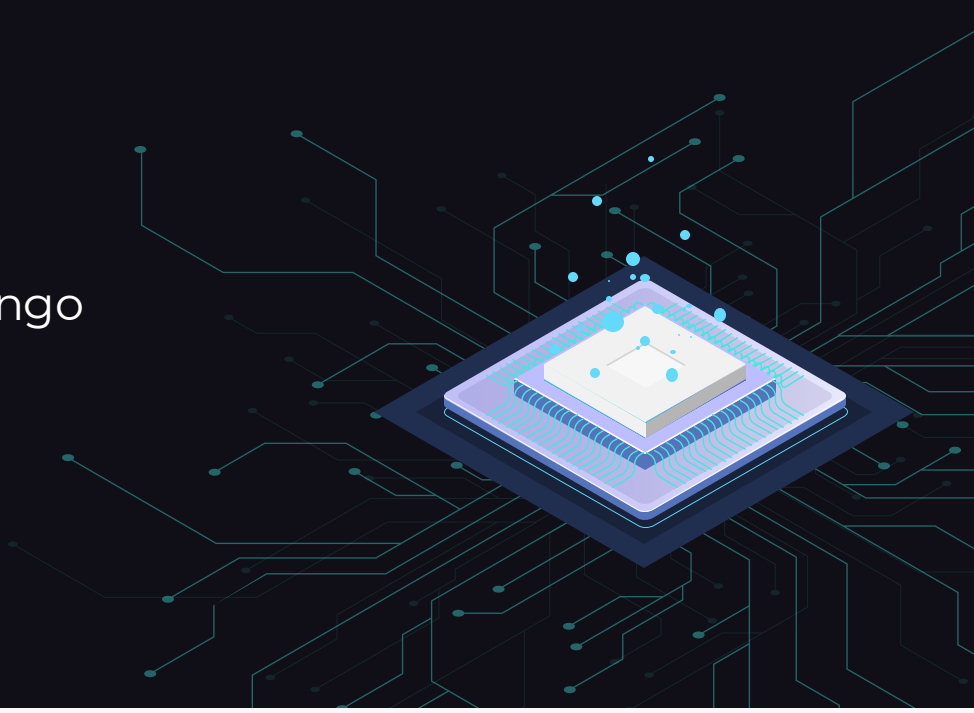
Partners: ac6 training citrinio

Zephyr RTOS Community Training

Day 2 – Multi-Tasking and Kernel Services



By: Jacob Beningo

BENINGO EMBEDDED GROUP

EMBEDDED SOFTWARE ACADEMY BY BENINGO

www.beningo.com
Contact: jacob@beningo.com

Table of Contents

Contents

Day 2 Overview – Multitasking, Scheduling, and Kernel Services in Zephyr

Modern embedded systems demand predictable behavior, concurrency, and efficiency. Whether you're building sensor networks, IoT devices, motor-control applications, or edge AI workloads, your firmware must handle multiple tasks in real time without falling into the traps of race conditions, blocking operations, or unpredictable latencies.

Yet many teams still rely on outdated threading models, ad-hoc synchronization patterns, and limited observability into runtime behavior. This slows development, makes systems brittle, and leads to difficult-to-diagnose failures—especially once devices are in the field.

Day 2 of this course focuses on the core of modern embedded design. You'll learn how to architect responsive, reliable, and deterministic systems using Zephyr's threading model and kernel services, and you'll gain hands-on experience with the tools that professionals use to analyze, debug, and optimize real-time behavior.

Through guided examples and exercises, we'll move from the fundamentals—creating and synchronizing threads—into more advanced concepts such as timeslicing, tickless scheduling, kernel ready queues, and work queues. You'll also learn how to observe your system in action with Percepio Tracealyzer/View, giving you visibility into performance bottlenecks, task interactions, and scheduling decisions.

We'll explore essential topics such as:

- Creating, prioritizing, and managing threads
- Understanding Zephyr's main thread, idle thread, and system initialization model
- Designing delays, timeouts, and deterministic timing behavior
- Using semaphores and mutexes to avoid race conditions
- How the scheduler works: tick-based vs tickless mode
- Offloading work using WorkQueues
- Capturing and analyzing runtime behavior using Percepio Tracealyzer/View

By applying these concepts in hands-on labs, you'll build the skills needed to design robust real-time systems and gain the confidence to debug complex timing and concurrency issues—long before they become field failures.

If you have questions, need guidance, or want to share feedback, feel free to reach out at jacob@beningo.com.

3-Day Zephyr Community Training Partners

The 3-Day Free Zephyr Community Edition Training is brought to you in partnership with AC6 and Citrinio. Each day is brought to you from one of the partners:

Day 1 - Roy Jamil

Day 2 – Jacob Beningo

Day 3 - Rodrigo Peixoto

The Instructor – Jacob Beningo

Jacob Beningo, CSDP. Jacob Beningo is an embedded software consultant who dramatically transforms businesses by improving software quality, cost, and time to market. He regularly publishes articles, blogs, webinars, and white papers about software architecture, processes, and development skills on over half a dozen platforms. He has completed projects across several industries, including automotive, defense, medical, and space. Jacob has worked with companies in over a dozen countries and has software operating on Earth, in orbit, around the moon, on the moon, and someday around Mars. Jacob holds bachelor's degrees in electrical engineering, Physics, and Mathematics from Central Michigan University and a master's degree in Space Systems Engineering from the University of Michigan.

Contact Jacob at jacob@beningo.com

www.beningo.com contains additional resources, templates, and Jacob's monthly Embedded Bytes newsletter. Check out his workshops at https://beningo.mykajabi.com/

Click the social media link below to follow Jacob and get more tips and tricks:

Recommended Materials

Development Board(s)

No development boards are required for this course. We'll be using a simulated development environment. If you have a development board, you can use it with these materials, some instructions may need to be modified.

Software Tool(s)

Use the tools provided in the Pre-Training Setup Document.

Recommendations for Completing the Labs

There are several options for how the attendee can complete each lab.

First, the most recommended method is to follow all the steps in the lab from scratch. Performing each lab from scratch will maximize your experience with the course material and topics. However, doing the labs from scratch is also the most time-consuming. This method may not allow you to complete the labs in the given timeframe in a time-constrained environment, such as during live on-site training.

Second, you can copy and paste the solution modules code for each step into your code project. This will decrease the time and potential error of writing all the lines of code yourself and reduce the time it takes to complete the labs.

Finally, you can import the lab solutions and play with them to understand the main concepts. This is the quickest way to complete the labs and may provide the most negligible value. The general way to master a technique or topic is to attend the lecture and hear about it, go through the lab hands-on to try it, and then implement that technique into your software.

Lab Exercises

Lab Exercise 1 – Zephyr Threading Fundamentals

Overview

Modern embedded systems require predictable, responsive behavior through effective multitasking. Understanding how a real-time operating system initializes, schedules threads, and manages system resources is fundamental to building reliable firmware.

In this lab, you'll explore Zephyr's threading model by examining the system initialization sequence, the main thread, and the idle thread. You'll learn how Zephyr boots, how threads are scheduled, and what happens when no application threads are ready to run.

Unlike bare-metal systems where you write a `main()` function that runs forever in a superloop, Zephyr provides a preemptive, priority-based scheduler that manages multiple threads of execution. Understanding this model is critical for building scalable, maintainable embedded applications.

Why This Matters:

Many developers transitioning from bare-metal to RTOS development don't fully understand the boot sequence, thread lifecycle, and scheduling behavior. This leads to confusion about:

- When and how `main()` is called
- What priority the main thread has
- When the idle thread runs and why it matters
- How the scheduler decides which thread to execute

By the end of this lab, you'll have hands-on experience observing these concepts in action, giving you the foundation needed for more advanced multitasking topics in subsequent labs.

Learning Objectives

By completing this lab, you will:

- Understand the Zephyr boot sequence and system initialization process
- Identify the main thread and its default configuration (priority, stack size)
- Explain the purpose and behavior of the idle thread
- Use Zephyr kernel APIs to query thread information at runtime
- Apply best practices for instrumenting and debugging threaded applications

Lab Instructions

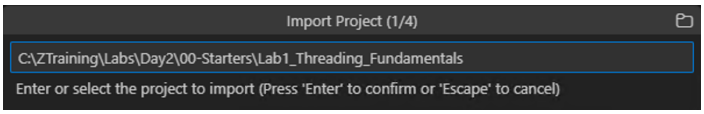
Part 1: Open the Starter Project

1) Start Visual Studio Code

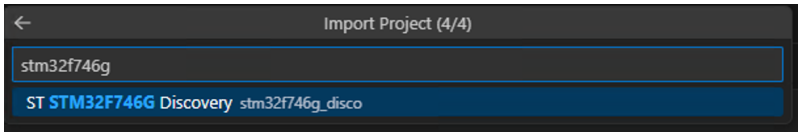
2) Click the Zephr Workbench Extension Icon. You'll noticed a section that says application. Click the down arrow to import the starter project.

Note: The + button is to create a new project.

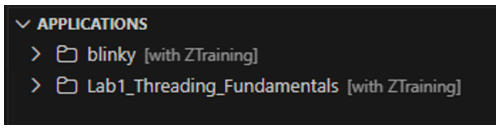
3) You'll notice in the command palette that you are being prompted. Use Explorer to find the location and path of the starter project. Below is an example path:



4) Accept the workspace and the toolchain by pressing enter. Then type stm32f746g and select the Discovery board as shown below:

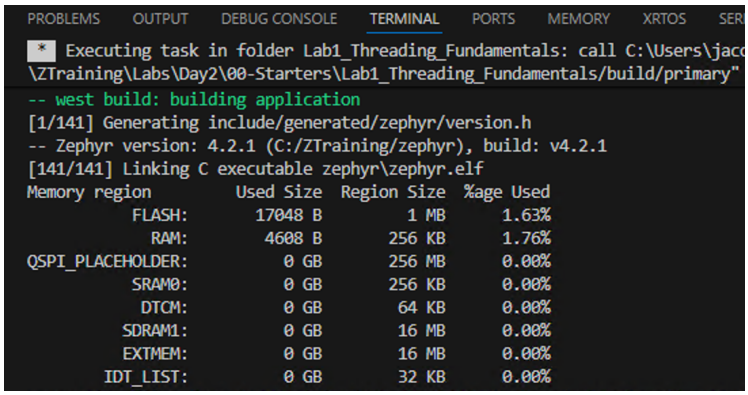


5) If everything went as expected, the application section should now show your imported project:



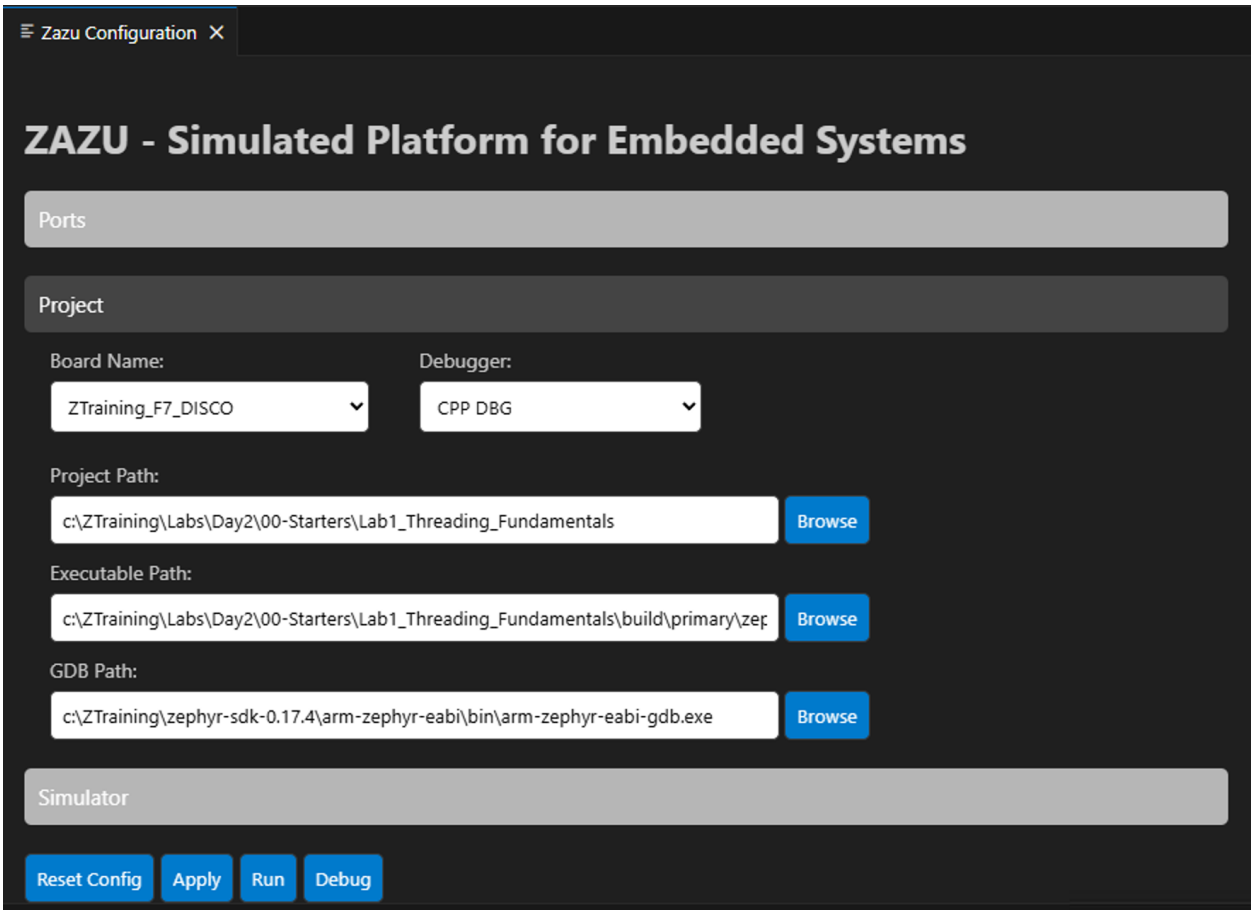
6) If you place your cursor over the Lab1 project, you see several icons appear. The gear can be used to build the project. The wrench to configure the project. The play button to run the project. The bug with play button to debug.

Click the gear button to build the project. If it is successful, you'll see the following:



7) Since we've compiled the project, let's run it using Zazu, in simulation mode. Click the Zazu extension.

8) Click Zazu Configuration and configure the project. Here are the settings for this first lab:



My full executable path is:

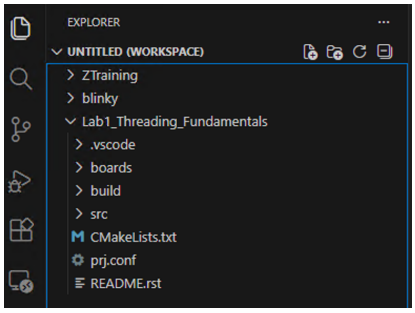
c:\ZTraining\Labs\Day2\00-Starters\Lab1_Threading_Fundamentals\build\primary\zephyr\zephyr.elf

In the other labs, you'll need to update the Project path and Executable path each time. I won't show you the settings again after this. (Just a reminder).

9) Click apply and Run. You should see the following:

There are two important points to note. First, on the left side of the board, the small green mark is an LED that should be blinking in the simulation. Second, our print messages go to the terminal at the bottom of the simulation area. It's printing whether that on-board LED is on or off.

10) Now, click Explorer in VS Code, you should see the starter project in the workspace:



11) Open src/main.c in your editor. You'll see a basic blinky application with several TODO comments indicating where you'll add code during the rest of this lab.

12) Review the existing code:

- The main() function is the entry point for the main thread
- GPIO initialization configures an LED
- A while(1) loop toggles the LED and sleeps

Part 2: Add System Initialization Information

Now that you have a general feel for the development environment, let's enhance the application a bit by adding print information about the system initialization and the main thread.

Objective: Create a function that prints startup information when the system boots.

Instructions:

13) Add a new function print_system_info() above the main() function:

```
/**
 * @brief Print system initialization information
 */
static void print_system_info(void)
{
    printk("\n");
    printk("=========================================\n");
    printk("  Lab 1: Threading Fundamentals\n");
    printk("=========================================\n");
    printk("System initialized successfully\n");
    printk("Current thread: %s\n", k_thread_name_get(k_current_get()));
    printk("Main thread priority: %d\n", k_thread_priority_get(k_current_get()));
    printk("=========================================\n");
    printk("\n");
}
```

14) Call this function at the beginning of main(), right after the function declaration:

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

    /* Print system information */
    print_system_info();

    /* ... rest of the code ... */
}
```

15) Rebuild and rerun Zazu.

Note: You can build the project in multiple ways.

- Use Zephyr Workbench Application Section
- Use Zephr Workbench West Workspace to open a terminal, navigate to the project directory and use the command line:
    - west build -b stm32f746g_disco
- Click the build button in the bottom left of VS Code

You should see the following output now when you start the project:

```
*** Booting Zephyr OS build v4.2.1 ***

=======================================
  Lab 1: Threading Fundamentals
=======================================
System initialized successfully
Current thread: (null)
Main thread priority: 0
=======================================

LED blinky application started
LED state: OFF
LED state: ON
```

Yikes! Our main thread's name is null! What is happening here, any ideas?

16) The reason the thread name is not reported is that thread name reporting is disabled by default. We need to modify our configuration in prj.conf. Open prj.conf and add the following configuration:

    CONFIG_THREAD_NAME=y

17) I would now recommend doing a pristine build. You can do this by going to the Zephyr Workbench extension. Under application, right click your project, select clean, and then rebuild/pristine. Once you've rebuilt, run the project in Zazu again. You should now see the following:

```
*** Booting Zephyr OS build v4.2.1 ***

=======================================
  Lab 1: Threading Fundamentals
=======================================
System initialized successfully
Current thread: main
Main thread priority: 0
=======================================

LED blinky application started
LED state: OFF
```

Key Observations:

- The thread name is "main" - this is the default name for the main thread
- The priority is 0, which is a cooperative priority in Zephyr
- This information is printed before the LED starts blinking

Understanding Thread Priority:

In Zephyr:

- Negative priorities (-1 to -CONFIG_NUM_COOP_PRIORITIES): Cooperative threads (never preempted by other threads)
- Priority 0: Highest preemptive priority
- Positive priorities (0 to CONFIG_NUM_PREEMPT_PRIORITIES): Preemptible threads
- Lower numbers = higher priority (priority -1 is higher than priority 5)

The main thread defaults to priority 0, making it a cooperative thread.

Step 3: Explore the Main Thread

Let's add more instrumentation to understand what the main thread is doing.

Objective: Add a counter and periodic informational messages to track main thread execution.

18) Add a counter variable in main() after the existing variable declarations:

```
int main(void)
{
   int ret;
   bool led_state = true;
   uint32_t blink_count = 0;  // Add this counter

   /* ... existing code ... */
}
```

19) Update the LED toggle section to increment the counter and provide more information:

```
/* Main application loop */
while (1) {
   /* Toggle the LED */
   ret = gpio_pin_toggle_dt(&led);
   if (ret < 0) {
      printk("Error: Failed to toggle LED\n");
      return ret;
   }

   led_state = !led_state;
   blink_count++;

   /* Print status every blink */
   printk("[Main Thread] Blink #%u - LED state: %s\n",
         blink_count,
         led_state ? "ON" : "OFF");

   /* Provide additional information periodically */
   if(blink_count % 5 == 0) {
      printk("  [Info] Main thread has executed %u blinks\n", blink_count);
      printk("  [Info] During k_msleep(), other threads can execute\n");
   }

   /* Sleep for the specified duration */
```

```
    k_msleep(SLEEP_TIME_MS);
}
```

20) Rebuild and run the application. You should see the following:

```
LED blinky application started
[Main Thread] Blink #1 - LED state: OFF
[Main Thread] Blink #2 - LED state: ON
[Main Thread] Blink #3 - LED state: OFF
[Main Thread] Blink #4 - LED state: ON
[Main Thread] Blink #5 - LED state: OFF
  [Info] Main thread has executed 5 blinks
  [Info] During k_msleep(), other threads can execute
```

Key Concept - Thread Suspension:

When k_msleep(1000) is called:

- The main thread is suspended for 1000 milliseconds
- The thread enters the “suspended” state and is removed from the ready queue
- The scheduler runs and selects the next highest-priority ready thread
- If no other threads are ready, the idle thread runs
- After the timeout expires, the main thread returns to the “ready” state
- The scheduler will eventually resume the main thread

Part 4: The Idle Thread

The idle thread is a special system thread that runs when no other threads are ready. Let’s add information about it.

Objective: Help users understand the idle thread’s purpose and when it executes.

Instructions:

21) Add a new function to explain the idle thread:

```
/**
 * @brief Print idle thread information
 */
static void print_idle_thread_info(void)
{
    printk("\n");
    printk("--- Idle Thread Information ---\n");
    printk("The idle thread runs when no other threads are ready.\n");
    printk("It has the lowest priority in the system.\n");
    printk("During idle, the CPU can enter low-power modes.\n");
```

```
    printk("The Idle thread can't be directly modified by the developer.\n");

    printk("--------------------------------\n");
    printk("\n");
}
```

22) Call this function in main() after GPIO initialization but before the main loop:

```
printk("GPIO initialization complete\n");
printk("LED will blink every %d ms\n", SLEEP_TIME_MS);
printk("Main thread entering blink loop...\n\n");

/* Print idle thread information once */
print_idle_thread_info();

/* Main application loop */
while (1) {
    /* ... */
}
```

23) Rebuild and run to have the idle information now included in your application.
24) In the the VS Code Explorer, navigate to where you have your main zephyr folder. Look for the folder named kernel. Within that folder is a file named idle.c. Open it.
25) Review the file. This file has the idle thread. You'll notice that the main function that's called is k_cpu_idle. That will call arch_cpu_idle. And that calls the idle specific code for the particular target that you're using.

Understanding the Idle Thread:

- **Purpose**: Runs when no application threads are ready
- **Priority**: Lowest in the system (runs only when nothing else can)
- **Behavior**: Can invoke power management to reduce energy consumption
- **Always Present**: Created automatically by Zephyr during initialization
- **Never Blocked**: Always ready to run (but lowest priority)

Real-World Relevance:

In production embedded systems, the idle thread is where power management happens. When your application threads are waiting for events (sensors, timers, communication), the idle thread runs and can put the CPU into low-power sleep modes, dramatically reducing power consumption in battery-powered devices.

Part 5: Zephyr Boot Sequence

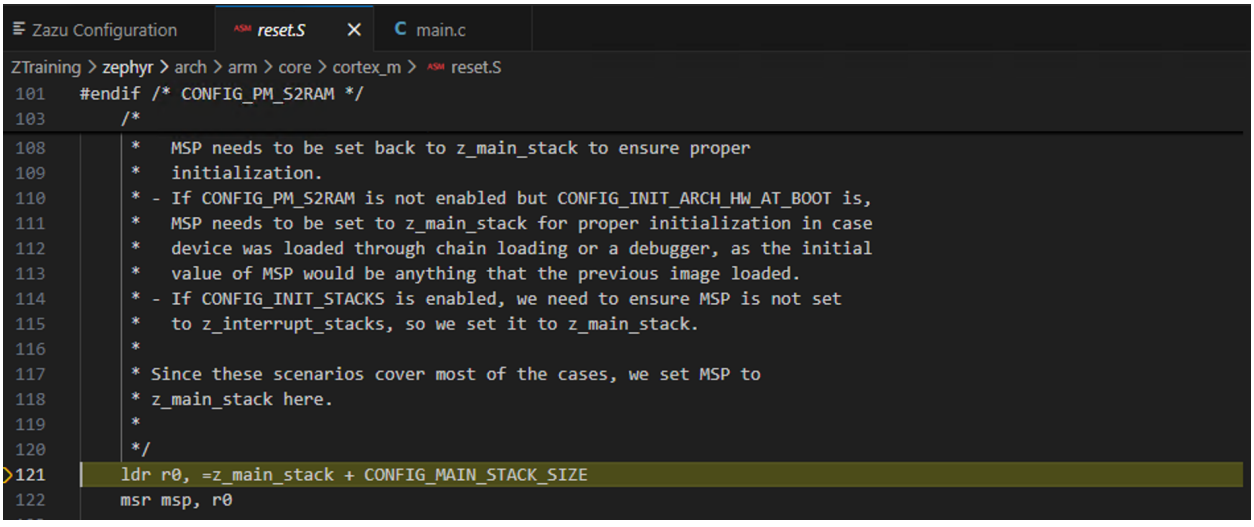
Understanding how Zephyr initializes is crucial for embedded development.

Boot Sequence Overview:

1. **Hardware Reset**: CPU starts executing from reset vector
2. **Early Assembly Code**: Sets up stack, initializes hardware basics
3. **C Runtime Initialization**: Initializes .data and .bss sections
4. **Kernel Initialization**: Zephyr kernel structures are initialized
5. **Driver Initialization**: Device drivers are initialized (using devicetree)
6. **Main Thread Creation**: Kernel creates the main thread
7. **Scheduler Start**: Scheduler begins, main thread starts running
8. **main() Execution**: Your main() function is called in the main thread context

Instructions:

26) Open the Zazu extension and navigate to the configuration.
27) Click Debug. You'll now find yourself at:

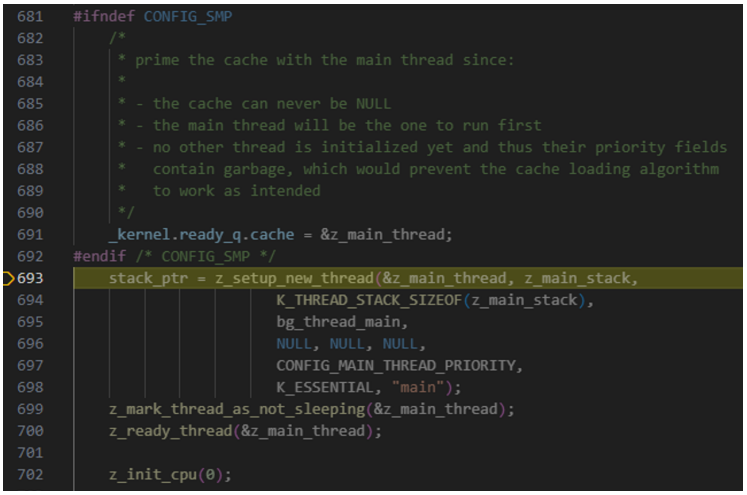


This is in the reset.S file. The entry point for the reset vector of the system.

28) Somewhere on your screen, usually at the top, you'll find the navigation panel:

29) Use the step and step over options to walk through the start-up process. As you step through you'll eventually reach z_cstart(). This is where the zephyr kernel starts. Step into that function.

30) You should find yourself in init.c. This is where all the zephyr initialization occurs. Previously, it was all the standard embedded initialization that we have all come to know and love over the years.
31) If you keep stepping into and through the code, you'll eventually find where we initialize the main thread! (Hint: init.c line 693):



32) Keep stepping. You'll find basic hardware initialization, mutli-level kernel boot, etc. You'll find where the kernel wants to switch to the main thread, and initializes a ready queue.
33) Once you get through this, you'll find that the main thread starts to run!

Important Notes:

- You don't create the main thread - Zephyr does it for you
- main() is not called from bare metal, it's called by the kernel in a thread context
- Before main() runs, the kernel and drivers are fully initialized
- The idle thread is also created during kernel initialization

Summary

Congratulations! You've completed Lab 1 and gained hands-on experience with Zephyr's threading fundamentals.

What You Learned

In this lab, you:

- Explored the Zephyr boot sequence and system initialization
- Identified the main thread and its default priority (0, preemptive)

- Understood the idle thread's purpose and when it executes
- Used Zephyr kernel APIs to query thread information
- Recognized thread state transitions (ready, running, suspended)
- Applied instrumentation techniques for debugging threaded applications

Key Takeaways

Main Thread:

- Automatically created by Zephyr during initialization
- Runs your main() function
- Default priority 0 (cooperative)
- Useful for initialization and lightweight tasks

Idle Thread:

- Runs when no other threads are ready
- Lowest priority in the system
- Enables power management and energy efficiency
- Created automatically by Zephyr

Thread Scheduling:

- Zephyr uses priority-based preemptive scheduling
- Threads transition between ready, running, and suspended states
- k_msleep() suspends a thread, allowing others to run
- The scheduler always runs the highest-priority ready thread

Next Steps

Now that you understand basic threading, you're ready to create custom threads and learn about inter-thread synchronization.

In Lab 2, you will:

- Create multiple custom threads with different priorities
- Implement semaphore-based synchronization between threads
- Design periodic and event-driven tasks
- Understand delays, timeouts, and blocking operations

Lab Exercise 2 – Synchronization

Overview

In real-time embedded systems, threads rarely work in isolation. They need to communicate, synchronize their activities, and coordinate access to shared resources. Semaphores are one of the fundamental synchronization primitives that enable threads to signal each other and coordinate their execution.

This lab focuses on creating a multi-threaded application where:

- A periodic thread runs at regular intervals and signals another thread
- An event-driven thread blocks waiting for signals and responds only when notified

You'll learn the difference between time-based (periodic) and event-driven task models, understand thread priorities, and implement production-quality modular code where each thread is in its own source file with no global variables.

Why This Matters:

> Many embedded applications combine periodic tasks (sensor sampling, control loops) with event-driven tasks (responding to button presses, communication events). Understanding how to synchronize these tasks efficiently is critical for responsive, deterministic firmware. Blocking on a semaphore consumes zero CPU cycles, making it far more efficient than polling.

Learning Objectives

By completing this lab, you will:

- Create custom threads using k_thread_create()
- Configure thread priorities and understand preemption
- Implement semaphore-based synchronization between threads
- Design periodic tasks using k_msleep()
- Design event-driven tasks that block on semaphores
- Apply production code architecture (modular, encapsulated)
- Understand the difference between polling and blocking

Step 1: Review the Project Structure

1) Navigate to the starter project:

   C:\ZTraining\Labs\Day2\00-Starters\Lab2_Synchronization

2) Review the file structure. Notice that the code is organized into modules:

- **sync_mgr**: Encapsulates semaphore management
- **periodic_thread**: Implements the periodic task
- **event_thread**: Implements the event-driven task
- **main**: System initialization

Each module has a .h (header) and .c (implementation) file. This modular architecture is production-quality code organization.

Step 2: Implement the Synchronization Manager

Task: Define and implement semaphore operations.

3) Open src/sync_mgr.c.

Define the Semaphore:

4) Add this line near the top of the file, after the includes:

```
static K_SEM_DEFINE(event_sem, 0, 1);
```

If you aren't familiar with this syntax, here is the explanation:

- K_SEM_DEFINE: Creates and initializes a semaphore at compile time
- event_sem: Semaphore name
- 0: Initial count (no events pending)
- 1: Maximum count (binary semaphore - only 0 or 1)
- static: Keeps it private to this module (encapsulation)

5) You now need to implement the signal function. In sync_mgr_signal_event(), add:

```
void sync_mgr_signal_event(void)
{
   k_sem_give(&event_sem);
}
```

6) Now we need a function that can be used to wait for the signal. In sync_mgr_wait_event(), add:

```
int sync_mgr_wait_event(k_timeout_t timeout)
{
   return k_sem_take(&event_sem, timeout);
}
```

Key Concepts:

- k_sem_give(): Increments the semaphore count, unblocking waiting threads
- k_sem_take(): Decrements the count (blocks if count is zero)
- The semaphore is hidden from other modules - they use the public API only

Step 3: Implement the Periodic Thread

Task: Create a thread that runs every 2 seconds and signals the event thread.

7) Open src/periodic_thread.c.
8) Let's now define the thread resources. Add these lines after the #define statements:

```
static K_THREAD_STACK_DEFINE(periodic_thread_stack, PERIODIC_THREAD_STACK_SIZE);
static struct k_thread periodic_thread_data;
```

9) Implement the thread entry function by adding the following code to the periodic_thread_entry() while loop:

```
while (1) {
    execution_count++;

    printk("[Periodic Thread] Execution #%u - Signaling event thread\n",
           execution_count);

    sync_mgr_signal_event();

    k_msleep(PERIOD_MS);
}
```

10) Now that we have the thread, we need it to also be created so it actually runs. In periodic_thread_init(), add:

```
int periodic_thread_init(void)
{
    k_tid_t tid;

    tid = k_thread_create(&periodic_thread_data,
                          periodic_thread_stack,
                          K_THREAD_STACK_SIZEOF(periodic_thread_stack),
                          periodic_thread_entry,
                          NULL, NULL, NULL,
                          PERIODIC_THREAD_PRIORITY,
                          0,
                          K_NO_WAIT);
```

```
    if (tid == NULL) {
        printk("[Periodic Thread] ERROR: Failed to create thread\n");
        return -1;
    }

    k_thread_name_set(tid, "periodic");

    printk("[Periodic Thread] Initialized\n");
    return 0;
}
```

Key Concepts:

- Thread stack must be defined with K_THREAD_STACK_DEFINE
- Thread control block (struct k_thread) holds thread state
- k_thread_create() initializes and starts the thread
- Priority 7 is preemptible (positive priority) - K_NO_WAIT means start immediately

Step 4: Implement the Event Thread

Task: Create a higher-priority thread that blocks waiting for semaphore signals.

11) Open src/event_thread.c.

12) Define the thread resources after the LED definitions by adding:

```
static K_THREAD_STACK_DEFINE(event_thread_stack, EVENT_THREAD_STACK_SIZE);
static struct k_thread event_thread_data;
```

13) In gpio_init(), add the following code to initialize the LED:

```
static int gpio_init(void)
{
    int ret;

    if (!gpio_is_ready_dt(&led)) {
        printk("[Event Thread] ERROR: LED device not ready\n");
        return -ENODEV;
    }

    ret = gpio_pin_configure_dt(&led, GPIO_OUTPUT_INACTIVE);
    if (ret < 0) {
        printk("[Event Thread] ERROR: Failed to configure LED\n");
        return ret;
    }
```

```
    return 0;
}
```

14) We now need code to process when an event occurs. In the event_thread_entry() while loop, add:

```
while (1) {
    ret = sync_mgr_wait_event(K_FOREVER);

    if (ret == 0) {
        event_count++;
        gpio_pin_toggle_dt(&led);
        printk("[Event Thread] Event #%u received - LED toggled\n",
               event_count);
    }
}
```

15) Finally, we can create the thread in event_thread_init() by adding:

```
int event_thread_init(void)
{
    k_tid_t tid;

    tid = k_thread_create(&event_thread_data,
                          event_thread_stack,
                          K_THREAD_STACK_SIZEOF(event_thread_stack),
                          event_thread_entry,
                          NULL, NULL, NULL,
                          EVENT_THREAD_PRIORITY,
                          0,
                          K_NO_WAIT);

    if (tid == NULL) {
        printk("[Event Thread] ERROR: Failed to create thread\n");
        return -1;
    }

    k_thread_name_set(tid, "event");

    printk("[Event Thread] Initialized\n");
    return 0;
}
```

Note: We could, and maybe should, use the zephyr kernel macros to create and initialize the threads at compile time. This would be a good exercise for you to do on your own!

Key Concepts:

- Priority 6 is higher than priority 7 (lower number = higher priority)

- K_FOREVER means block indefinitely until the semaphore is available
- This thread consumes zero CPU while blocked
- Event-driven = efficiency!
- When signaled, this thread preempts the periodic thread (higher priority)

Step 5: Complete Main Initialization

Task: Initialize all subsystems in the correct order.

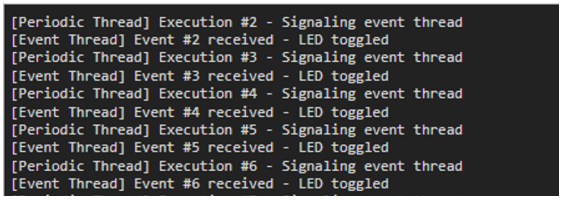
16) Open src/main.c.
17) In main(), add in the initializations we just created by replacing the TODO comments with:

```
ret = sync_mgr_init();
if (ret < 0) {
   printk("[Main] ERROR: Sync manager init failed\n");
   return ret;
}

ret = event_thread_init();
if (ret < 0) {
   printk("[Main] ERROR: Event thread init failed\n");
   return ret;
}

k_msleep(100);

ret = periodic_thread_init();
if (ret < 0) {
   printk("[Main] ERROR: Periodic thread init failed\n");
   return ret;
}
```

Key Point: Start the event thread before the periodic thread so it's ready to receive signals.

Step 6: Build and Test

18) Build the project.
19) Use Zazu to run the application in simulation mode. You should observe the following:
    a. The LED toggles when the signaling thread signals the event thread.
    b. The terminal will provide you with log information like the following:



Summary

Congratulations! You've implemented a multi-threaded application with semaphore synchronization.

What You Accomplished

- Created two custom threads with different priorities
- Implemented semaphore signaling between threads
- Designed a periodic task (time-based)
- Designed an event-driven task (semaphore-based blocking)
- Applied production code architecture
- Understood priority-based preemption

Key Takeaways

Semaphores enable inter-thread communication:

- Periodic thread signals events
- Event thread blocks efficiently (zero CPU while waiting)
- Higher-priority thread preempts when signaled

Thread priorities matter:

- Lower numbers = higher priority
- Higher-priority threads preempt lower-priority threads
- Design priorities based on response time requirements

Production code is modular:

- Each thread in separate files
- No global variables
- Clean public APIs
- Easy to test and maintain

Next Steps

In Lab 3, we'll take this same application, and I'll show you how to instrument it so that you can use Percepio View for Zephyr to visualize how the application is executing.

Lab 3: Percepio View for Zephyr

Overview

Understanding how your embedded system executes is critical for debugging timing issues, optimizing performance, and ensuring deterministic behavior. Percepio Tracealyzer is a powerful visualization and analysis tool that records kernel events and displays thread execution, CPU load, and system behavior.

This lab introduces a free, and limited version of Tracealyzer named Percepio View for Zephyr. It provides a snapshot mode where trace data is stored in a RAM buffer and extracted via GDB. You'll configure Percepio View, run the semaphore example we just looked at, capture trace data, and analyze.

Why This Matters:

When your system exhibits timing problems, race conditions, or unexpected behavior, Tracealyzer provides visibility into what's actually happening:

- Which threads are running and when
- How much CPU each thread consumes
- Where context switches occur
- Why threads are blocking
- Scheduling decisions in real-time

This lab teaches you how to instrument your code, capture traces, and analyze them, skills you'll use throughout your career debugging embedded systems.

Learning Objectives

By completing this lab, you will:

- Configure Tracealyzer in snapshot mode using prj.conf
- Understand menuconfig options for Tracealyzer
- Name threads for visibility in trace views
- Extract trace data from the RAM buffer
- Analyze thread execution in Tracealyzer View
- Interpret CPU load graphs
- Identify context switches and scheduling behavior
- Use free version features effectively

Step 1: Import the Starter Project

This is the same project as Lab 2. We are adding trace to it. You can either just continue to use the project you already have loaded, or you can import the Lab 3 starter project and configure it:

1) In Zephyr Workbench, import the project:

   C:\ZTraining\Labs\Day2\00-Starters\Lab3_Tracealyzer

2) Make sure to configure Zazu.

Step 2: Configure Percepio View for Zephyr in prj.conf

Task: Configure Percepio View for Zephyr

3) Open prj.conf. You'll add Tracealyzer configuration.

4) Enable tracing with Perceip by adding the following configuration lines:

   CONFIG_TRACING=y
   CONFIG_PERCEPIO_TRACERECORDER=y

5) Snapshot mode stores trace data in a RAM buffer (no streaming required). Configure it by adding the following lines to the configuration:

   CONFIG_PERCEPIO_TRC_CFG_STREAM_PORT_RINGBUFFER=y
   CONFIG_PERCEPIO_TRC_CFG_STREAM_PORT_RINGBUFFER_SIZE=8192

Explanation:

- STREAM_PORT_RINGBUFFER: Use RAM buffer for trace storage
- RINGBUFFER_SIZE: 8KB buffer (adjust based on RAM availability)
- Larger buffer = longer trace duration before overwrite

Step 3: Using menuconfig (Optional)

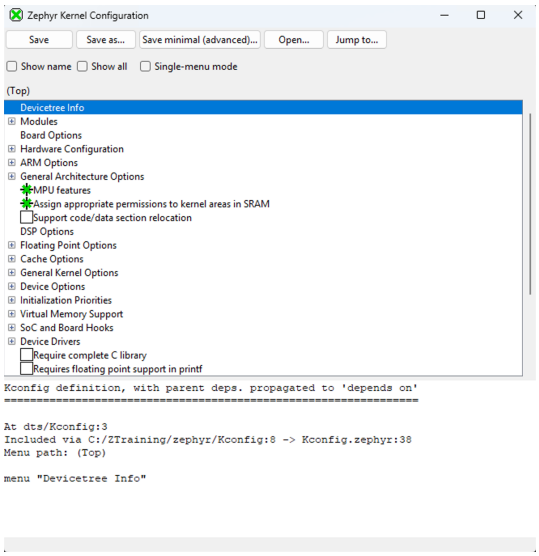
Task: Configure tracing using menuconfig

6) You can also configure Tracealyzer using menuconfig. If you were using a terminal, you could use:

   west build -b stm32f746g_disco -t menuconfig

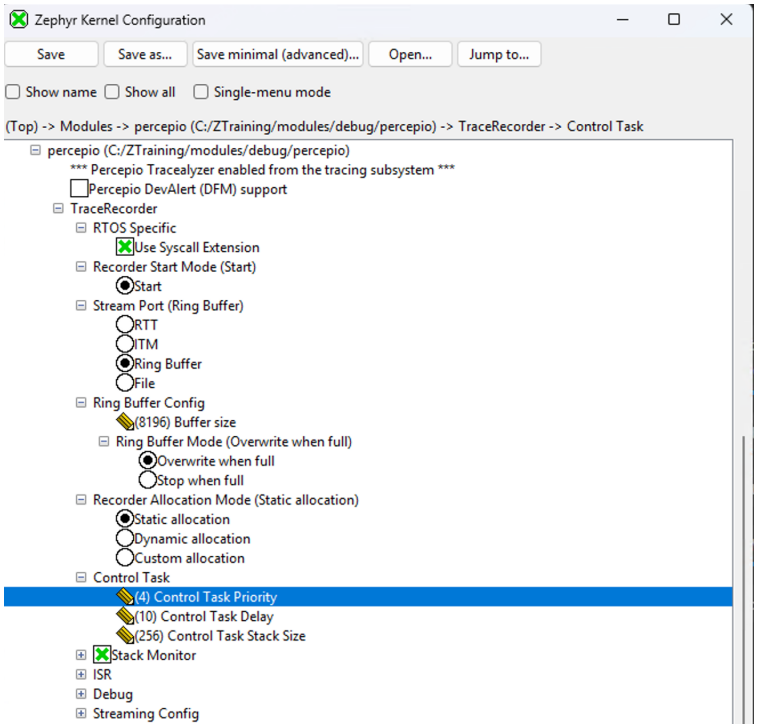
   This would then open a menu based interface to configure your environment.

   Since we are using Zephyr Workbench in VS Code, open the extension and right-click the Lab3_Tracealyzer application. Choose Configure->GUI Config. You should then see a screen like:



This is how you can use KConfig to configure your default Zephyr environment.

7) Expand Modules -> percepio -> Tracealyzer. You'll see all the defaults for working with View. You should see something like the following:



8) Since our application has tasks with lower priorities than 4, update the Control Task Priority to 20 . Then click save and exit GUI config.

Step 4: Verify Thread Naming

Thread names are critical for identifying threads in Tracealyzer. Without adding names to our threads, Tracealyzer will just show “Thread 0x20001234” or whatever the memory space is. Let’s add the thread names to the threads we created previously.

9) Open events_thread.c.
10) In event_thread_init, use k_thread_name_set to configure the thread name. Here is what I did:

```
if (tid == NULL) {
   printk("[Event Thread] ERROR: Failed to create thread\n");
   return -1;
}
else {
   k_thread_name_set(tid, "Event Thread");
}
```

11) Make the same change to the Periodic Thread.

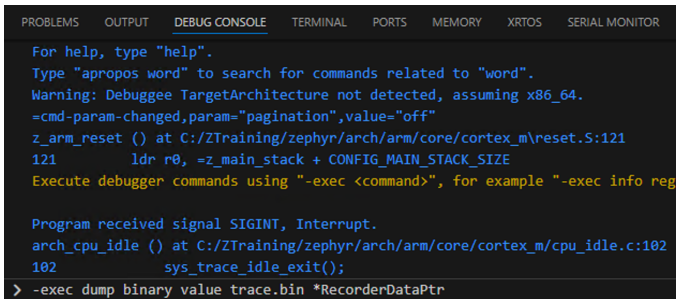
Step 5: Build and Analyze

Task: Run Percepio View to Analyze your Application

12) In the Zephyr Workbench, right-click the application and choose Clean -> Rebuild/Pristine to reb uild the application with the new configuration settings from KConfig and prj.config.
13) In Zazu, simulate the application using debug. Once you are at the reset vector, click run and wait for about 10 seconds. Pause execution.
14) Click on the Debug Console at the bottom of your screen. You’ll need to enter the following com mand to read the RAM Buffer:

```
-exec dump binary value trace.bin *RecorderDataPtr
```

The window to put this GDB command into looks like the following:



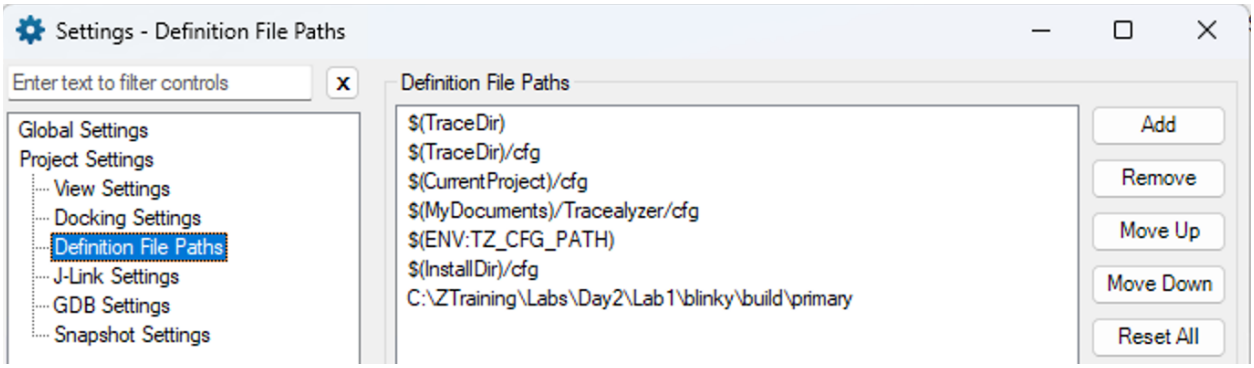
15) Once you've executed the command, go to the VS Code Explorer and look in your project directo ry. You should now see a trace.bin file.

16) Start Percepio View for Zephyr

17) Click Recording Settings on the right or File->Settings.

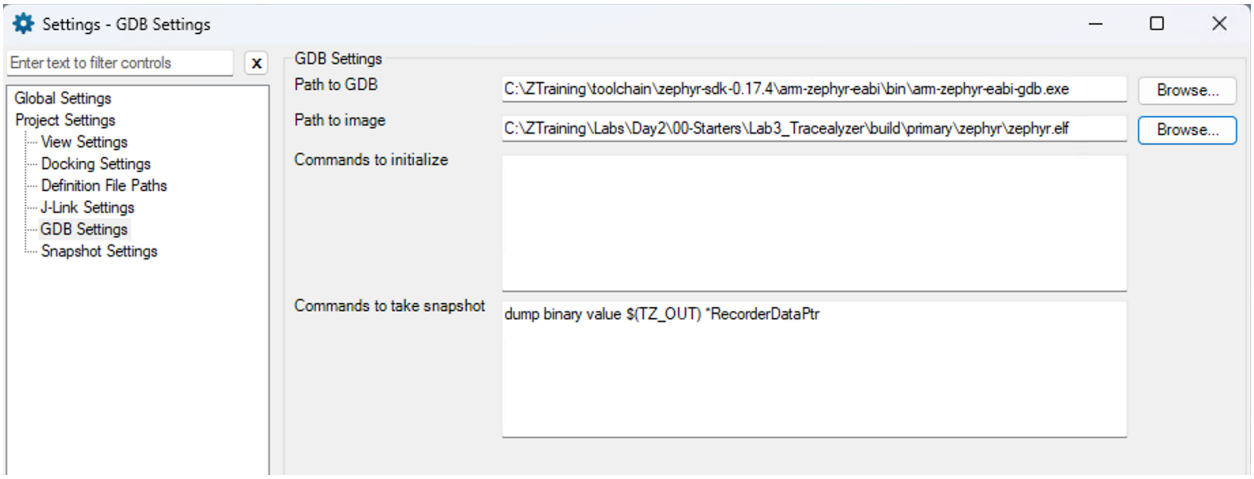
18) From the left menu, click Definition File Paths. View uses the syscall-v4.2.1.xml file that is genera ted by the toolchain when you compile your application. You can find the file in build\primary.

19) Add this path to the Definition File Paths. When I add mine, it looks like the following:



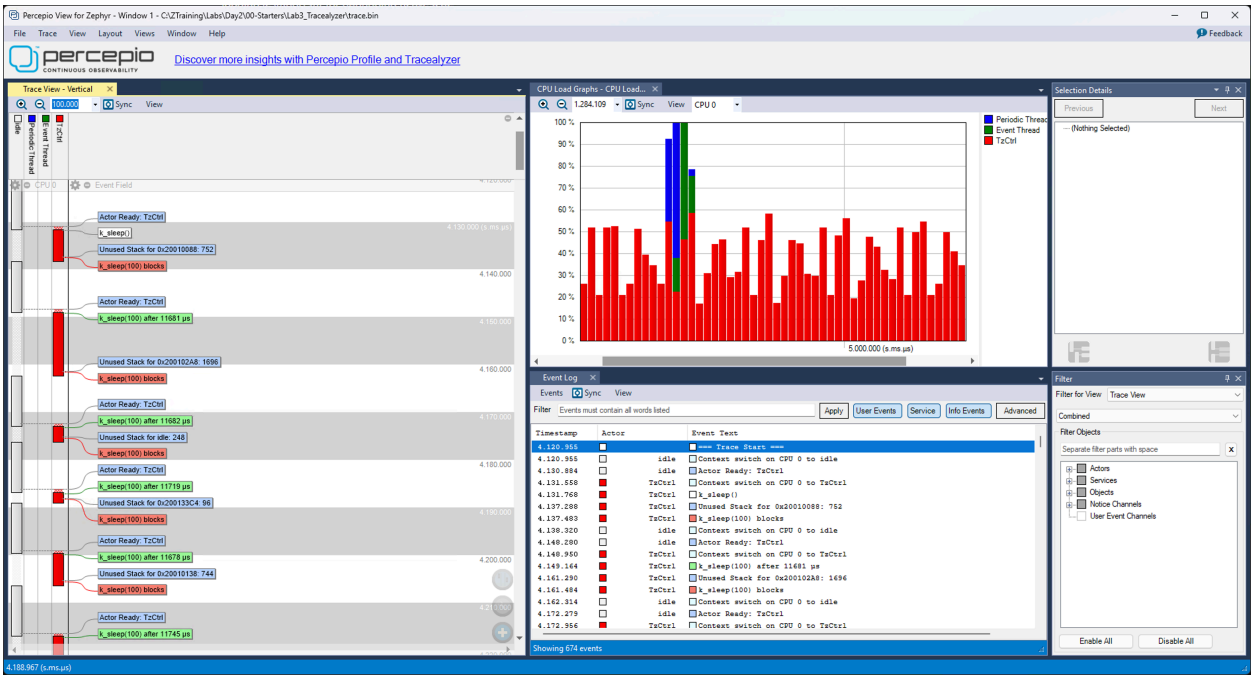
20) Next, click GDB Settings.

21) Add the path to your GDB and your image. Mine is as follows:

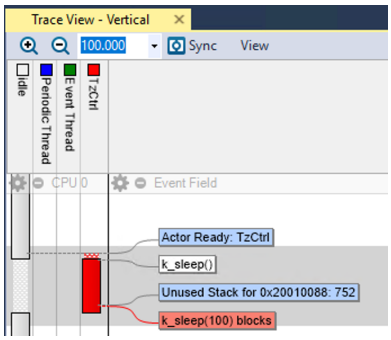


22) Click ok.

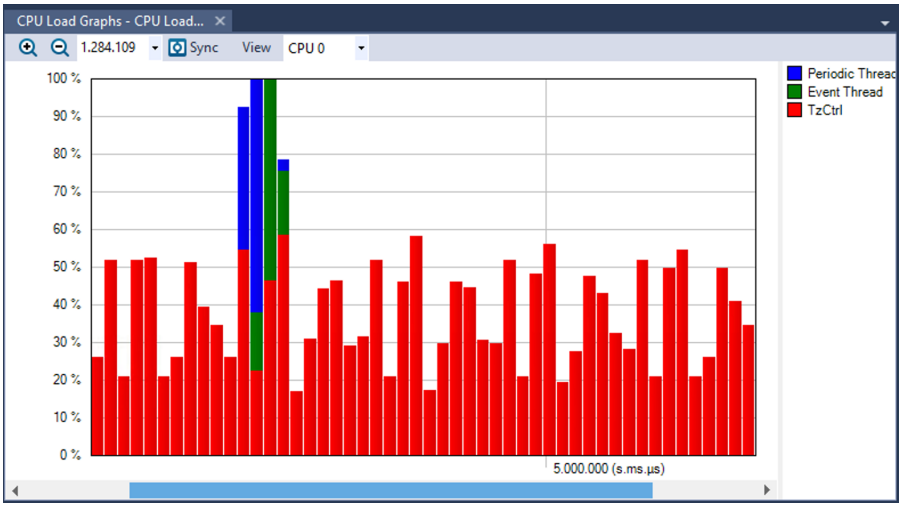
23) Now click File -> Open File. Navigate to your trace.bin file and open it. After a few moments, you should see something like the following:



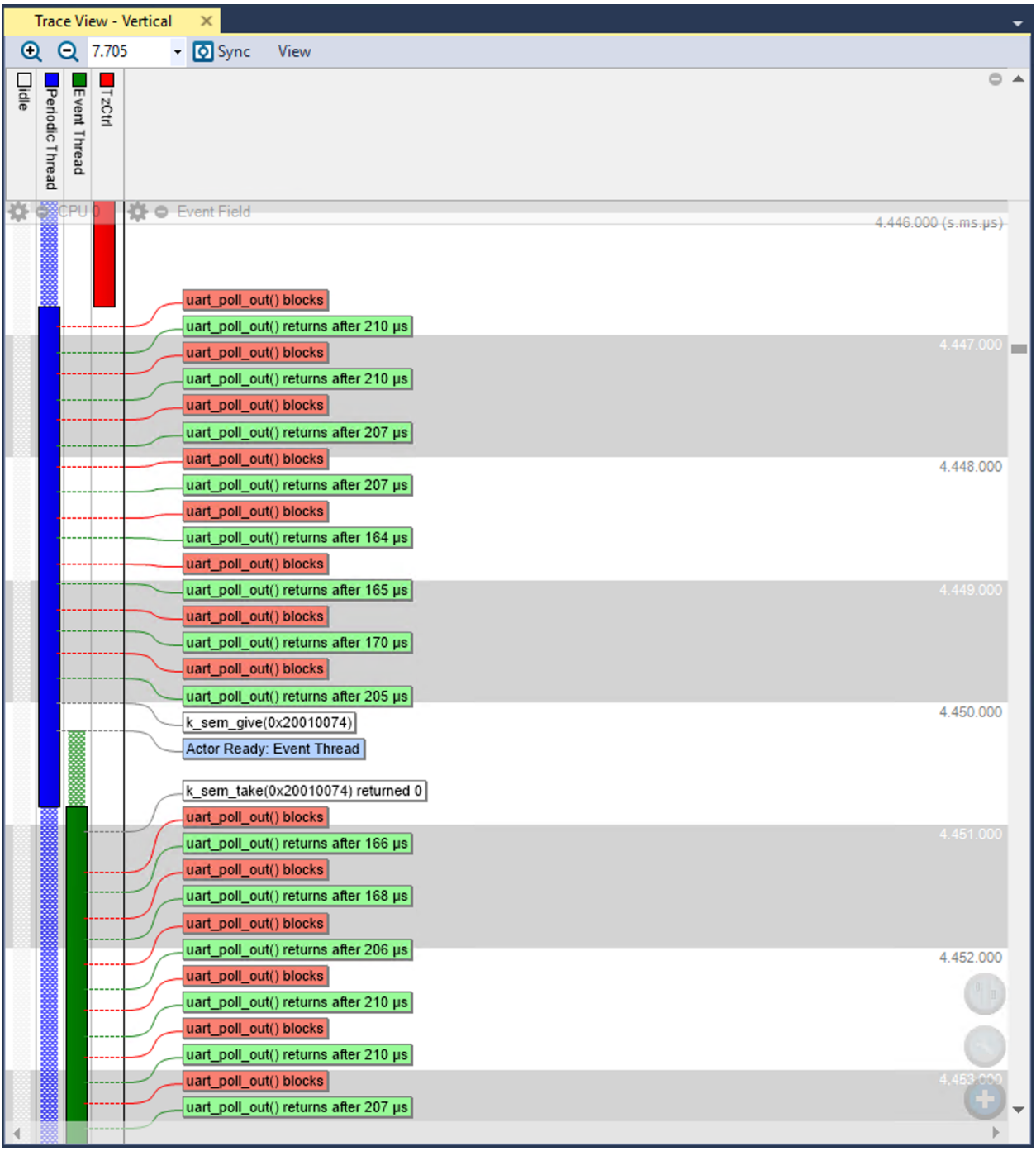
24) Notice how you can see the thread names, the idle thread, etc:



25) You can also see the CPU Load Graph:



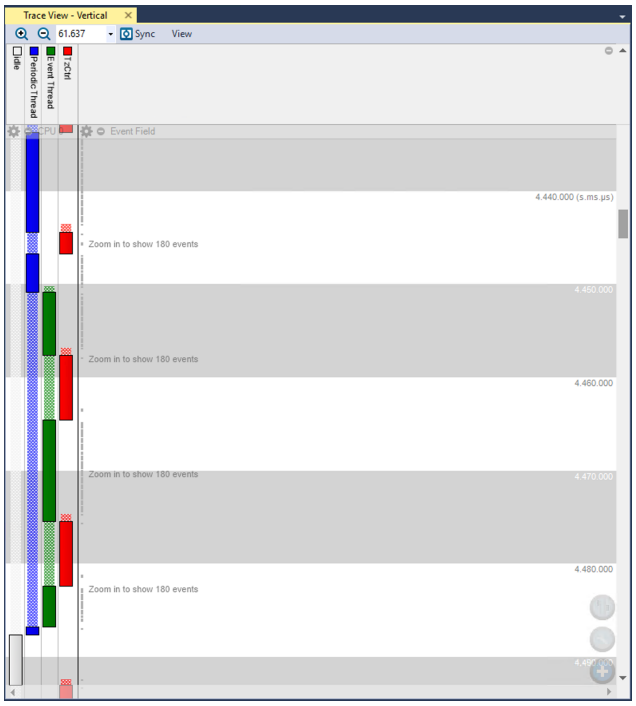
26) Navigate the vertical trace until you find when the periodic task fires. YOu should see the followi ng:



27) Notice when the Periodic Thread gives the semaphore, k_sem_give, the Event Thread becomes r eady to execute. The light green hashing show it's ready to run but not the highest priority yet.

28) Next, notice in the Event Thread how once it starts running, it notes that k_sem_take was execut ed.

29) We can use the trace to understand our code, how it's running, and identify issues. For example, look at this part of the trace below:



Our priorities are not set well! See how the TzCtrl task keeps interrupting our Periodic and Event threads? I would want to adjust the priorities of my tasks and TzCtrl task further so that they are not interrupted when they run.

Summary

Congratulations! You've configured Tracealyzer, captured trace data, and analyzed thread execution.

What You Accomplished

- Configured Tracealyzer in snapshot mode
- Enabled tracing with prj.conf settings
- Named threads for trace visibility
- Extracted trace buffer from RAM
- Analyzed thread execution timeline
- Examined CPU load distribution
- Identified context switches and scheduling patterns

Key Takeaways

Percepio View for Zephyr is essential for:

- Visualizing thread execution
- Debugging timing issues
- Understanding scheduling behavior
- Measuring CPU load
- Optimizing performance

Snapshot mode is ideal for:

- Quick debugging sessions
- Short-duration trace capture
- Systems without streaming infrastructure
- Learning and exploration

Thread naming is critical:

- Makes traces readable and understandable
- Always use k_thread_name_set()
- Use descriptive, meaningful names

CPU load analysis reveals:

- Which threads consume CPU time
- System capacity and headroom
- Optimization opportunities

Next Steps

Now that you've completed all four labs, you have the core skills for Zephyr RTOS development:

- **Lab 1**: Thread fundamentals
- **Lab 2**: Semaphore synchronization
- **Lab 3**: Percepio View for Zephyr

Advanced Topics to Explore:

- Message queues and mailboxes
- Work queues for deferred processing
- Memory pools and slabs
- Interrupt handling

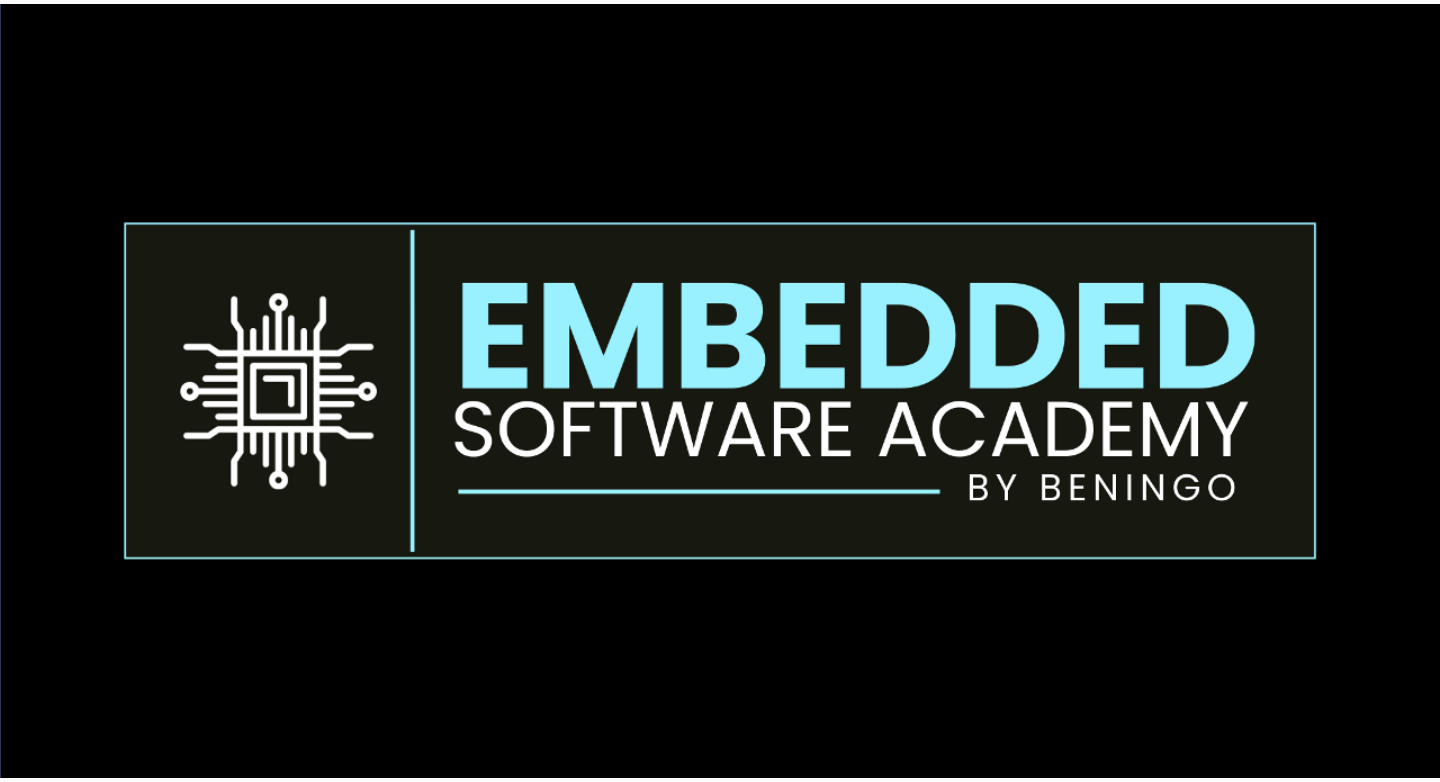
- Power management and low-power modes
- Device drivers and devicetree

Accelerate your RTOS project—consult with an expert to design, optimize, and implement real-time systems.

Are you struggling with project delays, rising costs, or an RTOS system that doesn't scale? We understand the pressure to deliver robust, real-time systems on time and within budget. Many teams face these challenges, but with the right guidance, you can overcome them.

Our expert consulting services help accelerate your RTOS project, ensuring your system is designed for scalability, maintainability, and optimal performance. Whether starting from scratch or refining an existing design, we'll help you create an architecture that meets your immediate needs and supports future growth, saving you money, reducing delays, and avoiding costly redesigns.

Contact jacob@beningo.com today to see how we can help you bring your RTOS projects to life.



Fast-track your career growth—get the expertise you need to deliver faster, better, and more reliable firmware.

Enhance your skills, streamline your processes, and elevate your architecture. Join my academy for on-demand, hands-on workshops and cutting-edge development resources designed to transform your career and keep you ahead of the curve.

What you'll get:

- Access to over eight hands-on Embedded Software Workshops
- Modernizing Embedded Software Core Courses
- Embedded Software Community Access
- Jacob's Webinar / Presentation Archive
- Embedded Development Q&A's with Jacob Beningo
- Embedded Software Development Resources

Learn more and subscribe by clicking here!

Level up your RTOS skills—design efficient, scalable embedded systems with expert-led training

Working with RTOS applications often leads to frustrating issues like poor performance, scalability issues, and debugging headaches. But it doesn't have to be that way. Our expert-led training helps you overcome these common challenges by teaching you how to design RTOS systems that are efficient, scalable, and ready for production.

Whether you're an individual developer looking to sharpen your skills or a team leader aiming to upskill your engineers, we've got you covered. With flexible training options—on-demand, live online, and customizable team workshops—you can learn how to avoid the pitfalls of RTOS design and build reliable, robust systems.

For more information on how we can help you level up your skills and streamline your RTOS development, contact **jacob@beningo.com** today!

BENINGO
EMBEDDED GROUP

THANK YOU

Let's Stay Connected

JacobBeningo

Jacob_Beningo

beningoembedded

Website | www.beningo.com
Contact | Jacob@beningo.com

EMBEDDED SOFTWARE ACADEMY
BY BENINGO