Microchip Lab Manual 24020 FRM6

Getting Started with ZephyrOS

MASTERs 24

EMPOWERING INNOVATION

Lab Manual for *Getting Started with ZephyrOS*

Contents

Introduction:

ZephyrOS is an open source RTOS targeted towards embedded systems that includes community support for many Microchip development boards. This class will introduce an engineer to the coding environment, SDK, and debug tools available within the ZephyrOS Ecosystem. Using hands-on examples, the engineer will gain experience with useful OS primitives and tasks, explore the hardware's Device Tree, build and deploy to target hardware.

Upon completion, you will:

- Learn how to install and navigate a ZephyrOS Project
- Gain an understanding of the basic ZephyrOS commands and mechanics
- Create a new ZephyrOS project structure, ready for a clean application build
- Use Zephyr's west tool to build and debug project code for a given target
- Become familiar with the usage of built in kernel API's for message queues, semaphores, tasks
- Gain insight into the ZephyrOS Devicetree system including sources and overlays

Prerequisites:

The lab material assumes you have prior experience with:

- Any Embedded RTOS (Real Time Operating System)
- C language programming

Conventions

- Commands that are run on the host machine are entered as shown below:

```
PS C:\> command -a -b -cdefgh

Command report
```

- Code snippets are enclosed in a box as shown below. Areas highlighted in YELLOW represent changes to existing code:

```
#include <stdio.h>

int main (void)
{
    printf("hello world \r\n");
    return 0;

}
```

INFO	Relevant information about a specific topic Select info style for text

WARNING	Important information Select warning style for text

Hardware Requirements:

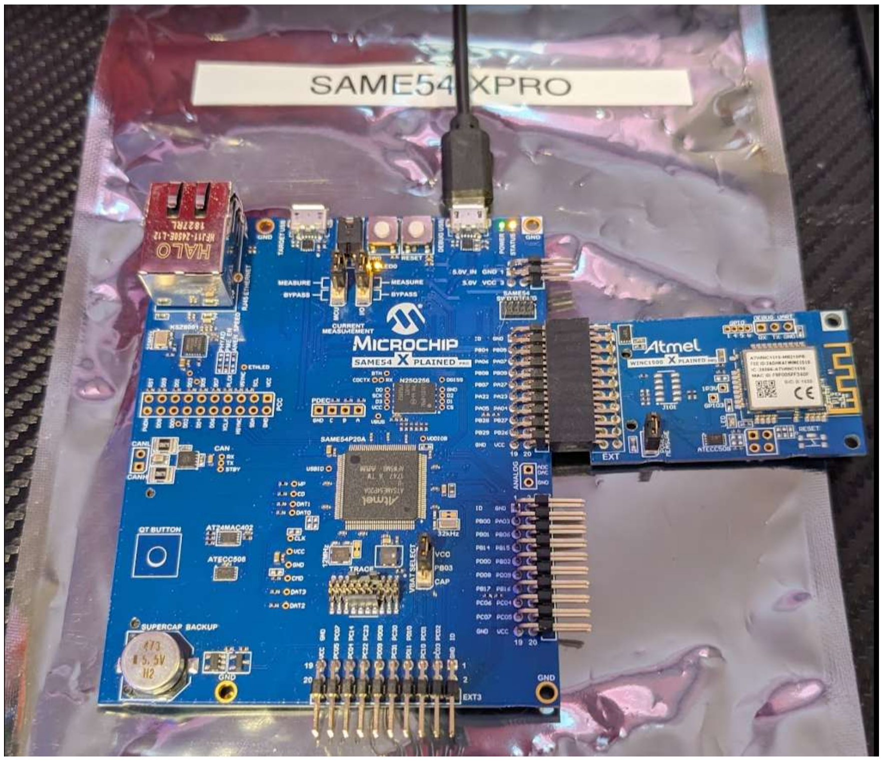
Here is the list of hardware needed to complete the lab:

The SAME54 XPLAINED PRO

1x Micro-USB cables

ATWINC1500-XPRO Extension Board

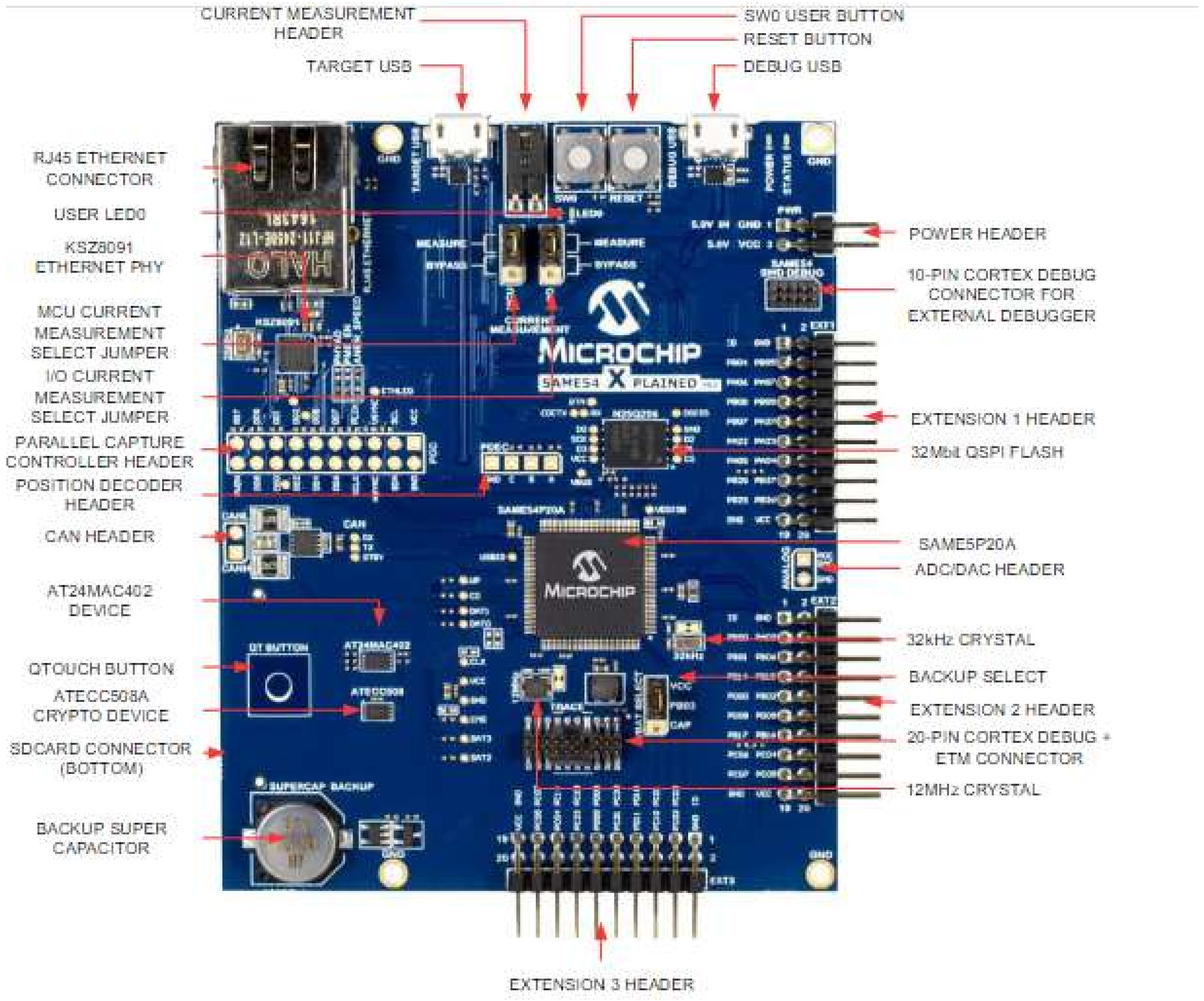
The picture below shows the main connectors and jumpers of the SAME54 XPLAINED PRO with a connected WINC1500 XPRO extension Board.



The SAM E54 Xplained Pro evaluation kit is a hardware platform for evaluating the ATSAME54P20A microcontroller (MCU). Supported by the Studio integrated development platform, the kit provides easy access to the features of the ATSAME54P20A and explains how to integrate the device into a custom design. This board is supported by MPLAB Integrated tools development environment and MPLAB Harmony

The ATWINC1500-XPRO is an extension board to the Xplained Pro evaluation platform. The ATWINC1500-XPRO extension board allows you to evaluate the ATWINC1500 low cost, low power 802.11 b/g/n Wi-Fi network controller module. Supported by the Atmel Studio integrated development platform, the kit provides easy access to the features of the ATWINC1500 and explains how to integrate the device in a custom design.

The picture below shows the main connectors and jumpers of the SAME54 XPLAINED PRO board.



More Information is available:

https://www.microchip.com/en-us/development-tool/atsame54-xpro

Software Requirements:

Here is a list of the software needed for this lab

- Windows 10/11
- Visual Studio Code
    - Serial Monitor Extension for VSCode - Install from VSCode Extensions Tab: https://marketplace.visualstudio.com/items?itemName=ms-vscode.vscode-serial-monitor
- Zephyr SDK including toolchain and host tool
- Windows Powershell will be used for all shell commands

All needed software will be pre-installed for this lab. For future reference, the Zephyr SDK can be installed on Windows, Linux, or MacOS using the instructions found here:

https://docs.zephyrproject.org/latest/develop/getting_started/index.html

The labs in their completed form can be found at:

https://github.com/dtang-microchip/ZephyrGettingStartedLab

Navigate the different branches “lab1”, “lab2”, “lab3” for the final code

Lab 1 - Build and Deploy a Sample Application, then Create a New Project Tree

Purpose:

In this lab, the student will build and deploy their first sample application, then copy the files from this sample application into their own project directory, making a skeleton ZephyrOS project that will be used for the rest of these labs.

Overview:

This lab will begin by opening VSCode and a terminal pane. You will then initiate your Python Virtual Environment (VENV) and use this terminal to issue "west" commands to build and flash code to your SAME54 XPro device. You will also open a second terminal window and use SerialMonitor to view serial output from your device

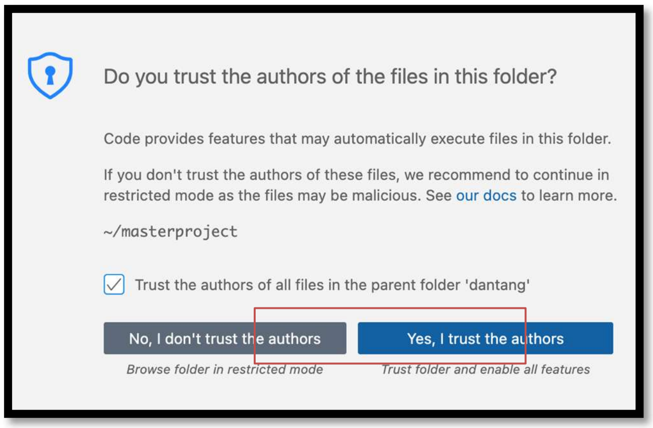
Procedure:

Step 1: Build a sample application using *west*

1.1: Launch a VSCode window on the Windows host by clicking on the VSCode icon or by changing directory to your project directory and typing:

```
PS C:\Users\[UserID]> cd zephyrproject
PS C:\Users\[UserID]\zephyrproject> code .
```

1.2: Opening VSCode will result in a confirmation window asking you to trust the folder, choose the button labeled "Yes, I trust the authors"


Do you trust the authors of the files in this folder?

Code provides features that may automatically execute files in this folder.

If you don't trust the authors of these files, we recommend to continue in restricted mode as the files may be malicious. See our docs to learn more.

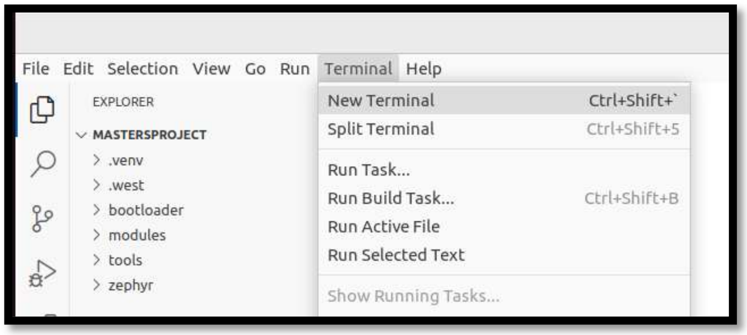
~/masterproject

☑ Trust the authors of all files in the parent folder 'dantang'

No, I don't trust the authors
Browse folder in restricted mode
Yes, I trust the authors
Trust folder and enable all features

1.3: When VSCode is loaded, enable autosave by navigating to "File-> AutoSave"

1.4: Open a terminal pane in VSCode by navigating to "Terminal-> New Terminal"


File Edit Selection View Go Run Terminal Help
EXPLORER
MASTERSPROJECT
.venv
.west
bootloader
modules
tools
zephyr
New Terminal Ctrl+Shift+`
Split Terminal Ctrl+Shift+5
Run Task...
Run Build Task... Ctrl+Shift+B
Run Active File
Run Selected Text
Show Running Tasks...

1.5: The west tool is a python executable that can be installed globally on a system, or locally within a Python Virtual Environment (venv). Our West install is in one such venv. Each time a new terminal window is opened, the environment may be sourced with the following command:

```
PS C:\Users\[UserID]\zephyrproject > .\.venv\Scripts\Activate.ps1
```

When complete, you terminal prompt will include the name of your virtual environment:

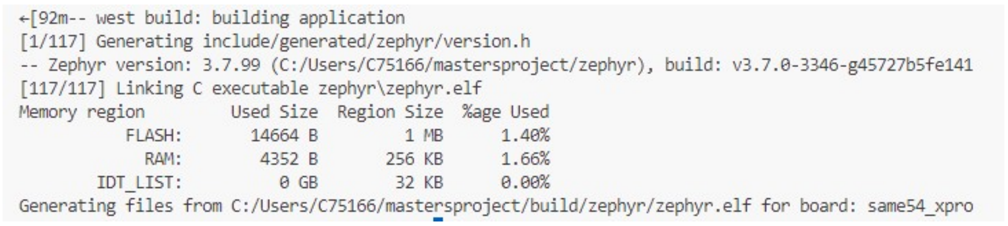
```
(.venv) PS C:\Users\[UserID]\zephyrproject>
```

You can learn more about Python Virtual Environments here:
https://docs.python.org/3/tutorial/venv.html

1.6: From within your terminal (and .venv), build your first sample project targeting the SAME54 XPRO board:

```
(.venv) PS C:\Users\[UserID]\zephyrproject> west build -p always -b
same54_xpro zephyr\samples\basic\blinky
```

Observe the output and success messages from your build, which may resemble the following:



```
←[92m-- west build: building application
[1/117] Generating include/generated/zephyr/version.h
-- Zephyr version: 3.7.99 (C:/Users/C75166/mastersproject/zephyr), build: v3.7.0-3346-g45727b5fe141
[117/117] Linking C executable zephyr\zephyr.elf
Memory region         Used Size  Region Size  %age Used
           FLASH:       14664 B         1 MB      1.40%
             RAM:        4352 B       256 KB      1.66%
        IDT_LIST:          0 GB        32 KB      0.00%
Generating files from C:/Users/C75166/mastersproject/build/zephyr/zephyr.elf for board: same54_xpro
```

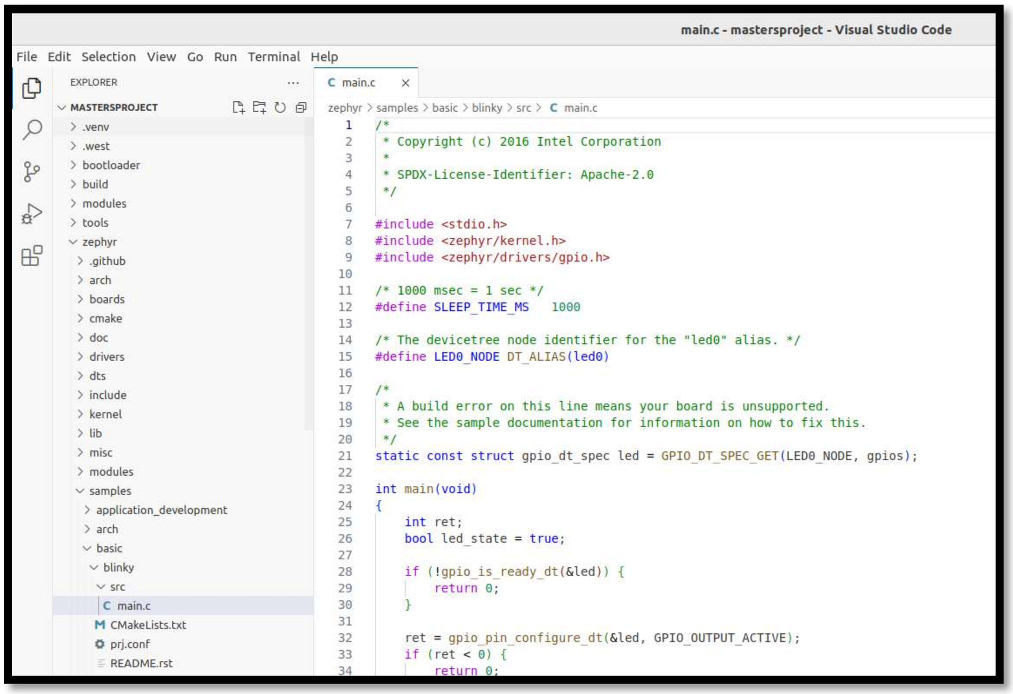
Step 2: Flash the sample application to your target device using *west*

2.1: To flash your SAME54 XPRO with the compiled code, connect your SAME54 XPRO Board with your MicroUSB cable connected to the “Debug USB” port of your target, allow your Operating System to find and mount the device, then type:

```
(.venv) PS C:\Users\[UserID]\zephyrproject> west flash
```

The device is now programmed and LED0 should be flashing at a rate of 1Hz

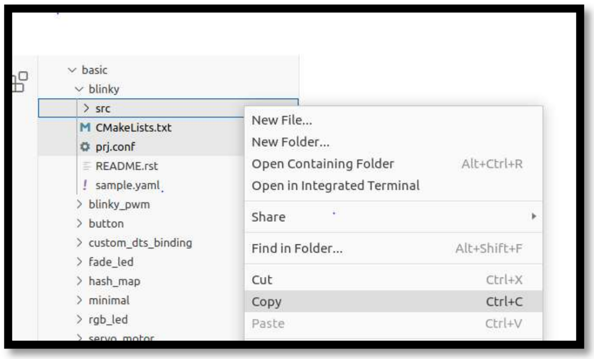
2.2: Observe the code of the sample application by using the VSCode Navigation pane to expand zephyr/samples/basic/blinky/src and opening main.c in the main panel. Many of these code elements will be discussed within the remainder of this course.



```c
/*
 * Copyright (c) 2016 Intel Corporation
 *
 * SPDX-License-Identifier: Apache-2.0
 */

#include <stdio.h>
#include <zephyr/kernel.h>
#include <zephyr/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)

/*
 * A build error on this line means your board is unsupported.
 * See the sample documentation for information on how to fix this.
 */
static const struct gpio_dt_spec led = GPIO_DT_SPEC_GET(LED0_NODE, gpios);

int main(void)
{
    int ret;
    bool led_state = true;

    if (!gpio_is_ready_dt(&led)) {
        return 0;
    }

    ret = gpio_pin_configure_dt(&led, GPIO_OUTPUT_ACTIVE);
    if (ret < 0) {
        return 0;
```

Step 3: Populate a new project tree, then build and deploy your custom project

3.1: It is often easiest to use collateral from a sample project to begin creating your own project. Right-Click/Copy **CMakeLists.txt**, **prj.conf**, and the **src\** folder from this sample project and paste them to the root of your current project.



You can also use your terminal window and run the following commands:

```
(.venv) PS C:\...\zephyrproject> cd ~\zephyrproject\zephyr\samples\basic\blinky
(.venv) PS C:\...\zephyrproject> copy CMakeLists.txt ~\zephyrproject\
(.venv) PS C:\...\zephyrproject> copy prj.conf ~\zephyrproject\
(.venv) PS C:\...\zephyrproject> copy -r src\ ~\zephyrproject\
(.venv) PS C:\...\zephyrproject> cd ~\zephyrproject\
```

3.2: Build this Blinky code in its new location.

```
(.venv) PS C:\...\zephyrproject> west build -p always -b same54_xpro .
```

3.3: Close any existing VSCode tabs for main.c, then reopen main.c from ./mastersproject/src/. Edit the Blink time in main.c to change the sleep time between blinks:

src/main.c Line 12:

```
#define SLEEP_TIME_MS 500
```

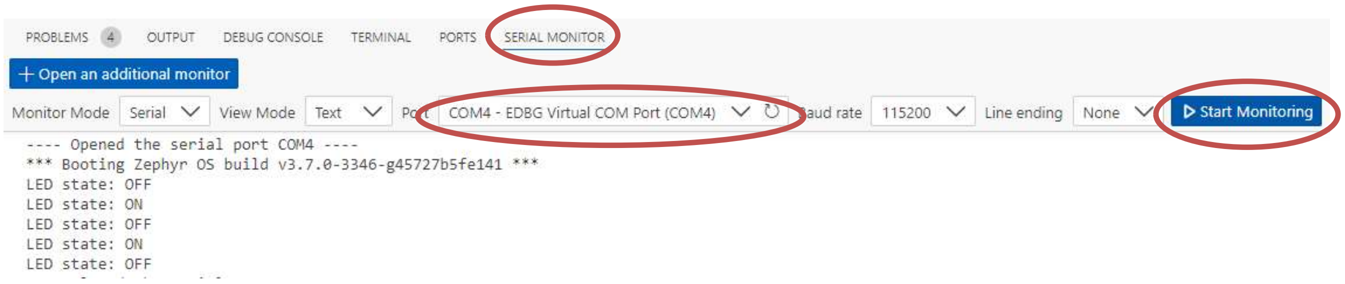
3.4: Rebuild your code with the custom changes included. If you do not need a clean build and your target options haven't changed since your last build, you can simply type:

```
(.venv) PS C:\...\zephyrproject> west build
```

3.5: Flash this newly built code to your SAME54 XPRO device and observe the new blink frequency

```
(.venv) PS C:\...\zephyrproject> west flash
```

3.6: In a **new terminal window** (Terminal->New Terminal), select the "SERIAL MONITOR" tab and connect to the serial port (Start Monitoring) of the device (EDBG Virtual COM Port) to see our print statements from our main() loop.



Monitor Mode Serial · View Mode Text · Port COM4 - EDBG Virtual COM Port (COM4) · Baud rate 115200 · Line ending None

```
---- Opened the serial port COM4 ----
*** Booting Zephyr OS build v3.7.0-3346-g45727b5fe141 ***
LED state: OFF
LED state: ON
LED state: OFF
LED state: ON
LED state: OFF
```

Results:

Congratulations! You have successfully built and flashed your first Zephyr projects!

Summary:

This lab introduced you to VSCode, *west*, the Zephyr sample application repository, and SerialMonitor – these are all of the tools you'll need to use for the rest of these labs

Lab 2 - Create Zephyr tasks and establish thread safe communications between tasks

Purpose:

Create your first tasks in Zephyr, exploring different options for creating tasks and passing in data and communication structures that your tasks might need to operate.

Overview:

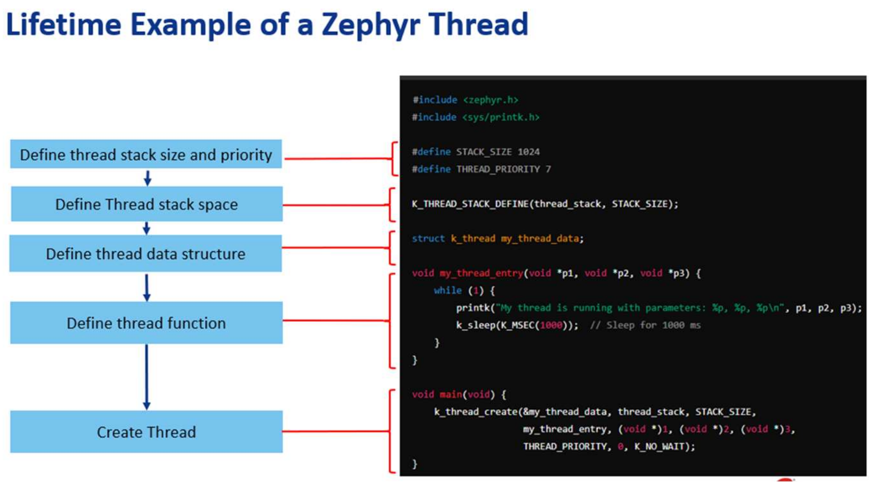
Zephyr allows the engineer to quickly and effectively create a task to be run by the scheduler system. Tasks can be created at compile time or at runtime and assigned priorities to help determine importance to the system.

When creating tasks, it's important to minimize "busy wait" or blocking functions where the scheduler may have difficultly taking control from the task.

We'll also create a message queue that will be used to deliver data from one task to another, and learn how to allow the tasks to access the message queue.

Procedure:

In the following steps, we'll complete the following high level actions in order to run a thread in Zephyr:


Lifetime Example of a Zephyr Thread
Define thread stack size and priority
Define Thread stack space
Define thread data structure
Define thread function
Create Thread

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

#define STACK_SIZE 1024
#define THREAD_PRIORITY 7

K_THREAD_STACK_DEFINE(thread_stack, STACK_SIZE);

struct k_thread my_thread_data;

void my_thread_entry(void *p1, void *p2, void *p3) {
    while (1) {
        printk("My thread is running with parameters: %p, %p, %p\n", p1, p2, p3);
        k_sleep(K_MSEC(1000));  // Sleep for 1000 ms
    }
}

void main(void) {
    k_thread_create(&my_thread_data, thread_stack, STACK_SIZE,
                    my_thread_entry, (void *)1, (void *)2, (void *)3,
                    THREAD_PRIORITY, 0, K_NO_WAIT);
}
```

Along the way in this lab, you may notice some *compiler warnings* when you build. These are due to the interative nature of this lab. By the time Lab 2 is complete all compiler warnings will be cleared!

Step 1: Create a Producer Thread

1.1: In the src\ folder, create a new file called producer.c, adding the following content:

```
#include <zephyr/kernel.h>
#include <zephyr/drivers/gpio.h>
#include "producer.h"

void producerThread(void *inLed, void*, void*) {
      struct gpio_dt_spec *toggleLED = (struct gpio_dt_spec*)inLed;
      bool led_state = true;
      int counter = 0;
      uint32_t data = 0;

      while (1) {
            gpio_pin_toggle_dt(toggleLED);

            led_state = !led_state;
            printf("LED state: %s\n", led_state ? "ON" : "OFF");
            k_msleep(SLEEP_TIME_MS);
      }
}
```

Notice that this function takes three void pointers and returns void. All Zephyr tasks must match this prototype. Void pointers can be cast to a useful type within the function as shown in this example. It's a good idea to create objects at a higher level and pass them in via this technique (called dependency injection) if multiple tasks need to share resources.

Zephyr provides a special delay function k_msleep(); that allows the individual task to wait for the specified timeout (in milliseconds) while allowing the scheduler to service other tasks within the system. If you need a delay, always use this sleep function (or one of its siblings – k_sleep(), k_usleep(), k_yield(), or k_busy_wait() [if you truly need a blocking delay])

1.2: In the src\ folder, create a file called "producer.h" with a define for SLEEP_TIME_MS and a prototype for producerThread():

```
#define SLEEP_TIME_MS     100
void producerThread(void* inLed, void*, void*);
```

1.3: From the top of main.c, remove the #define SLEEP_TIME_MS 500 to prevent a redefine. Further down in main(), delete the instantiation of bool led_state = true; since we moved this part of the application to producer.c

1.4: Tell CMake to compile producer.c and check the src/ folder for include files – open **CMakeLists.txt** and make the following changes at the bottom:

```
target_sources(app PRIVATE src/main.c src/producer.c)
target_include_directories(app PRIVATE src)
```

If you prefer to store your include files elsewhere in the file tree, you are free to do so. Just be sure to update this line in CMakeLists.txt accordingly. In any case, the linker will now be able to find any .h files that are added to this directory.

1.5: In main.c, remove all code from the while(1){} loop. We'll continue to blink the LED as in the old code here, but the blink functionality has moved to producer.c. Replace the deleted code with a simple sleep command:

```
while(1) {
     k_msleep(SLEEP_TIME_MS);
}
```

1.6: In main.c, include producer.h by adding the following line towards the top of the file:

```
#include "producer.h"
```

1.7: Still in main.c, use the following compile time macro to create a stack space for the task by adding the following code before main();

```
/*
 * Zephyr Thread defines
*/

#define STACKSIZE 1024
#define PRIORITY 7
K_THREAD_STACK_DEFINE(producerThreadstack_area, STACKSIZE);
struct k_thread producerThread_data;
```

1.8: Initialize the task in main() right before the while(1){} loop:

```
k_tid_t producer_tid = k_thread_create(&producerThread_data,
                         producerThreadstack_area,
                         K_THREAD_STACK_SIZEOF(producerThreadstack_area),
                         producerThread,
                         (void*)&led, (void*)NULL, (void*)NULL,
                         PRIORITY, 0, K_NO_WAIT);
```

1.9: **Build** and **flash** your new code using west flash. The LED should toggle every 100ms. (10Hz)

You can of course continue to type the full "west build -p always…" command, but it is often quicker to just run "west flash". West is generally smart enough to know when code has changed and recompile as necessary, then flash the resulting compiled code. If you ever find that newly added code doesn't seem to running as intended, try a pristine build again.

Step 2: Create a Consumer Thread

2.1: In the src/ folder, create a new file called consumer.c, adding the following content:

```
#include <zephyr/kernel.h>
#include "consumer.h"

void consumerThread(void*, void*, void*){
      uint32_t data = 0;

      while (1) {
            k_msleep(1000); // 'Magic Number' OK - We'll replace shortly
            printf("Consumer Thread: %d\n", data);
            data++;
      }
}
```

2.2: In the src/ folder, create a filed called "consumer.h" with a prototype for consumerThread():

```
void consumerThread(void*, void*, void*);
```

2.3: In main.c, follow the steps you learned previously to instantiate and run your new task with its own stack space and TID:

2.3.1: Include the header file for your new task at the top of main.c

```
#include "consumer.h"
```

2.3.2: Validate that CMakeLists.txt has a callout for your .c and .h files

```
target_sources(app PRIVATE src/main.c src/producer.c src/consumer.c)
```

2.3.3: In main.c, Create a stack area in memory for your thread (you can use the same STACKSIZE #define that you used for Producer Thread) and Create a new variable to hold your thread data:

```
/*
 * Zephyr Thread defines
*/

#define STACKSIZE 1024
#define PRIORITY 7
K_THREAD_STACK_DEFINE(producerThreadstack_area, STACKSIZE);
struct k_thread producerThread_data;
K_THREAD_STACK_DEFINE(consumerThreadstack_area, STACKSIZE);
struct k_thread consumerThread_data;
```

2.3.4: Call `k_thread_create()` with your new thread's information, being sure to check that you are passing expected void pointers – in this case, (void*)NULL for all three members

```
k_tid_t consumer_tid = k_thread_create(&consumerThread_data,
                          consumerThreadstack_area,
                          K_THREAD_STACK_SIZEOF(consumerThreadstack_area),
                          consumerThread,
                          (void*)NULL, (void*)NULL, (void*)NULL,
                          PRIORITY, 0, K_NO_WAIT);
```

2.3.5: Build and flash your code, examining the output in Serial Monitor. You should see output from both tasks appear in your console.

Step 3: Create a Message Queue to pass data between threads

3.1: In main.c – just before the main() function, declare a new global variable to hold the Message Queue and a buffer that will hold the number of queue items we wish to include (in this case, our buffer will hold a maximum of 4 32-bit queue items):

```
struct k_msgq consumerQueue;
char taskCommsBuffer[4 * sizeof(uint32_t)];

int main(void)
{
……
```

3.2: Initialize the Message Queue in main() before your thread creation by calling k_msgq_init():

```
k_msgq_init(&consumerQueue, taskCommsBuffer, sizeof(uint32_t), 4);

k_tid producer_tid = k_thread_create(……
```

3.3: Update producer.c/h prototypes of producerThread() to take the MessageQueue pointer as its second argument, as so:

```
void producerThread(void *inLed, void* inMsgQueue, void*)
```

3.4: Allow producer.c:producerThread to use this Message Queue by casting the void pointer in producerThread() just below where inLED is cast to
`struct gpio_dt_struct:`

```
void producerThread(void *inLed, void* inMsgQueue, void*) {
        struct gpio_dt_spec *toggleLED = (struct gpio_dt_spec*)inLed;
        struct k_msgq *notifyMsgQueue = (struct k_msgq*)inMsgQueue;
```

3.5: In producer.c, add the following block of code within the while(1){} loop to periodically place data into the message queue:

```
printf("LED state: %s\n", led_state ? "ON" : "OFF");
if(counter++>=10) {
      k_msgq_put(notifyMsgQueue, &data, K_NO_WAIT);
      data++;
      counter = 0;
}

k_msleep(SLEEP_TIME_MS);
```

The data we've chosen to pass in this example isn't particularly useful, but you can easily imagine an ADC collecting data and passing a rolling average value to other threads periodically

3.6: In main.c, update your `k_thread_create()` call to take the `&consumerQueue` pointer (created in step 1) as its ***second*** (void*) argument to match your updated producerThread() prototype.

```
    producer_tid = k_thread_create(&producerThread_data,
                    producerThreadstack_area,
                    K_THREAD_STACK_SIZEOF(producerThreadstack_area),
                    producerThread,
                    (void*)&led, (void*)&consumerQueue, (void*)NULL,
                    PRIORITY, 0, K_NO_WAIT);
```

3.7: Similarly, update consumer.c/h:

3.7.1: Accept (void*)inMsgQueue as its first argument for consumerThread() prototype

```
void consumerThread(void* inMsgQueue, void*, void*)
```

3.7.2: In consumer.c cast the incoming (void*)inMsgQueue pointer for use within the consumerThread function

```
void consumerThread(void* inMsgQueue, void*, void*) {
     struct k_msgq *notifyMsgQueue = (struct k_msgq*)inMsgQueue;
```

3.7.3: In consumer.c replace the k_msleep() command in consumerThread with a command to get a queue item in the message queue:

```
k_msgq_get(notifyMsgQueue, &data, K_FOREVER);
```

K_FOREVER seems like a long time. If your thread needs to wait that long for data, something may have gone wrong. In the real world, you may want to time bound this function and raise an error if the timer expires without a new queue entry showing up as expected.

3.7.4: In consumer.c remove the `data++;` incrementor from consumerThread(), as data is now being collected from the message queue and doesn't need to be internally incremented

3.7.5: In main.c update your `k_thread_create()` call to take your `&consumerQueue` pointer as it's **first** (void*) argument to match your updated consumerThread() prototype

```
    consumer_tid = k_thread_create(&consumerThread_data,
                     consumerThreadstack_area,
                     K_THREAD_STACK_SIZEOF(consumerThreadstack_area),
                     consumerThread,
                     (void*)&consumerQueue, (void*)NULL, (void*)NULL,
                     PRIORITY, 0, K_NO_WAIT);
```

3.8: **Build** and **flash** your code, observing in SerialMonitor that the value of "counter" in producerThread() is now delivered to consumerThread() via the shared message queue, and consumerThread now prints out the value whenever a new queue item is received.

Results:

Congratulations! You have completed Lab 2. Your device is now running two Zephyr OS tasks with a Message Queue to deliver data from one to the other!

Summary:

Learning to manage tasks and inter-task communication are important steps to building your RTOS application. In this Lab you created two tasks and learned to pass data between them. There are many other ways to pass data among tasks including Semaphores, Events, Pipes, and more. Browse the ZephyrOS API docs for more information and usage!

Lab 3 - Add a Shell and Winc1500 to your project

Purpose:

In this lab, you will add a uart shell, learn to interact with your running device over this shell, then add and configure a driver to enable a WINC1500-XPro Extension Header

Overview:

Now we'll learn how to add and configure software and drivers from the ZephyrOS "catalog".

The **shell** is a built in tool that exposes a command line to your embedded device. Using the shell, you can directly query information about the running system, including tasks and stack usage, devices installed, memory usage, and more.

In this lab, we'll enable the shell software for our project and explore some of the options for viewing system status. Next, we'll add a Devicetree entry to enable and configure the WINC1500-Xpro extension header and use the shell to scan for and connect to a wifi Access Point. Once connected, we'll end the lab by sending data to a server using the shell.

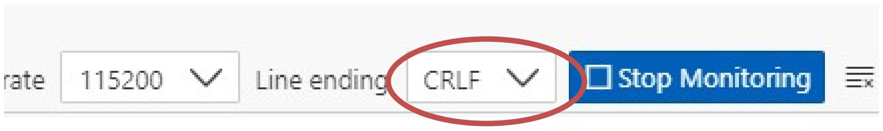
Procedure:

Step 1: Add a Shell to your project and browse options

1.1: Open prj.conf and add `CONFIG_SHELL=y`

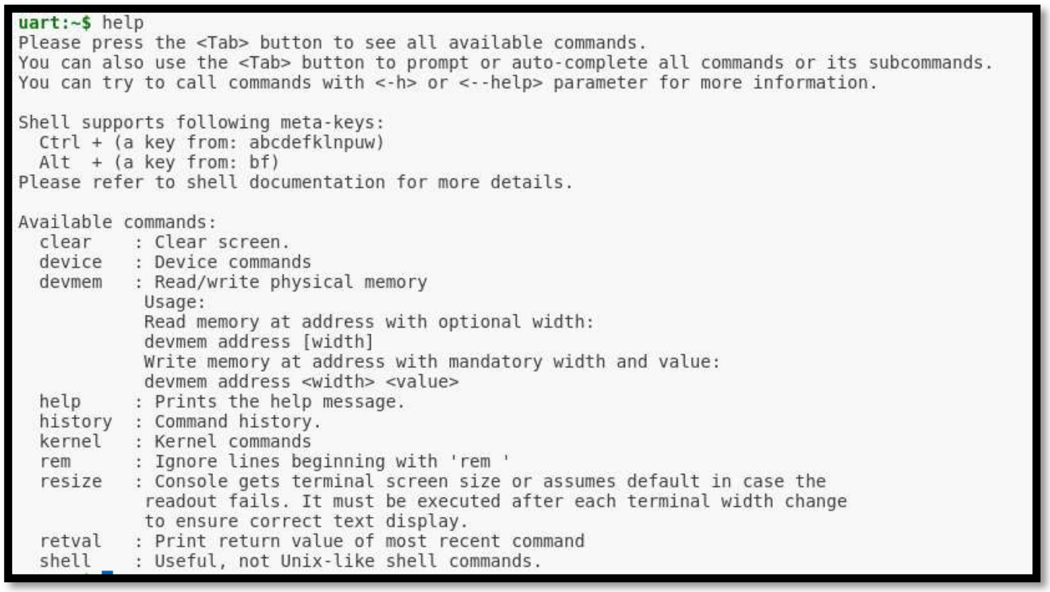
1.2: In producer.c, comment out or delete the `printf("LED state…");` line . in consumer.c, comment out the `printf("Consumer Thread:…);` line, then **Build** and **Flash** your board. This step ensures that we can use the UART console without the terminal being littered with other printf output.

1.3: Using SERIAL MONITOR, note that you now see a `uart:-$` prompt. Add **Line Ending** type CRLF: If you do not see the `uart:-$` prompt, hit enter in the message box to re-load the prompt



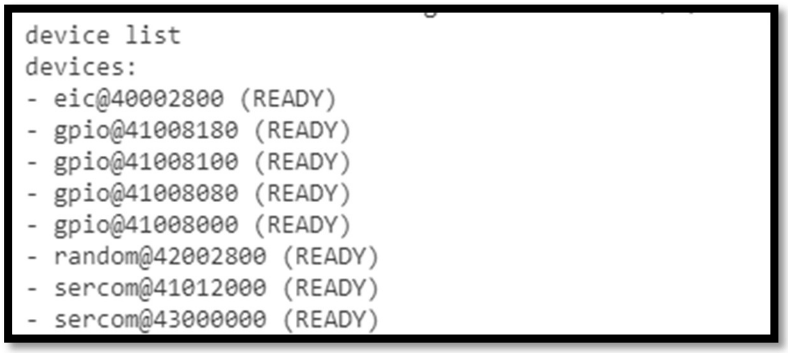
Then type `help` to see available options:

```
uart:~$ help
```



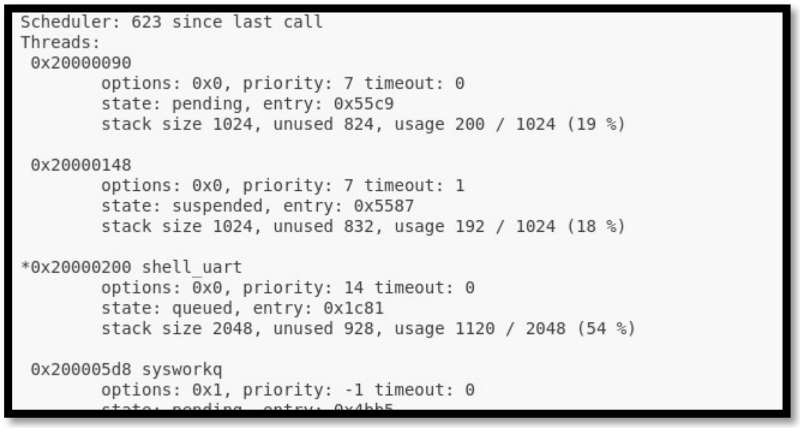
```
uart:~$ help
Please press the <Tab> button to see all available commands.
You can also use the <Tab> button to prompt or auto-complete all commands or its subcommands.
You can try to call commands with <-h> or <--help> parameter for more information.

Shell supports following meta-keys:
  Ctrl + (a key from: abcdefklnpuw)
  Alt  + (a key from: bf)
Please refer to shell documentation for more details.

Available commands:
  clear    : Clear screen.
  device   : Device commands
  devmem   : Read/write physical memory
            Usage:
            Read memory at address with optional width:
            devmem address [width]
            Write memory at address with mandatory width and value:
            devmem address <width> <value>
  help     : Prints the help message.
  history  : Command history.
  kernel   : Kernel commands
  rem      : Ignore lines beginning with 'rem '
  resize   : Console gets terminal screen size or assumes default in case the
            readout fails. It must be executed after each terminal width change
            to ensure correct text display.
  retval   : Print return value of most recent command
  shell    : Useful, not Unix-like shell commands.
```

1.4: Type `uart:~$ device list` to see a list of currently enabled devices.



```
device list
devices:
- eic@40002800 (READY)
- gpio@41008180 (READY)
- gpio@41008100 (READY)
- gpio@41008080 (READY)
- gpio@41008000 (READY)
- random@42002800 (READY)
- sercom@41012000 (READY)
- sercom@43000000 (READY)
```

1.5: Type `uart:~$` `kernel thread list`
to view a list of all running threads. Can you identify the producer and consumer threads you created earlier?

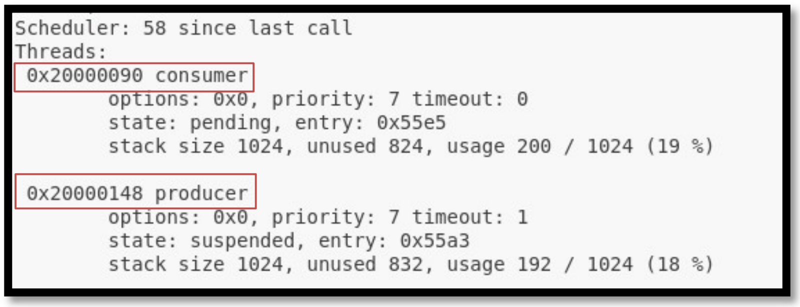


```
Scheduler: 623 since last call
Threads:
 0x20000090
        options: 0x0, priority: 7 timeout: 0
        state: pending, entry: 0x55c9
        stack size 1024, unused 824, usage 200 / 1024 (19 %)

 0x20000148
        options: 0x0, priority: 7 timeout: 1
        state: suspended, entry: 0x5587
        stack size 1024, unused 832, usage 192 / 1024 (18 %)

*0x20000200 shell_uart
        options: 0x0, priority: 14 timeout: 0
        state: queued, entry: 0x1c81
        stack size 2048, unused 928, usage 1120 / 2048 (54 %)

 0x200005d8 sysworkq
        options: 0x1, priority: -1 timeout: 0
        state: pending, entry: 0x4bb5
```

1.6: In main.c, after each thread is created, use the `k_tid_t` for each thread and the `k_thread_name_set()` function to add a friendly name for your threads. After the k_thread_create() calls, add the following:

```
k_thread_name_set(producer_tid, (const char *)"producer");
k_thread_name_set(consumer_tid, (const char *)"consumer");
```

1.7: **Build** and **Flash** your board, then use the previous shell commands to see more easily identify your threads in `kernel thread list`



```
Scheduler: 58 since last call
Threads:
 0x20000090 consumer
        options: 0x0, priority: 7 timeout: 0
        state: pending, entry: 0x55e5
        stack size 1024, unused 824, usage 200 / 1024 (19 %)

 0x20000148 producer
        options: 0x0, priority: 7 timeout: 1
        state: suspended, entry: 0x55a3
        stack size 1024, unused 832, usage 192 / 1024 (18 %)
```

1.8: **CHALLENGE**: Update your thread stack sizes to save memory space without impacting your application

Step 2: Update Device Tree Overlay for WINC1500 XPRO

2.1: Using the WINC1500-XPRO User Guide and the SAME54 XPlained Pro User Guide, find the correct pins to connect the WINC1500-XPRO to the SAME54 XPlained Pro's **EXT1** connector. Helpful charts can be found in Section 4.1.1 of each User Guide. User Guides are available in the code folder provided with this class or you can use the pins in the chart below.

	EXT1	EXT3
Chip Select	PB28	PC14
IRQ	PB7	PC30
Reset Pin	PA6	PC1
Enable Pin	PA7	PC10
Wake Pin	PA27	PC31

2.2: Create a new directory named boards\ and place a new file called same54_xpro.overlay

```
(.venv) PS C:\...\zephyrproject> mkdir boards
(.venv) PS C:\...\zephryproject> ni boards\same54_xpro.overlay
```

2.3 FOR EXT3: Open same54_xpro.overlay and add the following lines, replacing **bold** items with correct information from the User Guides. Use the format found in **cs-gpios** to help update **irq-gpios**, **reset-gpios**, and **enable-gpios**

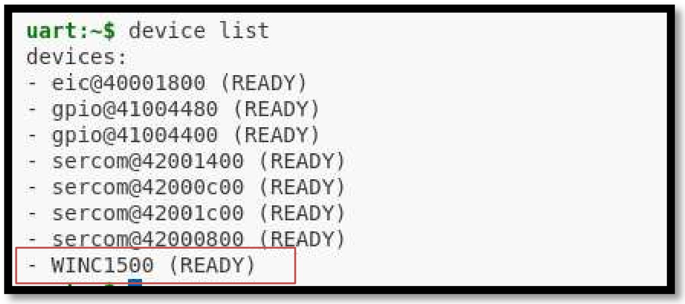
```
/
{
     aliases {
     };
};

&gmac {
    status = "disabled";
};

&pinctrl {
    sercom6_spi_overlay: sercome6_spi_overlay {
        group1 {
            pinmux = <PC5C_SERCOM6_PAD1>,
                     <PC4C_SERCOM6_PAD0>,
                     <PC7C_SERCOM6_PAD3>;
        };
    };
]:

&sercom6 {
     compatible = "atmel,sam0-spi";
     dipo = <3>;
     dopo = <0>;
     #address-cells = <1>;
     #size-cells = <0>;
     cs-gpios = <&portc 14 GPIO_ACTIVE_LOW>;
     pinctrl-0 = <&sercom6_spi_overlay>;
     pinctrl-names = "default";
     status = "okay";

     sercom6_cs0_winc1500: WINC1500@0 {
          compatible = "atmel,winc1500";
          reg = <0>;
          spi-max-frequency = <12000000>;
          irq-gpios = <&portx N GPIO_ACTIVE_LOW>;
          reset-gpios = <&portx N GPIO_ACTIVE_LOW>;
          enable-gpios = <&portx N GPIO_ACTIVE_HIGH>,
                         <&portx N GPIO_ACTIVE_HIGH>;
          status = "okay";
     };
};
```

enable-gpios lists two pins, comma separated. Since the CHIP_EN pin and the WAKE pin largely share functionality for our demo, listing them both allows the software driver to toggle them together when the chip is enabled or disabled. You could also use a Devicetree GPIO node to configure the WAKE pin separately to manage it on your own.

2.4: Open prj.conf (or use KConfig) to add the following configuration items to your project:

```
CONFIG_WIFI=y


CONFIG_WIFI_WINC1500=y
CONFIG_NETWORKING=y
CONFIG_NET_IPV4=y
CONFIG_TEST_RANDOM_GENERATOR=y

CONFIG_NET_PKT_RX_COUNT=16
CONFIG_NET_PKT_TX_COUNT=16
CONFIG_NET_BUF_RX_COUNT=32
CONFIG_NET_BUF_TX_COUNT=16
CONFIG_NET_MAX_CONTEXTS=16

CONFIG_NET_DHCPV4=n
CONFIG_DNS_RESOLVER=n


CONFIG_NET_TX_STACK_SIZE=2048
CONFIG_NET_RX_STACK_SIZE=2048

CONFIG_NET_SHELL=y
CONFIG_NET_L2_WIFI_SHELL=y
```

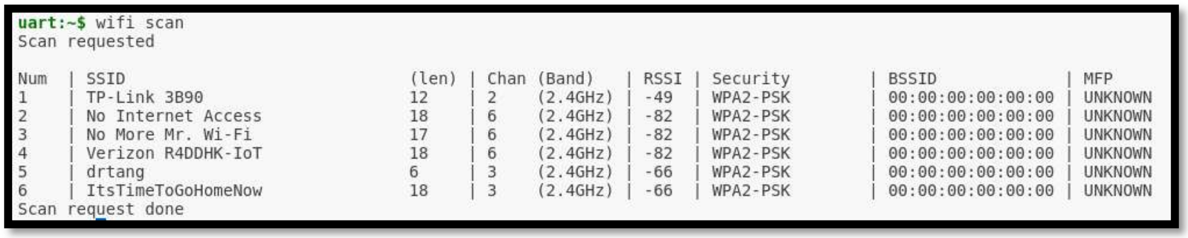
2.5: **Build** and **Flash** your board with a pristine build (west build -p always -b same54_xpro), then use the UART shell provided in SERIAL MONITOR to verify your newly added WINC1500 device is available to the system (with device list):



```
uart:~$ device list
devices:
- eic@40001800 (READY)
- gpio@41004480 (READY)
- gpio@41004400 (READY)
- sercom@42001400 (READY)
- sercom@42000c00 (READY)
- sercom@42001c00 (READY)
- sercom@42000800 (READY)
- WINC1500 (READY)
```

Step 3: Scan for available Wi-Fi and connect to an Access Point

3.1: Using SerialMonitor and your device's UART shell, type `uart:~$` `wifi scan` and wait for the local scan to complete



```
uart:~$ wifi scan
Scan requested

Num  | SSID                             (len) | Chan (Band)    | RSSI | Security        | BSSID             | MFP
1    | TP-Link 3B90                     12    | 2    (2.4GHz)  | -49  | WPA2-PSK        | 00:00:00:00:00:00 | UNKNOWN
2    | No Internet Access               18    | 6    (2.4GHz)  | -82  | WPA2-PSK        | 00:00:00:00:00:00 | UNKNOWN
3    | No More Mr. Wi-Fi                17    | 6    (2.4GHz)  | -82  | WPA2-PSK        | 00:00:00:00:00:00 | UNKNOWN
4    | Verizon R4DDHK-IoT               18    | 6    (2.4GHz)  | -82  | WPA2-PSK        | 00:00:00:00:00:00 | UNKNOWN
5    | drtang                           6     | 3    (2.4GHz)  | -66  | WPA2-PSK        | 00:00:00:00:00:00 | UNKNOWN
6    | ItsTimeToGoHomeNow               18    | 3    (2.4GHz)  | -66  | WPA2-PSK        | 00:00:00:00:00:00 | UNKNOWN
Scan request done
```

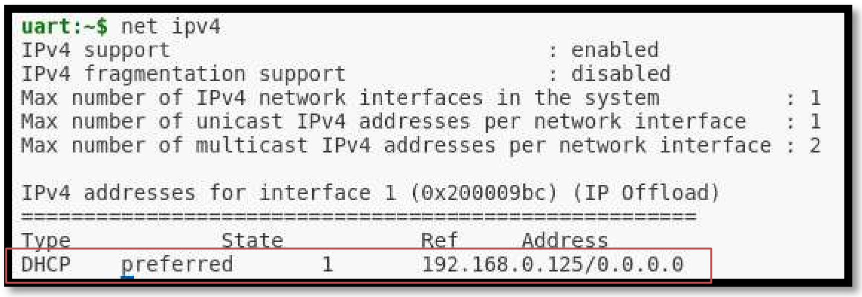
3.2: Verify that the SSID of the router provided by your instructor is shown in the list, then make a connection:

wifi connect -s <SSID> -p <Passphrase> -k <Security type>

```
uart:~$ wifi connect -s "MCHPZephyr" -p "Zephyr4Microchip" -k1
```

3.3: Verify your connection and IP address using the `net` shell module

```
uart:~$ net ipv4
```



```
uart:~$ net ipv4
IPv4 support                                      : enabled
IPv4 fragmentation support                        : disabled
Max number of IPv4 network interfaces in the system          : 1
Max number of unicast IPv4 addresses per network interface   : 1
Max number of multicast IPv4 addresses per network interface : 2

IPv4 addresses for interface 1 (0x200009bc) (IP Offload)
=======================================================
Type            State           Ref     Address
DHCP    preferred       1       192.168.0.125/0.0.0.0
```

3.4: Using the `net` shell module, send a UDP packet to the Presenter's UDP listener server, and confirm that your message appears on the Projector Screen

```
uart:~$ net udp send 192.168.0.100 8085 "Hello from [yourName]"
```

Results:

If all went well, you have added your first Devicetree entry to anable and configure the WINC1500, and used the wifi and net shells to create your first connection to "the cloud"!

Summary:

In this lab, we briefly learned about the usefulness of the shell for project and hardware debugging. Many features within ZephyrOS have a shell for developer interaction, and you can even build your own shell menu's for things like manufactur testing and calibration routines. The built-in shell programs are also a great way to see code for different modules and devices in action – for instance you can view shell_wifi.c and shell_net.c for sample usage of the function calls to scan, connect, and communicate. If you want to build your own shell, check out the docs for the Macro SHELL_CMD_REGISTER();

Appendix A – Host PC setup guide

The Host PC used in this lab was a generic x86_64 architecture type with a native installation of Windows 10.

ZephyrOS can be installed and used on most recent builds of Windows, MacOS, and Linux. In order to re-create this lab on your own host PC you can install Windows 10 and the required dependencies.

Below is a general setup guide for installing the required dependencies for everything we have covered in this lab.

1. Install Windows 10 and setup your user account (username masters).
2. Follow the directions in the Zephyr OS Getting Started Guide found here: https://docs.zephyrproject.org/latest/develop/getting_started/index.html
3. Install OpenOCD and add it to your PATH, then reboot. : https://docs.zephyrproject.org/latest/develop/flash_debug/host-tools.html#openocd-debug-host-tools
    a. (https://www.architectryan.com/2018/03/17/add-to-the-path-on-windows-10/)
4. Install VSCode from https://code.visualstudio.com/download
    a. From the Extensions Marketplace, Install Serial Monitor

Appendix B - Debug your project in VSCode

Purpose:

Explore options for debugging a target within a compiled application.

Overview:

Debugging is an important tool for embedded development. Zephyr and VSCode offer several options for connecting a debugger to the target to see code flow and interrogate variables at different points of your project's operation.

Procedure:

1. These line numbers assume you have just completed Lab1. If you are at another point in your lab work, please either revert to the end of lab 1 or adjust line numbers and variables to match your current code state.

2. Ensure that your latest code is compiled and flashed to your SAME54XPRO board, then type

```
(.venv) C:\...\mastersproject> west debug
```

3. After this command completes the console will leave you in a (gdb) prompt. Depending on your terminal window size, you may need to hit the **Return** key a few times to reach this prompt. There are several ways to interact with gdb (type `help` to learn more) but one useful option is to us the Text User Interface:

```
(gdb) tui enable
```

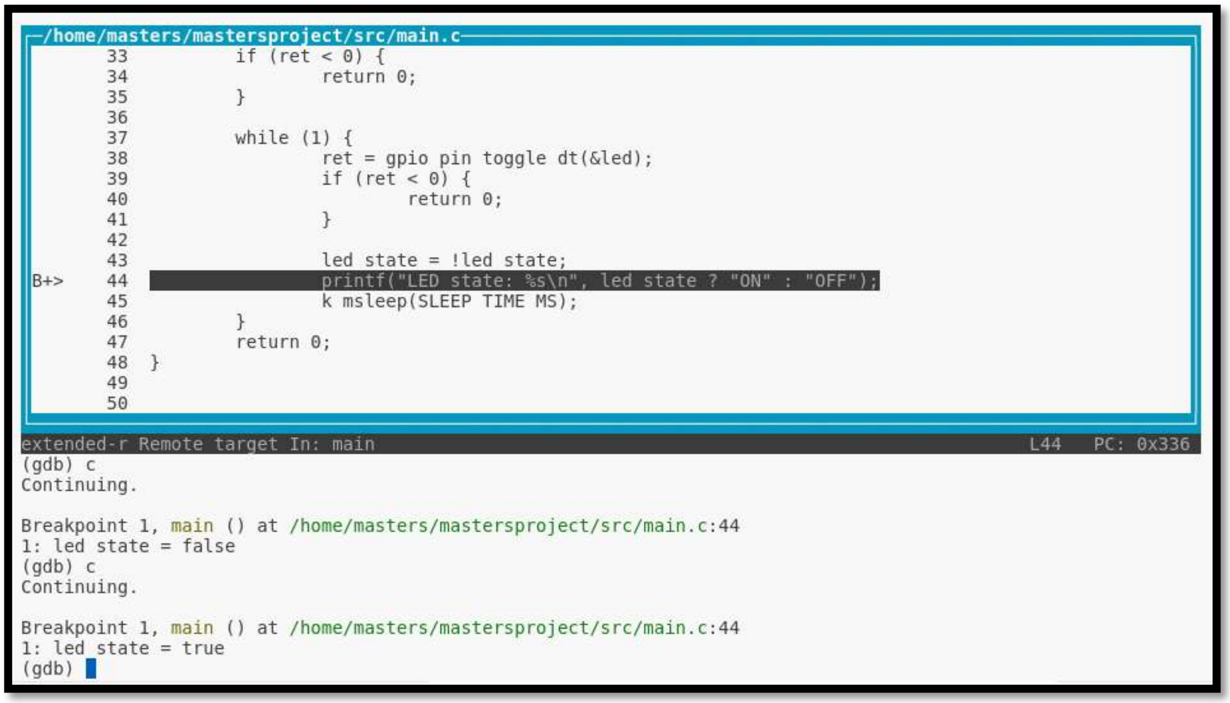
4. Add a breakpoint to your GDB session, the continue operation until the breakpoint is hit

```
(gdb) break main.c:44
(gdb) continue
```

5. The device will now run until the breakpoint at main.c line 44 is reached. When operation is broken at that time, view the value of a variable

```
(gdb) display led_state
(gdb) c
```

Observe the displayed value of the variable led_state on subsequent loops of the main loop, and verify that this variable matches the state of LED0 on your SAME54XPRO:



```
/home/masters/mastersproject/src/main.c
        33          if (ret < 0) {
        34                  return 0;
        35          }
        36
        37          while (1) {
        38                  ret = gpio pin toggle dt(&led);
        39                  if (ret < 0) {
        40                          return 0;
        41                  }
        42
        43                  led state = !led state;
B+>     44                  printf("LED state: %s\n", led state ? "ON" : "OFF");
        45                  k msleep(SLEEP TIME MS);
        46          }
        47          return 0;
        48  }
        49
        50
extended-r Remote target In: main                                          L44   PC: 0x336
(gdb) c
Continuing.

Breakpoint 1, main () at /home/masters/mastersproject/src/main.c:44
1: led state = false
(gdb) c
Continuing.

Breakpoint 1, main () at /home/masters/mastersproject/src/main.c:44
1: led state = true
(gdb)
```

Exit GDB and return to your venv command prompt:

```
(gdb) exit
```

If prompted, choose y to quit the active session

Supplement C - Flashing the WINC1500 Firmware

Introduction

Just as in MPLABX/Harmony applications, it's important for the Firmware verison on the WINC1500 and the Driver version in software to be in sync. In Zephyr, the driver version used is based on version 19.5.2, thus our WINC1500 Firmware version should match.

While firmware can be updated from other platforms, this procedure is generally accomplished easiest from a MS Windows computer. Updating from other platforms is subject to change and is outside the scope of this supplement.

z

Procedure

Download ASF standalone version 3.35.1, which includes WINC1500 Firmware version 19.5.2 from here.

If other versions are needed, the full list of previous ASF versions are available for download here

UnZip the download to a folder, then navigate to the following path within the folder:

asf-standalone-archive-3.35.1.54\xdk-asf-3.35.1\common\components\wifi\winc1500\firmware_update_project

Connect your WINC1500 module to EXT1 on a *Supported Board,* then connect your supported board over the Debug USB port to your host computer

At the time of writing, supported boards are:

SAMW25 XPRO

SAM4S XPRO

SAMD21 XPRO

SAMG53 XPRO

SAMG55 XPRO

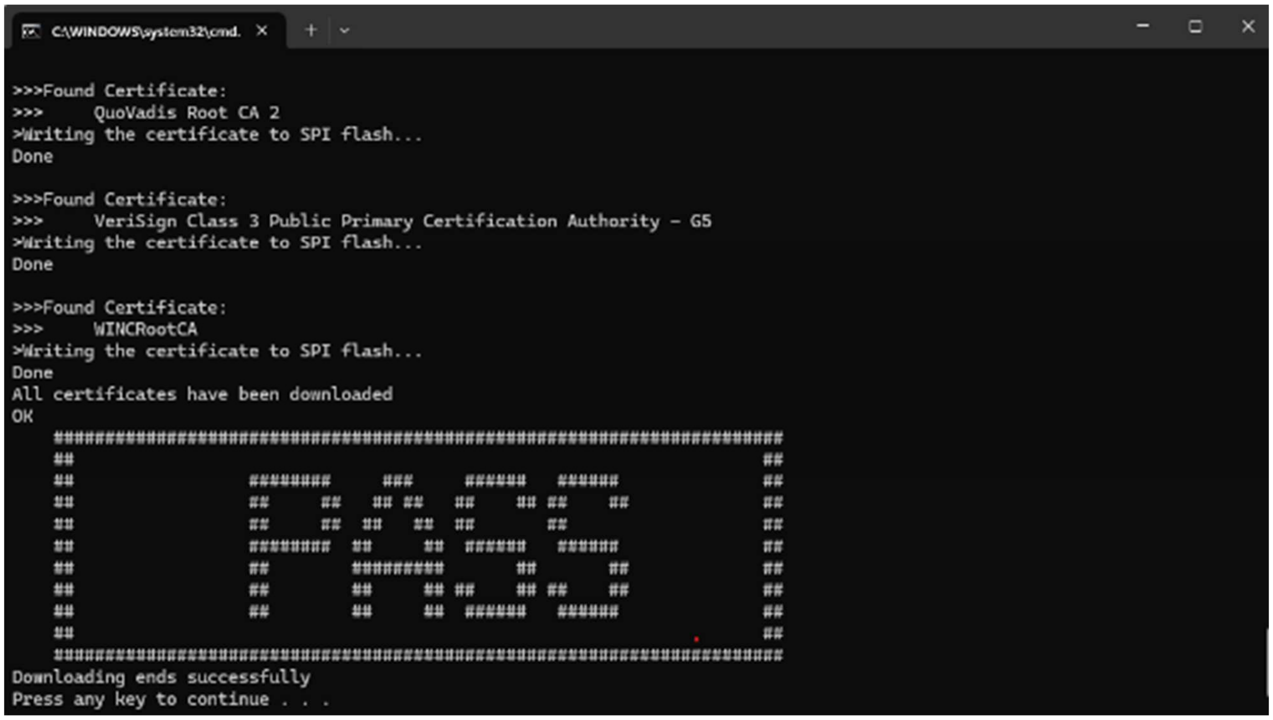
SAML21 XPRO

SAML22 XPRO

SAMR21 XPRO

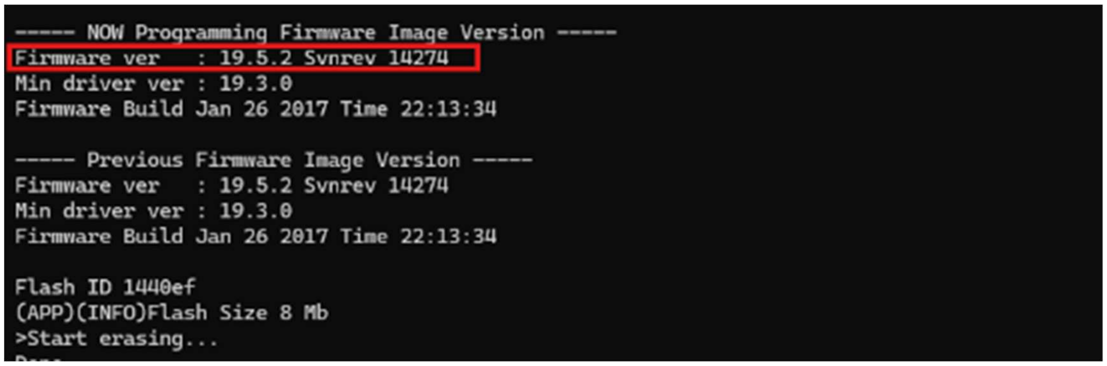
A sample program can be created for other boards as well by following the guidance of an existing program. Alternatively, you can use a USB UART dongle connected to the Debug UART port on the WINC1500XPlained Pro extension board or on your custom hardware. Check the Application Note in the doc/ folder entitled *WINC_Devices_Integrated_Serial_Flash_Download_Procedure.pdf* for more information regarding this process.

In the "firwmare_update_project" folder, choose the .bat file that matches your chosen supported board, and wait while the Firmware is updated automatically. This process may take up to 2 minutes. When complete, you will see a command window with the following output:



```
>>>Found Certificate:
>>>     QuoVadis Root CA 2
>Writing the certificate to SPI flash...
Done

>>>Found Certificate:
>>>     VeriSign Class 3 Public Primary Certification Authority - G5
>Writing the certificate to SPI flash...
Done

>>>Found Certificate:
>>>     WINCRootCA
>Writing the certificate to SPI flash...
Done
All certificates have been downloaded
OK
    ########################################################################
    ##                                                                    ##
    ##                 ########     ###     ######   ######               ##
    ##                 ##     ##   ## ##   ##     ## ##    ##              ##
    ##                 ##     ##  ##   ##  ##        ##                    ##
    ##                 ########  ##     ##  ######   ######               ##
    ##                 ##        #########       ##       ##              ##
    ##                 ##        ##     ## ##    ## ##    ##              ##
    ##                 ##        ##     ##  ######   ######               ##
    ##                                                                    ##
    ########################################################################
Downloading ends successfully
Press any key to continue . . .
```

Scrolling up within the cmd window output, verify that your firmware has been set to v 19.5.2 as seen below:



```
----- NOW Programming Firmware Image Version -----
Firmware ver   : 19.5.2 Svnrev 14274
Min driver ver : 19.3.0
Firmware Build Jan 26 2017 Time 22:13:34

----- Previous Firmware Image Version -----
Firmware ver   : 19.5.2 Svnrev 14274
Min driver ver : 19.3.0
Firmware Build Jan 26 2017 Time 22:13:34

Flash ID 1440ef
(APP)(INFO)Flash Size 8 Mb
>Start erasing...
```

WINC1500 firmware update has now been completed, and you may continue with this lab manual.

You may need to restore your WINC1500 firmware version at a later time. To achieve this, simply follow these same steps with a different standalone ASF version from the link found in Step 1

2.3: FOR EXT1 Open same54_xpro.overlay and add the following lines, replacing **bold** items with correct information from the User Guides. Use the format found in **cs-gpios** to help update **irq-gpios**, **reset-gpios**, and **enable-gpios**

```
/
{
     aliases {
     };
};

&gmac {
    status = "disabled";
};

&sercom4 {
     compatible = "atmel,sam0-spi";
     dipo = <3>;
     dopo = <0>;
     #address-cells = <1>;
     #size-cells = <0>;
     cs-gpios = <&portb 28 GPIO_ACTIVE_LOW>;
     pinctrl-0 = <&sercom4_spi_default>;
     pinctrl-names = "default";
     status = "okay";

     sercom4_cs0_winc1500: WINC1500@0 {
          compatible = "atmel,winc1500";
          reg = <0>;
          spi-max-frequency = <12000000>;
          irq-gpios = <&portx N GPIO_ACTIVE_LOW>;
          reset-gpios = <&portx N GPIO_ACTIVE_LOW>;
          enable-gpios = <&portx N GPIO_ACTIVE_HIGH>,
                         <&portx N GPIO_ACTIVE_HIGH>;
          status = "okay";
     };
};
```

enable-gpios lists two pins, comma separated. Since the CHIP_EN pin and the WAKE pin largely share functionality for our demo, listing them both allows the software driver to toggle them together when the chip is enabled or disabled. You could also use a Devicetree GPIO node to configure the WAKE pin separately to manage it on your own.