

**THE IMAGINATION UNIVERSITY PROGRAMME**

**RVfpga-SoC Lab 4**

**Running Zephyr on SweRVolf**

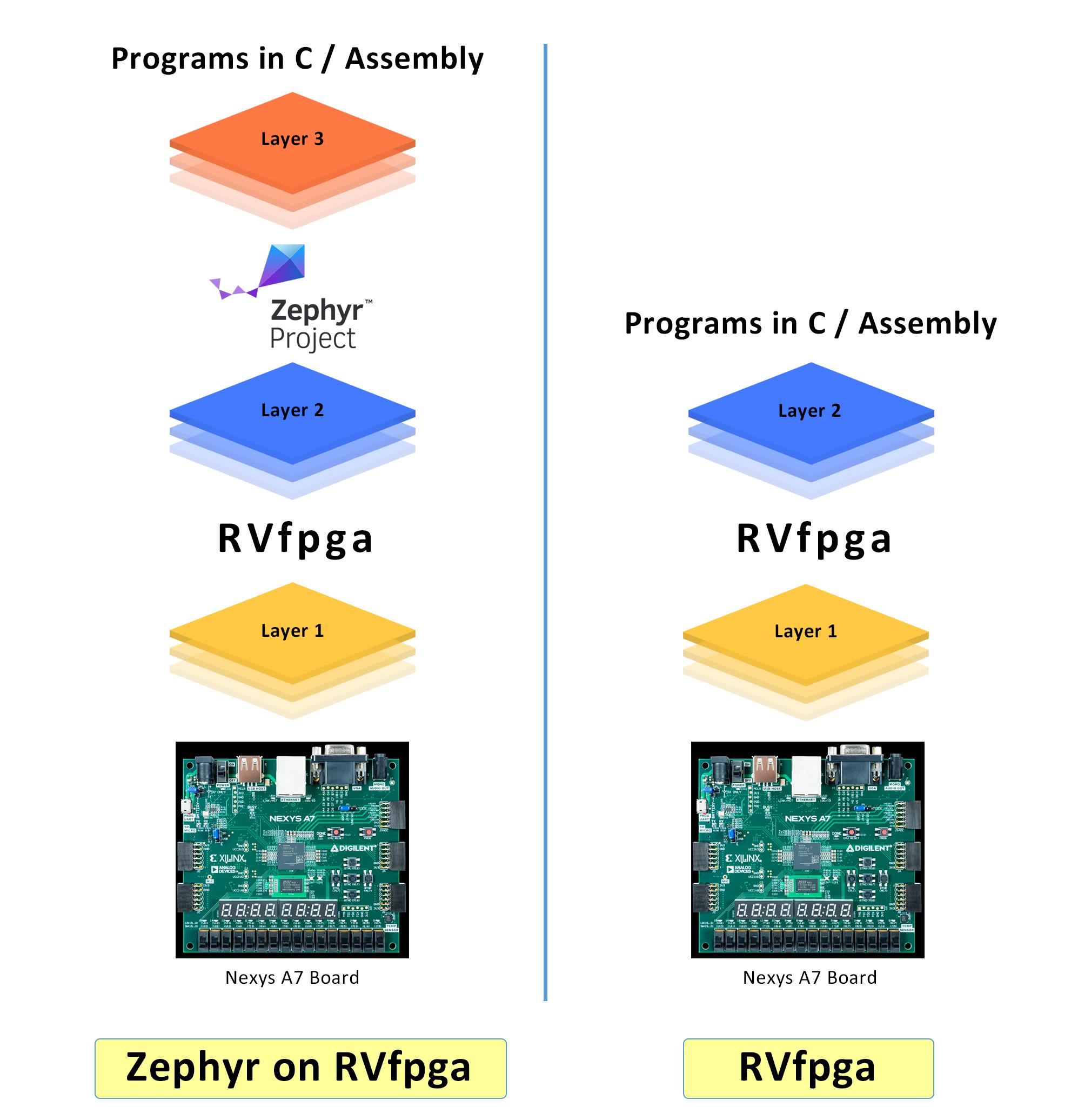
# Introduction :

In this Lab, we show how to run the Zephyr real-time operating system (RTOS) on SweRVolf. A real-time operating system (RTOS) is an operating system intended to serve real-time applications that process data as it comes in, mostly without buffer delay.

In Labs 2 and 3, we have been running simple programs written in the RISC-V assembly or C language. In practical applications, an SoC will almost always be running an operating system, and applications will be running on top of the operating system.

Two overall categories of operating systems for embedded systems exist: embedded Linux-based operating systems and real-time operating systems (RTOS). When an SoC is designed with a particular CPU, the design is usually tuned to use one or the other type of operating system. SweRVolf has been built with the intention of running a real-time operating system. The SweRV EH1 CPU does not have a memory management unit and would, thus, struggle to run embedded Linux.

Figure 1 shows an illustration of the different hardware/software layers in the overall system.



**Figure 1. Layers on the top of FPGA Boards**

In this Lab, we will describe the Zephyr RTOS, build and run the Zephyr RTOS on RVfpga-SoC, and build & run applications that run on Zephyr.

# Requirements :

To complete this lab, you will need to install the following:

* Vivado 2019.2 Web Pack (Refer to Installation Guide (Page No.04))
* Verilator (v4.106) (Refer to Installation Guide (Page No.13))
* FuseSoC (Refer to Installation Guide (Page No.14))
* OpenOCD (RISC-V-specific version) (Refer to Installation Guide (Page No.14))
* Zephyr Prerequisites (Refer to Installation Guide (Page No.15))
* Zephyr SDK (v0.12.3) (Refer to Installation Guide (Page No.15))

**IMPORTANT:** Before starting RVfpga-SoC Labs, we highly recommend completing the RVfpga-SoC Installation Guide.

For example, if you have not already, install Xilinx’s Vivado and Verilator following the instructions in the RVfpga-SoC Installation Guide. Make sure that you have copied the RVfpga-SoC folder that you downloaded from Imagination’s University Programme to your machine.

# Zephyr Overview:

The Zephyr Project is a scalable real-time operating system supporting multiple hardware architectures, optimized for resource-constrained devices, and built with security in mind. The Zephyr OS is based on a small-footprint kernel designed for use on resource-constrained systems: from simple embedded environmental sensors and LED wearables to sophisticated smart watches and IoT wireless gateways.

Zephyr offers a number of familiar services for development: Multi-threading, Interrupts, Memory Allocation, Inter-thread Synchronization, Inter-thread Data Passing, and Power Management.

Zephyr supports a wide variety of boards with different CPU architectures and developer tools. Contributors have added support for an increasing number of SoCs, platforms, and drivers.

The Zephyr kernel supports multiple architectures, including:

* RISC-V (32- and 64-bit)

For more detailed information on the Zephyr Project, read the Zephyr project documentation at [http://docs.zephyrproject.org](http://docs.zephyrproject.org/).

In this lab, we first show how to add Zephyr’s version 2.4 to our Workspace. Then we will build the code for a few sample examples that come with Zephyr. This lab will show examples of using Zephyr both in hardware and simulation.

# Understanding the Hardware/Software Layers:

In Labs 2 and 3, our process of running programs on the FPGA board followed these steps:

**Step 1**. **Download RVfpga onto the FPGA board**

First, we download the RVfpga, the RISC-V system targeted to an FPGA, to the

Nexys A7 FPGA board. We download the RVfpga onto the board by either uploading the bitstream to the board using PlatformIO or by using the FuseSoC run command, which uploads the generated bitstream to the board if it's connected.

**Step 2**. **Build and run programs on RVfpga**

The second step is to download the program onto RVfpga.

In this Lab, we will amend these steps to add another layer, the Zephyr RTOS (real-time operating system) onto RVfpga, and run programs on Zephyr. The steps for doing this are as follows:

**Step 1**. **Download RVfpga onto the FPGA board**

Same as above.

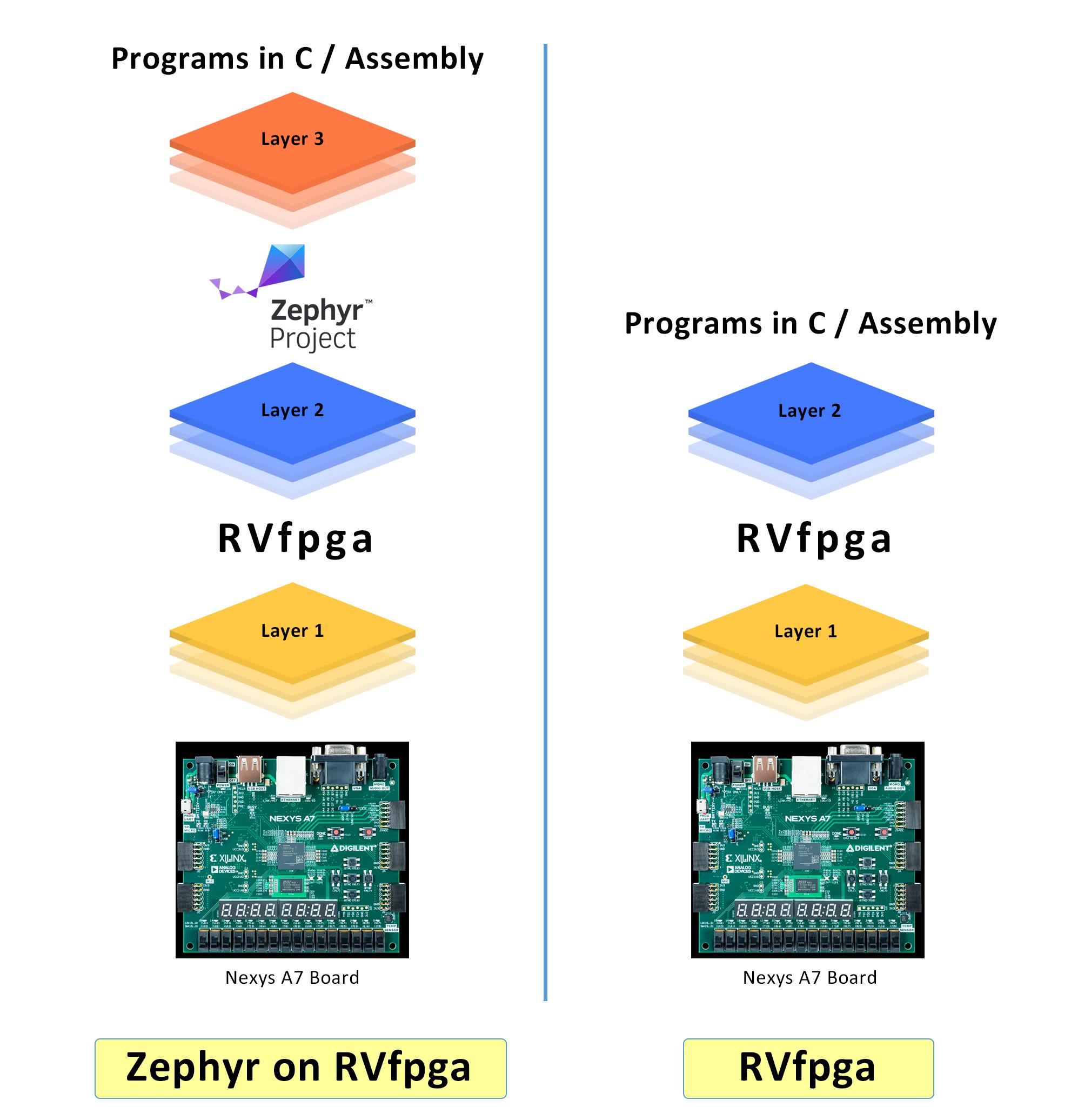
**Step 2. Add Zephyr to RVfpga-SoC/SweRVolf**

In this step, we loadZephyr onto RVfpga-SoC/SweRVolf.

**Step 3. Build and run programs on Zephyr, which is running on RVfpga**

In this step, we build an application for Zephyr and then run the application on Zephyr.

The side-by-side Illustration of both modes of running a program is shown in Figure 2.



**Figure 2. Layers on the top of FPGA Boards**

Now we will show how to build Zephyr applications and then run those applications on Zephyr.

# Adding Zephyr Support In SweRVolf :

In this section of the lab, we show how to add Zephyr to your WORKSPACE.

Open your Ubuntu terminal and complete the following steps :

**Step 1.** Navigate to the directory “**SweRVolf**” in which we created our workspace in the previous lab, to use as the root of the project. We called it **$WORKSPACE*.*** Now we have to set the same shell variables again. To do that, we run the following:

**$** export WORKSPACE=$(pwd)

**$** export SWERVOLF\_ROOT=$WORKSPACE/fusesoc\_libraries/swervolf

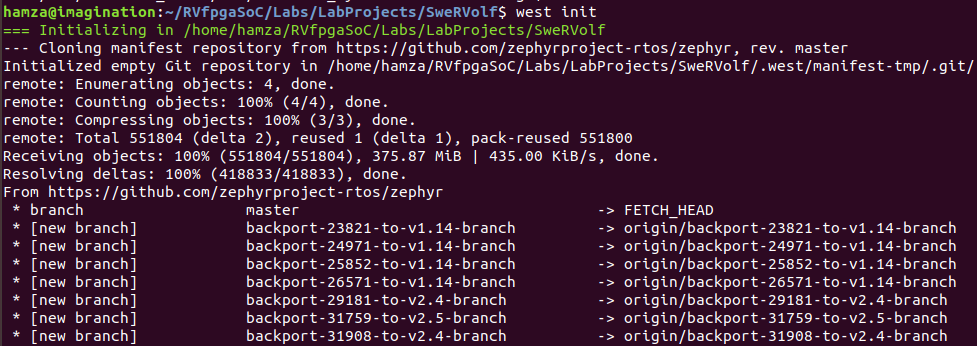


**Figure 3. Set the shell variables**

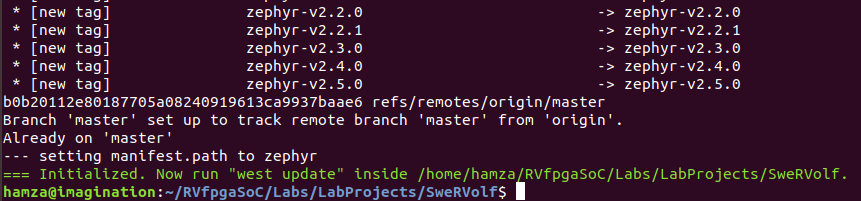
**Step 2.** Add Zephyr & SweRVolf-specific drivers

Create a West (Zephyr's build tool) workspace in the same directory as the FuseSoC workspace by running

**$** west init

****

**………..**

****

**Figure 4. west initialized**

**Step 3.** Add the SweRVolf-specific drivers and board support package (BSP) using the following command:

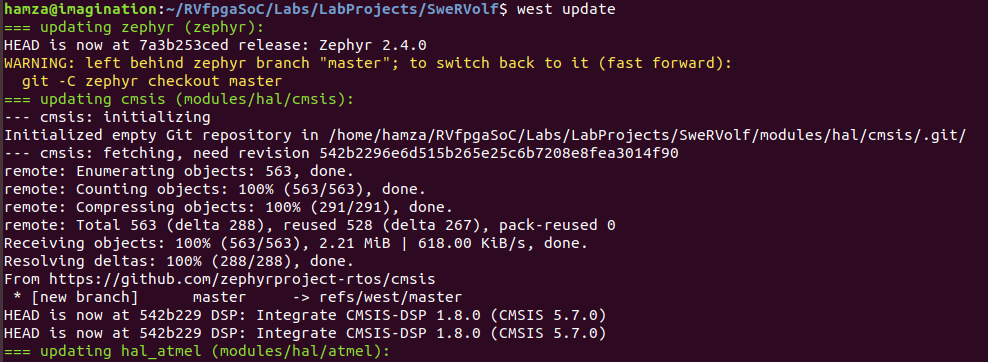
**$** west config manifest.path fusesoc\_libraries/swervolf

****

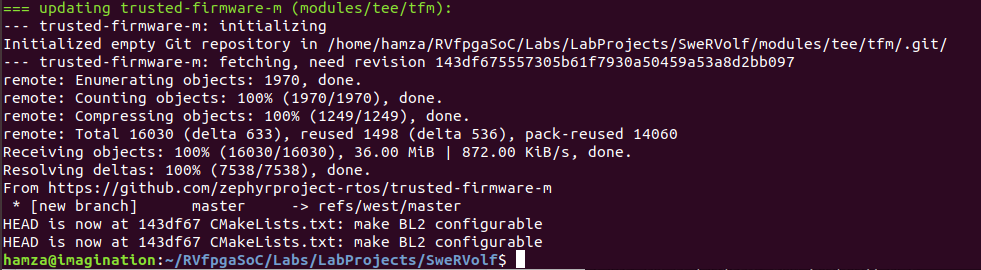
**Figure 5. west config**

**$** west update

This may take several minutes to complete the downloading process, depending on your Internet download speed.

****

**………..**

****

**Figure 6. west update**

The Workspace will now look like this :

$WORKSPACE

├──fusesoc\_libraries

| ├──...

| └──swervolf

├──...

└──zephyr

# Building and Running Zephyr Applications On Verilator:

In this section, we step through how to build programs that can run on Zephyr. Then we show how to simulate such programs on the Verilator simulator.We show two example programs in this section.

1. **Zephyr Hello World Example :**

This example prints “**Hello World**” + “**Configured Board Name**” on the terminal.

See Figure 7 for the source code.

|  |
| --- |
| 1  2 #include <zephyr.h>  3 #include <sys/printk.h>  4  5 void main(void)  6 {  7 printk("Hello World! %s\n", CONFIG\_BOARD);  8 }  9 |

**Figure 7. main.c of hello\_world example**

**Step 1.** Go to the directory for this example, which is located at the following path:

$WORKSPACE/zephyr/samples/hello\_world

To do so, use the following command:

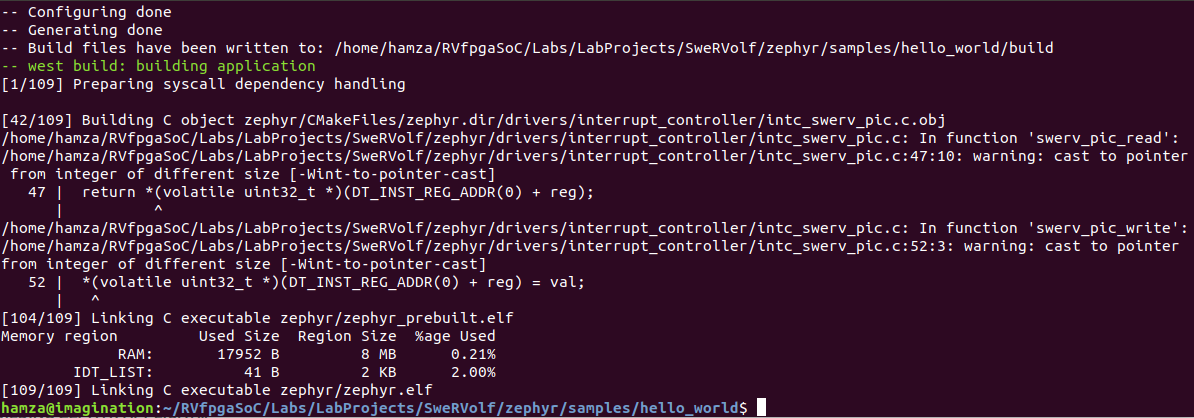
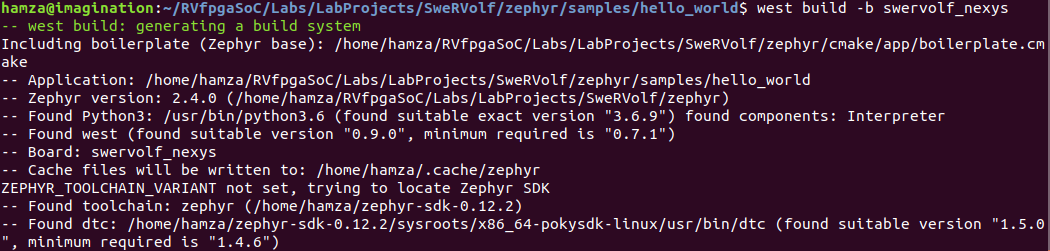
**$** cd zephyr/samples/hello\_world



**Figure 8. Navigate to the hello\_world directory**

**Step 2.** Build the code for the “**hello\_world**” Example using the following command :

**$** west build -b swervolf\_nexys



**Figure 9. hello\_world build**

This will create the zephyr.elf and zephyr.bin files for the **hello\_world** example. We will use the “**.bin**” file in a simulator, but it must first be converted into a suitable Verilog hex file.

**Step 3.**  Convert the .bin file to .hex file :

To create the .hex file, run the following command from the hello\_world directory :

**$** python3 $SWERVOLF\_ROOT/sw/makehex.py build/zephyr/zephyr.bin > /home/{YourUsername}/RVfpgaSoC/Labs/LabProjects/SweRVolf/zephyr/samples/hello\_world/App.hex



**Figure 10. hello\_world hex file created**

**Step 4.** Navigate to the WORKSPACE directory :

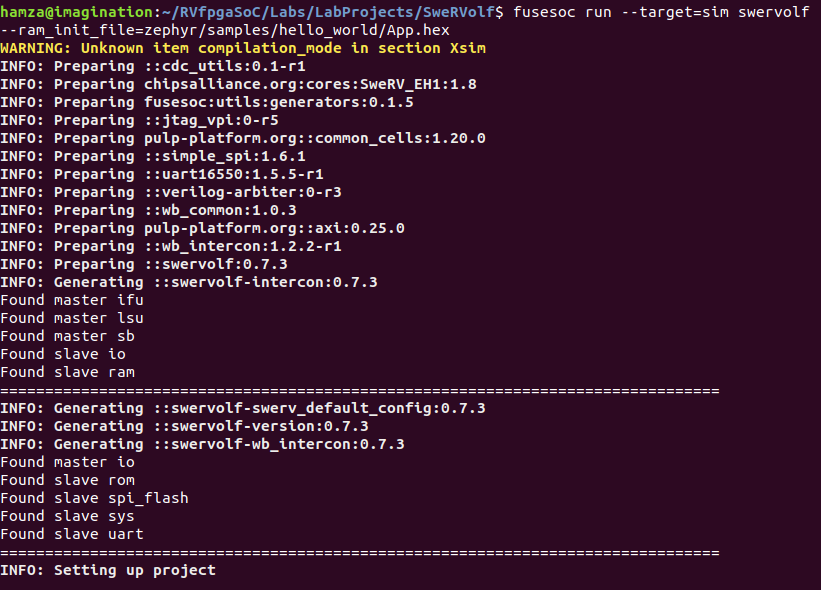
**$** cd $WORKSPACE



**Figure 11. Navigate to the main Workspace directory**

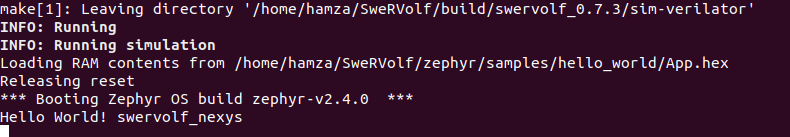
**Step 5.** Load the .hex file in the simulator :

**$** fusesoc run --target=sim swervolf --ram\_init\_file=zephyr/samples/hello\_world/App.hex



**Figure 12. fusesoc run**

The terminal will show the following output (see Figure 13).



**Figure 13. hello\_world example output**

Press “**Ctrl + C**” to stop the program.

1. **Zephyr Philosophers Example :**

An implementation of a solution to the Dining Philosophers Problem (a classic multi-thread synchronization problem). This particular implementation demonstrates the usage of multiple preemptible and cooperative threads of differing priorities, as well as dynamic mutexes and causing a thread to sleep.

The philosopher always tries to get the lowest fork first (f1 then f2). When done, he will give back the forks in the reverse order (f2 then f1). If he gets two forks, he is EATING. Otherwise, he is THINKING. Transitional states are shown as well, such as STARVING when the philosopher is hungry, but the forks are not available, and HOLDING ONE FORK when a philosopher is waiting for the second fork to be available.

Each Philosopher will randomly alternate between the EATING and THINKING state.

Go to the following path to see the source code of this example :

$WORKSPACE/zephyr/samples/philosophers/src/main.c

For this example, we will repeat the same process again but in the philosophers directory

**Step 1.** This example program is in the following directory:

$WORKSPACE/zephyr/samples/philosophers

Change to that directory using the following command:

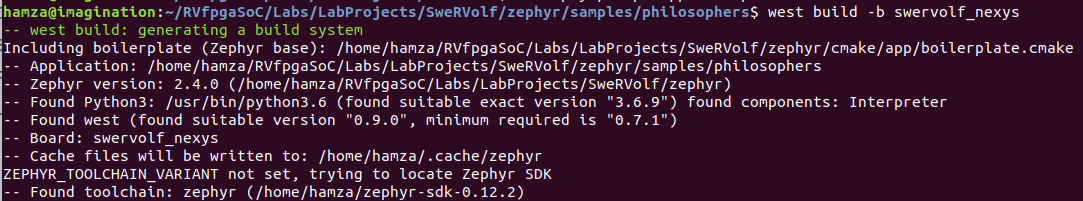
**$** cd zephyr/samples/philosophers

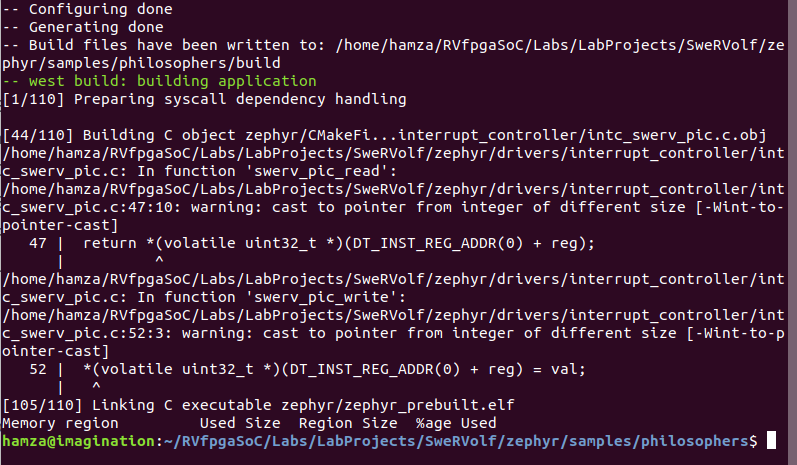


**Figure 14. Navigate to philosophers directory**

**Step 2.** Build the code for the philosophers example using the following command:

**$** west build -b swervolf\_nexys





**Figure 15. philosophers build**

This will create the zephyr.elf and zephyr.bin files for the philosophers example. Again we will convert the .bin file into a suitable Verilog hex file.

**Step 3.**  Convert the .bin file to .hex file

To create the Hex file, run the following command from the hello\_world directory :

**$** python3 $SWERVOLF\_ROOT/sw/makehex.py build/zephyr/zephyr.bin > /home/{YourUsername}/RVfpgaSoC/Labs/LabProjects/SweRVolf/zephyr/samples/philosophers/App.hex



**Figure 16. Create philosophers hex file**

**Step 4.** Navigate to the WORKSPACE directory:

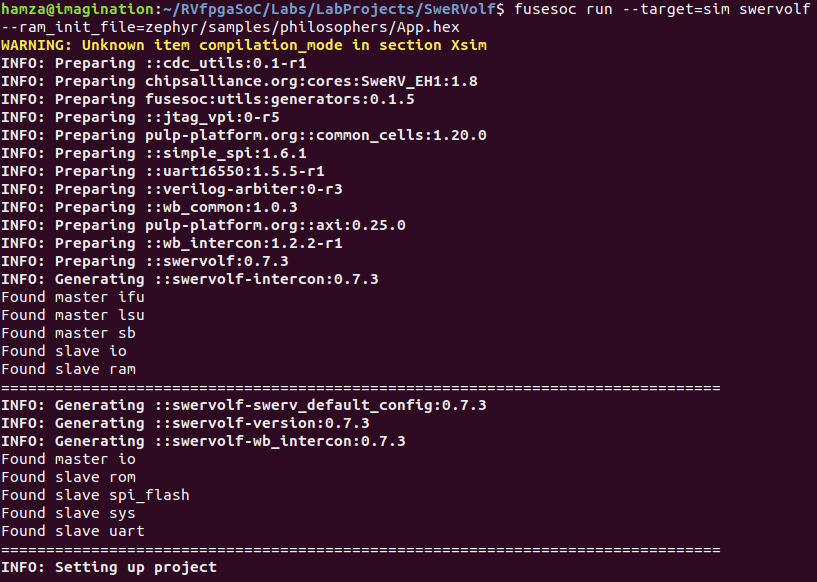
**$** cd $WORKSPACE



**Figure 17. main directory**

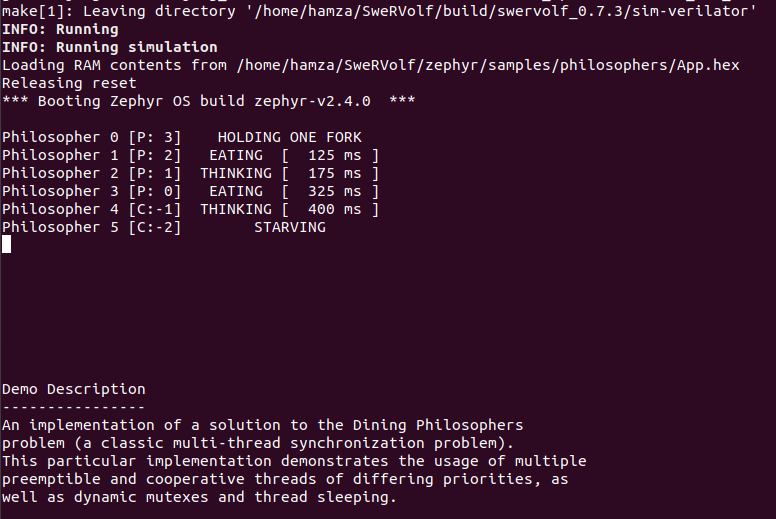
**Step 5.** Load the .hex file in the simulator:

**$** fusesoc run --target=sim swervolf --ram\_init\_file=zephyr/samples/philosophers/App.hex



**Figure 18. fusesoc run**

Now you will see the following output:



**Figure 19. Zephyr philosophers Output**

# Building Zephyr Application for Hardware:

Now we show how to build programs for the RVfpga-SoC running Zephyr in hardware.

1. **Zephyr Blinky Example:**

Blinky is a simple application that blinks an LED forever using the:`GPIO

API <gpio\_api>`. The source code shows how to configure GPIO pins as outputs,

then turn them on and off.

|  |
| --- |
| 1  2 /\*  3 \* Copyright (c) 2016 Intel Corporation  4 \*  5 \* SPDX-License-Identifier: Apache-2.0  6 \*/  7  8 #include <zephyr.h>  9 #include <device.h>  10 #include <devicetree.h>  11 #include <drivers/gpio.h>  12  13 /\* 1000 msec = 1 sec \*/  14 #define SLEEP\_TIME\_MS 1000  15  16 /\* The devicetree node identifier for the "led0" alias. \*/  17 #define LED0\_NODE DT\_ALIAS(led0)  18  19 #if DT\_NODE\_HAS\_STATUS(LED0\_NODE, okay)  20 #define LED0 DT\_GPIO\_LABEL(LED0\_NODE, gpios)  21 #define PIN DT\_GPIO\_PIN(LED0\_NODE, gpios)  22 #define FLAGS DT\_GPIO\_FLAGS(LED0\_NODE, gpios)  23 #else  24 /\* A build error here means your board isn't set up to blink an LED. \*/  25 #error "Unsupported board: led0 devicetree alias is not defined"  26 #define LED0 ""  27 #define PIN 0  28 #define FLAGS 0  29 #endif  30  31 void main(void)  32 {  33 const struct device \*dev;  34 bool led\_is\_on = true;  35 int ret;  36  37 dev = device\_get\_binding(LED0);  38 if (dev == NULL) {  39 return;  40 }  41  42 ret = gpio\_pin\_configure(dev, PIN, GPIO\_OUTPUT\_ACTIVE | FLAGS);  43 if (ret < 0) {  44 return;  45 }  46  47 while (1) {  48 gpio\_pin\_set(dev, PIN, (int)led\_is\_on);  49 led\_is\_on = !led\_is\_on;  50 k\_msleep(SLEEP\_TIME\_MS);  51 }  52 } |

**Figure 20. main.c of blinky example**

The path for this example is here:

$WORKSPACE/zephyr/samples/basic/blinky/

Navigate to the above path, and then run the following command in the terminal to build the example and generate “.elf” and “.bin” files :

**$** west build -b swervolf\_nexys

After building the code, there will now be an executable **.elf** file in build/zephyr/zephyr.elf and a **.bin** file in build/zephyr/zephyr.bin.

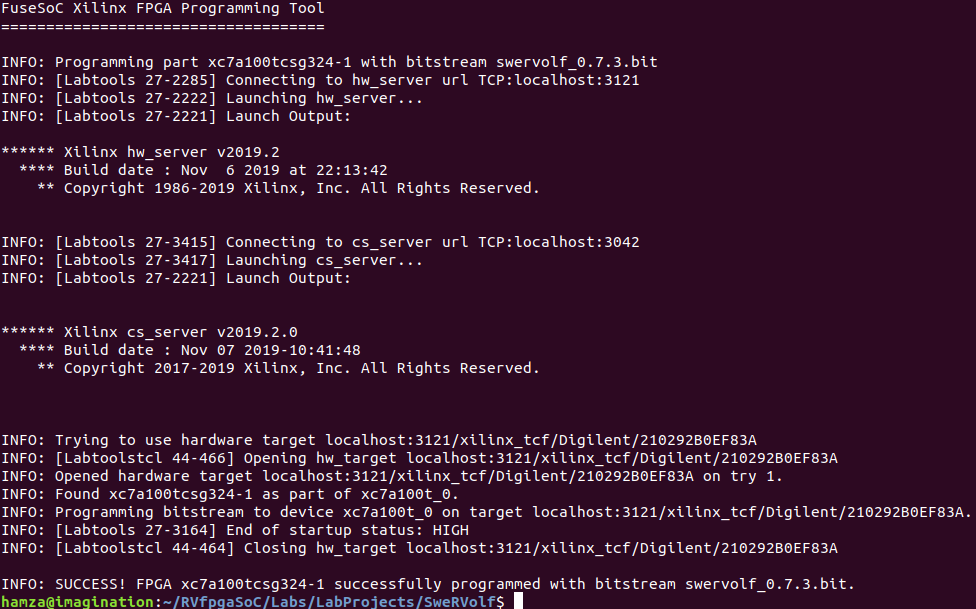
The executable file can be loaded into SweRVolf with a debugger, and the binary file can be converted to a .hex file and loaded into RAM for simulations, as described in the next section.

# Running Zephyr Application on Hardware:

To run the applications on the Nexys A7 board, we need to load the programs using OpenOCD:

**Step 1**. Connect the Nexys A7 board to your computer and then run the FPGA build command in the Workspace directory.

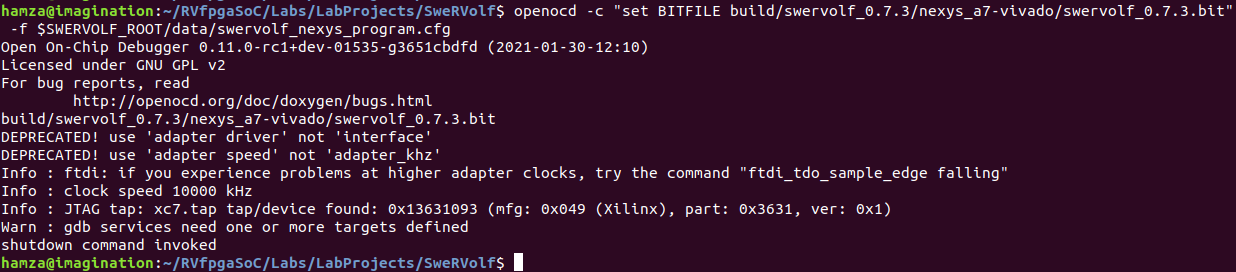
**$** fusesoc run --target=nexys\_a7 --run swervolf



**Figure 21. Run FPGA Build**

**Step 2.** Program the board with OpenOCD.

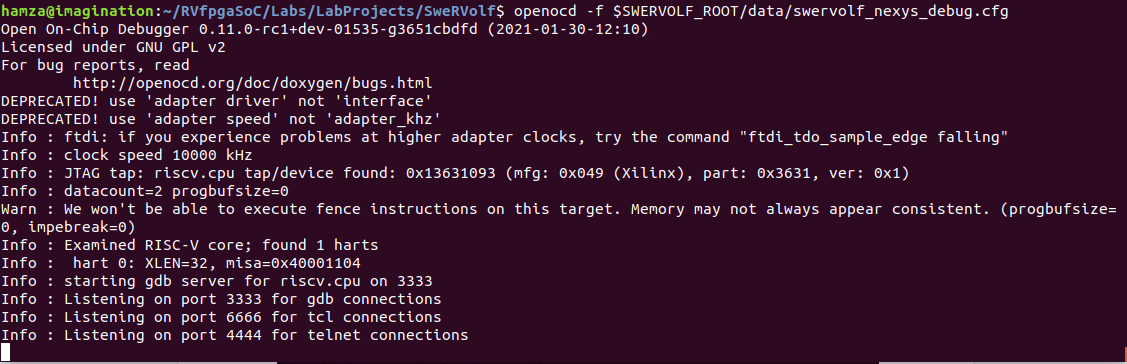
**$** openocd -c "set BITFILE build/swervolf\_0.7.3/nexys\_a7-vivado/swervolf\_0.7.3.bit" -f $SWERVOLF\_ROOT/data/swervolf\_nexys\_program.cfg



**Figure 22. Run OpenOCD**

**Step 3.** Connect OpenOCD with SweRVolf.

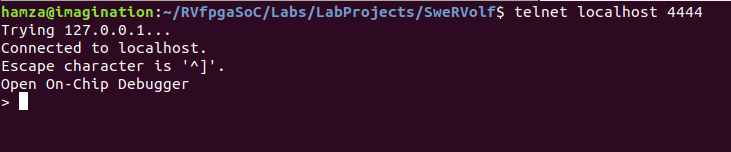
**$** openocd -f $SWERVOLF\_ROOT/data/swervolf\_nexys\_debug.cfg



**Figure 23. OpenOCD Connected**

**Step 3**. Open a third terminal using “Ctrl + Shift + t” & connect to the debug session through OpenOCD using the following command:

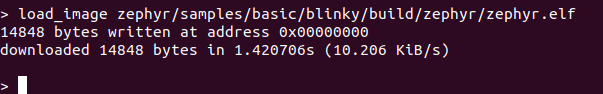
**$** telnet localhost 4444



**Figure 24. telnet**

OpenOCD supports loading ELF program files by running *load\_image /path/to/file.elf*. Remember that the path is relative to the directory from where OpenOCD was launched.

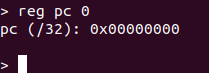
> load\_image zephyr/samples/basic/blinky/build/zephyr/zephyr.elf

****

**Figure 25. load image .elf file**

After the program has been loaded, set the program counter to address zero using the following command:

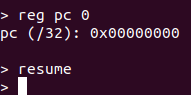
**>** reg pc 0

****

**Figure 26. Set program counter to zero**

Now start the program using this command:

**>** resume

****

**Figure 27. Start the program**

Now you will see the right-most LED of the Nexys A7 board will start blinking.



**Figure 28.LED Blinking**

# Zephyr Application Development Overview:

Zephyr’s build system is based on CMake. The build system is application-centric and requires Zephyr-based applications to initiate building the kernel source tree. The application build controls the configuration and builds a process of both the application and Zephyr itself, compiling them into a single binary.

Zephyr’s base directory hosts Zephyr’s source code, its kernel configuration options, and its build definitions.

The files in the application directory link Zephyr with the application. This directory contains all application-specific files, such as configuration options and source code.

An application in its simplest form has the content listed here and described below:

/App

├── CMakeLists.txt

├── prj.conf

└── src

└── main.c

**CMakeLists.txt**: This file tells the build system where to find the other application files and links the application directory with Zephyr’s CMake build system. This link provides features supported by Zephyr’s build system, such as board-specific kernel configuration files, the ability to run and debug compiled binaries on real or emulated hardware, and more.

**Kernel configuration files**: An application typically provides a Kconfig configuration file (usually called prj.conf) that specifies application-specific values for one or more kernel configuration options. These application settings are merged with board-specific settings to produce a kernel configuration.

**Application source code files**: An application typically provides one or more application-specific files written in C or assembly language. These files are usually located in a subdirectory called **src**.

# Creating a New Zephyr Application:

Follow these steps to create a new application directory.

**Step 1**. Change to the Samples directory:

**$** cd zephyr/samples

**Step 2**. Create a new directory for your application:

**$** mkdir my\_first\_app



**Figure 29. Make project directory**

**Step 3**. It is recommended to place all application source code in a subdirectory named src.

This makes it easier to distinguish between project files and source files:

**$** cd my\_first\_app

**$** mkdir src



**Figure 30. Make src directory inside the project directory**

**Step 4**. Enter the src directory and then create the application’s main source file, “main.c”.

**$** cd src

**$** nano main.c



**Figure 31. create “main.c” file**

Nano Editor will open up in your ubuntu terminal as shown in the figure below :



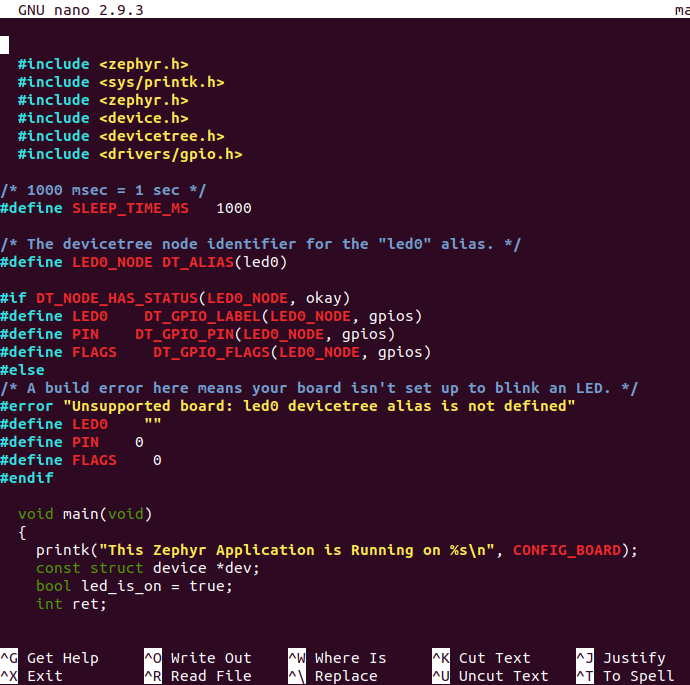
**Figure 32. GNU nano Editor**

**Step 5**. Copy the following code in the nano editor. This code is the mixture of both the “hello\_world” and “blinky” examples source code.

|  |
| --- |
| #include <zephyr.h>  #include <sys/printk.h>  #include <zephyr.h>  #include <device.h>  #include <devicetree.h>  #include <drivers/gpio.h>  /\* 1000 msec = 1 sec \*/  #define SLEEP\_TIME\_MS 1000  /\* The devicetree node identifier for the "led0" alias. \*/  #define LED0\_NODE DT\_ALIAS(led0)  #if DT\_NODE\_HAS\_STATUS(LED0\_NODE, okay)  #define LED0 DT\_GPIO\_LABEL(LED0\_NODE, gpios)  #define PIN DT\_GPIO\_PIN(LED0\_NODE, gpios)  #define FLAGS DT\_GPIO\_FLAGS(LED0\_NODE, gpios)  #else  /\* A build error here means your board isn't set up to blink an LED. \*/  #error "Unsupported board: led0 devicetree alias is not defined"  #define LED0 ""  #define PIN 0  #define FLAGS 0  #endif  void main(void)  {  printk("This Zephyr Application is Running on %s\n", CONFIG\_BOARD);  const struct device \*dev;  bool led\_is\_on = true;  int ret;  dev = device\_get\_binding(LED0);  if (dev == NULL) {  return;  }  ret = gpio\_pin\_configure(dev, PIN, GPIO\_OUTPUT\_ACTIVE | FLAGS);  if (ret < 0) {  return;  }  while (1) {  gpio\_pin\_set(dev, PIN, (int)led\_is\_on);  led\_is\_on = !led\_is\_on;  k\_msleep(SLEEP\_TIME\_MS);  }  } |

**Figure 33. “main.c” code**

After you are finished writing the code, press “**Ctrl + x**” to exit.



**Figure 34. main.c file code**

Then it will ask you if you want to save the file, and you have to press “**y**” for Yes.

****

**Figure 35. save main.c file**

Press “**Enter**” to save the file with the name “main.c”.

****

**Figure 36. confirm the name main.c**

**Step 6**. Now we need to navigate out of the src directory and then create the “**CMakeLists.txt**” and “**prj.conf**” files :

**$** cd ..

**$** nano CMakeLists.txt



**Figure 37. Create CMakeLists.txt**

Copy the following code in the nano editor:

|  |
| --- |
| cmake\_minimum\_required(VERSION 3.13.1)  find\_package(Zephyr REQUIRED HINTS $ENV{ZEPHYR\_BASE})  project(my\_first\_app)  target\_sources(app PRIVATE src/main.c) |

**Figure 38. “CMakeLists.txt” file code**

Now perform the same steps that you have done to save the “main.c” file.



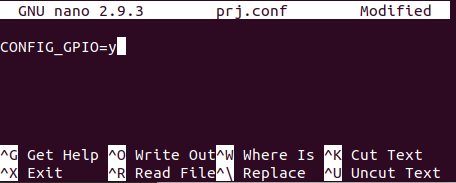
**Figure 39. nano editor**

Now create the project configuration file.

****

**Figure 40. create a project configuration file**

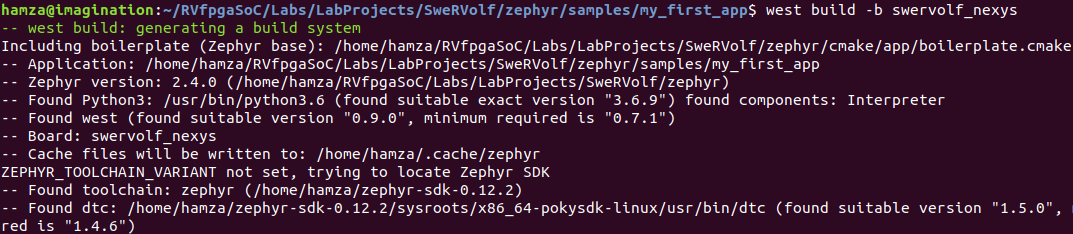
Application configuration options are set in prj.conf in the application directory. Since we are using the LED in our example so we set the “CONFIG\_GPIO” parameter is yes.

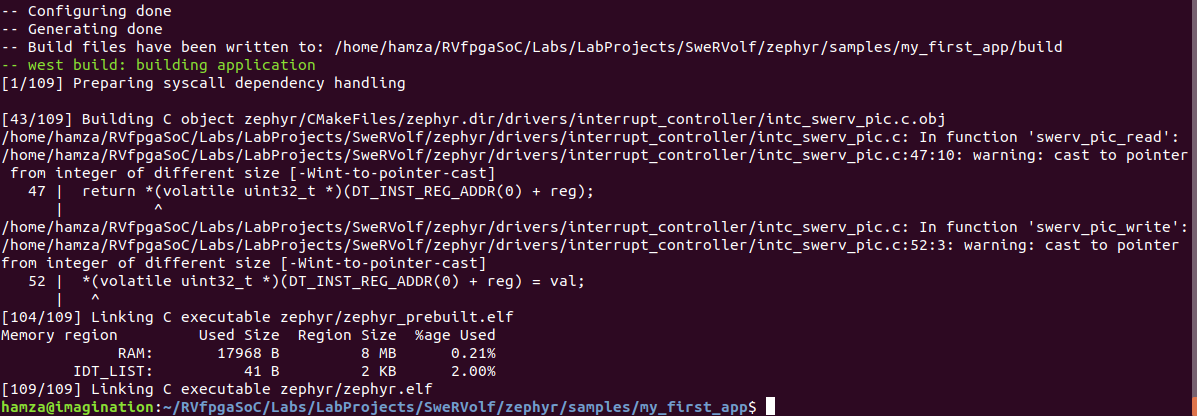
****

**Figure 41. “prj.conf” nano editor**

**Step 7**. Build the code for “**my\_first\_app**”:

**$** west build -b swervolf\_nexys

****

****

**Figure 42. “my\_first\_app” build**

**Step 8.**  Convert the .bin file to .hex file :

To create the .hex file run the following command from the my\_first\_app directory :

**$** python3 $SWERVOLF\_ROOT/sw/makehex.py build/zephyr/zephyr.bin > /home/{YourUsername}/RVfpgaSoC/Labs/LabProjects/SweRVolf/zephyr/samples/my\_first\_app/App.hex

****

**Figure 43. Convert “.bin” file to “.hex”**

**Step 9.** Navigate to the WORKSPACE directory:

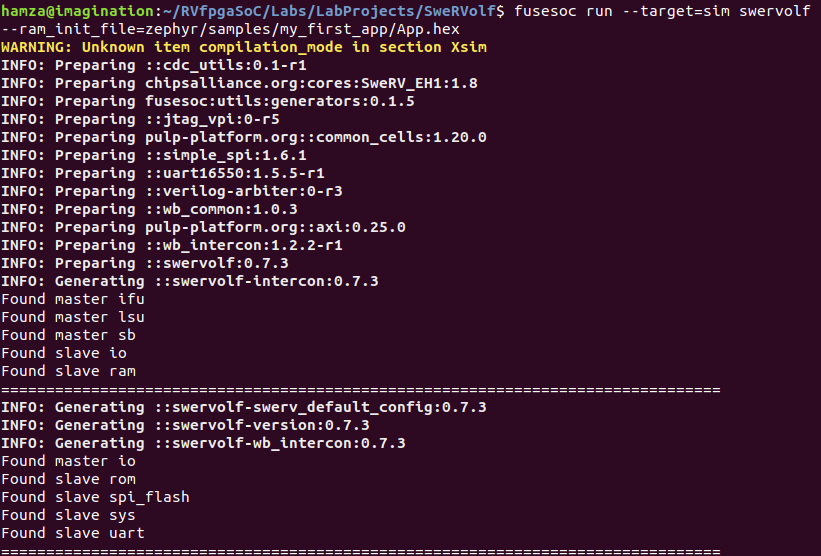
**$** cd $WORKSPACE

****

**Figure 44. Navigate to the Workspace directory**

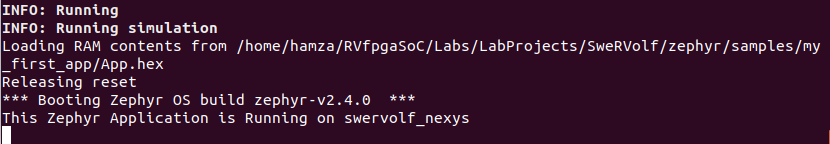
**Step 10.** Load the .hex file into the simulator:

**$** fusesoc run --target=sim swervolf --ram\_init\_file=zephyr/samples/my\_first\_app/App.hex

****

**Figure 45 Load the .hex file into the simulator**

Here we can see our Application is successfully running on Zephyr.

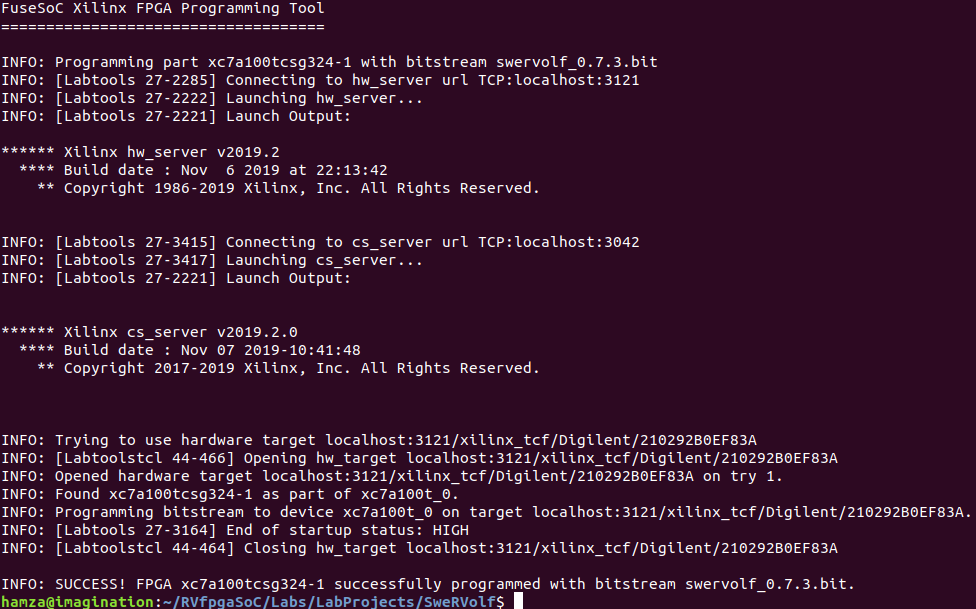
****

**Figure 46. “my\_first\_app” output**

Now we will run the program on the Nexys A7 board.

**Step 1**. Connect the Nexys A7 board to your computer and then run the FPGA build command in the Workspace directory.

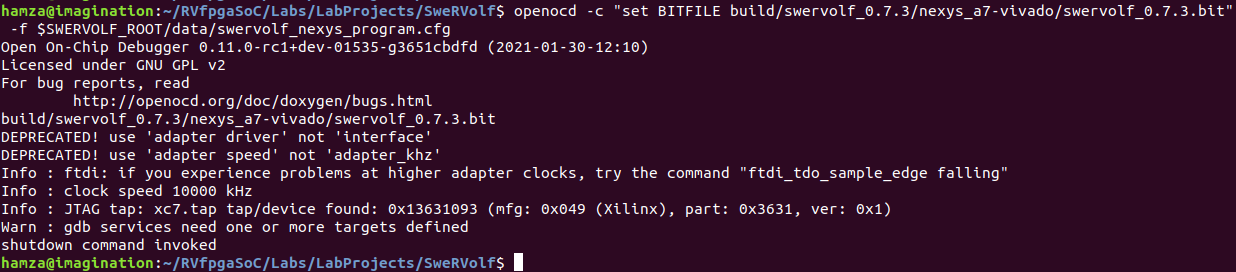
**$** fusesoc run --target=nexys\_a7 --run swervolf



**Figure 21. Run FPGA Build**

**Step 2.** Program the board with OpenOCD.

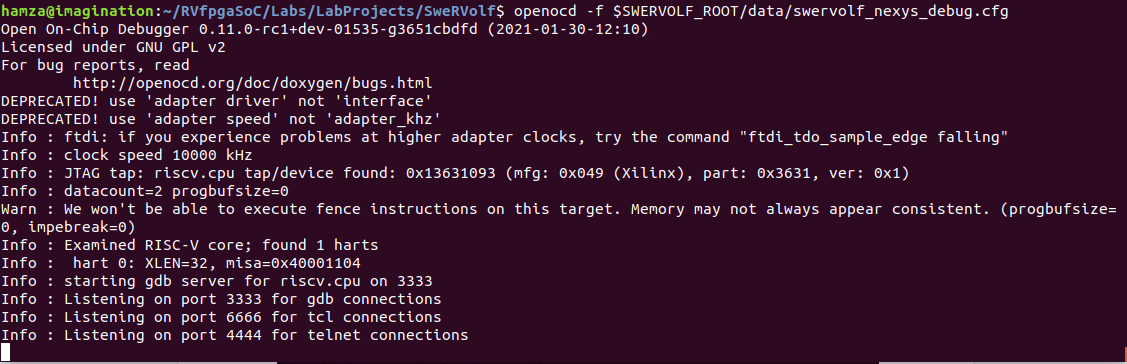
**$** openocd -c "set BITFILE build/swervolf\_0.7.3/nexys\_a7-vivado/swervolf\_0.7.3.bit" -f $SWERVOLF\_ROOT/data/swervolf\_nexys\_program.cfg



**Figure 22. Run OpenOCD**

**Step 3.** Connect OpenOCD with SweRVolf.

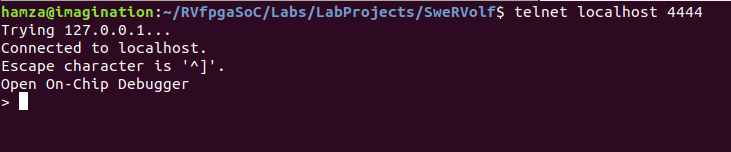
**$** openocd -f $SWERVOLF\_ROOT/data/swervolf\_nexys\_debug.cfg



**Figure 23. OpenOCD Connected**

**Step 3**. Open a third terminal using “Ctrl + Shift + t” & connect to the debug session through OpenOCD using the following command:

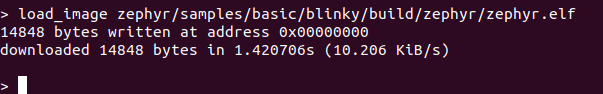
**$** telnet localhost 4444



**Figure 24. telnet**

OpenOCD supports loading ELF program files by running *load\_image /path/to/file.elf*. Remember that the path is relative to the directory from where OpenOCD was launched.

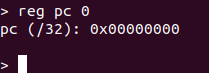
> load\_image zephyr/samples/basic/blinky/build/zephyr/zephyr.elf

****

**Figure 25. load image .elf file**

After the program has been loaded, set the program counter to address zero using the following command:

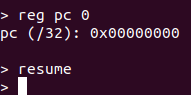
**>** reg pc 0

****

**Figure 26. Set program counter to zero**

Now start the program using this command:

**>** resume

****

**Figure 27. Start the program**