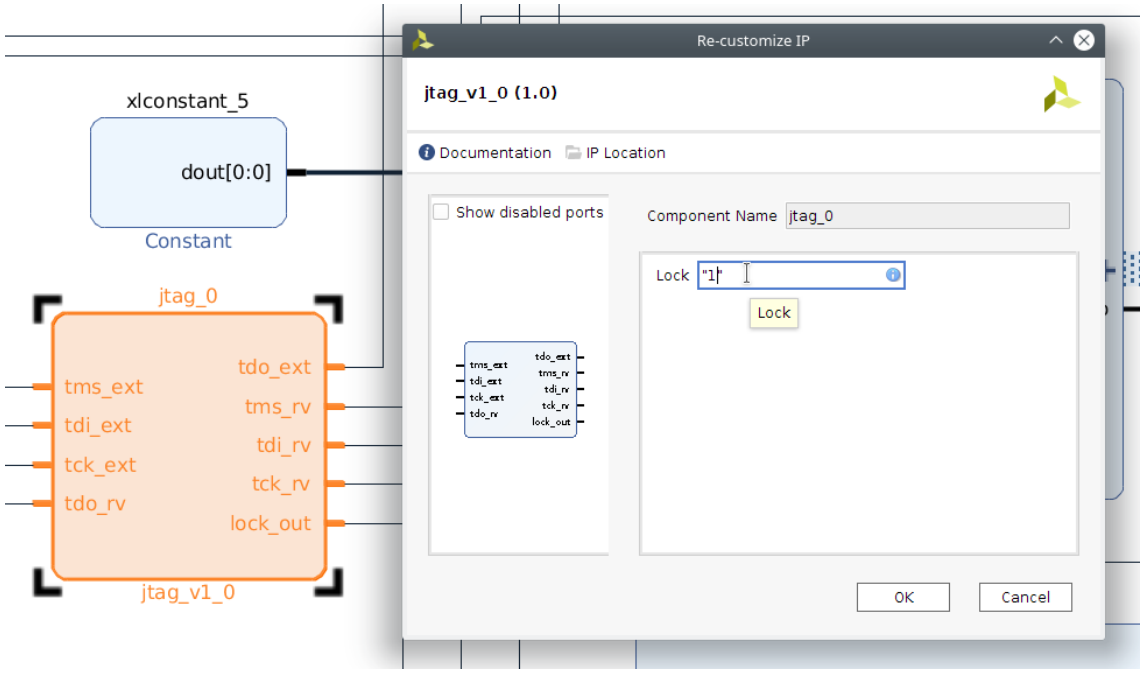
For debugging purposes, you can upload firmware to the ROM in the RISC-V soft CPU through the RISC-V JTAG. For production applications, this JTAG interface is insecure because it allows others to access the firmware. Therefore, when you finish developing your application, you can complete the following steps to secure the system.

1. Lock the JTAG
2. Change the API key
3. Change the secret key
4. Enable bitstream encryption

4.6.1 Locking the JTAG

Locking the JTAG stops others from accessing the ROM in your RISC-V soft CPU.

1. In the Vivado Block Design, set the variable 'Lock' of the component 'jtag_0' to '1'



2. Synthesize the system to remove the JTAG interface from the hardware.

Firmware can no longer can be uploaded through the RISC-V JTAG. Instead, it must be embedded in the bitstream.
Tip: The Lock flag can be read by the CPU by reading GPIO pins 31.

4.6.2 Changing the API Key

The API key is used to authenticate calls to API commands that read the seed from the secure element. Because UART communication is not encrypted, the API key ensures that data hasn't been modified on the way to the ICCFPGA module. Currently, the API key is stored in the 'iccfpga-rv/secure/secure.cpp' file:

Listing 4.6: API Key

```
const uint8_t g_api_key[48] = {
    0xb2, 0x33, 0x12, 0x56, 0x41, 0xf0, 0xcc, 0x60, 0x9c, 0x6f, 0x36, 0x30, 0xe7,
    0xf3, 0xb2, 0xfd, 0xb7, 0x2b, 0xbd, 0x2e, 0x94, 0x40, 0xa5, 0xb0, 0x0a, 0xb5,
    0x83, 0x65, 0x5b, 0x01, 0xb1, 0x43, 0xdf, 0x31, 0x3e, 0x9a, 0xa0, 0x90, 0x72,
    0x02, 0xdf, 0x5f, 0x16, 0x40, 0x8e, 0x64, 0xf4, 0xd4
};
```

If you change this key, you must recompile the RISC-V firmware.

4.6.3 Changing the Secret Key

The secret key is used to encrypt and decrypt data sent between the RISC-V firmware and the secure element. Currently, the secret key is stored in the ‘iccfpga-rv/secure/secure.cpp‘ file:

Listing 4.7: AES Key

```
// this has to be changed by the user to ensure security
const uint8_t g_slot4_key[32] = {
    0x37, 0x80, 0xe6, 0x3d, 0x49, 0x68, 0xad, 0xe5, 0xd8, 0x22, 0xc0, 0x13, 0xfc,
    0xc3, 0x23, 0x84, 0x5d, 0x1b, 0x56, 0x9f, 0xe7, 0x05, 0xb6, 0x00, 0x06, 0xfe,
    0xec, 0x14, 0x5a, 0x0d, 0xb1, 0xe3
};
```

If you change the secret key, you must use the API to initialize the secure element to set the new secret key. See 4.9.9.

4.6.4 Enabling Bitstream Encryption

To stop others from being able to read your bitstream, you can encrypt it before uploading or flashing it to the ICCFPGA module.

1. Clone the 'iccfpga' repository. The key for bitstream encryption is in the 'iccfpga-core/iccfpga/iccfpga.xdc' file.

Listing 4.8: Cloning the Repositories

```
git clone --recursive https://gitlab.com/iccfpga-rv/iccfpga
cd iccfpga/iccfpga-eclipse/submodule/ArduinoJson
git checkout 5.x
```

2. In the constraints file, comment out either the 'BRAM' or 'eFUSE' line, depending on how you want to store the AES keys (BRAM is volatile memory, and eFUSE is non-volatile memory). The AES key is a 64 digit hexadecimal string.

Listing 4.9: Constraints File

```
#encryption settings
set_property BITSTREAM.ENCRYPTION.ENCRYPT YES [current_design]
set_property BITSTREAM.ENCRYPTION.ENCRYPTKEYSELECT BBRAM [current_design]
#set_property BITSTREAM.ENCRYPTION.ENCRYPTKEYSELECT eFUSE [current_design]
set_property BITSTREAM.ENCRYPTION.KEY0 256'h7821461...2382B4D [current_design]
```

3. In Vivado, generate the encrypted bitstream

If encryption is enabled in the constraints file, you should now have an NKY file

Listing 4.10: NKY Example File

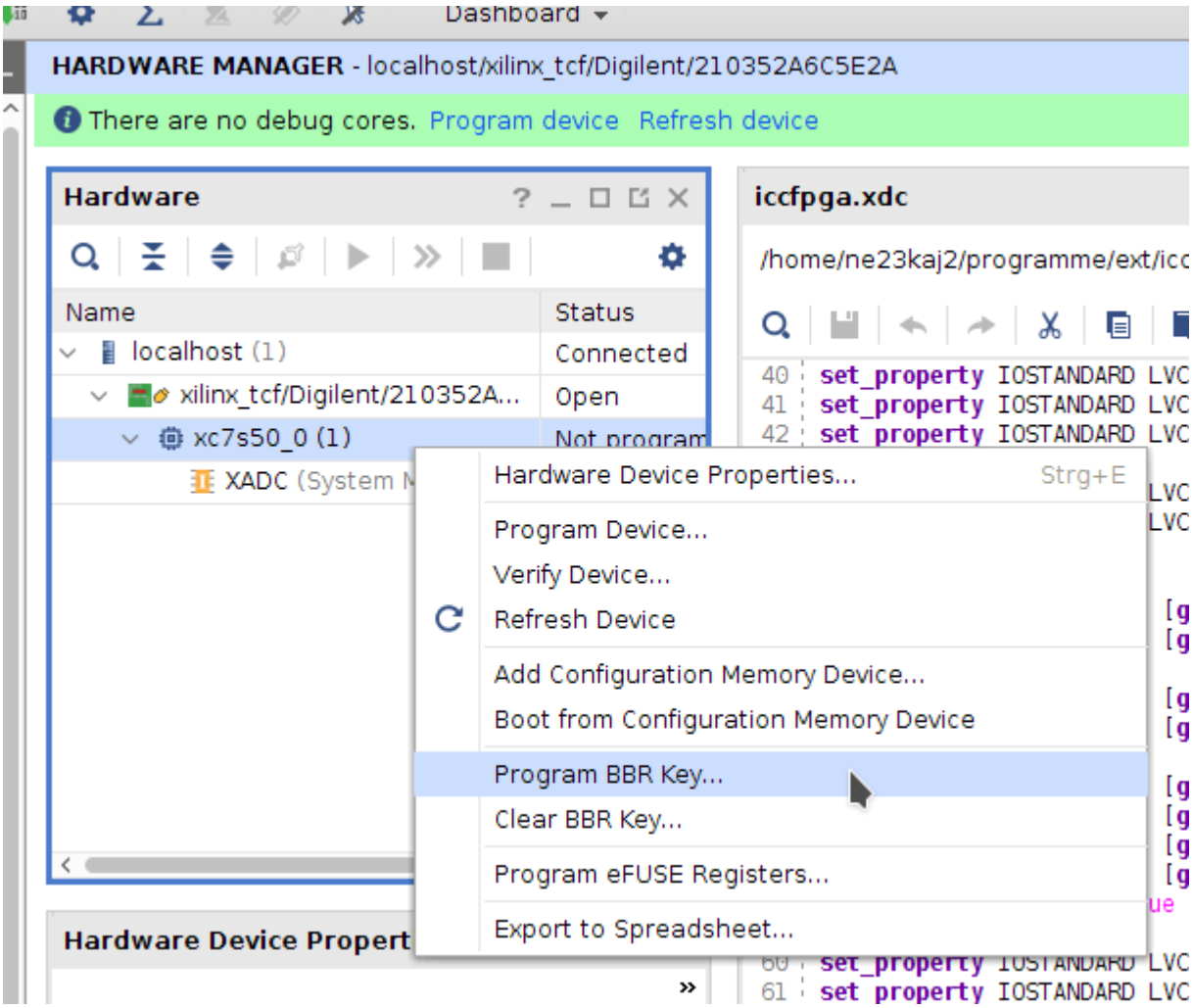
```
iccfpga/iccfpga.runs/impl_1$ cat design_iccfpga_wrapper.nky
Device xc7s50;
Key 0 78214125442A472D4B614E645267556B58703273357638792F423F4528482B4D;
Key StartCBC 8b4bd4762870cf71061cce9ef1401f07;
Key HMAC 25fd439344cfccf18562bc9f67301b1559186382f82af6a6d207db8aa5ba4b62;
```

Now, you can program your encryption key into memory, using one of the following options:

- Volatile BBRAM memory (useful for testing encryption)
- Non-volatile eFUSE memory (useful for production environments)

Programming the FPGA keys to BBRAM (volatile)

1. In Vivado, click **Program BBR Key**

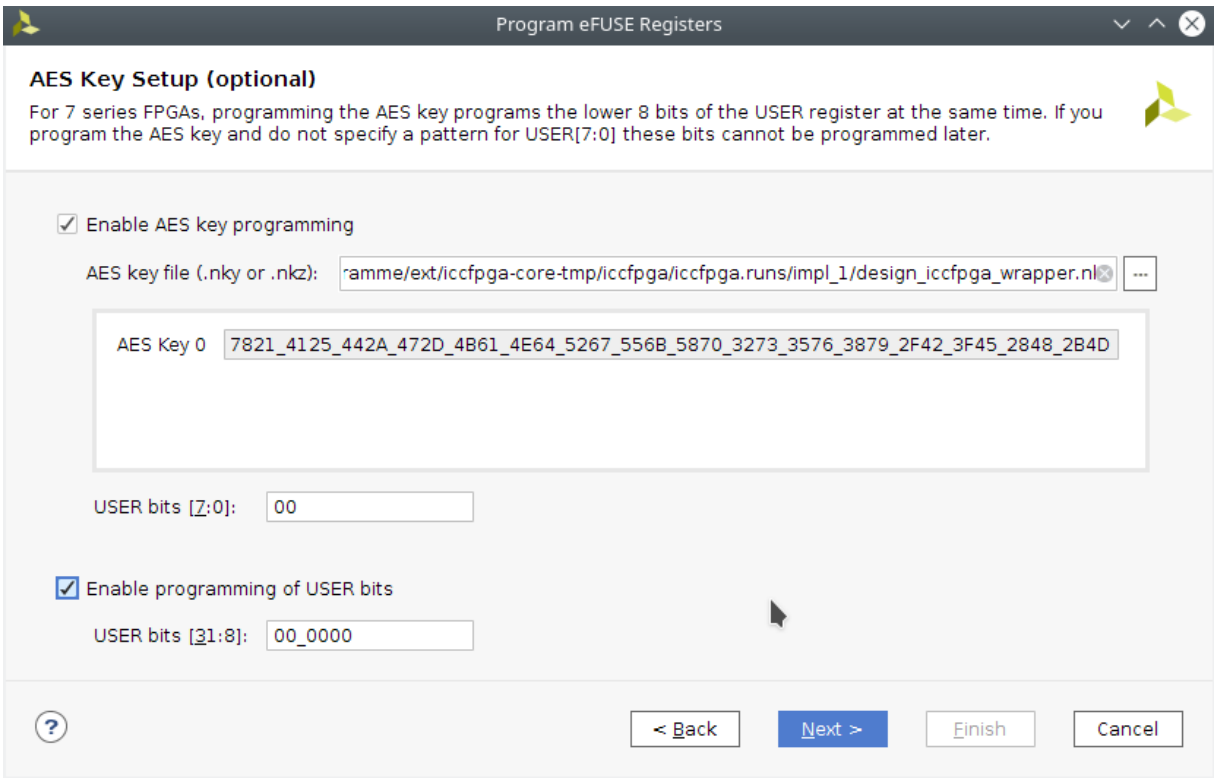


After programming the BBRAM file, you can upload the encrypted bitstream to the ICCFPGA.
Please note that the BBRAM is volatile and there is no backup battery on the ICCFPGA module.
To store the encrypted bitstream in the flash memory on the module, the key has to be programmed to eFUSE.

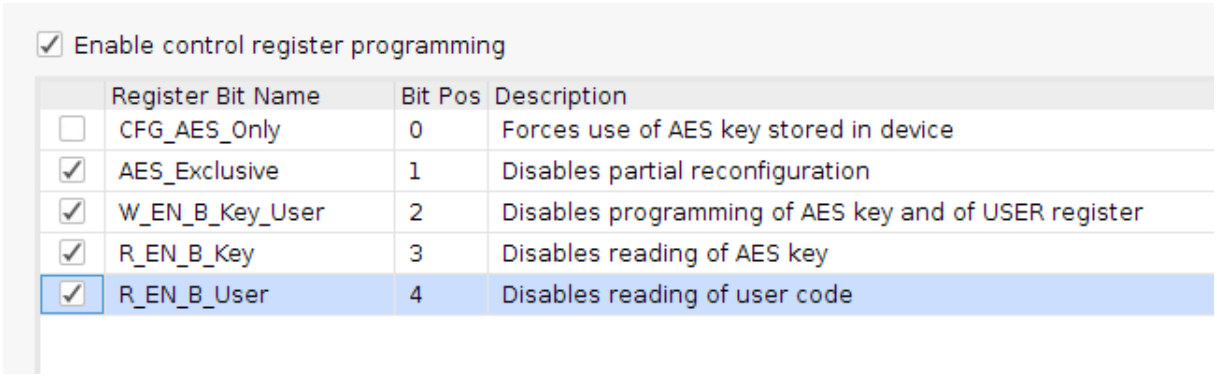
Programming the FPGA keys to eFUSE (permanent)

Note: This option is permanent and if you lose the key, you won't be able to recover the ICCFPGA.

1. In Vivado, click **Program eFUSE registers**



2. Select the NKY file and click **Next**
3. Complete the fields, using the following recommended values. **IMPORTANT:** Don't select the first option unless you know what you are doing. After setting CFG_AES_ONLY, it will no longer be possible to program the QSPI flash memory via the Vivado hardware manager. This is a known problem without solution: `https://www.xilinx.com/support/answers/44116.html`.



Now, only the RISC-V firmware has access to the QSPI flash memory and is able to write updated bitstreams to the flash.

4.7 Compiling the RISC-V Firmware

4.8 Embedding RISC-V firmware in the bitstream

TODO

Note: ROM can be embedded (in the fast way) only if encryption is disabled. If bitstream encryption is enabled, the ROM code has to be synthesized together with the FPGA system. Encrypting bitstreams after replacing the ROM is not possible (perhaps it is possible but currently unknown how it works - probably via some TCL-magic in Vivado).

4.9 API

The CryptoCore exposes an RS232 API, which you can call over a UART connection. For example, you could send API calls through a device that's connected to the CryptoCore over USB.

4.9.1 setFlags

Sets the given flags.

The following flags are available:

- keepSeedInRAM: Reads the seed from the secure element and caches it in RAM. (Caching the seed saves 1-2 seconds for each request of the seed.)
- debugRS232Output: Enables debugging output on the SWD line or RS232.

Before you can set the keepSeedInRAM flag, you must initalize the secure element by calling the 'initSecureElement' command.

Parameters

Parameter	Type	Description
flags	object	The name/value pairs of the flags that you want to set

Example Request

```
{
    "command":"setFlags",
    "flags":{
        "keepSeedInRAM":true
    }
}
```

Example Responses

```
{
    "code": 200,
    "command":"setFlags",
    "duration": 250
}
```

```
{
    "code": 400,
    "command":"setFlags",
    "error": "Error message"
}
```

4.9.2 testHardwareAcceleration

Tests the hardware acceleration on the ICCFPGA module.

The following functions are tested and compared with the results of unaccelerated functions:

- Type conversions (bytes/trytes and bytes/trits)
- Hashing (proof of work, Curl, Keccak384, and Troika)

Example Request

```
{
    "command":"testHardwareAcceleration"
}
```

Example Responses

```
{
    "code": 200,
    "command":"testHardwareAcceleration"
```

```
    "bytesToTritsSingle":"pass",
    "tritsToBytesSingle":"pass",
    "pow":"pass",
    "keccak384":"pass",
    "bytesToTritsRandomRepeated":"pass",
    "tritsToBytesRandomRepeated":"pass",
    "trytesToBigintRandomRepeated":"pass",
    "bigintToTrytesRandomRepeated":"pass",
    "troika":"pass",
    "curl":"pass",
    "duration":1867
}
```

```
{
    "code": 400,
    "command":"testHardwareAcceleration"
    "error": "Error message"
}
```

4.9.3 generateRandomSeed

Generates a random seed and stores it in one of eight available memory addresses in the secure element.

Before you can call this command, you must initalize the secure element by calling the 'initSecureElement' command.

Parameters

Parameter	Type	Description
key	integer	An integer between 0 and 7, which specifies the memory address in which to save the seed

Example Request

```
{
    "command":"generateRandomSeed",
    "key": 0
}
```

Example Responses

```
{
    "code": 200,
    "command":"generateRandomSeed",
    "duration":1800
}
```

```
{
    "code": 400,
    "command":"generateRandomSeed",
    "error": "Error message"
}
```

4.9.4 generateAddress

Generates addresses for the seed in the secure element's given key.

Before you can call this command, you must initalize the secure element by calling the 'initSecureElement' command.

Parameters

Parameter	Type	Description
key	integer	The memory address of the seed from which you want to derive the address
firstIndex	integer	The address index from which to start generating addresses
number	integer	The number of addresses to generate, starting from the first index
security	integer	The security level of the address that you want to generate

Example Request

```
{
    "command":"generateAddress",
    "key": 0,
    "firstIndex": 0,
    "number": 10,
    "security": 2
}
```

Example Responses

```
{
    "code": 200,
    "command":"generateAddress",
    "trytes": ["....","....",...],
    "duration": 1800
}
```

```
{
    "code": 400,
    "command":"generateAddress",
    "error": "Error message"
}
```

4.9.5 attachToTangle

Chains the transactions into a bundle, using the trunkTransaction and branchTransaction parameters, and does proof of work on all of them, using the given minimum weight magnitude.

This command can do proof of work for a bundle that contains up to eight transactions. If you want to do proof of work for larger bundles in a single command, you can add the branch transaction hash, trunk transaction hash, and timestamp to the transaction trytes yourself before passing them to the 'doPow' command.

Parameters

Parameter	Type	Description
trunkTransaction	string	Trunk transaction hash to use to attach the bundle to the Tangle
branchTransaction	string	Branch transaction hash to use to attach the bundle to the Tangle
minWeightMagnitude	integer	The minimum weight magnitude to use during proof of work
timestamp	integer	A Unix epoch timestamp to add to the transaction's 'timestamp' fields
trytes	array of strings	Transaction trytes of up to eight transactions in a bundle

Example Request

```
{
    "command":"attachToTangle",
    "trunkTransaction": "...",
    "branchTransaction": "...",
    "minWeightMagnitude": 14,
    "timestamp": 1552571227826,
    "trytes": ["....", "....", ...]
}
```

Example Responses

```
{
    "code": 200,
    "command":"attachToTangle",
    "trytes": ["....","....",...],
    "duration": 1800
}
```

```
{
    "code": 400,
    "command":"attachToTangle",
    "error": "Error message"
}
```

4.9.6 doPow

Does proof of work on an array of transaction trytes.

This command can do proof of work for up to 10 transactions at once.

Parameters

Parameter	Type	Description
minWeightMagnitude	integer	The minimum weight magnitude to use during proof of work
trytes	array of strings	Transaction trytes of the transactions

Example Request

```
{
    "command":"doPow",
    "minWeightMagnitude": 14,
    "trytes": ["....", "....", ...]
}
```

Example Responses

```
{
    "code": 200,
    "command":"doPow",
    "trytes": ["....","....",...],
    "duration": 1800
}
```

```
{
    "code": 400,
    "command":"doPow",
    "error": "Error message"
}
```

4.9.7 signTransaction

Signs a single input transaction, using the seed in the secure element's given key.

Before you can call this command, you need to do the following calculation and add the result to the 'auth' parameter: keccak384(key+addressIndex+bundleHash+apiKey)

Before you can call this command, you must initalize the secure element by calling the 'initSecureElement' command.

Parameters

Parameter	Type	Description
key	string	The memory address of the seed that owns the address
addressIndex	string	The index of the input transaction's address
bundleHash	integer	The bundle hash in the transaction's 'bundle' field
securityLevel	integer	The security level of the input transaction's address
auth	string	The Keccak384 hash of the key, addressIndex, bundleHash, and the API key

Example Request

```
{
    "command":"signTransaction",
    "key": 0,
    "addressIndex": 0,
    "bundleHash": "...",
    "securityLevel": 2,
    "auth": "..."
}
```

Example Responses

```
{
    "code": 200,
    "command":"signTransaction",
    "trytes": ["....","....",...],
    "duration": 1800
}
```

```
{
    "code": 400,
    "command":"signTransaction",
    "error": "Error message"
}
```

4.9.8 jsonDataTX

Creates a zero-value transaction that contains the given JSON data in the 'signatureMessageFragment' field.

This command returns the transaction trytes (including proof of work) of the zero-value transaction.

These trytes are ready for sending to a node.

Example Request

```
{
    "command": "jsonDataTX",
    "trunkTransaction":"...",
    "branchTransaction":"...",
    "minWeightMagnitude": 14,
    "tag":"...",
    "address":"...",
    "timestamp":1566907523000,
    "data":{"test":"myFirstCryptoCoreTransaction"}
}
```

Examples Responses

```
{
    "code": 200,
    "command": "jsonDataTX",
    "hash": "...9999",
    "trytes": ["....","....",...],
    "duration": 1800
}
```

```
{
    "code": 400,
    "command": "jsonDataTX",
    "error": "Error message"
}
```

4.9.9 initSecureElement

Initializes the secure element so that the API can access the seed on it.

This command is a security measure that prevents attackers from removing the secure element, replacing it with another and reading the AES key from the RISC-V firmware. Before RISC-V shares the AES key with the secure element, you must call this command to prove that you know the key.

This command needs to be called only once.

Example Request

```
{
    "command": "initSecureElement",
    "key": "3780e63d4968ade5d822c013fcc323845d1b569fe705b60006feec145a0db1e3"
}
```

Example Responses

```
{
    "code": 200,
    "command": "initSecureElement",
    "duration": 1800
}
```

```
{
    "code": 400,
    "command": "initSecureElement",
    "error": "Error message"
}
```

4.9.10 readFlashPage

The readFlashPage command is used for reading pages (4kB) from QSPI flash memory. It is the same QSPI flash the FPGA uses when starting the configuration process. There is no (software) restriction other than valid page-numbers (0-4095) because the QSPI flash is not a secured memory. The output data is in base64 format.

```
{
    "command": "readFlashPage",
    "page": 0
}
```

Example Responses

```
{
    "code": 200,
    "command": "readFlashPage",
    "duration": 12
    "data:"..b64.."
}
```

```
{
    "code": 400,
    "command": "readFlashPage",
    "error": "Error message"
}
```

4.9.11 writeFlashPage

The writeFlashPage command is used for writing pages (4kB) into the QSPI flash memory. It is the same QSPI flash the FPGA uses when starting the configuration process. This way, the soft-cpu can update the entire system by writing new bitstreams into the flash. This API-call is restricted. The 'auth'-parameter is calculated this way:

Parameters

Parameter	Type	Description
page	integer	page number in QSPI flash. Valid values are [0...4095]
data	string	4kB data in Base64 format
auth	string	Checksum and authentication auth = hexString(keccak384(page+data+apiKey))

```
{
    "command": "writeFlashPage",
    "page": 0
    "data": "..b64..",
    "auth": "....",
}
```

Example Responses

```
{
    "code": 200,
    "command": "writeFlashPage",
    "duration": 100
}
```

```
{
    "code": 400,
    "command": "writeFlashPage",
```

```
4       "error": "Error message"
5   }
```

Chapter 5

Appendix

5.1 Programmer's Manual

5.1.1 RISC-V Soft CPU

The firmware in the CryptoCore uses a soft CPU with a 32-bit implementation of the RISC-V instruction set called "VexRiscv". This implementation won the RISC-V foundation's soft CPU competition in 2018, and comes with the following benefits:

- Written in the Scala programming language, which means you can extend it through plugins.
- Licensed under MIT, which allows you to use it in commercial environments.

Memory Map

Address	Size	Comment
0x00000000	128kB	ROM
0x80000000	128kB	RAM
0x84000000	4kB	Secured RAM
0xF1030000	64kB	GPIO pins and alternate functions (AF) switch
0xF1040000	64kB	SPI master
0xF1070000	64kB	UART
0xF10A0000	64kB	I2C master
0xF4000000	64kB	Secure element
0xF4010000	64kB	System timer
0xF4020000	64kB	QSPI master for QSPI flash
0xF4030000	4kB	RAM for the converter
0xF4040000	64kB	Converter (trits/trytes ⇔ bits/bytes)
0xF4050000	64kB	PiDiver (hashing)

ROM, RAM, Secured RAM The ICCFPGA module supports 128 kB ROM, 128 kB RAM, as well as an additional 4 kB of RAM, which only is accessible in privileged modes. It is used as key-store for the AES encryption key (for decrypting communicatios received from the Secure Element) and API key. Also it is used in Machine-mode as protected memory for stack.

Secure Element The secure element is attached to the I2C interface, which is accessible only in Supervisor mode. This interface also is used for booting alternative bitstreams from flash memory.

PiDiver The PiDiver component accelerates hashing such as Curl-P81, Keccak384 (SHA3) and Troika. It also does Proof-of-Work. All hashing-algorithms need a single clock-cycle per hashing-round.

Converter The converter converts binary data to/from ternary.

GPIO Pins and the Alternate Functions (AF) Switch There are 19 GPIO pins, which can be used as digital inputs or outputs. Pins 0 to 8 can changed to use as SPI, I2C, and UART. The base component is an AXI GPIO, for which documentation can be found here: `https://www.xilinx.com/support/documentation/ip_documentation/axi_gpio/v2_0/pg144-axi-gpio.pdf`.

The GPIO periphery is used for reading/writing the GPIO pins and for changing their function.

The first GPIO channel is used for the actual GPIO pins, where each one can be configured to be an input or an output.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
LCK	x	x	x	x	x	x	x	x	x	x	LEDL	LEDU	IO18	IO17	IO16

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IO15	IO14	IO13	IO12	IO11	IO10	IO9	IO8	IO7	IO6	IO5	IO4	IO3	IO2	IO1	IO0

GPIO pins 19 and 20 are wired to two LEDs on the ICCFPGA module. Pin IO19 is connected to "LED User", and pin IO20 is connected to "LED Lock". Pin 31 always reads the LOCK bit in the JTAG peripheral, which is routed to the GPIO pins.

Alternate function for GPIO pins:

- IO0 (MOSI), IO1 (MISO), IO2 (SCK), IO3 (SS0), IO4 (SS1): SPI master with two slave-select lines

- IO5 (SCL), IO6 (SDA): I2C master
- IO7 (RXD) , IO8 (TXD): UART

Alternate functions can be selected by writing a '1' to the corresponding bit location of the second GPIO channel of the AXI GPIO.

SPI Master The SPI master supports two slaves, and is connected to the AF switch (pins 0 to 4). AXI documentation: `https://www.xilinx.com/support/documentation/ip_documentation/axi_quad_spi/v3_2/pg153-axi-quad-spi.pdf`

QSPI Master The QSPI master is connected to the 16 MB flash storage on the ICCFPGA module, and is used for automatically configuring it at boot. and is accessible from the soft CPU, which allows you to update the bitstream. AXI-Documentation: `https://www.xilinx.com/support/documentation/ip_documentation/axi_quad_spi/v3_2/pg153-axi-quad-spi.pdf`

UART The UART is used for input and output of data to/from the ICCFPGA module. If no UART is needed, the UART pins can be used as GPIO pins by disabling the function in the AF switch (GPIO pins 7 to 8). AXI documentation: `https://www.xilinx.com/support/documentation/ip_documentation/axi_uartlite/v2_0/pg142-axi-uartlite.pdf`

Converter RAM The converter RAM is used by the converter when converting data to/from binary and ternary.

System Timer The system timer generates timer interrupts with a frequency of 1 kHz for time measurements or delays in software.

I2C Master The I2C master can be used to attach I2C peripherals to the ICCFPGA module. It also is connected to the AF switch (pins 5 and 6).

5.1.2 Software-Security

Privilege Modes

The VexRiscv in the IOTA Crypto Core project supports the following privilege modes:

Machine-Mode is the highest privilege mode, which gives code access to all functionality. By default, the CPU starts in Machine mode is then switched to lower privilege modes on boot.

User Mode is the lowest privilege mode. Code in User mode must call code running in higher privilege modes via an special interface.

The specification of RISC-V also mentions Supervisor- und Hypervisor-Mode but both are neither used nor implemented in the soft-cpu.

Physical Memory Protection

Although the firmware only executes "benign code", it could be compromised via external interfaces and "malicious code" could be injected or attack techniques such as buffer overflows could lead to exposing data that should not be accessible. Physical Memory Protection tries to prevent such instances but assigning privileges to blocks of memory which are checked with each access. There are three permissions which can be set: read, write, execute.

The PMP plugin for the VexRiscv soft-cpu implements the PMP proposal from the Instruction Set Manual Volume II (Privileged Architecture) from the RISC-V Foundation[1] with one custom extension.

Normally, it is assumed that code running in Machine-mode has access to everything. It can be restricted by locking protected memory regions which enforces privilege checks also in machine-mode but there are some restrictions which lead to compromises.

[1] `https://riscv.org/specifications/privileged-isa/`

For instance, it is not possible to give one block of memory different permissions for Machine- and User-mode. If Machine-mode is restricted (locked region), the same privileges also are applied to User-mode. In the other case (unlocked region) User-mode is restricted but everything is allowed in Machine-mode.

For this reason, the implementation of PMP was extended using two unused Bits in the configuration registers that encode for which mode the privileges are applied.

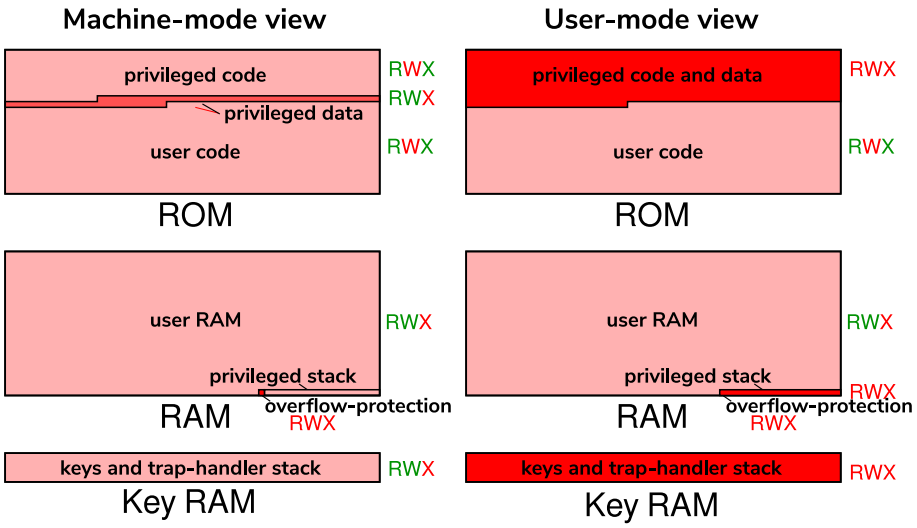
Protecting memory follows an opt-in, opt-out principe:

- Everything which is not explicitely allowed in User-mode is forbidden
- Everything which is not explicitely forbidden in Machine-mode is allowed

This only applies if there is at least one active protection rule.

Protected Memory Areas

The following describes how the memory was protected in the reference firmware for the IOTA Crypto Core module.



The characters R, W, X in the diagram above encode the privileges ‘readable‘, ‘writable‘ and ‘executable‘.

In Machine-mode the entire ROM is executable with exception of a small section used for privileged data - the rest of the memory is neither in Machine- nor in User-mode executable. There is an extra stack for privileged code that is protected against overflows by an ‘overflow-protection‘ (a single 32bit address which is not accessible in Machine- or User-mode) which prevents leaking of sensitive data into user-RAM (either intended by trying to exploit stack-overflows or unintended by using to much stack for local variables in privileged code). This single 32bit address is completly locked for Machine- and User-mode and access causes the system to halt completly. Additionally, there is a 4kB memory block in which AES- and API-keys are stored. It also is used as stack for the trap-handler.

In User-mode, only access to memory for user-Code and user-RAM or peripherals such as GPIOs, I2C, SPI and UART peripherals is allowed.

One further speciality is that protection for privileged code and data is dynamically initialized at the start of the firmware - so no static memory protection layout is needed which allows to use the rest that is not used by privileged code or data for user code and data.

Though, there is one detail the developer should be aware of: Privileged code and data has to be explicitely assigned to protected memory sections in the source-code.

This is done by simply adding ‘MMTEXT‘ (readable, executable, not writable), ‘MMDATA‘ (readable, writeable, not executable) or ‘MMRODATA‘ (readable, not writable, not executable) to functions or variables.

```
1 uint32_t MMDATA protected_variable;
2
3 const uint32_t MMRODATA protected_constant = 0xc001cafe;
4
5 uint32_t MMTEXT protected_function() {
6 ...
7 }
```

Calls to Privileged Code

Most peripherals are protected by PMP, preventing User-code to use it directly. Instead, the code in User-mode has to call a binary interface (supervisor binary interface) which executes privileged code.

The program flow is as following:

- User-code calls a user-callable function in the SBI-namespace

```
StatusResponse* reboot(uint32_t address) {
        return (StatusResponse*) SBI_CALL_1(REBOOT, address);
}
```

- The function in the SBI-namespace executes an 'ECALL' (jump to the trap-handler that is running in machine-mode)

```
#define SBI_CALL(func, arg0, arg1, arg2, arg3, arg4) ({          \
    register uintptr_t a0 asm ("a0") = (uintptr_t)(func);        \
    register uintptr_t a1 asm ("a1") = (uintptr_t)(arg0);        \
    register uintptr_t a2 asm ("a2") = (uintptr_t)(arg1);        \
    register uintptr_t a3 asm ("a3") = (uintptr_t)(arg2);        \
    register uintptr_t a4 asm ("a4") = (uintptr_t)(arg3);        \
    register uintptr_t a5 asm ("a5") = (uintptr_t)(arg4);        \
    asm volatile ("ecall"                                        \
          : "+r" (a0)                                            \
          : "r" (a1), "r" (a2), "r" (a3), "r" (a4), "r" (a5)     \
          : "memory");                                           \
    a0;                                                          \
})

#define SBI_CALL_1(which, arg0) \
        SBI_CALL(which, arg0, 0, 0, 0, 0)
```

- The trap-handler returns in machine-mode to the dispatcher-function that evaluates the 'func'-parameter and calls the right function

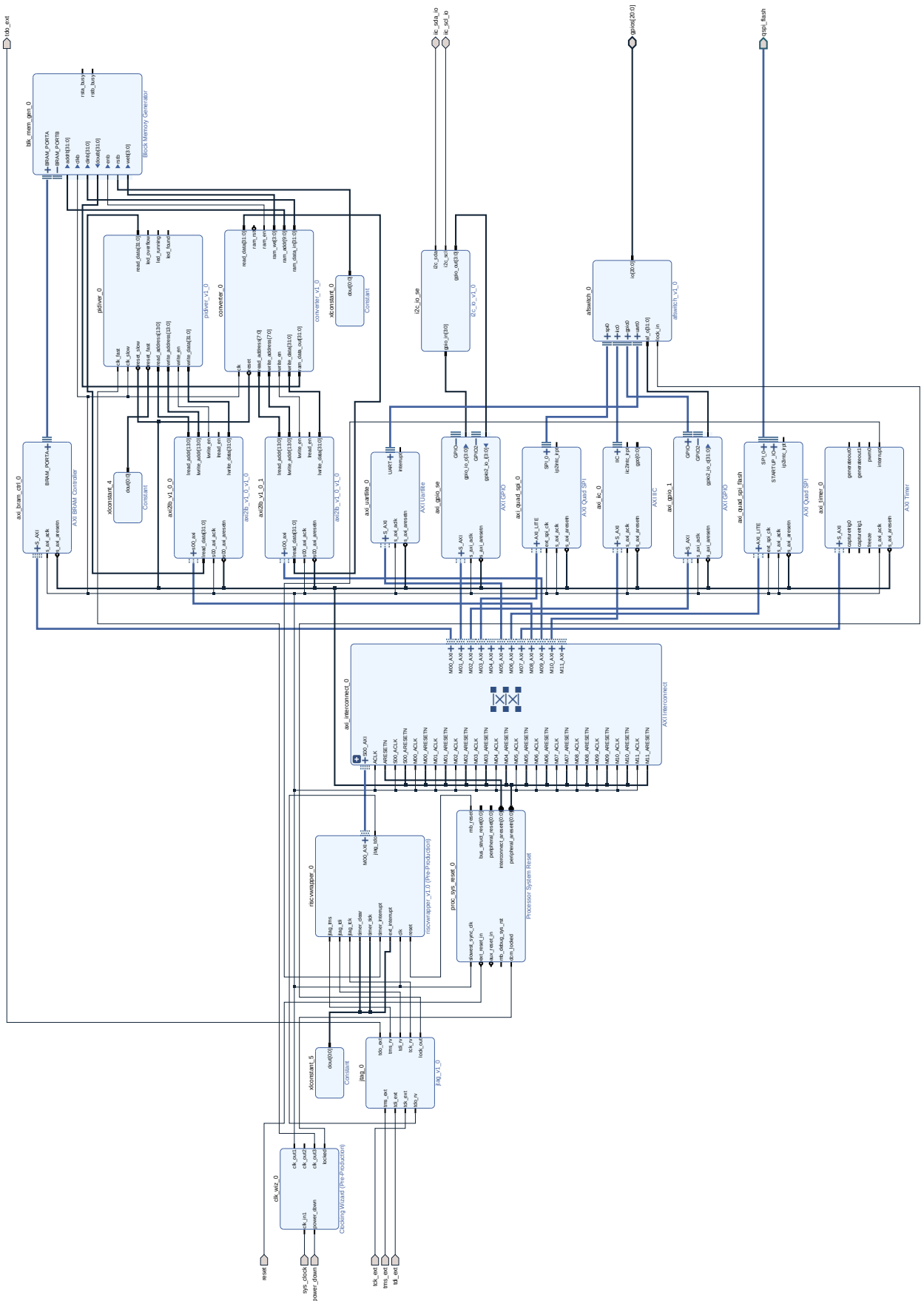
```
uint32_t* MMTEXT sbiCall(
    uint32_t func, uint32_t arg0, uint32_t arg1,
    uint32_t arg2, uint32_t arg3, uint32_t arg4) {

    switch (func) {
// ...
    case REBOOT:
        return (uint32_t*) _reboot(arg0);
// ...
    return 0;
}
```

- Following an example of the privileged 'reboot' command:

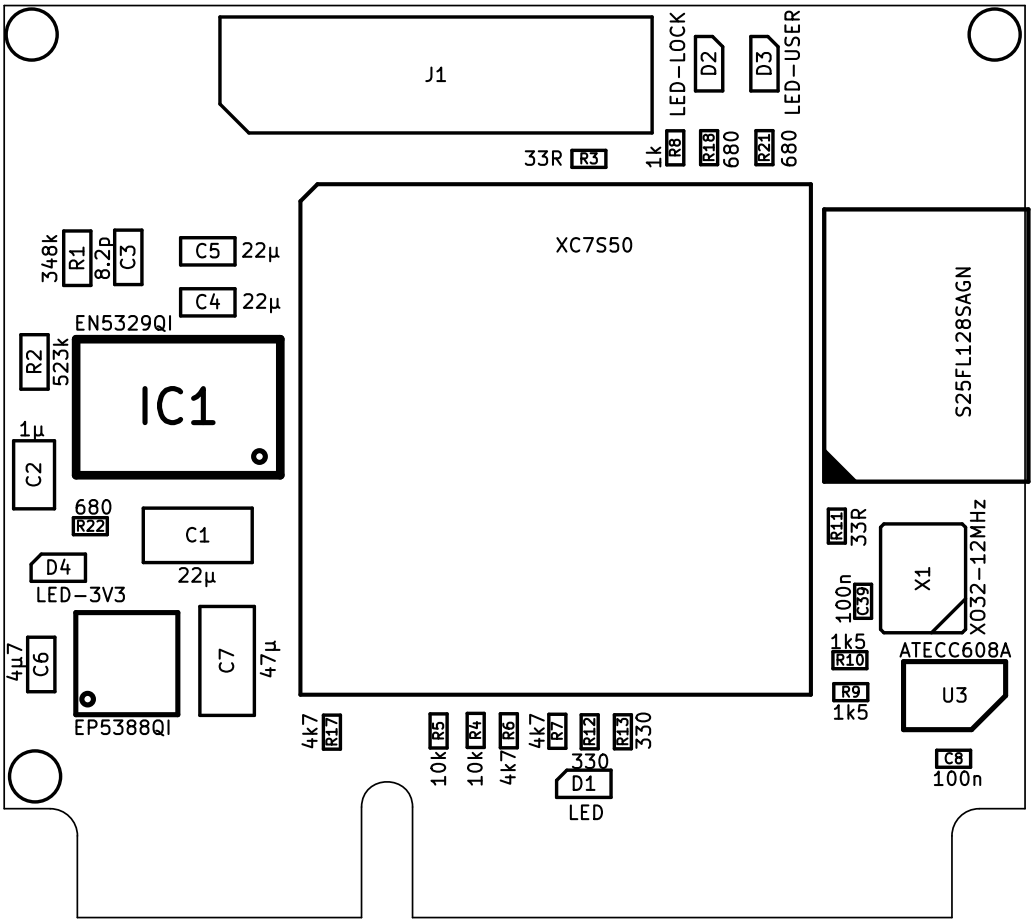
```
StatusResponse* _reboot(uint32_t address) {
    StatusResponse* resp = &statusResponse;
    do {
        if ((address & 0x0000ffff) || address > 0x1000000) {
            resp->error(SBIError::INVALID_ADDRESS);
            break;
        }
        // iprog is triggered one second after
        SecureGPIO::triggerIPROG(address);
        resp->error(SBIError::NONE);
    } while(0);
    return resp;
}
```

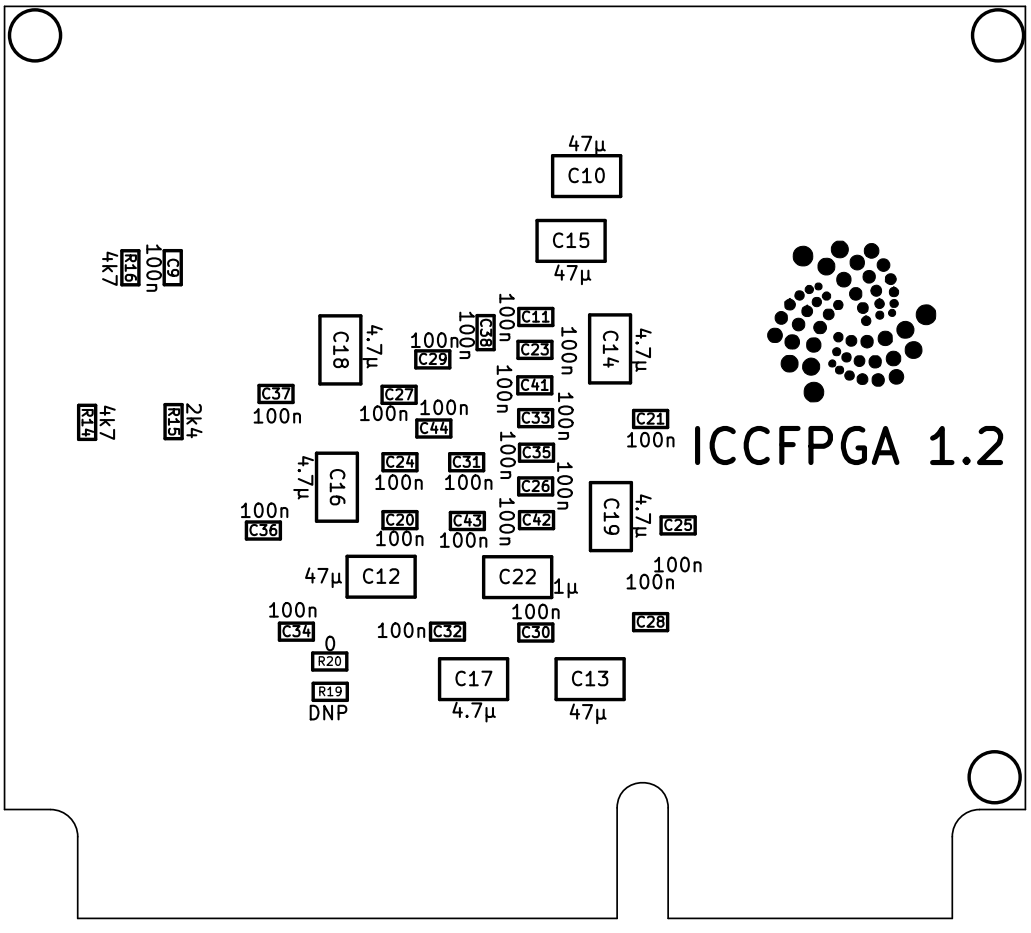
- After executing the privileged function, the CPU returns to the user-callable function, returning a pointer to an object. This object could contain only a status code or data or pointers to data. The return-objects are statically defined and are reused with each call for safety reasons.

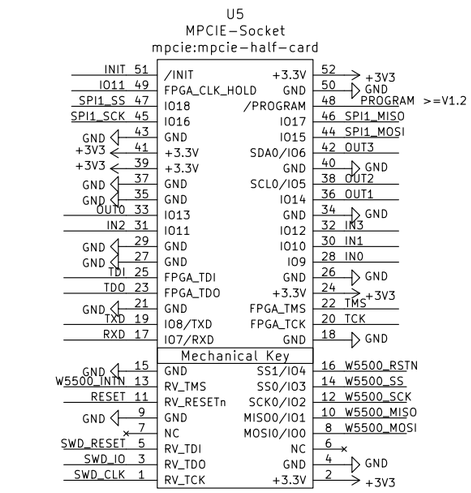
5.2 SoC System Overview

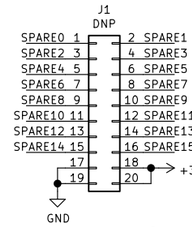


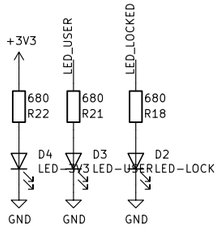
5.3 ICCFPGA Module Component Placement and Schematic

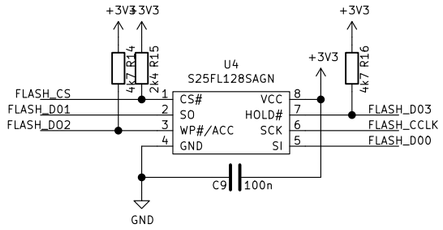


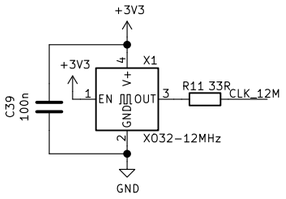


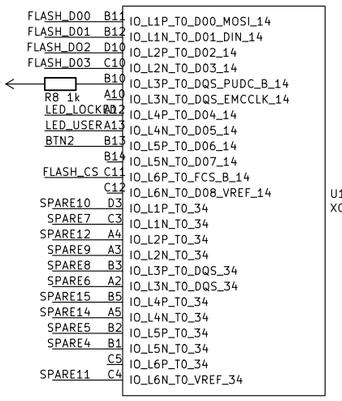


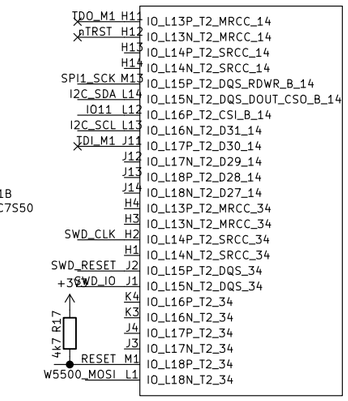


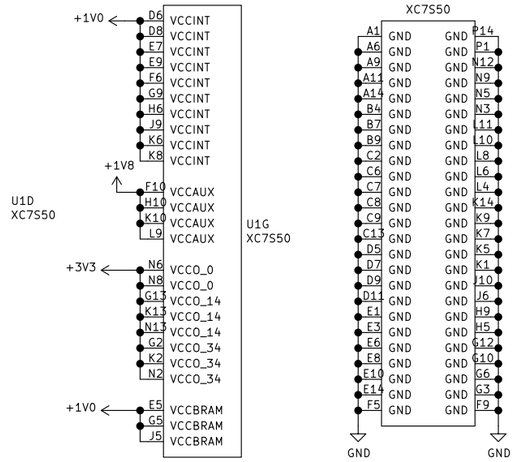


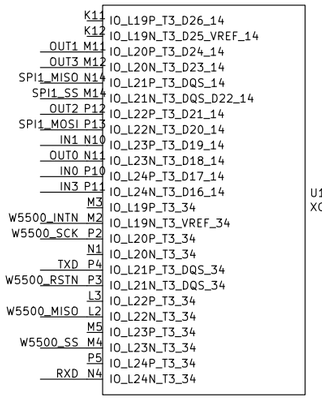


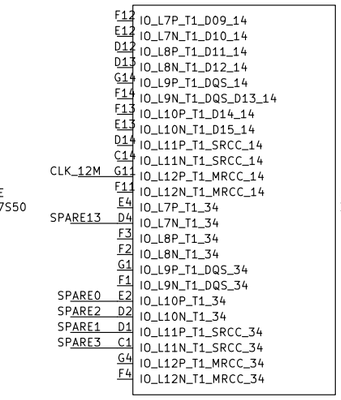


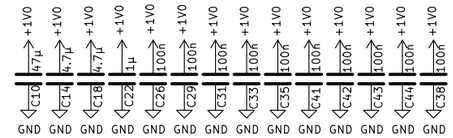


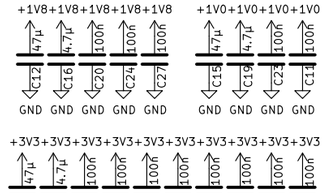


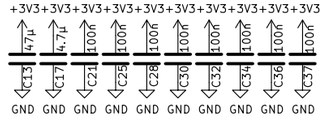


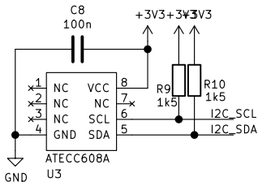


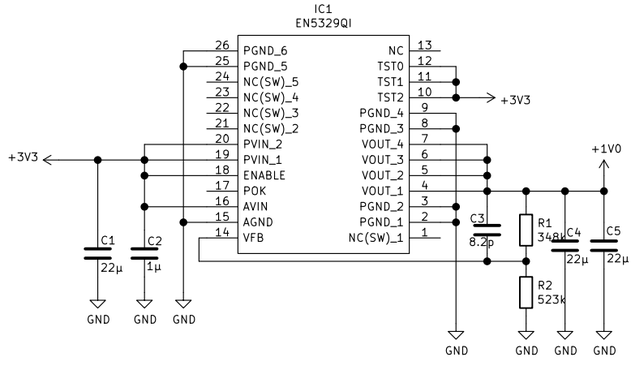


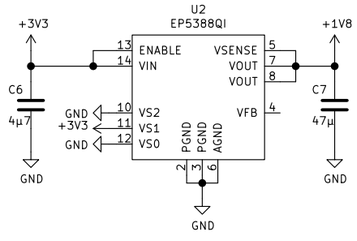


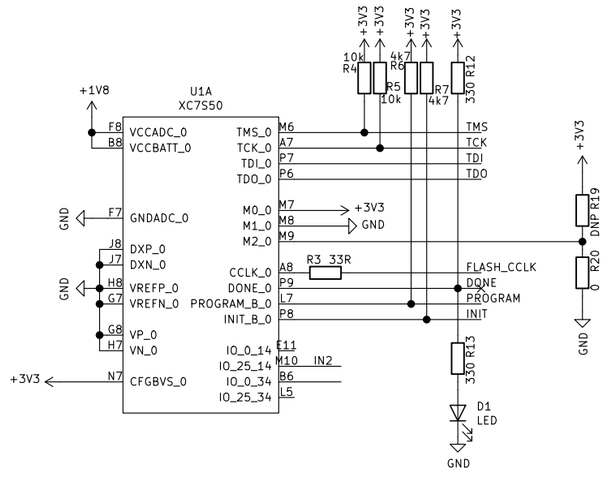










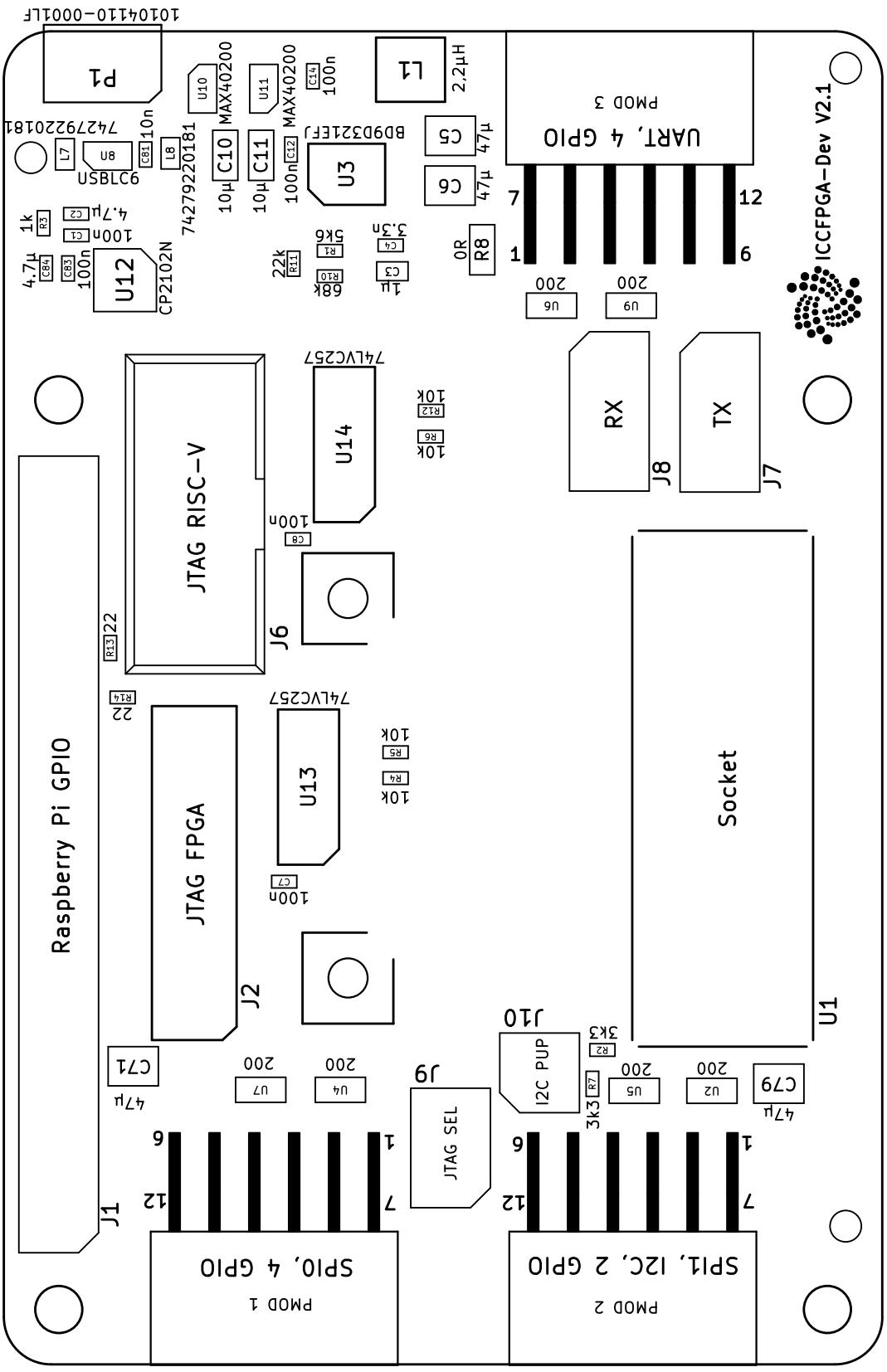


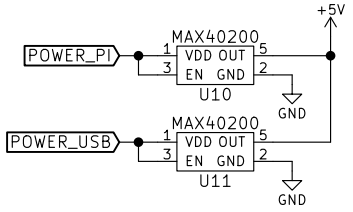
Sheet: /
File: iccfpga.sch

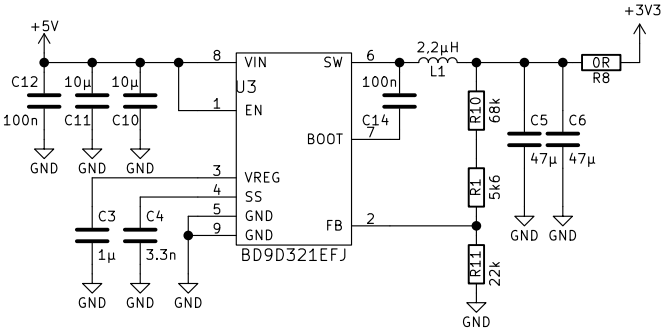
Title:

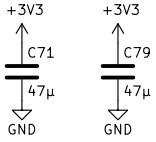
Size: A3	Date:	Rev:
KiCad E.D.A. kicad 5.1.4-e60b26684ubuntu18.04.1		Id: 1/1

5.4 Development Board Component Placement and Schematics





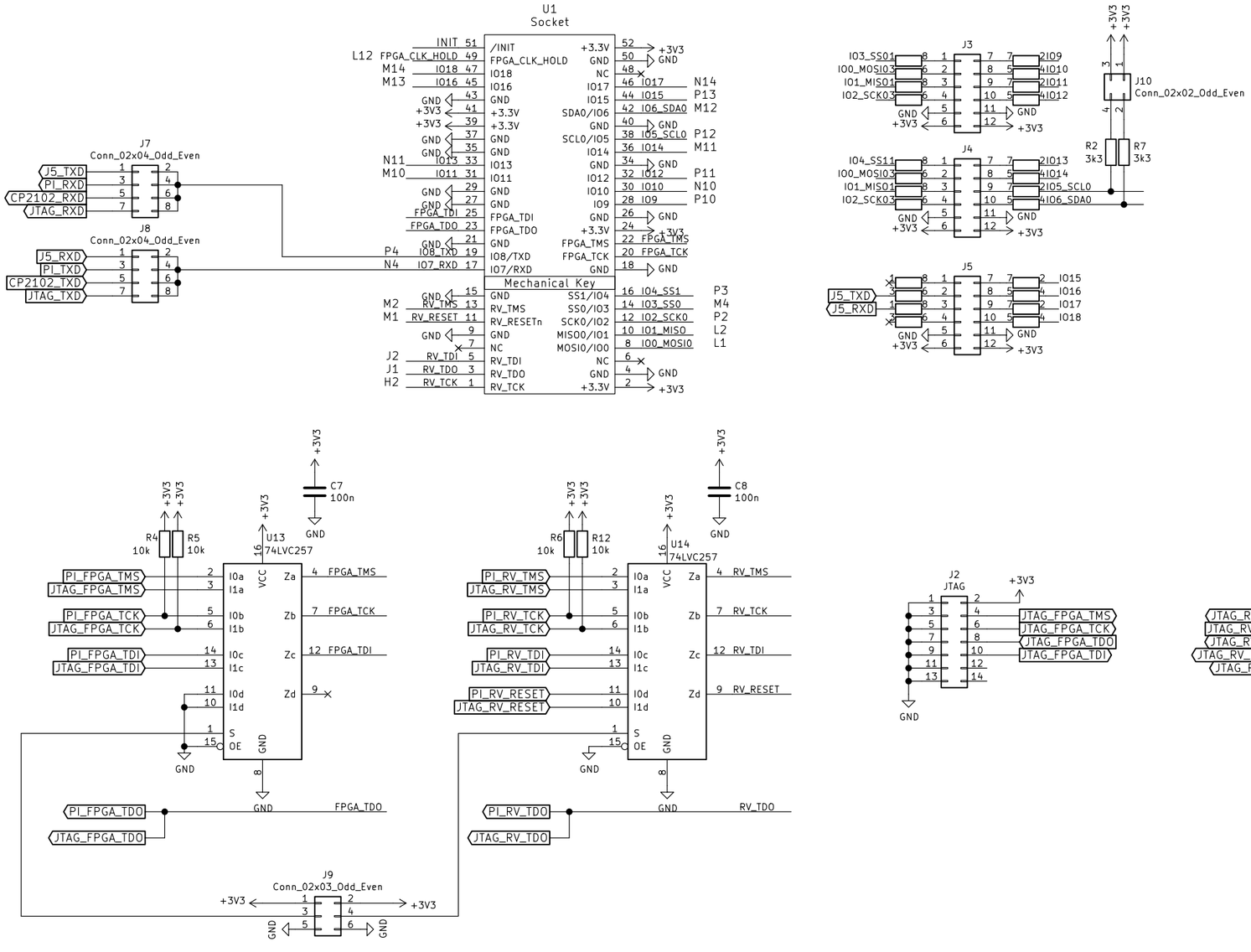




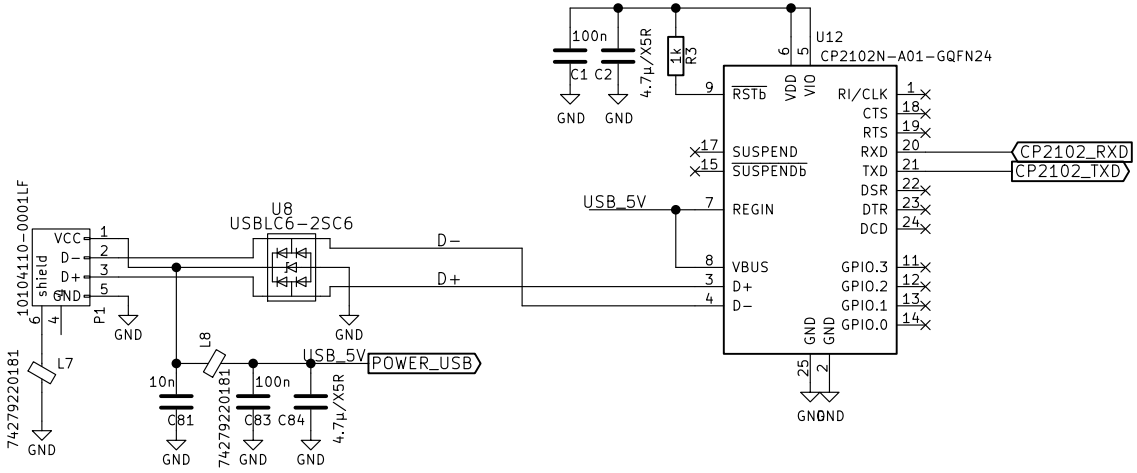
Sheet: /power/
File: power.sch

Title:

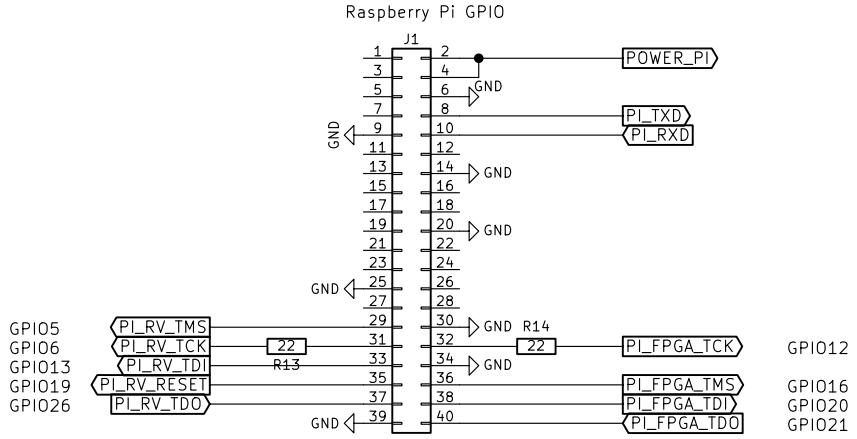
Size: A4	Date:	**Rev:**
KiCad E.D.A. kicad 5.1.4-e60b26684ubuntu18.04.1		Id: 2/5



Sheet: /fpga/	
File: fpga.sch	
Title:	
Size: A3	Date:
KiCad E.D.A. kicad 5.1.4-e60b26684ubuntu18.04.1	Id: 3/5



Sheet: /cp2102usb/		
File: cp2102usb.sch		
Title:		
Size: A4	Date:	**Rev:**
KiCad E.D.A. kicad 5.1.4-e60b26684ubuntu18.04.1		Id: 4/5



Sheet: /rpigpio/
File: rpigpio.sch

Title:

Size: A4	Date:	**Rev:**
KiCad E.D.A. kicad 5.1.4-e60b26684ubuntu18.04.1		Id: 5/5