Zephyr RTOS Mini-Project Practice Sheet

A collection of 10 hands-on projects designed for all-round practice of Zephyr RTOS application development using QEMU emulation.

Prerequisites

- Zephyr SDK installed and configured
- QEMU installed for ARM emulation
- Familiarity with C programming
- Basic understanding of RTOS concepts

Target Board for QEMU: `qemu_cortex_m3` or `qemu_x86`

Build & Run Commands:

```
# Build
west build -b qemu_cortex_m3 -p always

# Run in QEMU
west build -t run
```

Project 1: Blinking LED Simulator

Difficulty: □ Beginner

Objective

Create a simulated LED blinker that prints "LED ON" and "LED OFF" messages alternately at 1-second intervals using a software timer.

Requirements

1. Use `K_TIMER_DEFINE()` to create a timer
2. Implement expiry callback to toggle LED state
3. Print current LED state on each toggle
4. LED should start in OFF state

Expected Output

```
[00:00:01.000] LED ON
[00:00:02.000] LED OFF
[00:00:03.000] LED ON
...
```

Hints

- Use a static boolean variable to track LED state
- Timer period should be 1000ms
- Use `printk()` for output

Learning Outcomes

- Software timer initialization and usage
- Callback functions in ISR context
- Basic Zephyr project structure

Project 2: Multi-Threaded Counter

Difficulty: □ Beginner

Objective

Create two threads that increment a shared counter. Thread A increments by 1, Thread B increments by 2. Print the counter value after each increment.

Requirements

1. Define two threads using `K_THREAD_DEFINE()`
2. Use a mutex to protect the shared counter
3. Each thread should have a 500ms delay between increments
4. Print which thread performed the increment
5. Run for 10 seconds total

Expected Output

```
[Thread A] Counter: 1
[Thread B] Counter: 3
[Thread A] Counter: 4
[Thread B] Counter: 6
...
```

Hints

- Use `K_MUTEX_DEFINE()` for the mutex
- Lock mutex before accessing counter, unlock after
- Use `k_msleep()` for delays
- Consider thread priorities

Learning Outcomes

- Thread creation and management

- Mutex usage for shared resource protection
- Thread synchronization basics

Project 3: Producer-Consumer with Message Queue

Difficulty: □□ Intermediate

Objective

Implement a producer-consumer pattern where one thread produces sensor readings and another thread consumes and processes them.

Requirements

1. Producer thread generates random "temperature" readings (20-40 range)
2. Consumer thread reads and calculates running average
3. Use message queue for communication
4. Producer generates data every 200ms
5. Consumer processes and prints every 500ms
6. Print queue status (items pending)

Expected Output

```
[Producer] Generated temp: 25
[Producer] Generated temp: 31
[Consumer] Received: 25, 31 | Average: 28.0
[Producer] Generated temp: 22
...
```

Hints

- Use `K_MSGQ_DEFINE()` with appropriate size
- Store temperature in a struct with timestamp
- Use `k_msgq_num_used_get()` for queue status
- Handle queue full/empty conditions

Learning Outcomes

- Message queue operations
- Inter-thread communication
- Data structure design for messaging

Project 4: Traffic Light State Machine

Difficulty: □□ Intermediate

Objective

Implement a traffic light controller using a state machine with timers. Simulate RED, YELLOW, and GREEN states with appropriate timings.

Requirements

1. States: RED (5s) → GREEN (4s) → YELLOW (1s) → RED...
2. Print state transitions with timestamps
3. Implement a pedestrian button interrupt simulation
4. When "button pressed", immediately transition to YELLOW (if GREEN)
5. Use a semaphore to signal button press from another thread

Expected Output

```
[00:00:00.000] Traffic Light: RED
[00:00:05.000] Traffic Light: GREEN
[00:00:07.500] PEDESTRIAN BUTTON PRESSED
[00:00:07.500] Traffic Light: YELLOW
[00:00:08.500] Traffic Light: RED
...
```

Hints

- Use enumeration for states
- Create separate timer for each state duration
- Button simulation thread gives semaphore every 7-15 seconds randomly
- Main thread takes semaphore with `K_NO_WAIT` to check

Learning Outcomes

- State machine implementation
- Semaphore signaling
- Event-driven programming
- Timer management

Project 5: UART Command Processor

Difficulty: □□ Intermediate

Objective

Create a UART-based command processor that accepts text commands and executes corresponding actions.

Requirements

1. Accept commands via UART input
2. Implement at least 5 commands:
    - `help` - List available commands

    - `status` - Print system uptime and thread info
    - `led on/off` - Toggle simulated LED
    - `count` - Print a running counter value
    - `echo <text>` - Echo back the text
3. Handle unknown commands gracefully
4. Support command history (last 3 commands)

Expected Output

```
> help
Available commands: help, status, led, count, echo
> status
Uptime: 5230 ms | Active threads: 2
> led on
LED is now ON
> echo Hello Zephyr
Hello Zephyr
> unknown
Error: Unknown command 'unknown'
```

Hints

- Use UART interrupt callback for receiving
- Parse commands using `strcmp()` or custom parser
- Store command history in circular buffer
- Use `k_uptime_get()` for uptime

Learning Outcomes

- UART driver usage
- String parsing and command handling
- Circular buffer implementation
- System information APIs

Project 6: Workqueue Task Scheduler

Difficulty: □□□ Advanced

Objective

Create a task scheduler using workqueues that processes different types of work items with priorities.

Requirements

1. Define 3 types of work items:
    - HIGH priority: Simulated "alarm" processing (immediate)
    - MEDIUM priority: Data logging (every 2 seconds)

    - LOW priority: Statistics calculation (every 5 seconds)
2. Use system workqueue for HIGH priority
3. Create custom workqueue for MEDIUM and LOW
4. Each work handler prints its execution with timestamp
5. Demonstrate work submission and execution order

Expected Output

```
[00:00:00.100] [HIGH] Alarm processed: Sensor threshold exceeded
[00:00:02.000] [MED] Data logged: temp=25, humidity=60
[00:00:02.050] [LOW] Stats: avg_temp=24.5 over 10 samples
[00:00:04.000] [MED] Data logged: temp=26, humidity=58
...
```

Hints

- Use `K_WORK_DEFINE()` and `k_work_submit()`
- Create delayable work for periodic tasks
- Use `k_work_queue_start()` for custom workqueue
- Pay attention to stack size for custom workqueue

Learning Outcomes

- Workqueue concepts and usage
- Deferred work processing
- Priority-based task execution
- Custom workqueue creation

Project 7: Ring Buffer Data Logger

Difficulty: □□□ Advanced

Objective

Implement a data logging system using a ring buffer that stores sensor samples and allows retrieval of historical data.

Requirements

1. Ring buffer stores last 20 sensor readings
2. Producer thread adds new readings every 100ms
3. Logger thread writes oldest 5 entries to "storage" every 1 second
4. Implement commands:
    - `log` - Dump current buffer contents
    - `stats` - Show buffer usage statistics
    - `clear` - Clear the buffer
5. Handle buffer overflow gracefully (overwrite oldest)

Expected Output

```
[Producer] Added sample #45: value=127
[Logger] Writing batch: [123, 124, 125, 126, 127]
> stats
Buffer: 15/20 used | Total samples: 45 | Overflows: 25
> log
[0] 126 [1] 127 [2] 128 [3] 129 [4] 130 ...
```

Hints

- Use `struct ring_buf` from `<zephyr/sys/ring_buffer.h>`
- Use `ring_buf_put()` and `ring_buf_get()`
- Track statistics in a separate struct
- Use mutex for thread-safe access

Learning Outcomes

- Ring buffer API usage
- Memory-efficient data storage
- Producer-consumer patterns
- Statistics tracking

Project 8: Event-Driven Sensor Monitor

Difficulty: □□□ Advanced

Objective

Create a sensor monitoring system using Zephyr's event objects to handle multiple sensor events concurrently.

Requirements

1. Simulate 3 sensors: Temperature, Humidity, Pressure
2. Each sensor has its own thread generating random values
3. Use events to signal:
    - `TEMP_READY` (bit 0)
    - `HUMID_READY` (bit 1)
    - `PRESSURE_READY` (bit 2)
    - `ALARM` (bit 3) - when any value exceeds threshold
4. Monitor thread waits for events and processes them
5. Support waiting for any event OR all events

Expected Output

```
[Sensor] Temp: 35°C (ALARM - threshold exceeded!)
[Monitor] Event received: TEMP_READY | ALARM
[Monitor] Processing temperature alarm...
[Sensor] Humidity: 65%
[Sensor] Pressure: 1013 hPa
[Monitor] Event received: HUMID_READY | PRESSURE_READY
[Monitor] All sensors read - logging data...
```

Hints

- Use `K_EVENT_DEFINE()` for event object
- Use `k_event_post()` to signal events
- Use `k_event_wait()` with appropriate flags
- `K_EVENT_WAIT_ANY` vs `K_EVENT_WAIT_ALL`

Learning Outcomes

- Event objects and event-driven design
- Bitwise event handling
- Multi-sensor data aggregation
- Alarm/threshold monitoring

Project 9: Priority Inversion Demonstration

Difficulty: □□□□ Expert

Objective

Create a demonstration of the priority inversion problem and show how mutex priority inheritance solves it.

Requirements

1. Create 3 threads:
   - HIGH priority (pri=1): Needs shared resource
   - MEDIUM priority (pri=3): CPU-intensive, no shared resource
   - LOW priority (pri=5): Holds shared resource
2. Demonstrate priority inversion scenario
3. Show execution timeline with timestamps
4. Run scenario twice:
   - First without priority inheritance
   - Then with priority inheritance enabled
5. Compare and print execution times

Expected Behavior

Without Priority Inheritance:

```
[LOW] Acquired mutex
[HIGH] Waiting for mutex...
[MED] Running CPU task (blocking HIGH indirectly)
[MED] Done
[LOW] Released mutex
[HIGH] Acquired mutex - waited 500ms!
```

With Priority Inheritance:

```
[LOW] Acquired mutex
[HIGH] Waiting for mutex (LOW boosted to pri=1)
[LOW] Released mutex
[HIGH] Acquired mutex - waited 50ms!
[MED] Running CPU task
```

Hints

- Use `k_thread_priority_get()` to show priority changes
- Careful timing setup to trigger inversion
- MEDIUM thread should have loop with `k_busy_wait()`
- Compare wall-clock times

Learning Outcomes

- Priority inversion understanding
- Priority inheritance mechanism
- Real-time scheduling concepts
- Performance measurement

Project 10: Mini Shell with Background Tasks

Difficulty: □□□□ Expert

Objective

Build a mini shell that can start, stop, and monitor background tasks, similar to a simple task manager.

Requirements

1. Implement shell commands:
   - `start <task_name>` - Start a background task
   - `stop <task_id>` - Stop a running task
   - `list` - List all tasks with status
   - `info <task_id>` - Show detailed task info
   - `mem` - Show memory usage

2. Support at least 4 concurrent background tasks:
    - `counter` - Continuously incrementing counter
    - `blinker` - LED blink simulation
    - `monitor` - System resource monitor
    - `logger` - Periodic log writer
3. Tasks should be suspendable/resumable
4. Show task CPU time estimation

Expected Output

```
shell> start counter
Task 'counter' started with ID=1
shell> start blinker
Task 'blinker' started with ID=2
shell> list
ID  NAME      STATUS    RUNTIME
1   counter   RUNNING   1250ms
2   blinker   RUNNING   800ms
shell> stop 1
Task 1 stopped
shell> info 2
Task: blinker
Status: RUNNING
Priority: 5
Stack: 512/1024 used
Runtime: 2100ms
shell> mem
Total heap: 8192 bytes
Used: 2048 bytes (25%)
Free: 6144 bytes
```

Hints

- Use array of thread structures for task management
- `k_thread_suspend()` and `k_thread_resume()` for task control
- Track task start time for runtime calculation
- Use `k_thread_stack_space_get()` for stack usage
- Consider using thread names via `k_thread_name_set()`

Learning Outcomes

- Dynamic thread management
- System monitoring and introspection
- Shell implementation patterns
- Resource tracking and management

Bonus Challenges

Challenge A: Add Persistence

Modify Project 7 to save logged data to simulated flash storage using Zephyr's flash API or settings subsystem.

Challenge B: Network Integration

Extend Project 5 to also accept commands via a simulated network socket (for `qemu_x86` with networking).

Challenge C: Power Management

Add sleep modes to Project 8 where the system enters low-power state when no sensor activity occurs for 10 seconds.

Project Template

Use this template structure for each project:

```
project_name/
├── CMakeLists.txt
├── prj.conf
├── src/
│   └── main.c
└── run_qemu.sh
```

CMakeLists.txt:

```
cmake_minimum_required(VERSION 3.20.0)
find_package(Zephyr REQUIRED HINTS $ENV{ZEPHYR_BASE})
project(project_name)
target_sources(app PRIVATE src/main.c)
```

prj.conf:

```
CONFIG_PRINTK=y
CONFIG_HEAP_MEM_POOL_SIZE=4096
# Add project-specific configs
```

run_qemu.sh:

```
#!/bin/bash
west build -b qemu_cortex_m3 -p always
west build -t run
```

Evaluation Criteria

For each project, evaluate based on:

Criteria	Weight
Correct functionality	40%
Code quality & organization	20%
Proper use of Zephyr APIs	20%
Error handling	10%
Documentation/comments	10%

Additional Resources

- Zephyr Project Documentation
- Zephyr API Reference
- Zephyr Samples
- Local guides in this repository:
  - `Zephyr_Threads_And_Synchronization_Guide.md`
  - `Zephyr_Software_Timer_Guide.md`
  - `Zephyr_Devicetree_Overlay_Guide.md`
  - `Zephyr_Project_Configuration_Guide.md`

Happy Coding with Zephyr RTOS!