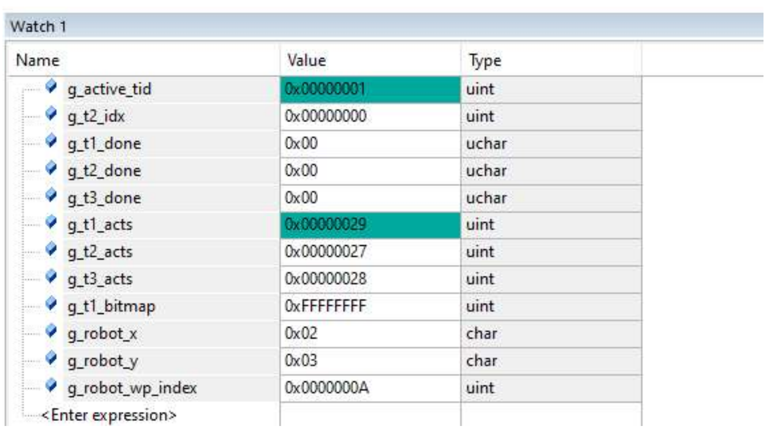
# Table of Contents

# Introduction

This lab had two aims. First, to build and verify a time-sliced round-robin scheduler on the LPC17xx using CMSIS-RTOS RTX with a configured quantum of about fifteen milliseconds. Second, to design a compact operating system style workload with realistic inter-task dependencies and shared resources, then measure and explain its behavior. I produced two builds for each part of the assignment. The demo build makes execution visible on hardware through LEDs and the LCD, with deliberate two-second windows so the active thread is easy to see. The analysis build removes board I/O and instead exposes watchable state variables so that the Watch Window, Event Viewer, Performance Analyzer, and System or Thread Viewer can show the scheduler itself. For Question 1, I implemented three finite and nontrivial tasks, a slice programmable LCD painter, a Morse-style symbol worker, and a robotics themed waypoint follower. For Question 2, I implemented five finite roles, Memory Management, CPU Management, Application Interface, Device Management, and User Interface, coordinated using signals and a mutex protected logger, together with an SRAM bit-band exercise and a short monitor thread to bound execution time. Across both questions, I verified timing, ordering, mutual exclusion, and clean termination.

# Question 1 - Analysis

I started by running the analysis build and opening the Watch window to confirm that each thread was doing real, finite work. In **Figure 1**, you can see the live counters and state variables changing as the system runs. The counters for the painter, Morse, and robot tasks (g_t1_acts, g_t2_acts, and g_t3_acts) advance steadily and close to each other, which is what you expect when all three threads share the CPU fairly under RR. The painter's bitmap (g_t1_bitmap) shows all low 32 columns painted, and the robot pose (g_robot_x, g_robot_y) moves toward successive waypoints while the waypoint index increments. This snapshot also shows g_active_tid flipping among 1, 2, and 3 during the run, which is a quick sanity check that the scheduler is alternating the runnable threads.


Watch 1

| Name | Value | Type |
|---|---|---|
| g_active_tid | 0x00000001 | uint |
| g_t2_idx | 0x00000000 | uint |
| g_t1_done | 0x00 | uchar |
| g_t2_done | 0x00 | uchar |
| g_t3_done | 0x00 | uchar |
| g_t1_acts | 0x00000029 | uint |
| g_t2_acts | 0x00000027 | uint |
| g_t3_acts | 0x00000028 | uint |
| g_t1_bitmap | 0xFFFFFFFF | uint |
| g_robot_x | 0x02 | char |
| g_robot_y | 0x03 | char |
| g_robot_wp_index | 0x0000000A | uint |
| <Enter expression> | | |

**Figure 1.** Watch Window showing live counters and state variables.

To quantify fairness, I captured the Performance Analyzer in **Figure 2**. While all three application threads are alive, their CPU time bars grow at nearly the same rate, reflecting an even split of processor time. Any small differences come from the per-activation work each task does, but the overall share is balanced, which matches the design goal for RR with equal priorities and short quanta.


**Figure 2.** Performance Analyzer indicating near-equal CPU share under round-robin.

The RR quantum itself was verified in the Event Viewer. **Figure 3** zooms in on a single thread's slices; the tooltip over Thread_Painter reports a duration of about 15.0 ms for each contiguous run, with a minimum/maximum tightly bracketing 15 ms. That's exactly the configured RR timeout (3 ticks × 5 ms tick). The staircase pattern confirms that threads are being preempted by the scheduler at the end of each quantum.

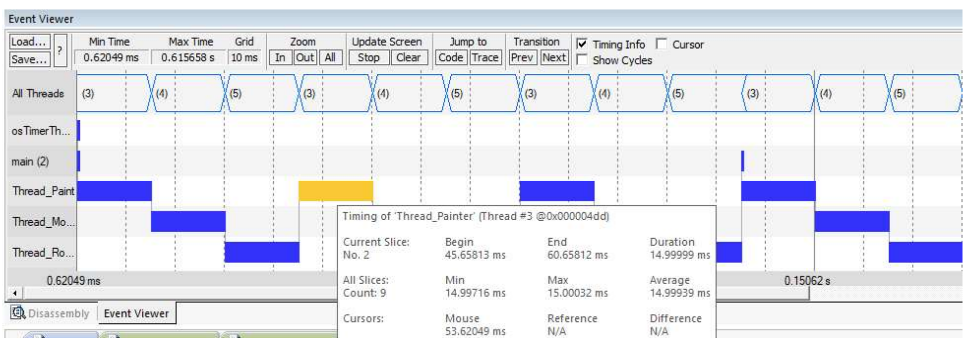



**Figure 3.** Event Viewer zoom verifying a consistent ~15 ms round-robin time slice.

Finally, **Figure 4** shows the entire execution timeline. You can see the repeating triplet of slices across the three threads for most of the run, followed by a brief period where one or two tasks have already terminated. After all three finish their finite workloads, the only activity left is the kernel's Idle thread, which is visible as the solid region at the tail end of the trace. That transition to idle time is important because it proves every task completed and deleted itself instead of spinning forever.


**Figure 4.** Event Viewer overview showing task switching, thread termination, and final Idle time.

# Question 2 - Analysis

Question 2 models a small operating system made of five cooperating roles that share the same round-robin priority but coordinate through explicit synchronization. Control flow is shaped by signals that hand work from memory to CPU, from application to device, and finally to the user interface, while a mutex protects a shared logger so concurrent writes are consistent. A volatile word is manipulated through SRAM bit-banding to exercise low-level memory access, and each role performs a finite unit of work, inserts a one-tick delay where required, and then terminates. The resulting trace begins at Memory Management and unfolds through the remaining roles in that dependency order, which makes the interactions and their timing easy to observe in the analysis tools.

The Watch Window in **Figure 5** shows that each role performs a single pass and then deletes, which is why every counter is exactly one. This is the expected outcome for a finite workload that runs to completion once per role. The mutex-protected logger can also be inspected here to confirm that the application writes its prefix and the device appends its ending without tearing.

Watch 1

| Name | Value | Type |
|---|---|---|
| mem_access_counter | 0x00000001 | uint |
| cpu_access_counter | 0x00000001 | uint |
| app_counter | 0x00000001 | uint |
| dev_counter | 0x00000001 | uint |
| ui_user_count | 0x00000001 | uint |
| logger[128] | 0x4D 'M' | uchar |
| bb_word | 0x00000009 | uint |

**Figure 5.** Watch Window displaying Q2 counters and shared state after the single-pass execution.

The Performance Analyzer in **Figure 6** records brief execution time for each thread with near-zero overall percentages, which is natural for a short, dependency-driven run. The two bit-band helper calls appear as small functions with modest self time, and the monitor thread shows minimal activity. More importantly, the tool confirms that each role did run and consume some CPU before terminating, which supports the correctness of the single-pass implementation.

| Module/Function | Calls | Time(Sec) | Time(%) |
|---|---|---|---|
| ../HAL_CM.c | | 24.830 us | 0% |
| thread2_analysis.c | | 15.310 us | 0% |
| bb_read_bit | 2 | 0.420 us | 0% |
| Monitor | 1 | 6.030 us | 0% |
| Th_UserInterface | 1 | 0.260 us | 0% |
| Th_DeviceManagement | 1 | 0.700 us | 0% |
| Th_ApplicationInterface | 1 | 0.700 us | 0% |
| Th_CPUManagement | 1 | 0.690 us | 0% |
| Th_MemoryManagement | 1 | 0.790 us | 0% |
| Init_Thread | 1 | 1.160 us | 0% |
| bb_write_bit | 3 | 0.800 us | 0% |
| bb_alias_addr | 5 | 0.650 us | 0% |
| tl_mark | 5 | 3.110 us | 0% |

**Figure 6.** Performance Analyzer summarizing brief execution time for each Q2 role.

The Event Viewer in **Figure 7** captures the causal ordering. The first slice belongs to Memory Management, followed by CPU Management when the signal arrives. After the CPU returns its signal, the application appears, then the device after the application's wakeup, and finally the user interface. Each thread then shows a brief final slice corresponding to the one-tick delay before termination. At the end of the timeline the system transitions to Idle, which demonstrates that all work is finite and no thread spins.

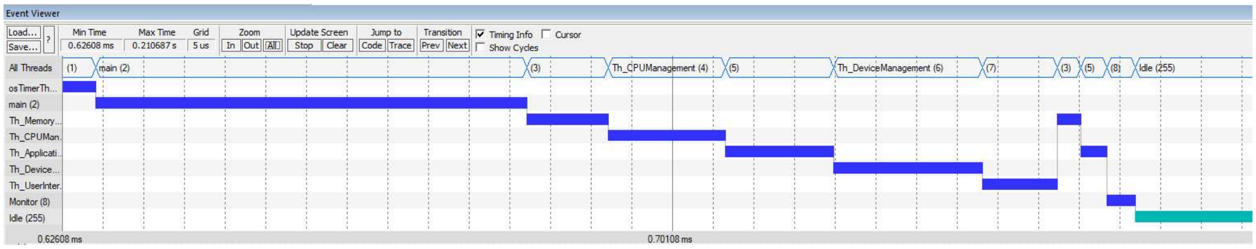



**Figure 7.** Event Viewer timeline illustrating the Q2 ordering from Memory → CPU → Application → Device → UI.

For this operating system problem, round-robin is suitable because all roles run at the same priority and the real sequencing is enforced by signals and the mutex rather than by priority differences. While threads are ready, round-robin provides predictable progress and prevents starvation. When a role must wait for another, it blocks, which cleanly removes it from competition until the dependency is satisfied. The main trade-off is latency. A blocked thread does not contend, so the next ready thread receives the processor, but when several become ready together, they will share the processor at the quantum, even if one is on the critical path. In a larger system, priorities or shorter quanta could reduce response time along critical chains such as application to device, while round-robin would continue to work well for short, background tasks that do not impose ordering on others.

# Conclusion

In conclusion, this lab met its objectives. For Question 1, a five millisecond tick with a three-tick round-robin timeout produced a measured fifteen millisecond quantum; the three tasks shared CPU time evenly while ready and each terminated cleanly. For Question 2, the five-thread system honored its signal-based ordering, protected the shared logger via a mutex, and demonstrated SRAM bit-banding within a bounded run. The demo builds made behavior visible on hardware, and the analysis builds made timing and scheduling straightforward to measure. Overall, round-robin delivered predictable progress for equal-priority work, while signals and mutexes enforced ordering.

# Appendix

## Appendix A: Q1 Debug Code

```
/* COE718 Lab 3a — ANALYSIS VERSION*/

#include "cmsis_os.h"
#include <stdint.h>

/* ===== RR proof knobs ===== */
#define ACTIVATIONS_PER_TASK   150u   /* keep all three alive ~few seconds */
#define WORK_UNITS_PAINTER     28000u /* per-activation busy work */
#define WORK_UNITS_MORSE       28000u
#define WORK_UNITS_ROBOT       28000u

/* ===== Painter parameters ===== */
#ifndef LCD_TOTAL_COLUMNS
# define LCD_TOTAL_COLUMNS     30u
#endif

/* ===== Morse message ===== */
#ifndef MORSE_TMU
# define MORSE_TMU            "-  --  ..-"
#endif

/* ===== Robot waypoints ===== */
typedef struct { int8_t x, y; } wp_t;
#ifndef WP_COUNT
static const wp_t WAYPOINTS_LOCAL[] = { {2,0},{6,0},{6,3},{3,3},{0,3},{0,0} };
# define WAYPOINTS WAYPOINTS_LOCAL
# define WP_COUNT  (sizeof(WAYPOINTS_LOCAL)/sizeof(WAYPOINTS_LOCAL[0]))
#endif

/* ===== Watchable debug vars ===== */
volatile uint32_t g_active_tid = 0;   /* 1,2,3 = which thread currently running */
volatile uint32_t g_t1_acts    = 0;   /* activations done by Painter */
volatile uint32_t g_t2_acts    = 0;   /* activations done by Morse   */
volatile uint32_t g_t3_acts    = 0;   /* activations done by Robot   */
volatile uint32_t g_t1_bitmap  = 0;   /* low 32 bits: painted cols */
volatile uint32_t g_t2_idx     = 0;   /* morse index */
volatile int8_t   g_robot_x    = 0, g_robot_y = 0;
volatile uint32_t g_robot_wp_index = 0;
volatile uint8_t  g_t1_done    = 0, g_t2_done = 0, g_t3_done = 0;

/* ===== Threads ===== */
void Thread_Painter (void const *argument);
void Thread_Morse   (void const *argument);
void Thread_Robot   (void const *argument);

osThreadId tid_painter, tid_morse, tid_robot;
osThreadDef(Thread_Painter, osPriorityNormal, 1, 0);
osThreadDef(Thread_Morse,   osPriorityNormal, 1, 0);
osThreadDef(Thread_Robot,   osPriorityNormal, 1, 0);
```

```c
static void do_busy_work(uint32_t units){
  volatile uint32_t acc = 0u;
  while (units--) {
    acc ^= units;
    acc = (acc << 1) | (acc >> 31);
  }
}

int Init_Thread(void) {
  int ok = 1;
  tid_painter = osThreadCreate(osThread(Thread_Painter), NULL); if (!tid_painter) ok = 0;
  tid_morse   = osThreadCreate(osThread(Thread_Morse),   NULL); if (!tid_morse)   ok = 0;
  tid_robot   = osThreadCreate(osThread(Thread_Robot),   NULL); if (!tid_robot)   ok = 0;
  return ok ? 0 : -1;
}

/* --------------- Task A: Slice-Programmable Painter --------------- */
void Thread_Painter (void const *argument) {
  uint32_t act = 0;
  uint32_t col = 0;
  (void)argument;

  while (act < ACTIVATIONS_PER_TASK) {
    uint32_t i;    g_active_tid = 1u;

    /* mark a few columns per activation */
    for (i = 0; i < 3u; ++i) {
      g_t1_bitmap |= (1u << (col & 31u));
      col++;
    }

    do_busy_work(WORK_UNITS_PAINTER);   /* keep READY; let RR do the preemption */
    g_t1_acts++;
    act++;
  }

  g_t1_done = 1u;
  osThreadTerminate(osThreadGetId());
}

/* -------------------- Task B: Morse “TMU” ------------------------- */
void Thread_Morse (void const *argument) {
  const char *msg = MORSE_TMU;
  uint32_t act = 0;
  uint32_t mlen = 0;
  (void)argument;

  /* compute length once */
  { const char *p = msg; while (*p) { mlen++; p++; } if (mlen == 0) mlen = 1; }

  while (act < ACTIVATIONS_PER_TASK) {
    char c;
```

```c
    g_active_tid = 2u;

    c = msg[g_t2_idx];
    g_t2_idx++;
    if (g_t2_idx >= mlen) g_t2_idx = 0;

    if (c == '-')      do_busy_work(WORK_UNITS_MORSE + (WORK_UNITS_MORSE>>3));
    else               do_busy_work(WORK_UNITS_MORSE);

    g_t2_acts++;
    act++;
  }

  g_t2_done = 1u;
  osThreadTerminate(osThreadGetId());
}

/* --------------- Task C: Differential-Drive Waypoint Tracker ---------------- */
static int8_t sgn_i8(int8_t v){ return (int8_t)((v > 0) - (v < 0)); }

void Thread_Robot (void const *argument) {
  uint32_t act = 0;
  (void)argument;

  g_robot_x = 0; g_robot_y = 0; g_robot_wp_index = 0;

  while (act < ACTIVATIONS_PER_TASK) {
    int8_t tx, ty, dx, dy;

    g_active_tid = 3u;

    tx = WAYPOINTS[g_robot_wp_index % WP_COUNT].x;
    ty = WAYPOINTS[g_robot_wp_index % WP_COUNT].y;

    dx = (int8_t)(tx - g_robot_x);
    dy = (int8_t)(ty - g_robot_y);

    if (dx == 0 && dy == 0) {
      g_robot_wp_index++; /* reached this waypoint; move to next (wraps) */
    } else {
      if ((int16_t)dx * (int16_t)dx >= (int16_t)dy * (int16_t)dy) {
        g_robot_x = (int8_t)(g_robot_x + sgn_i8(dx));
      } else {
        g_robot_y = (int8_t)(g_robot_y + sgn_i8(dy));
      }
    }

    do_busy_work(WORK_UNITS_ROBOT);
    g_t3_acts++;
    act++;
  }

  g_t3_done = 1u;
  osThreadTerminate(osThreadGetId());}
```

## Appendix B: Q2 Debug Code

```
#include "cmsis_os.h"
#include <stdint.h>
#include <string.h>

/* COE718 Lab 3a Q2 - Analysis version */

// ------------------- Watch-friendly globals -------------------
volatile uint32_t mem_access_counter = 0;
volatile uint32_t cpu_access_counter = 0;
volatile uint32_t app_counter       = 0;
volatile uint32_t dev_counter       = 0;
volatile uint32_t ui_user_count     = 0;

/* Make these volatile so the Watch window always sees updates */
volatile uint32_t bb_word = 0;
volatile char    logger[128] = {0};

volatile const char *logger_str   = logger;
// ------------------- Timeline monitor -------------
static char timeline[128];
static volatile uint32_t tl_head = 0;
osMutexDef(tl_mutex);
static osMutexId tl_mutex;

volatile const char *timeline_str = timeline;

static __inline void tl_mark(char tag) {
  osMutexWait(tl_mutex, osWaitForever);
  if (tl_head < sizeof(timeline) - 1) {
    if (tl_head == 0 || timeline[tl_head - 1] != tag) {
      timeline[tl_head++] = tag;
      timeline[tl_head]   = '\0';
    }
  }
  osMutexRelease(tl_mutex);
}

// ------------------- Signals -------------------
#define SIG_MM_TO_CPU   (1U << 0)
#define SIG_CPU_TO_MM   (1U << 1)
#define SIG_APP_READY   (1U << 2)
#define SIG_DEV_DONE    (1U << 3)

// ------------------- Single global mutex for logger ------------
osMutexDef(log_mutex);
static osMutexId log_mutex;

// ------------------- Bit-band helpers (SRAM) -------------------
#define BB_SRAM_REF   (0x20000000UL)
#define BB_SRAM_ALIAS (0x22000000UL)

static __inline volatile uint32_t* bb_alias_addr(volatile void* addr, uint32_t bit)
```

```
{
  uint32_t byte_offset = (uint32_t)addr - BB_SRAM_REF;
  uint32_t bit_word_offset = (byte_offset * 32U) + (bit * 4U);
  return (volatile uint32_t*)(BB_SRAM_ALIAS + bit_word_offset);
}
static __inline void bb_write_bit(volatile void* addr, uint32_t bit, uint32_t val)
{
  *bb_alias_addr(addr, bit) = (val ? 1U : 0U);
}
static __inline uint32_t bb_read_bit(volatile void* addr, uint32_t bit)
{
  return *bb_alias_addr(addr, bit);
}

// ------------------- Rotate-right (barrel-shift demo) ----------
static __inline uint32_t ror32(uint32_t x, unsigned n)
{
  n &= 31U;
  return (x >> n) | (x << (32U - n));
}

// ------------------- Thread IDs & defs -------------------------
static osThreadId tid_mem, tid_cpu, tid_app, tid_dev, tid_ui, tid_mon;

void Th_MemoryManagement(const void *arg);
void Th_CPUManagement(const void *arg);
void Th_ApplicationInterface(const void *arg);
void Th_DeviceManagement(const void *arg);
void Th_UserInterface(const void *arg);
void Monitor(const void *arg);

/* Explicit stacks */
osThreadDef(Th_MemoryManagement,    osPriorityNormal, 1, 0);
osThreadDef(Th_CPUManagement,       osPriorityNormal, 1, 0);
osThreadDef(Th_ApplicationInterface,osPriorityNormal, 1, 0);
osThreadDef(Th_DeviceManagement,    osPriorityNormal, 1, 0);
osThreadDef(Th_UserInterface,       osPriorityNormal, 1, 0);
osThreadDef(Monitor,                osPriorityLow,    1, 0);

int Init_Thread (void) {
  tl_mutex  = osMutexCreate(osMutex(tl_mutex));
  log_mutex = osMutexCreate(osMutex(log_mutex));

  tid_mem = osThreadCreate(osThread(Th_MemoryManagement),    NULL);
  tid_cpu = osThreadCreate(osThread(Th_CPUManagement),       NULL);
  tid_app = osThreadCreate(osThread(Th_ApplicationInterface),NULL);
  tid_dev = osThreadCreate(osThread(Th_DeviceManagement),    NULL);
  tid_ui  = osThreadCreate(osThread(Th_UserInterface),       NULL);
  if (!tid_mem || !tid_cpu || !tid_app || !tid_dev || !tid_ui) return -1;

  /* Finite monitor */
  tid_mon = osThreadCreate(osThread(Monitor), NULL);
  return 0;
}
```

```
// ------------------- Implementations --------------------------
void Th_MemoryManagement(const void *arg)
{
  uint32_t b0;
  mem_access_counter++;
  tl_mark('M');

  /* Bit-band demo: set bit3, clear bit2, toggle bit0 */
  bb_write_bit((void*)&bb_word, 3U, 1U);
  bb_write_bit((void*)&bb_word, 2U, 0U);
  b0 = bb_read_bit((void*)&bb_word, 0U);
  bb_write_bit((void*)&bb_word, 0U, b0 ^ 1U);

  /* Signal CPU and wait for response */
  osSignalSet(tid_cpu, SIG_MM_TO_CPU);
  (void)osSignalWait(SIG_CPU_TO_MM, osWaitForever);

  osDelay(1);
  osThreadTerminate(osThreadGetId());
}

void Th_CPUManagement(const void *arg)
{
  uint32_t x;
  unsigned rot;

  (void)osSignalWait(SIG_MM_TO_CPU, osWaitForever);
  tl_mark('C');

  cpu_access_counter++;

  /* Conditional rotate: if bit3 set in bb_word, rotate by 7 else by 3 */
  x = (uint32_t)bb_word;
  rot = bb_read_bit((void*)&bb_word, 3U) ? 7U : 3U;
  x = ror32(x ^ 0xA5A5A5A5UL, rot);

  osSignalSet(tid_mem, SIG_CPU_TO_MM);
  osThreadTerminate(osThreadGetId());
}

void Th_ApplicationInterface(const void *arg)
{
  tl_mark('A');

  osMutexWait(log_mutex, osWaitForever);
  strcpy((char*)logger, "App: begin write -> ");
  osMutexRelease(log_mutex);

  osSignalSet(tid_dev, SIG_APP_READY);
  (void)osSignalWait(SIG_DEV_DONE, osWaitForever);

  app_counter++;
  osDelay(1);
```

```
  osThreadTerminate(osThreadGetId());
}

void Th_DeviceManagement(const void *arg)
{
  (void)osSignalWait(SIG_APP_READY, osWaitForever);
  tl_mark('D');

  osMutexWait(log_mutex, osWaitForever);
  strcat((char*)logger, "Device: append + close.");
  osMutexRelease(log_mutex);

  osSignalSet(tid_app, SIG_DEV_DONE);

  dev_counter++;
  osDelay(1);
  osThreadTerminate(osThreadGetId());
}

void Th_UserInterface(const void *arg)
{
  tl_mark('U');
  ui_user_count++;
  //osDelay(1);
  osThreadTerminate(osThreadGetId());
}

/* Finite Monitor: terminates after ~2s */
void Monitor(const void *arg)
{
  static volatile uint32_t hb = 0;
  uint32_t i;
  for (i = 0; i < 40U; ++i) {  /* 40 * 50 ms = ~2 seconds */
    hb++;
    osDelay(50);
  }
  osThreadTerminate(osThreadGetId());
}
```

## Appendix C: Q1 Demo Code

```
#include "thread_tasks.h"   /* SLICES_TO_FINISH, LCD_TOTAL_COLUMNS, MORSE_TMU, WAYPOINTS... */
#include "cmsis_os.h"
#include "LPC17xx.h"
#include "GLCD.h"
#include <stdint.h>

/* ====== 2-second window per thread (RTX tick = 5 ms) ====== */
#ifndef RTX_TICK_US
# define RTX_TICK_US        5000u
#endif
#define WINDOW_TICKS        400u   /* 400 * 5 ms = 2000 ms = 2 s */

/* ---------- Fallback waypoints ---------- */
#ifndef WP_COUNT
typedef struct { int8_t x, y; } wp_t_local;
static const wp_t_local DFLT_WP[] = { {2,0},{6,0},{6,3},{3,3},{0,3},{0,0} };
#define WP_COUNT (sizeof(DFLT_WP)/sizeof(DFLT_WP[0]))
#define WP_AT(k) (DFLT_WP[(k)])
#else
#define WP_AT(k) (WAYPOINTS[(k)])
#endif

/* ---------- LED helpers ---------- */
/* 0 -> P1.28, 1 -> P2.2, 2 -> P1.31 */
static void leds_init(void) {
  LPC_GPIO1->FIODIR |= (1u<<28) | (1u<<31);
  LPC_GPIO2->FIODIR |= (1u<<2);
}
static void leds_all_off(void) {
  LPC_GPIO1->FIOCLR = (1u<<28) | (1u<<31);
  LPC_GPIO2->FIOCLR = (1u<<2);
}
static void led_show(int idx) {
  leds_all_off();                      /* exactly one LED on */
  if (idx == 0)      LPC_GPIO1->FIOSET = (1u<<28);
  else if (idx == 1) LPC_GPIO2->FIOSET = (1u<<2);
  else if (idx == 2) LPC_GPIO1->FIOSET = (1u<<31);
}

/* ---------- LCD helpers ---------- */
#define LCD_W  21
static void pad_copy(unsigned char *dst, unsigned int maxw, const char *src){
  unsigned int i = 0;
  while (src && src[i] && i < maxw) { dst[i] = (unsigned char)src[i]; i++; }
  while (i < maxw) { dst[i++] = ' '; }
  dst[maxw] = 0;
}
static void lcd_line(unsigned int line, const char *txt){
  static unsigned char buf[LCD_W+1];
  pad_copy(buf, LCD_W, txt);
  GLCD_DisplayString(line, 0, 1, buf);
}
```

```
static void lcd_title(const char *msg){ lcd_line(0, msg); }
static void lcd_active_text(const char *tn){
  /* prints "Active: <name>" padded */
  static char tmp[32];
  unsigned int i = 0;
  const char *prefix = "Active: ";
  while (prefix[i]) { tmp[i] = prefix[i]; i++; }
  while (tn && *tn && i < sizeof(tmp)-1) { tmp[i++] = *tn++; }
  tmp[i] = 0;
  lcd_line(1, tmp);
}

/* progress bar to line 3, exact width */
static void lcd_bar_line3(unsigned int filled, unsigned int total){
  static char bar[32];
  unsigned int i, width = (LCD_W < 30u) ? LCD_W : 30u;
  if (total == 0) total = 1;
  if (filled > total) filled = total;
  /* compute chars to draw */
  {
    unsigned int n = (filled * width) / total;
    for (i = 0; i < n && i < width; i++) bar[i] = '#';
    while (i < width) bar[i++] = ' ';
    bar[i] = 0;
  }
  lcd_line(3, bar);
}

/* show small status on line 2, padded */
static void lcd_status_line2(const char *txt){ lcd_line(2, txt); }

/* plot dot for robot on text grid (lines 5..) */
static void lcd_plot_dot(uint32_t ln, uint32_t col){
  if (col < LCD_W) {
    GLCD_DisplayChar(5 + ln, col, 1, '.');
  }
}

/* ---------- tiny utils ---------- */
static int8_t sgn(int8_t v){ return (v>0) - (v<0); }

/* ---------- Threads & token (signals) ---------- */
void Thread_Painter (void const *argument);
void Thread_Morse   (void const *argument);
void Thread_Robot   (void const *argument);

osThreadId tid_painter, tid_morse, tid_robot;
static volatile uint8_t t1_done, t2_done, t3_done;

/* use default RTX stack (0) */
osThreadDef(Thread_Painter, osPriorityNormal, 1, 0);
osThreadDef(Thread_Morse,   osPriorityNormal, 1, 0);
osThreadDef(Thread_Robot,   osPriorityNormal, 1, 0);
```

```
#define SIG_TOKEN   (0x1u)

/* choose next alive thread in round-robin order */
static void pass_token_from(uint8_t from_id){
  /* from_id: 1=T1, 2=T2, 3=T3 */
  if (from_id == 1u) {
    if (!t2_done)      osSignalSet(tid_morse, SIG_TOKEN);
    else if (!t3_done) osSignalSet(tid_robot, SIG_TOKEN);
  } else if (from_id == 2u) {
    if (!t3_done)      osSignalSet(tid_robot, SIG_TOKEN);
    else if (!t1_done) osSignalSet(tid_painter, SIG_TOKEN);
  } else { /* from T3 */
    if (!t1_done)      osSignalSet(tid_painter, SIG_TOKEN);
    else if (!t2_done) osSignalSet(tid_morse, SIG_TOKEN);
  }
}

/* ---------- Init: create threads, give token to T1 ---------- */
int Init_Thread(void) {
  leds_init();
  leds_all_off();

  GLCD_Init();
  GLCD_SetTextColor(White);
  GLCD_SetBackColor(Black);
  GLCD_Clear(Black);

  t1_done = t2_done = t3_done = 0;

  lcd_title("Round-Robin Demo");
  lcd_active_text("(waiting)");
  lcd_status_line2("                    ");
  lcd_bar_line3(0, 1);

  tid_painter = osThreadCreate(osThread(Thread_Painter), NULL);
  tid_morse   = osThreadCreate(osThread(Thread_Morse),   NULL);
  tid_robot   = osThreadCreate(osThread(Thread_Robot),   NULL);

  if (!tid_painter) lcd_status_line2("ERR: T1 create");
  if (!tid_morse)   lcd_status_line2("ERR: T2 create");
  if (!tid_robot)   lcd_status_line2("ERR: T3 create");

  if (tid_painter && tid_morse && tid_robot) {
    osSignalSet(tid_painter, SIG_TOKEN);   /* start with T1 */
    return 0;
  }
  return -1;
}

/* ============================================================
   Task A — Slice-Programmable Pixel Painter
   One "activation" per 2-second window; finishes in SLICES_TO_FINISH.
   ============================================================ */
void Thread_Painter (void const *argument) {
```

```
  uint32_t slices, per_activation, painted = 0;
  (void)argument;

#ifndef LCD_TOTAL_COLUMNS
# define LCD_TOTAL_COLUMNS 30u
#endif
#ifndef SLICES_TO_FINISH
# define SLICES_TO_FINISH 3u
#endif

  slices = (SLICES_TO_FINISH < 1) ? 1 : SLICES_TO_FINISH;
  per_activation = (LCD_TOTAL_COLUMNS + slices - 1u) / slices;

  while (1) {
    osEvent ev = osSignalWait(SIG_TOKEN, osWaitForever);
    (void)ev;

    if (painted >= LCD_TOTAL_COLUMNS) {
      t1_done = 1u;
      lcd_status_line2("T1 Done");
      pass_token_from(1u);
      osThreadTerminate(osThreadGetId());
    }

    /* do one chunk */
    {
      uint32_t todo = per_activation, i;
      char label[24];
      if (painted + todo > LCD_TOTAL_COLUMNS) todo = LCD_TOTAL_COLUMNS - painted;

      led_show(0);
      lcd_active_text("T1");

      {
        unsigned int k = (painted + todo);
        unsigned int N = LCD_TOTAL_COLUMNS;
        label[0]='P';label[1]='a';label[2]='i';label[3]='n';label[4]='t';label[5]='e';label[6]='r';label[7]=':';label[8]=' ';
        label[9]  = (char)('0' + ((k*10u)/N));
        label[10] = (char)('/'); label[11]='1'; label[12]='0'; label[13]=0;
      }
      lcd_status_line2(label);

      /* show bar progress */
      lcd_bar_line3(painted + todo, LCD_TOTAL_COLUMNS);

      painted += todo;
    }

    /* hold token ~2s */
    {
      uint32_t t;
      for (t = 0; t < WINDOW_TICKS; t++) { osDelay(1); }
    }
```

```
    pass_token_from(1u);
  }
}

/* ============================================================
   Task B — Morse “TMU”
   One symbol per 2-second window;
   ============================================================ */
static void morse_symbol_consume(char c){
  if (c == '.')      osDelay(12);  /* ~60 ms */
  else if (c == '-') osDelay(24);  /* ~120 ms */
  else               osDelay(12);
}
static unsigned int morse_len(const char *s){
  unsigned int n=0; while (s && s[n]) n++; return n;
}

void Thread_Morse (void const *argument) {
  const char *p; unsigned int idx=0, total;
  (void)argument;
#ifndef MORSE_TMU
  #define MORSE_TMU "-  --  ..-"
#endif
  p = MORSE_TMU; total = morse_len(MORSE_TMU);

  while (1) {
    osSignalWait(SIG_TOKEN, osWaitForever);

    if (idx >= total) {
      t2_done = 1u;
      lcd_status_line2("T2 Done");
      pass_token_from(2u);
      osThreadTerminate(osThreadGetId());
    }

    led_show(1);
    lcd_active_text("T2");

    {
      char l2[32];
      const char *kind = (p[idx]=='.') ? "dot" : (p[idx]=='-') ? "dash" : "gap";
      l2[0]='M';l2[1]='o';l2[2]='r';l2[3]='s';l2[4]='e';l2[5]=':';l2[6]=' ';l2[7]=0;
      {
        unsigned int j=7, k=0;
        while (kind[k] && j<31){ l2[j++] = kind[k++]; }
        l2[j++]=' '; l2[j++]='(';
        l2[j++] = (char)('0' + ((idx+1)/10u));
        l2[j++] = (char)('0' + ((idx+1)%10u));
        l2[j++] = '/';
        l2[j++] = (char)('0' + (total/10u));
        l2[j++] = (char)('0' + (total%10u));
        l2[j++]=')'; l2[j]=0;
      }
      lcd_status_line2(l2);
```

```
    }

    /* line 3: progress bar over total symbols */
    lcd_bar_line3(idx+1, total);

    morse_symbol_consume(p[idx]);
    idx++;

    /* hold token ~2s */
    {
      uint32_t t;
      for (t = 0; t < WINDOW_TICKS; t++) { osDelay(1); }
    }

    pass_token_from(2u);
  }
}

/* =========================================================
   Task C — Differential-Drive Waypoint Tracker
   One grid step per 2-second window;
   ========================================================= */
void Thread_Robot (void const *argument) {
  int8_t x = 0, y = 0;
  uint32_t i = 0; (void)argument;

  while (1) {
    osSignalWait(SIG_TOKEN, osWaitForever);

    if (i >= WP_COUNT) {
      t3_done = 1u;
      lcd_status_line2("T3 Done");
      pass_token_from(3u);
      osThreadTerminate(osThreadGetId());
    }

    led_show(2);
    lcd_active_text("T3");

    /* one step toward current waypoint */
    {
      int8_t dx = (int8_t)(WP_AT(i).x - x);
      int8_t dy = (int8_t)(WP_AT(i).y - y);

      if (dx == 0 && dy == 0) {
        i++;  /* reached this waypoint */
      } else {
        if ((dx*dx) >= (dy*dy)) x = (int8_t)(x + sgn(dx));
        else                    y = (int8_t)(y + sgn(dy));
      }

      {
        char l2[32];
        int xi = (int)x, yi = (int)y;
```

```
        l2[0]='R';l2[1]='o';l2[2]='b';l2[3]='o';l2[4]='t';l2[5]=':';l2[6]=' ';l2[7]=0;
        {
          unsigned int j=7;
          l2[j++]='(';
          if (xi<0){ l2[j++]='-'; xi=-xi; }
          l2[j++] = (char)('0' + (xi/10)%10);
          l2[j++] = (char)('0' + (xi%10));
          l2[j++]=','; l2[j++]=' ';
          if (yi<0){ l2[j++]='-'; yi=-yi; }
          l2[j++] = (char)('0' + (yi/10)%10);
          l2[j++] = (char)('0' + (yi%10));
          l2[j++]=')'; l2[j++]=' ';
          /* "i/N" */
          {
            unsigned int ii = (i<100)? i:99;
            unsigned int NN = (WP_COUNT<100)? WP_COUNT:99;
            l2[j++] = (char)('0' + (ii/10));
            l2[j++] = (char)('0' + (ii%10));
            l2[j++] = '/';
            l2[j++] = (char)('0' + (NN/10));
            l2[j++] = (char)('0' + (NN%10));
            l2[j]=0;
          }
        }
        lcd_status_line2(l2);
      }

      /* plot dot for current pose */
      lcd_plot_dot((uint32_t)((y >= 0) ? y : 0), (uint32_t)((x >= 0) ? x : 0));
    }

    /* bar on line 3 = waypoint progress */
    lcd_bar_line3(i, WP_COUNT);

    /* hold token ~2s */
    {
      uint32_t t;
      for (t = 0; t < WINDOW_TICKS; t++) { osDelay(1); }
    }

    pass_token_from(3u);
  }
}
```

# Appendix D: Q2 Demo Code

```
#include "cmsis_os.h"
#include "LPC17xx.h"
#include "GLCD.h"
#include <stdint.h>
#include <string.h>

/* ====== 2-second window per task (RTX tick = 5 ms) ====== */
#ifndef RTX_TICK_US
# define RTX_TICK_US        5000u
#endif
#define WINDOW_TICKS        400u   /* 400 * 5 ms = 2000 ms = 2 s */

/* ------------------- Shared demo/analysis state ------------------- */
volatile uint32_t mem_access_counter = 0;
volatile uint32_t cpu_access_counter = 0;
volatile uint32_t app_counter        = 0;
volatile uint32_t dev_counter        = 0;
volatile uint32_t ui_user_count      = 0;

/* Bit-band demo target and shared logger */
volatile uint32_t bb_word = 0;
volatile char     logger[128] = {0};

/* ------------------- Signals ------------------- */
#define SIG_MM_TO_CPU   (1u << 0)
#define SIG_CPU_TO_MM   (1u << 1)
#define SIG_APP_READY   (1u << 2)
#define SIG_DEV_DONE    (1u << 3)

/* ------------------- Mutex for logger ------------------- */
osMutexDef(log_mutex);
static osMutexId log_mutex;

/* ------------------- Bit-band helpers (SRAM) ------------------- */
#define BB_SRAM_REF   (0x20000000UL)
#define BB_SRAM_ALIAS (0x22000000UL)

static volatile uint32_t* bb_alias_addr(volatile void* addr, uint32_t bit)
{
  uint32_t byte_off = (uint32_t)addr - BB_SRAM_REF;
  uint32_t word_off = (byte_off * 32u) + (bit * 4u);
  return (volatile uint32_t*)(BB_SRAM_ALIAS + word_off);
}
static void bb_write_bit(volatile void* addr, uint32_t bit, uint32_t val)
{
  *bb_alias_addr(addr, bit) = (val ? 1u : 0u);
}
static uint32_t bb_read_bit(volatile void* addr, uint32_t bit)
{
  return *bb_alias_addr(addr, bit);
}
```

```
/* ------------------- Barrel rotate (CPU conditional) -------------- */
static uint32_t ror32(uint32_t x, unsigned n)
{
  n &= 31u;
  return (x >> n) | (x << (32u - n));
}

/* ------------------- LED helpers --------------- */
/* LED map: 0=P1.28, 1=P2.2, 2=P1.31, 3=P2.3, 4=P2.4 */
static void leds_init(void) {
  LPC_GPIO1->FIODIR |= (1u<<28) | (1u<<31);
  LPC_GPIO2->FIODIR |= (1u<<2)  | (1u<<3) | (1u<<4);
}
static void leds_all_off(void) {
  LPC_GPIO1->FIOCLR = (1u<<28) | (1u<<31);
  LPC_GPIO2->FIOCLR = (1u<<2)  | (1u<<3) | (1u<<4);
}
static void led_show(int idx) {
  leds_all_off();  /* exactly one LED on */
  if      (idx == 0) LPC_GPIO1->FIOSET = (1u<<28);
  else if (idx == 1) LPC_GPIO2->FIOSET = (1u<<2);
  else if (idx == 2) LPC_GPIO1->FIOSET = (1u<<31);
  else if (idx == 3) LPC_GPIO2->FIOSET = (1u<<3);
  else if (idx == 4) LPC_GPIO2->FIOSET = (1u<<4);
}

/* ------------------- LCD helpers --------------- */
#define LCD_W  21   /* safe width for classic font */

static void pad_copy(unsigned char *dst, unsigned int maxw, const char *src){
  unsigned int i = 0;
  while (src && src[i] && i < maxw) { dst[i] = (unsigned char)src[i]; i++; }
  while (i < maxw) { dst[i++] = ' '; }
  dst[maxw] = 0;
}
static void lcd_line(unsigned int line, const char *txt){
  static unsigned char buf[LCD_W+1];
  pad_copy(buf, LCD_W, txt);
  GLCD_DisplayString(line, 0, 1, buf);
}
static void lcd_title(const char *msg){ lcd_line(0, msg); }
static void lcd_active_text(const char *tn){
  static char tmp[32];
  unsigned int i = 0;
  const char *prefix = "Active: ";
  while (prefix[i]) { tmp[i] = prefix[i]; i++; }
  while (tn && *tn && i < sizeof(tmp)-1) { tmp[i++] = *tn++; }
  tmp[i] = 0;
  lcd_line(1, tmp);
}
static void lcd_status_line2(const char *txt){ lcd_line(2, txt); }
static void lcd_line3(const char *txt){ lcd_line(3, txt); }

/* show a short “spotlight” so each role is clearly visible */
```

```
static void hold_window(void){
  uint32_t t;
  for (t = 0; t < WINDOW_TICKS; ++t) { osDelay(1); }
}

/* ------------------- Threads & defs ------------------------------- */
static osThreadId tid_mem, tid_cpu, tid_app, tid_dev, tid_ui;

void Th_MemoryManagement     (void const *arg);
void Th_CPUManagement        (void const *arg);
void Th_ApplicationInterface (void const *arg);
void Th_DeviceManagement     (void const *arg);
void Th_UserInterface        (void const *arg);

osThreadDef(Th_MemoryManagement,     osPriorityNormal, 1, 0);
osThreadDef(Th_CPUManagement,        osPriorityNormal, 1, 0);
osThreadDef(Th_ApplicationInterface, osPriorityNormal, 1, 0);
osThreadDef(Th_DeviceManagement,     osPriorityNormal, 1, 0);
osThreadDef(Th_UserInterface,        osPriorityNormal, 1, 0);

/* ------------------- Init: LEDs/LCD + threads --------------------- */
int Init_Thread (void) {
  leds_init();
  leds_all_off();

  GLCD_Init();
  GLCD_SetTextColor(White);
  GLCD_SetBackColor(Black);
  GLCD_Clear(Black);
  lcd_title("Q2 Demo: OS Roles");
  lcd_active_text("(waiting)");
  lcd_status_line2("logger ready");
  lcd_line3("                ");

  log_mutex = osMutexCreate(osMutex(log_mutex));

  /* Create all roles; RR with equal priorities, ordering via signals */
  tid_mem = osThreadCreate(osThread(Th_MemoryManagement),     NULL);
  tid_cpu = osThreadCreate(osThread(Th_CPUManagement),        NULL);
  tid_app = osThreadCreate(osThread(Th_ApplicationInterface), NULL);
  tid_dev = osThreadCreate(osThread(Th_DeviceManagement),     NULL);
  tid_ui  = osThreadCreate(osThread(Th_UserInterface),        NULL);

  if (!tid_mem || !tid_cpu || !tid_app || !tid_dev || !tid_ui) {
    lcd_status_line2("ERR: thread create");
    return -1;
  }
  return 0;
}

/* ======================== Implementations ======================== */

void Th_MemoryManagement(const void *arg)
{
```

```
  uint32_t b0; (void)arg;

  mem_access_counter++;
  led_show(0);
  lcd_active_text("Memory");
  lcd_status_line2("bit-band ops...");
  /* Bit-band demo: set bit3, clear bit2, toggle bit0 */
  bb_write_bit((void*)&bb_word, 3u, 1u);
  bb_write_bit((void*)&bb_word, 2u, 0u);
  b0 = bb_read_bit((void*)&bb_word, 0u);
  bb_write_bit((void*)&bb_word, 0u, b0 ^ 1u);

  /* show bb_word hex briefly */
  {
    char line[32];
    unsigned int i = 0;
    static const char H[16] = "0123456789ABCDEF";
    uint32_t v = bb_word;
    line[i++]='b'; line[i++]='b'; line[i++]='='; line[i++]='0'; line[i++]='x';
    line[i++] = H[(v>>28)&0xF]; line[i++] = H[(v>>24)&0xF];
    line[i++] = H[(v>>20)&0xF]; line[i++] = H[(v>>16)&0xF];
    line[i++] = H[(v>>12)&0xF]; line[i++] = H[(v>>8)&0xF];
    line[i++] = H[(v>>4)&0xF];  line[i++] = H[(v>>0)&0xF];
    line[i]=0;
    lcd_line3(line);
  }

  /* hand off to CPU and wait the reply so ordering is visible */
  osSignalSet(tid_cpu, SIG_MM_TO_CPU);
  (void)osSignalWait(SIG_CPU_TO_MM, osWaitForever);

  hold_window();              /* ~2 s spotlight */
  lcd_status_line2("Memory done");
  osDelay(1);                 /* 1 tick per spec */
  osThreadTerminate(osThreadGetId());
}

void Th_CPUManagement(const void *arg)
{
  uint32_t x; unsigned rot; (void)arg;

  (void)osSignalWait(SIG_MM_TO_CPU, osWaitForever);
  cpu_access_counter++;
  led_show(1);
  lcd_active_text("CPU");
  lcd_status_line2("rotate & reply");

  /* conditional rotate based on bb_word bit3 */
  x = (uint32_t)bb_word;
  rot = bb_read_bit((void*)&bb_word, 3u) ? 7u : 3u;
  x = ror32(x ^ 0xA5A5A5A5u, rot);
  (void)x;

  osSignalSet(tid_mem, SIG_CPU_TO_MM);
```

```
  hold_window();
  lcd_status_line2("CPU done");
  osThreadTerminate(osThreadGetId());
}

void Th_ApplicationInterface(const void *arg)
{
  (void)arg;

  led_show(2);
  lcd_active_text("App");
  lcd_status_line2("write prefix...");

  osMutexWait(log_mutex, osWaitForever);
  strcpy((char*)logger, "App: begin -> ");
  osMutexRelease(log_mutex);

  osSignalSet(tid_dev, SIG_APP_READY);
  (void)osSignalWait(SIG_DEV_DONE, osWaitForever);

  app_counter++;
  lcd_line3((const char*)logger);

  hold_window();
  lcd_status_line2("App done");
  osDelay(1);
  osThreadTerminate(osThreadGetId());
}

void Th_DeviceManagement(const void *arg)
{
  (void)arg;

  (void)osSignalWait(SIG_APP_READY, osWaitForever);

  led_show(3);
  lcd_active_text("Device");
  lcd_status_line2("append & signal");

  osMutexWait(log_mutex, osWaitForever);
  strcat((char*)logger, "Device: close.");
  osMutexRelease(log_mutex);

  osSignalSet(tid_app, SIG_DEV_DONE);

  dev_counter++;
  lcd_line3((const char*)logger);

  hold_window();
  lcd_status_line2("Device done");
  osDelay(1);
  osThreadTerminate(osThreadGetId());
}
```

```
void Th_UserInterface(const void *arg)
{
  (void)arg;

  led_show(4);
  lcd_active_text("User IF");
  lcd_status_line2("one-shot task");

  ui_user_count++;

  hold_window();
  lcd_status_line2("UI done");
  osDelay(1);
  osThreadTerminate(osThreadGetId());
}
```

## Appendix E: Q1 Header file

```
#pragma once
#include "cmsis_os.h"
#include <stdint.h>

#define LCD_TOTAL_COLUMNS      30u
#define SLICES_TO_FINISH       3u   // Task A: finish in this many activations

// Task B (Morse for TMU)
static const char *MORSE_TMU = "-  --  ..-";

// ===== Task C (Robotics): Waypoints (grid units) =====
// Keep these small; each activation moves by exactly 1 grid step.
typedef struct { int8_t x, y; } wp_t;
static const wp_t WAYPOINTS[] = {
  {2,  0}, {6,  0}, {6, 3}, {3, 5}, {0, 5}, {0, 0} // a little loop
};
static const uint32_t WP_COUNT = sizeof(WAYPOINTS)/sizeof(WAYPOINTS[0]);

// Shared prototypes
int  Init_Thread(void);
void Thread_Painter (void const *argument);
void Thread_Morse   (void const *argument);
void Thread_Robot   (void const *argument);
```