Zephyr RTOS: Threads and Synchronization Primitives Guide

Table of Contents

Thread Setup and Creation

Basic Thread Creation

In Zephyr, threads are created using the `K_THREAD_DEFINE()` macro or dynamically using `k_thread_create()`.

Static Thread Definition (Recommended)

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

// Define a thread stack (memory for the thread)
K_THREAD_STACK_DEFINE(my_thread_stack, 1024);

// Define the thread
struct k_thread my_thread_data;

// Thread function
void my_thread_func(void *arg1, void *arg2, void *arg3)
{
    // Thread code here
    printk("Thread running with arg1: %p\n", arg1);
}

void main(void)
{
    // Create the thread
    k_thread_create(&my_thread_data, my_thread_stack, 1024,
                    my_thread_func,
                    (void *)1, (void *)2, NULL,  // Arguments
                    0, 0, K_NO_WAIT);             // Priority, options,
delay
}
```

Macro-based Thread Definition (Simpler)

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

K_THREAD_DEFINE(my_thread, 1024,
                my_thread_func, (void *)1, (void *)2, NULL,
                0, 0, 0);  // Priority, options, delay

void my_thread_func(void *arg1, void *arg2, void *arg3)
{
    // Thread code here
}
```

Dynamic Thread Creation

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

K_THREAD_STACK_DEFINE(my_dynamic_stack, 2048);
struct k_thread my_dynamic_thread;

void create_thread_dynamically(void)
{
    k_thread_create(&my_dynamic_thread,
                    my_dynamic_stack, 2048,
                    my_thread_func,
                    NULL, NULL, NULL,
                    5, 0, K_NO_WAIT);
}
```

Thread Parameters

Parameter	Description	Example
Stack Size	Memory allocated for thread stack	1024 bytes, 2048 bytes
Priority	Thread priority (-128 to 127)	0 (cooperative), 5 (preemptive)
Options	Thread flags	0, K_ESSENTIAL, K_FP_REGS
Delay	Delay before thread starts	K_NO_WAIT, K_MSEC(100)

Thread Lifecycle

Thread States

```
CREATED → READY → RUNNING → WAITING → READY → ... → DEAD
```

- **CREATED**: Thread object created but not yet scheduled
- **READY**: Thread is ready to run but not currently running
- **RUNNING**: Thread is currently executing
- **WAITING**: Thread is blocked (waiting on synchronization primitive, delay, etc.)
- **DEAD**: Thread has exited

Thread Control Functions

```
// Suspend the current thread for a duration
k_msleep(1000);          // Sleep for 1000 milliseconds
k_usleep(1000);          // Sleep for 1000 microseconds

// Get thread information
k_thread_name_set(&my_thread, "worker");
const char *name = k_thread_name_get(&my_thread);

// Exit current thread
k_thread_abort(&my_thread);
return;  // Thread function returns

// Get current thread
struct k_thread *current = k_current_get();

// Query thread status
int priority = k_thread_priority_get(&my_thread);
k_thread_priority_set(&my_thread, 10);  // Change priority
```

Thread Properties

Priority Levels

- **Cooperative Threads**: Priority 0 to -128
  - Non-preemptible, only yield voluntarily
  - Lower priority values = higher priority
- **Preemptive Threads**: Priority 1 to 127
  - Can be preempted by higher priority threads
  - Higher priority values = higher priority

```
// Cooperative thread (priority 0)
k_thread_create(&thread, stack, 1024,
                my_func, NULL, NULL, NULL,
                0, 0, K_NO_WAIT);

// Preemptive thread (priority 5)
k_thread_create(&thread, stack, 1024,
```

```
                my_func, NULL, NULL, NULL,
                5, 0, K_NO_WAIT);

// High priority preemptive (priority 1)
k_thread_create(&thread, stack, 1024,
                my_func, NULL, NULL, NULL,
                1, 0, K_NO_WAIT);
```

Thread Options

```
// K_ESSENTIAL: Thread is essential (system won't panic if it dies)
// K_FP_REGS: Thread uses floating-point (needs FPU context)
// K_INHERIT_PERMS: Thread inherits object permissions

k_thread_create(&thread, stack, 1024,
                my_func, NULL, NULL, NULL,
                5, K_ESSENTIAL | K_FP_REGS, K_NO_WAIT);
```

Thread Usage Patterns

Pattern 1: Worker Thread

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

K_THREAD_STACK_DEFINE(worker_stack, 1024);
struct k_thread worker_thread;

void worker_func(void *arg1, void *arg2, void *arg3)
{
    while (1) {
        printk("Worker thread working...\n");
        k_msleep(1000);
    }
}

void main(void)
{
    k_thread_create(&worker_thread, worker_stack, 1024,
                    worker_func, NULL, NULL, NULL,
                    5, 0, K_NO_WAIT);

    // Main thread continues
    while (1) {
        printk("Main thread...\n");
        k_msleep(500);
    }
}
```

Pattern 2: Thread with Event Waiting

```c
#include <zephyr/kernel.h>

K_EVENT_DEFINE(my_event);
K_THREAD_STACK_DEFINE(listener_stack, 1024);
struct k_thread listener_thread;

void listener_func(void *arg1, void *arg2, void *arg3)
{
    while (1) {
        // Wait for event (bits 0 and 1)
        uint32_t events = k_event_wait(&my_event, 0x03, false, K_FOREVER);
        printk("Event received: 0x%x\n", events);
    }
}

void main(void)
{
    k_thread_create(&listener_thread, listener_stack, 1024,
                    listener_func, NULL, NULL, NULL,
                    5, 0, K_NO_WAIT);

    k_msleep(2000);
    k_event_post(&my_event, 0x01);  // Signal event
}
```

Pattern 3: Multiple Threads Communication

```c
#include <zephyr/kernel.h>

K_MSGQ_DEFINE(my_msgq, sizeof(uint32_t), 10, 4);
K_THREAD_STACK_DEFINE(producer_stack, 1024);
K_THREAD_STACK_DEFINE(consumer_stack, 1024);

void producer_func(void *arg1, void *arg2, void *arg3)
{
    uint32_t count = 0;
    while (1) {
        k_msgq_put(&my_msgq, &count, K_MSEC(100));
        printk("Producer sent: %u\n", count++);
        k_msleep(100);
    }
}

void consumer_func(void *arg1, void *arg2, void *arg3)
{
    uint32_t msg;
    while (1) {
        if (k_msgq_get(&my_msgq, &msg, K_FOREVER) == 0) {
```

```
            printk("Consumer received: %u\n", msg);
        }
    }
}

void main(void)
{
    k_thread_create(&producer_thread, producer_stack, 1024,
                    producer_func, NULL, NULL, NULL,
                    5, 0, K_NO_WAIT);

    k_thread_create(&consumer_thread, consumer_stack, 1024,
                    consumer_func, NULL, NULL, NULL,
                    5, 0, K_NO_WAIT);
}
```

Synchronization Primitives

1. Mutex (Mutual Exclusion Lock)

A mutex ensures only one thread can access a critical section at a time. Supports priority inheritance to prevent priority inversion.

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

K_MUTEX_DEFINE(my_mutex);

// Critical resource
int shared_counter = 0;

void thread_func(void *arg1, void *arg2, void *arg3)
{
    while (1) {
        // Lock the mutex
        k_mutex_lock(&my_mutex, K_FOREVER);

        // Critical section - only one thread at a time
        shared_counter++;
        printk("Counter: %d\n", shared_counter);

        // Unlock the mutex
        k_mutex_unlock(&my_mutex);

        k_msleep(100);
    }
}
```

Key Features:

- Supports priority inheritance
- Recursive locking available with `K_RECURSIVE_MUTEX_DEFINE()`
- Prevents deadlocks with timeout handling

```
// Mutex with timeout
int ret = k_mutex_lock(&my_mutex, K_MSEC(100));
if (ret == 0) {
    // Lock acquired
    k_mutex_unlock(&my_mutex);
} else {
    printk("Timeout waiting for mutex\n");
}

// Recursive mutex (same thread can lock multiple times)
K_RECURSIVE_MUTEX_DEFINE(recursive_mutex);
k_mutex_lock(&recursive_mutex, K_FOREVER);
k_mutex_lock(&recursive_mutex, K_FOREVER);  // Same thread, OK
k_mutex_unlock(&recursive_mutex);
k_mutex_unlock(&recursive_mutex);
```

2. Semaphore

A semaphore maintains a counter and is used for controlling access to shared resources or signaling between threads.

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

// Binary semaphore (initial count = 1, max count = 1)
K_SEMAPHORE_DEFINE(binary_sem, 1, 1);

// Counting semaphore (initial count = 0, max count = 10)
K_SEMAPHORE_DEFINE(counting_sem, 0, 10);

void producer_func(void *arg1, void *arg2, void *arg3)
{
    while (1) {
        // Produce something
        printk("Producing...\n");
        k_msleep(500);

        // Signal consumer
        k_sem_give(&counting_sem);
    }
}

void consumer_func(void *arg1, void *arg2, void *arg3)
{
    while (1) {
        // Wait for producer
        k_sem_take(&counting_sem, K_FOREVER);
```

```
        printk("Consuming...\n");
    }
}
```

Semaphore vs Mutex:

Feature	Mutex	Semaphore
Ownership	Thread-owned	No ownership
Priority Inheritance	Yes	No
Recursive	Can be	No
Use Case	Mutual exclusion	Signaling, Resource counting

3. Condition Variable

A condition variable allows threads to wait for a specific condition, released by other threads.

```c
#include <zephyr/kernel.h>

K_MUTEX_DEFINE(cond_mutex);
K_CONDVAR_DEFINE(my_condvar);

int condition = 0;

void waiter_func(void *arg1, void *arg2, void *arg3)
{
    k_mutex_lock(&cond_mutex, K_FOREVER);

    while (!condition) {
        printk("Waiting for condition...\n");
        // Wait and release mutex atomically
        k_condvar_wait(&my_condvar, &cond_mutex, K_FOREVER);
    }

    printk("Condition met!\n");
    k_mutex_unlock(&cond_mutex);
}

void signaler_func(void *arg1, void *arg2, void *arg3)
{
    k_msleep(2000);

    k_mutex_lock(&cond_mutex, K_FOREVER);
    condition = 1;
    // Wake up one waiting thread
    k_condvar_signal(&my_condvar);
    // Or wake up all waiting threads
    // k_condvar_broadcast(&my_condvar);
```

```
    k_mutex_unlock(&cond_mutex);
}
```

4. Event Object

Event objects allow one or more threads to wait for one or more bits to be set by other threads.

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

K_EVENT_DEFINE(my_event);

void waiter_func(void *arg1, void *arg2, void *arg3)
{
    while (1) {
        // Wait for bits 0 or 1 (auto-clear after wait)
        uint32_t events = k_event_wait(&my_event, 0x03, false, K_FOREVER);
        printk("Event received: 0x%x\n", events);
    }
}

void signaler_func(void *arg1, void *arg2, void *arg3)
{
    k_msleep(1000);

    // Post event bit 0
    k_event_post(&my_event, 0x01);

    k_msleep(1000);

    // Post event bit 1
    k_event_post(&my_event, 0x02);
}

// Advanced usage with flags
void advanced_waiter_func(void *arg1, void *arg2, void *arg3)
{
    // Wait for bits 0 AND 1 (both must be set)
    uint32_t events = k_event_wait(&my_event, 0x03, true, K_FOREVER);

    // Wait for bits 0 OR 1 (either can be set, auto-clear)
    events = k_event_wait(&my_event, 0x03, false, K_TIMEOUT_MS_INFINITE);

    // Check pending bits without waiting
    uint32_t pending = k_event_test(&my_event, 0x03);
}
```

5. Spinlock

A spinlock is a low-level synchronization primitive where threads busy-wait. Used in interrupt contexts and performance-critical sections.

```c
#include <zephyr/kernel.h>

K_SPINLOCK_DEFINE(my_spinlock);

int shared_value = 0;

void thread_func(void *arg1, void *arg2, void *arg3)
{
    k_spinlock_key_t key = k_spin_lock(&my_spinlock);

    // Critical section
    shared_value++;

    k_spin_unlock(&my_spinlock, key);
}

// In interrupt context
void my_irq_handler(void *unused)
{
    k_spinlock_key_t key = k_spin_lock(&my_spinlock);

    // ISR critical section
    shared_value++;

    k_spin_unlock(&my_spinlock, key);
}
```

6. Message Queue

A message queue allows threads to exchange fixed-size messages asynchronously.

```c
#include <zephyr/kernel.h>

// Define a message queue: sizeof(item), max items, alignment
K_MSGQ_DEFINE(my_msgq, sizeof(uint32_t), 10, 4);

// Or with custom structure
struct message {
    uint32_t value;
    char text[32];
};

K_MSGQ_DEFINE(custom_msgq, sizeof(struct message), 5, 4);

void producer_func(void *arg1, void *arg2, void *arg3)
{
    uint32_t msg = 0;
    while (1) {
        // Put message with timeout
        int ret = k_msgq_put(&my_msgq, &msg, K_MSEC(100));
```

```
        if (ret == 0) {
            printk("Message sent: %u\n", msg);
        } else if (ret == -ENOMSG) {
            printk("Queue full\n");
        }
        msg++;
        k_msleep(100);
    }
}

void consumer_func(void *arg1, void *arg2, void *arg3)
{
    uint32_t msg;
    while (1) {
        // Get message with timeout
        if (k_msgq_get(&my_msgq, &msg, K_FOREVER) == 0) {
            printk("Message received: %u\n", msg);
        }
    }
}

// Useful functions
k_msgq_num_used_get(&my_msgq);   // Get number of messages in queue
k_msgq_reset(&my_msgq);           // Clear all messages
```

Practical Examples

Example 1: Thread-Safe Counter

```
#include <zephyr/kernel.h>
#include <zephyr/device.h>

K_MUTEX_DEFINE(counter_mutex);
int counter = 0;
const int TARGET_COUNT = 100;

K_THREAD_STACK_DEFINE(incrementer1_stack, 512);
K_THREAD_STACK_DEFINE(incrementer2_stack, 512);
K_THREAD_STACK_DEFINE(printer_stack, 512);

void incrementer_func(void *arg1, void *arg2, void *arg3)
{
    int thread_id = (int)arg1;

    while (counter < TARGET_COUNT) {
        k_mutex_lock(&counter_mutex, K_FOREVER);

        if (counter < TARGET_COUNT) {
            counter++;
            printk("Thread %d incremented counter to %d\n", thread_id,
```

```
counter);
        }

        k_mutex_unlock(&counter_mutex);
        k_msleep(10);
    }
}

void printer_func(void *arg1, void *arg2, void *arg3)
{
    while (counter < TARGET_COUNT) {
        k_mutex_lock(&counter_mutex, K_FOREVER);
        printk("Current counter value: %d\n", counter);
        k_mutex_unlock(&counter_mutex);

        k_msleep(500);
    }
}

void main(void)
{
    k_thread_create(&incrementer1, incrementer1_stack, 512,
                    incrementer_func, (void *)1, NULL, NULL,
                    5, 0, K_NO_WAIT);

    k_thread_create(&incrementer2, incrementer2_stack, 512,
                    incrementer_func, (void *)2, NULL, NULL,
                    5, 0, K_NO_WAIT);

    k_thread_create(&printer, printer_stack, 512,
                    printer_func, NULL, NULL, NULL,
                    6, 0, K_NO_WAIT);
}
```

Example 2: Producer-Consumer Pattern

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

#define BUFFER_SIZE 100
#define ITEM_SIZE sizeof(uint32_t)

K_MSGQ_DEFINE(buffer, ITEM_SIZE, BUFFER_SIZE, 4);
K_THREAD_STACK_DEFINE(producer_stack, 1024);
K_THREAD_STACK_DEFINE(consumer_stack, 1024);

void producer_func(void *arg1, void *arg2, void *arg3)
{
    uint32_t item = 0;

    while (1) {
        // Simulate production
```

```
        k_msleep(100);

        // Send item to consumer
        int ret = k_msgq_put(&buffer, &item, K_MSEC(100));
        if (ret == 0) {
            printk("Produced: %u\n", item);
        } else {
            printk("Buffer full, skipping\n");
        }
        item++;
    }
}

void consumer_func(void *arg1, void *arg2, void *arg3)
{
    uint32_t item;

    while (1) {
        // Wait for item
        if (k_msgq_get(&buffer, &item, K_FOREVER) == 0) {
            // Simulate consumption
            printk("Consumed: %u\n", item);
            k_msleep(200);
        }
    }
}

void main(void)
{
    k_thread_create(&producer, producer_stack, 1024,
                    producer_func, NULL, NULL, NULL,
                    5, 0, K_NO_WAIT);

    k_thread_create(&consumer, consumer_stack, 1024,
                    consumer_func, NULL, NULL, NULL,
                    6, 0, K_NO_WAIT);
}
```

Example 3: Thread Synchronization with Barriers

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

#define NUM_THREADS 3
K_BARRIER_DEFINE(sync_barrier, NUM_THREADS);
K_THREAD_STACK_DEFINE(worker_stacks[NUM_THREADS], 512);
struct k_thread workers[NUM_THREADS];

void worker_func(void *arg1, void *arg2, void *arg3)
{
    int id = (int)arg1;
```

```
    // Phase 1: Each thread does its work
    printk("Thread %d starting phase 1\n", id);
    k_msleep(100 * (id + 1));  // Different timing
    printk("Thread %d done with phase 1\n", id);

    // Wait for all threads to finish phase 1
    k_barrier_wait(&sync_barrier);

    // Phase 2: All threads start together
    printk("Thread %d starting phase 2\n", id);
    k_msleep(50);
    printk("Thread %d done with phase 2\n", id);
}

void main(void)
{
    for (int i = 0; i < NUM_THREADS; i++) {
        k_thread_create(&workers[i], worker_stacks[i], 512,
                        worker_func, (void *)i, NULL, NULL,
                        5, 0, K_NO_WAIT);
    }
}
```

Best Practices

1. Stack Size Planning

```
// Rule of thumb: 512-2048 bytes for most tasks
// Monitor with: printk("Stack usage: %u/%u\n",
//                      k_thread_stack_space_get(&my_thread), 1024);

K_THREAD_STACK_DEFINE(critical_task_stack, 2048);  // More complex tasks
K_THREAD_STACK_DEFINE(simple_task_stack, 512);     // Simple tasks
```

2. Priority Assignment

```
// Typical priority scheme:
// -1 to 0: Cooperative (idle-like) threads
// 1-3: High priority real-time tasks
// 4-7: Medium priority tasks
// 8+: Lower priority background tasks

const int PRIORITY_CRITICAL = 1;
const int PRIORITY_HIGH = 3;
const int PRIORITY_NORMAL = 5;
const int PRIORITY_LOW = 8;
```

3. Avoiding Deadlocks

```
// Bad: Two mutexes acquired in different orders
// Thread A: lock(m1), lock(m2)
// Thread B: lock(m2), lock(m1)  // Potential deadlock!

// Good: Always acquire mutexes in same order
// Thread A: lock(m1), lock(m2)
// Thread B: lock(m1), lock(m2)  // Safe

// Good: Use timeouts to prevent indefinite blocking
k_mutex_lock(&mutex, K_MSEC(1000));
```

4. Critical Section Best Practices

```
// Minimize critical section length
K_MUTEX_DEFINE(critical_mutex);

void good_critical_section(void)
{
    // Prepare data outside critical section
    int temp_value = calculate_value();

    k_mutex_lock(&critical_mutex, K_FOREVER);
    // Only access shared data inside critical section
    shared_data = temp_value;
    k_mutex_unlock(&critical_mutex);

    // Process results outside critical section
    process_result(shared_data);
}
```

5. Thread Naming for Debugging

```
k_thread_name_set(&my_thread, "worker_0");
printk("Thread: %s (priority: %d)\n",
       k_thread_name_get(&my_thread),
       k_thread_priority_get(&my_thread));
```

6. Proper Error Handling

```
void proper_synchronization(void)
{
    // Always handle return values
    if (k_mutex_lock(&my_mutex, K_MSEC(100)) == 0) {
```

```
        // Lock acquired
        on_shared_data();
        k_mutex_unlock(&my_mutex);
    } else {
        // Timeout occurred
        printk("Failed to acquire mutex\n");
        // Handle error appropriately
    }
}
```

7. Thread Lifecycle Management

```
// Use K_ESSENTIAL for critical threads
k_thread_create(&critical, stack, 1024,
                critical_func, NULL, NULL, NULL,
                1, K_ESSENTIAL, K_NO_WAIT);

// Use K_FP_REGS if thread uses floating-point
k_thread_create(&fp_thread, stack, 1024,
                fp_func, NULL, NULL, NULL,
                5, K_FP_REGS, K_NO_WAIT);

// Abort thread if needed (use sparingly!)
k_thread_abort(&my_thread);
```

8. Synchronization Selection Guide

```
Use MUTEX when:
  - Need mutual exclusion of a resource
  - Multiple threads access shared data
  - Thread ownership matters (priority inheritance)

Use SEMAPHORE when:
  - Need simple signaling between threads
  - Resource counting (e.g., buffer slots)
  - No priority inheritance needed

Use CONDITION VARIABLE when:
  - Threads wait for a specific condition
  - Condition is protected by a mutex
  - Multiple waiter threads

Use EVENT when:
  - Simple bit-based signaling
  - Immediate-return wait checking needed
  - Multiple independent event bits

Use MESSAGE QUEUE when:
  - Threads exchange data/commands
```

```
  - FIFO ordering required
  - Asynchronous communication needed

Use SPINLOCK when:
  - Interrupt context synchronization
  - Very short critical sections
  - Not suitable for thread context (busy-wait)
```

Configuration Requirements

Add these to your `prj.conf`:

```
# Enable threading features
CONFIG_MULTITHREADING=y

# Enable specific synchronization primitives
CONFIG_MUTEX=y
CONFIG_SEMAPHORE=y
CONFIG_EVENTS=y
CONFIG_MSGQ=y
CONFIG_CONDVAR=y
CONFIG_BARRIER=y

# Thread stack statistics (for debugging)
CONFIG_THREAD_STACK_INFO=y
CONFIG_THREAD_NAME=y

# Kernel timing
CONFIG_SYS_CLOCK_HW_CYCLES_PER_SEC=1000000
```

References

- Zephyr Kernel Services Documentation
- Zephyr Kernel API Reference
- Thread API: `<zephyr/kernel.h>`
- Synchronization Primitives: All in `<zephyr/kernel.h>`