THE IMAGINATION UNIVERSITY PROGRAMME

RVfpga-SoC Lab 4

Running Zephyr on SweRVolf

Table 1. RVfpga Terms

Name	Description
Courses	
RVfpga	A course that shows how to use RVfpgaNexys and RVfpgaSim, RISC-V system-on-chips (SoCs), to run programs and extend the system by adding peripherals (RVfpga Labs 1-10), and explore the core and memory system by running simulations, measuring performance, adding instructions, and modifying the memory system (RVfpga Labs 11-20). Throughout the course, users are also shown how to use the RISC-V toolchain (compilers and debuggers) and simulators, the Verilator HDL simulator, and Western Digital's Whisper instruction set simulator (ISS).
RVfpga-SoC	A course that shows how to build a subset SweRVolfX SoC from scratch using building blocks such as the SweRV core, memories, and peripherals. The course also shows how to load the Zephyr real-time operating system (RTOS) onto SweRVolf and run programs including Tensorflow Lite's hello world example on top of the operating system.
Cores and SoCs	
SweRV EH1 Core	Open-source commercial RISC-V core developed by Western Digital (https://github.com/chipsalliance/Cores-SweRV).
SweRV EH1 Core Complex	SweRV EH1 core with added memory (ICCM, DCCM, and instruction cache), programmable interrupt controller (PIC), bus interfaces, and debug unit (https://github.com/chipsalliance/Cores-SweRV).
SweRVolfX	The System on Chip that we use in the RVfpga course. It is an extension of SweRVolf. **SweRVolf** (https://github.com/chipsalliance/Cores-SweRVolf): An open-source SoC built around the SweRV EH1 Core Complex. It adds a boot ROM, UART interface, system controller, interconnect (AXI Interconnect, Wishbone Interconnect, and AXI-to-Wishbone bridge), and an SPI controller. **SweRVolfX**: It adds four new peripherals to SweRVolf: a GPIO, a PTC, an additional SPI, and a controller for the 8 Digit 7-Segment Displays.
RVfpgaNexys	The SweRVolfX SoC targeted to the Nexys A7 board and its peripherals. It adds a DDR2 interface, CDC (clock domain crossing) unit, BSCAN logic (for the JTAG interface), and clock generator. RVfpgaNexys is the same as SweRVolf Nexys (https://github.com/chipsalliance/Cores-SweRVolf), except that the latter is based on SweRVolf.
RVfpgaSim	The SweRVolfX SoC with a testbench wrapper and AXI memory intended for simulation. RVfpgaSim is the same as SweRVolf Sim, (https://github.com/chipsalliance/Cores-SweRVolf), except that the latter is based on SweRVolf.

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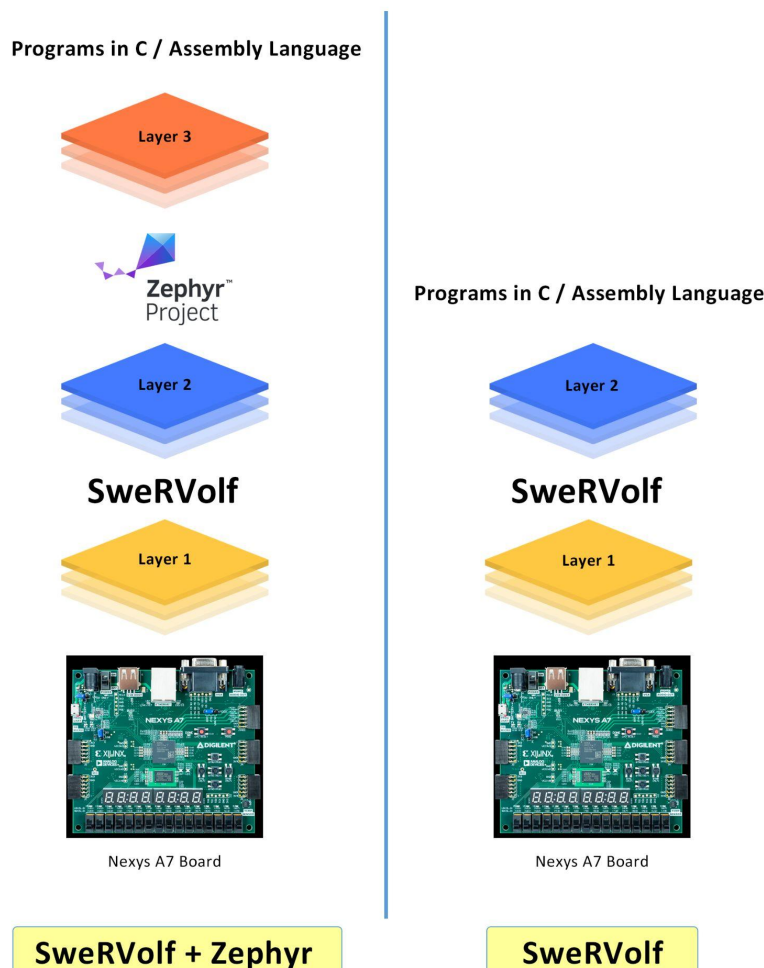
Figure 1. Layers on the top of FPGA Boards

In this Lab, we will describe the Zephyr RTOS, build and run the Zephyr RTOS on SweRVolf, and build & run Zephyr applications.

2. 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.08))
- FuseSoC   (Refer to Installation Guide (Page No.08))
- OpenOCD (RISC-V-specific version)   (Refer to Installation Guide (Page No.09))
- Zephyr Prerequisites   (Refer to Installation Guide (Page No.09))
- Zephyr SDK (v0.12.4)   (Refer to Installation Guide (Page No.10))
- PuTTY   (Refer to Installation Guide (Page No.11))

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.

---

3. 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.

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.

4. 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 SweRVolf onto the FPGA board**
First, we download the SweRVolf, the RISC-V system targeted to an FPGA, to the Nexys A7 FPGA board. We download the SweRVolf 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 SweRVolf**
The second step is to build RISC-V programs and then download them onto SweRVolf.

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

Step 1. **Download SweRVolf onto the FPGA board**
Same as above.

Step 2. Build Zephyr
In this step, build an application for Zephyr. The process of building an application also builds the underlying Zephyr RTOS. The output is an elf file.

Step 3. Load programs on SweRVolf.
In this step, we load the elf file generated during Step 2 onto SweRVolf.

The side-by-side Illustration of both modes of running a program is shown in Figure 1 above.

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

5. Adding Zephyr Support In SweRVolf

In this section of the lab, we show how to add Zephyr to your WORKSPACE.
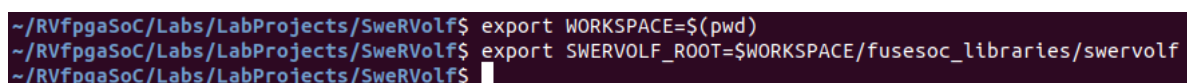
Open your Ubuntu terminal and complete the following steps:

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`

You can also enter the "`printenv <variable-name>`" command in the terminal window to verify if the shell variables have been successfully set or not.


```
~/RVfpgaSoC/Labs/LabProjects/SweRVolf$ export WORKSPACE=$(pwd)
~/RVfpgaSoC/Labs/LabProjects/SweRVolf$ export SWERVOLF_ROOT=$WORKSPACE/fusesoc_libraries/swervolf
~/RVfpgaSoC/Labs/LabProjects/SweRVolf$
```

Figure 2. 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 3. 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 4. west config

> ➢ `west update`

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

Figure 5. west update

The Workspace will now look like this:

```
$WORKSPACE
├──fusesoc_libraries
│    ├──...
│    └──swervolf
├──...
└──zephyr
```

6. 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 6 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 6. 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 7. 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 8. 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/`**<Username>**`/RVfpgaSoC/Labs/LabProjects/SweRVolf/zephyr/sa`
`mples/hello_world/App.hex`

```
hamza@imagination:~/RVfpgaSoC/Labs/LabProjects/SweRVolf/zephyr/samples/hello_world$ python3 $SWERVOLF_ROOT/sw/makehex.py
build/zephyr/zephyr.bin  > /home/hamza/RVfpgaSoC/Labs/LabProjects/SweRVolf/zephyr/samples/hello_world/App.hex
hamza@imagination:~/RVfpgaSoC/Labs/LabProjects/SweRVolf/zephyr/samples/hello_world$
```

Figure 9. hello_world hex file created

Step 4. Navigate to the WORKSPACE directory:

➤ `cd $WORKSPACE`

```
~/RVfpgaSoC/Labs/LabProjects/SweRVolf/zephyr/samples/hello_world$ cd $WORKSPACE
~/RVfpgaSoC/Labs/LabProjects/SweRVolf$
```

Figure 10. 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`

```
hamza@imagination:~/RVfpgaSoC/Labs/LabProjects/SweRVolf$ fusesoc run --target=sim swervolf
--ram_init_file=zephyr/samples/hello_world/App.hex
WARNING: Unknown item compilation_mode in section Xsim
INFO: Preparing ::cdc_utils:0.1-r1
INFO: Preparing chipsalliance.org:cores:SweRV_EH1:1.8
INFO: Preparing fusesoc:utils:generators:0.1.5
INFO: Preparing ::jtag_vpi:0-r5
INFO: Preparing pulp-platform.org::common_cells:1.20.0
INFO: Preparing ::simple_spi:1.6.1
INFO: Preparing ::uart16550:1.5.5-r1
INFO: Preparing ::verilog-arbiter:0-r3
INFO: Preparing ::wb_common:1.0.3
INFO: Preparing pulp-platform.org::axi:0.25.0
INFO: Preparing ::wb_intercon:1.2.2-r1
INFO: Preparing ::swervolf:0.7.3
INFO: Generating ::swervolf-intercon:0.7.3
Found master ifu
Found master lsu
Found master sb
Found slave io
Found slave ram
=============================================================================
INFO: Generating ::swervolf-swerv_default_config:0.7.3
INFO: Generating ::swervolf-version:0.7.3
INFO: Generating ::swervolf-wb_intercon:0.7.3
Found master io
Found slave rom
Found slave spi_flash
Found slave sys
Found slave uart
=============================================================================
INFO: Setting up project
```

Figure 11. fusesoc run

The terminal will show the following output (see Figure 12).

```
make[1]: Leaving directory '/home/hamza/SweRVolf/build/swervolf_0.7.3/sim-verilator'
INFO: Running
INFO: Running simulation
Loading RAM contents from /home/hamza/SweRVolf/zephyr/samples/hello_world/App.hex
Releasing reset
*** Booting Zephyr OS build zephyr-v2.4.0  ***
Hello World! swervolf_nexys
```

Figure 12. hello_world example output

Press "**ctrl + c**" to stop the program.

2. 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`

```
hamza@hamza-lenovo:~/SweRVolf$ cd zephyr/samples/philosophers/
hamza@hamza-lenovo:~/SweRVolf/zephyr/samples/philosophers$ ▮
```

Figure 13. Navigate to philosophers directory

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

➢ `west build -b swervolf_nexys`

Figure 14. 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 philosophers directory :

> ➢ ```python3 $SWERVOLF_ROOT/sw/makehex.py build/zephyr/zephyr.bin
> >
> /home/{YourUsername}/RVfpgaSoC/Labs/LabProjects/SweRVolf/zephyr
> /samples/philosophers/App.hex```

Figure 15. Create philosophers hex file

Step 4. Navigate to the WORKSPACE directory:

> ➢ ```cd $WORKSPACE```

```
~/RVfpgaSoC/Labs/LabProjects/SweRVolf/zephyr/samples/philosophers$ cd $WORKSPACE
~/RVfpgaSoC/Labs/LabProjects/SweRVolf$
```

Figure 16. main directory

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

➢ `fusesoc run --target=sim swervolf`
  `--ram_init_file=zephyr/samples/philosophers/App.hex`

```
hamza@imagination:~/RVfpgaSoC/Labs/LabProjects/SweRVolf$ fusesoc run --target=sim swervolf
--ram_init_file=zephyr/samples/philosophers/App.hex
WARNING: Unknown item compilation_mode in section Xsim
INFO: Preparing ::cdc_utils:0.1-r1
INFO: Preparing chipsalliance.org:cores:SweRV_EH1:1.8
INFO: Preparing fusesoc:utils:generators:0.1.5
INFO: Preparing ::jtag_vpi:0-r5
INFO: Preparing pulp-platform.org::common_cells:1.20.0
INFO: Preparing ::simple_spi:1.6.1
INFO: Preparing ::uart16550:1.5.5-r1
INFO: Preparing ::verilog-arbiter:0-r3
INFO: Preparing ::wb_common:1.0.3
INFO: Preparing pulp-platform.org::axi:0.25.0
INFO: Preparing ::wb_intercon:1.2.2-r1
INFO: Preparing ::swervolf:0.7.3
INFO: Generating ::swervolf-intercon:0.7.3
Found master ifu
Found master lsu
Found master sb
Found slave io
Found slave ram
=============================================================================
INFO: Generating ::swervolf-swerv_default_config:0.7.3
INFO: Generating ::swervolf-version:0.7.3
INFO: Generating ::swervolf-wb_intercon:0.7.3
Found master io
Found slave rom
Found slave spi_flash
Found slave sys
Found slave uart
=============================================================================
INFO: Setting up project
```

Figure 17. fusesoc run

Now you will see the following output:

Figure 18. Zephyr philosophers Output

7. Building Zephyr Application for Hardware

Now we show how to build programs for the SwerVolf 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 19. 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.

8. 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 turn it on, then run the FPGA build command in the Workspace directory.

➢ `fusesoc run --target=nexys_a7 --run swervolf`

Figure 20. 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 21. Run OpenOCD

Step 3. Connect OpenOCD with SweRVolf.

➢ `openocd -f $SWERVOLF_ROOT/data/swervolf_nexys_debug.cfg`

Figure 22. OpenOCD Connected

Step 4. Open a third terminal using "ctrl + shift + t" and connect to the debug session through OpenOCD using the following command:

➢ `telnet localhost 4444`

Figure 23. 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 24. 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 25. Set program counter to zero

Now start the program using this command:

➢ `resume`

Figure 26. Start the program

Now you will see the right-most LED of the Nexys A7 board will start blinking.
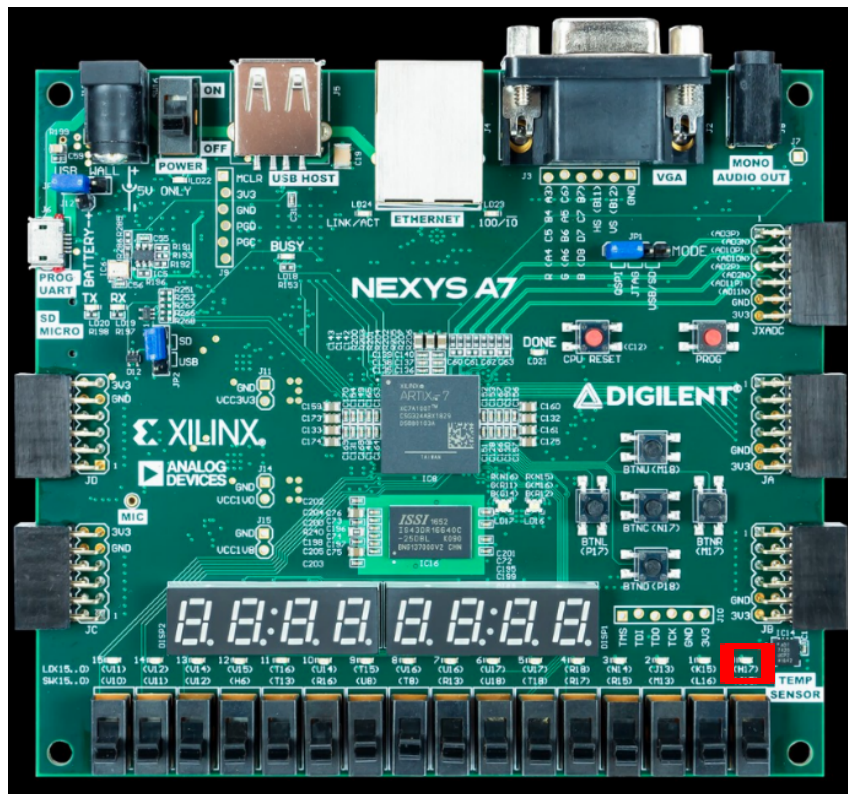
Figure 27.LED Blinking

Now you can press "ctrl + c" to exit out of the program.

9. 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**.

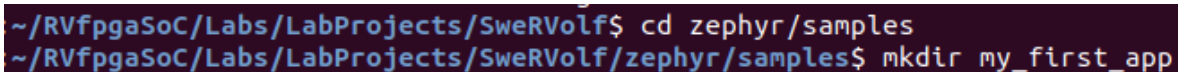## 10. Creating a New Zephyr Application

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:
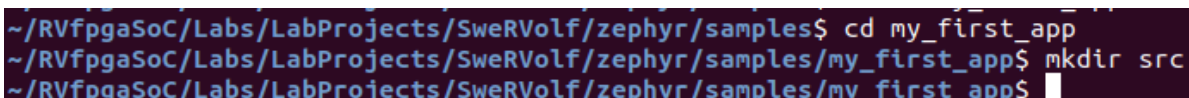
➢ `mkdir my_first_app`

```
~/RVfpgaSoC/Labs/LabProjects/SweRVolf$ cd zephyr/samples
~/RVfpgaSoC/Labs/LabProjects/SweRVolf/zephyr/samples$ mkdir my_first_app
```

Figure 28. 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`

```
~/RVfpgaSoC/Labs/LabProjects/SweRVolf/zephyr/samples$ cd my_first_app
~/RVfpgaSoC/Labs/LabProjects/SweRVolf/zephyr/samples/my_first_app$ mkdir src
~/RVfpgaSoC/Labs/LabProjects/SweRVolf/zephyr/samples/my_first_app$ 
```

Figure 29. 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 30. create "main.c" file

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

Figure 31. GNU nano Editor

Step 5. Copy the following code in the nano editor. This code is the mixture of both the "hello_world" and the "blinky" example source code.

```c
  #include <zephyr.h>
  #include <sys/printk.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)
```

```
{
    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);
        printk("This Zephyr Application is Running on %s\n", CONFIG_BOARD);
    }
}
```

Figure 32. "main.c" code

After you are finished writing the code, press "**ctrl + x**" to exit.

```c
  GNU nano 2.9.3

  #include <zephyr.h>
  #include <sys/printk.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)
  {

    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);
        printk("This Zephyr Application is Running on %s\n", CONFIG_BOARD);
    }
  }

^G Get Help    ^O Write Out    ^W Where Is    ^K Cut Text     ^J Justify
^X Exit        ^R Read File    ^\ Replace     ^U Uncut Text   ^T To Spell
```

Figure 33. 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 34. save main.c file

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

Figure 35. 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`

```
~/SweRVolf/zephyr/samples/my_first_app/src$ cd ..
~/SweRVolf/zephyr/samples/my_first_app$ nano CMakeLists.txt
```

Figure 36. Create CMakeLists.txt

Copy the following code to 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 37. "CMakeLists.txt" file code

Now perform the same steps that you have done in order to save the "main.c" file.

```
  GNU nano 2.9.3                 CMakeLists.txt                 Modified

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)




















^G Get Help    ^O Write Out   ^W Where Is    ^K Cut Text    ^J Justify
^X Exit        ^R Read File   ^\ Replace     ^U Uncut Text  ^T To Spell
```

Figure 38. nano editor

Now create the project configuration file.
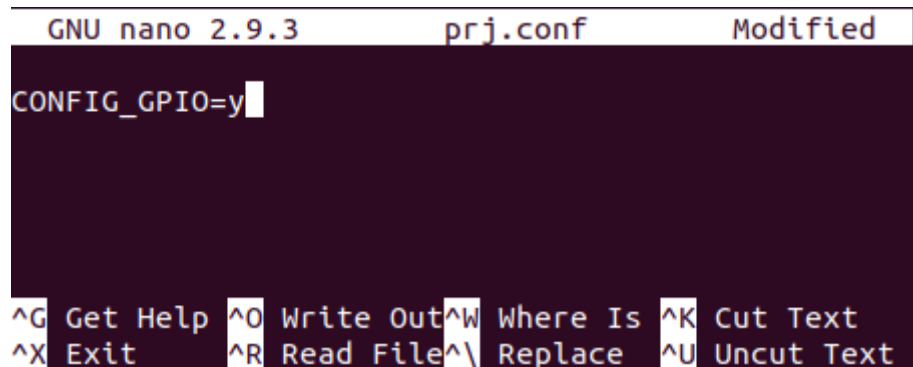
```
~/SweRVolf/zephyr/samples/my_first_app$ nano prj.conf
~/SweRVolf/zephyr/samples/my_first_app$
```

Figure 39. create a project configuration file

Application configuration options are set in prj.conf in the application directory. Since we are using an LED in our source code, we have to set the "CONFIG_GPIO" parameter as yes.
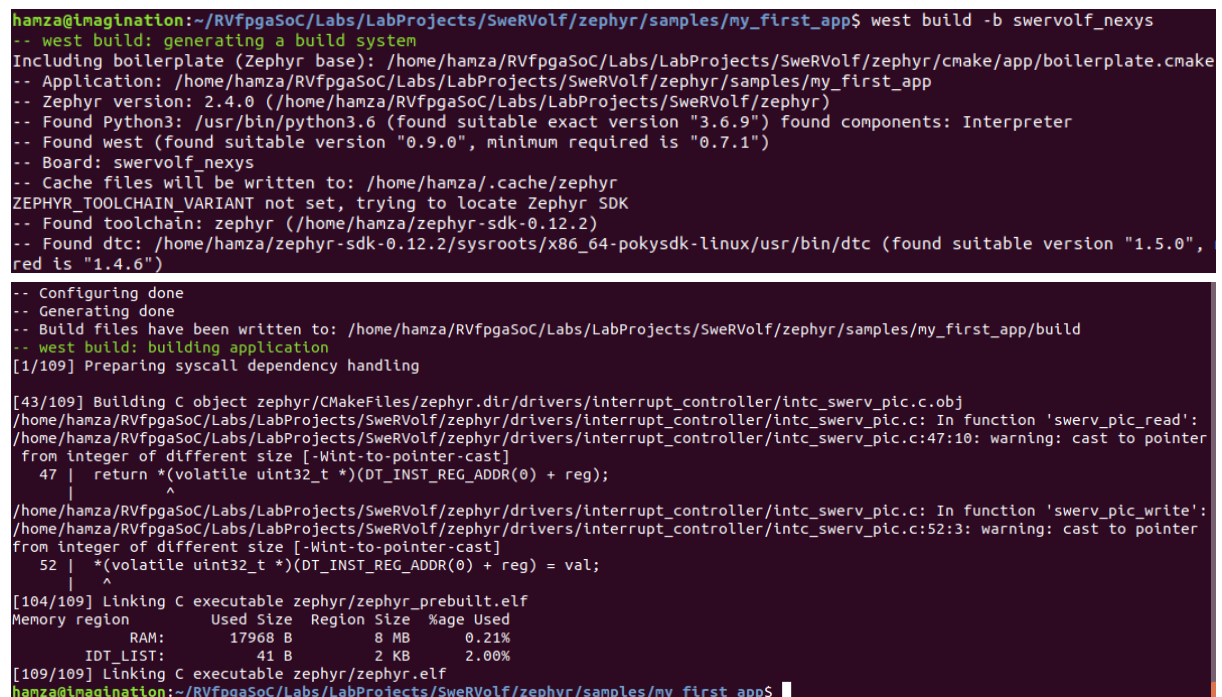
```
CONFIG_GPIO=y
```

Figure 40. "prj.conf" code

Figure 41. "prj.conf" nano editor

Now save the "prj.conf" file.

Step 7. Build the code for "**my_first_app**":

➤ `west build -b swervolf_nexys`

Figure 42. "my_first_app" build

The binaries have been generated successfully. Now we will run the "my_first_app" program on the Nexys A7 board.

Step 9. Navigate to the WORKSPACE directory:

> ➢ `cd $WORKSPACE`

```
~/RVfpgaSoC/Labs/LabProjects/SweRVolf/zephyr/samples/my_first_app$ cd $WORKSPACE
~/RVfpgaSoC/Labs/LabProjects/SweRVolf$
```

Figure 43. Navigate to the Workspace directory

Step 10. 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`

```
****** Xilinx cs_server v2019.2.0
 **** Build date : Nov 07 2019-10:41:48
  ** Copyright 2017-2019 Xilinx, Inc. All Rights Reserved.


INFO: Trying to use hardware target localhost:3121/xilinx_tcf/Digilent/210292B0EF83A
INFO: [Labtoolstcl 44-466] Opening hw_target localhost:3121/xilinx_tcf/Digilent/210292B0EF83A
INFO: Opened hardware target localhost:3121/xilinx_tcf/Digilent/210292B0EF83A on try 1.
INFO: Found xc7a100tcsg324-1 as part of xc7a100t_0.
INFO: Programming bitstream to device xc7a100t_0 on target localhost:3121/xilinx_tcf/Digilent/210292B0EF83A.
INFO: [Labtools 27-3164] End of startup status: HIGH
INFO: [Labtoolstcl 44-464] Closing hw_target localhost:3121/xilinx_tcf/Digilent/210292B0EF83A

INFO: SUCCESS! FPGA xc7a100tcsg324-1 successfully programmed with bitstream swervolf_0.7.3.bit.
hamza@imagination:~/RVfpgaSoC/Labs/LabProjects/SweRVolf$
```

Figure 44. Run FPGA Build

Step 11. 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`

```
hamza@imagination:~/RVfpgaSoC/Labs/LabProjects/SweRVolf$ openocd -c "set BITFILE build/swervolf_0.7.3/nexys_a7-vivado/swervolf_0.7.3.bit"
 -f $SWERVOLF_ROOT/data/swervolf_nexys_program.cfg
Open On-Chip Debugger 0.11.0-rc1+dev-01535-g3651cbdfd (2021-01-30-12:10)
Licensed under GNU GPL v2
For bug reports, read
        http://openocd.org/doc/doxygen/bugs.html
build/swervolf_0.7.3/nexys_a7-vivado/swervolf_0.7.3.bit
DEPRECATED! use 'adapter driver' not 'interface'
DEPRECATED! use 'adapter speed' not 'adapter_khz'
Info : ftdi: if you experience problems at higher adapter clocks, try the command "ftdi_tdo_sample_edge falling"
Info : clock speed 10000 kHz
Info : JTAG tap: xc7.tap tap/device found: 0x13631093 (mfg: 0x049 (Xilinx), part: 0x3631, ver: 0x1)
Warn : gdb services need one or more targets defined
shutdown command invoked
hamza@imagination:~/RVfpgaSoC/Labs/LabProjects/SweRVolf$
```

Figure 45. Run OpenOCD

Step 12. Connect OpenOCD with SweRVolf.

> ➢ `openocd -f $SWERVOLF_ROOT/data/swervolf_nexys_debug.cfg`

Figure 46. OpenOCD Connected

Step 13. Open a third terminal using "Ctrl + Shift + t" & connect to the debug session through OpenOCD using the following command:

➤ `telnet localhost 4444`

Figure 47. 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/my_first_app/build/zephyr/zephyr.elf`

Figure 48. 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 49. Set program counter to zero
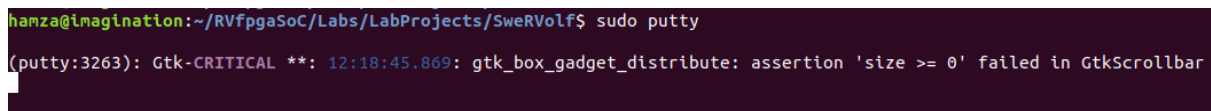
Now start the program using this command:

➢ `resume`

Figure 50. Start the program

The LED on the board will start blinking.

Step 14. Open a new terminal tab using "Ctrl + Shift + t" and open PuTTY using the following command:

➢ `sudo putty`

Figure 51. open PuTTY

We will be using PuTTY here as a serial console for our Nexys A7 board.

Step 15. Set the following configuration:
Select the connection type as "**serial**", then enter "**/dev/ttyUSB1**" as the serial line, and set the speed equal to "**115200**". Now click "Open" to start the serial console.
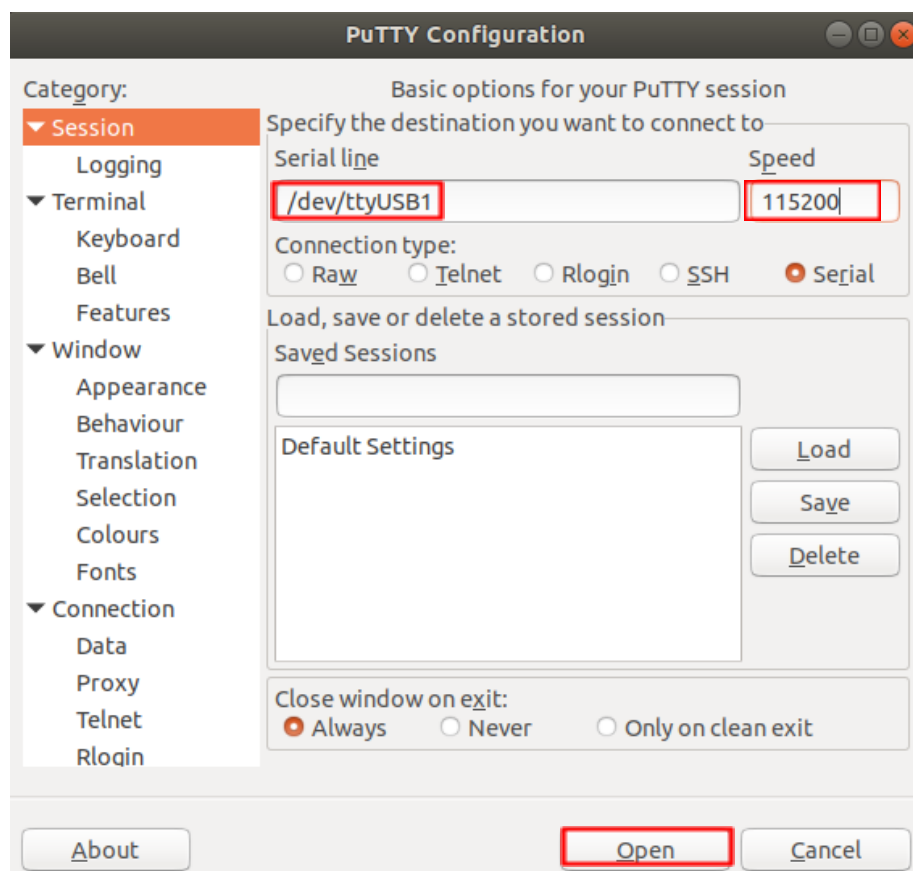

Figure 52. PuTTY Configuration

In the serial console, we can see the output of our program "my_first_app" (see Figure 53).
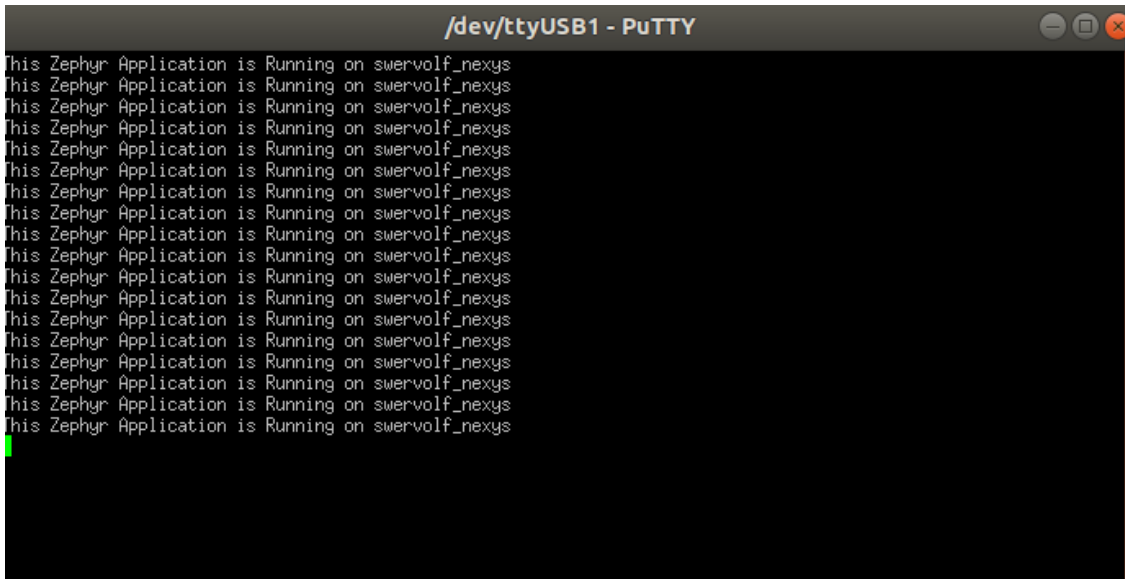

Figure 53. serial console output

Note: If you are unable to open a serial console, try "/dev/ttyUSB0" as the serial line.

As we can see in the output text on the serial console, this zephyr application is running on SweRVolf Nexys.