Operating systems and concurrency B04

David Kendall

Northumbria University

Introduction

- Towards an operating system: CMSIS and Mbed
- Multi-tasking operating system services
- μC/OS-II (uC/OS-II)
- Task management in uC/OS-II

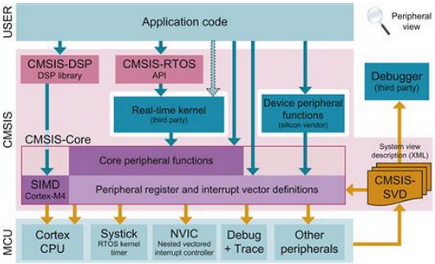
Standard names for the microcontroller registers

```c
#include <stdbool.h>
#include <stdint.h>
#include <LPC407x_8x_177x_8x.h>

#define LED1PIN    (1UL << 18)

void delay(uint32_t ms);

int main() {
  LPC_IOCON->P1_18 &= ~0x1FUL;
  LPC_GPIO1->DIR |= LED1PIN;
  while (true) {
    LPC_GPIO1->SET = LED1PIN;
    delay(1000);
    LPC_GPIO1->CLR = LED1PIN;
    delay(1000);
  }
}
```

- Header file `LPC407x_8x_177x_8x.h` provided by the manufacturer of the microcontroller (NXP) to give a set of standard names for the microcontroller registers
- No need to write `#define`s after looking up the addresses yourself.
- Header file written in standard ANSI C - CMSIS-complaint - portable between tool providers.

CMSIS

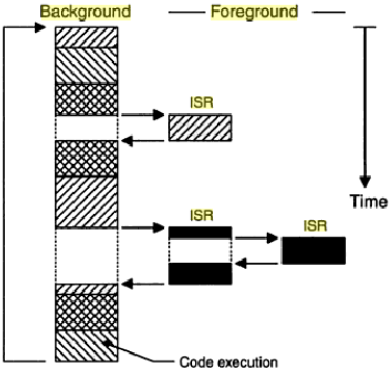
- Cortex Microcontroller Software Interface Standard
- CMSIS-Core
  - Standard set of names provided by ARM for accessing the microprocessor
  - Standard startup files for bringing up the CPU out of reset:
    - configure system clocks
    - define the vector table
    - run the `main` function
- Improves software portability and reusability across different microcontrollers and toolchains



Martin, T. *The Designer's Guide to the Cortex-M Processor Family: A Tutorial Approach*, Newnes, 2013

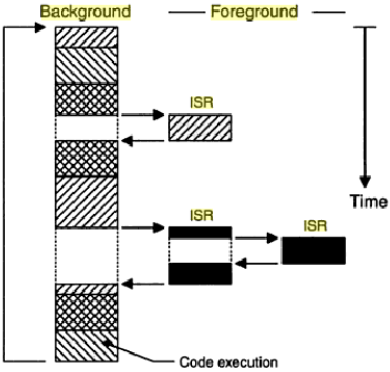
Foreground/background tasks

- Simple multitasking
- Super loop calls functions for computation (background)
- Interrupt service routines (ISRs) handle asynchronous events (foreground)



Foreground/background tasks

- Simple multitasking
- Super loop calls functions for computation (background)
- Interrupt service routines (ISRs) handle asynchronous events (foreground)



Problem

Model with only one computation task is not flexible enough

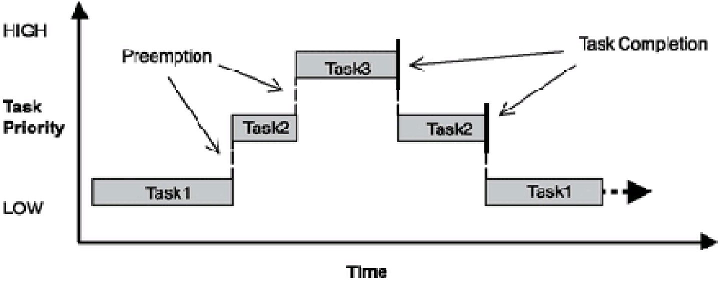
Multitasking

- Easier to structure the application as a set of concurrent tasks rather than as a single program or as foreground/background tasks: each task is responsible for some well-defined part of the system's overall behaviour
- But only the illusion of concurrency - the OS switches quickly between tasks, executing some instructions from one task before moving on to another task
- switching from one text to another requires a context switch
- deciding which task to switch to requires a scheduling algorithm

Scheduling

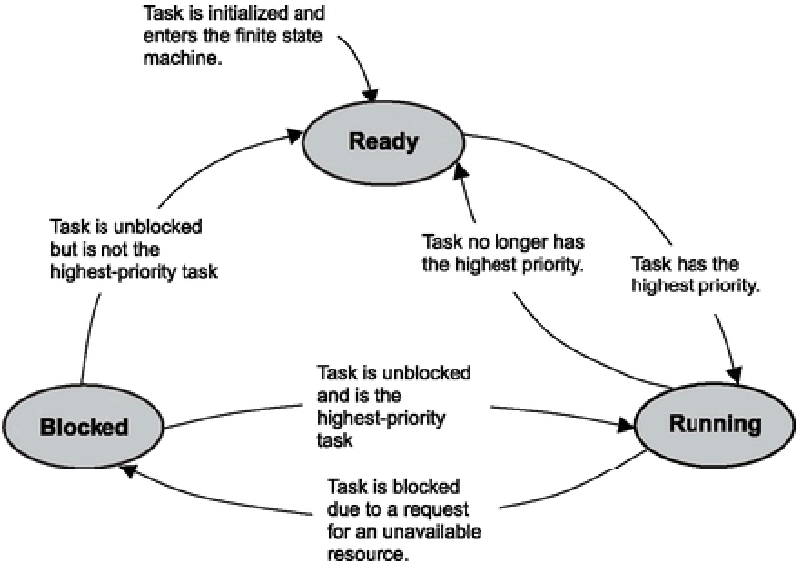
- Deciding when one task should stop executing and which one should begin next is a scheduling problem
- Two main approaches to scheduling:
  - Preemptive scheduling
    - Task is forced to yield the CPU
    - Round robin
    - Priority-based
  - Non-preemptive (cooperative) scheduling
    - Task voluntarily yields the CPU and signals the next task to begin

Fixed-priority preemptive scheduling



- We focus on fixed-priority premptive scheduling

A task and its execution states



uC/OS-II: A small operating system

- Main features:
  - Multi-tasking
  - Preemptive
- Other features:
  - Predictable
  - Robust and reliable
  - Standards-compliant
  - Portable
  - Scalable
  - Source code available

uC/OS-II Services

- Task management
- Delay management

uC/OS-II Services

- Task management
- Delay management
- Semaphores
- Mutual exclusion semaphores
- Event flags
- Message mailboxes
- Message queues
- Memory management
- Timers
- Miscellaneous

uC/OS-II Services

- Task management
- Delay management
- Semaphores
- Mutual exclusion semaphores
- Event flags
- Message mailboxes
- Message queues
- Memory management
- Timers
- Miscellaneous

See uC/OS-II Quick Reference

Tasks behaviour

- The behaviour of a task is defined by a C function that:
  1. never terminates
  2. blocks repeatedly

Example of task behaviour definition

```
static void appTaskLED2(void *pdata) {
  while (true) {
    OSTimeDlyHMSM(0,0,0,500);
    gpioPinToggle(&pin[LED2]);
  }
}
```

Tasks: other requirements

Priority

- Used for fixed-priority pre-emptive scheduling
- a number between 0 and `OS_LOWEST_PRIO`
- low number ⇒ high priority
- high number ⇒ low priority
- OS reserves priorities 0 to 3 and `OS_LOWEST_PRIO` - 3 to `OS_LOWEST_PRIO`
- Advice: define an enumeration of task priority constants, starting at priority level 4.
- Example

```
enum {
  APP_TASK_BUTTONS_PRIO = 4,
  APP_TASK_LED1_PRIO,
  APP_TASK_LED2_PRIO
};
```

Tasks: other requirements

Stack

- Each task needs its own data area (stack) for storing
  - context
  - local variables
- Example stack definition

```
enum {APP_TASK_LED2_STK_SIZE = 256};
static OS_STK appTaskLED2Stk[APP_TASK_LED2_STK_SIZE];
```

Tasks: other requirements

Stack

- Each task needs its own data area (stack) for storing
  - context
  - local variables
- Example stack definition

```
enum {APP_TASK_LED2_STK_SIZE = 256};
static OS_STK appTaskLED2Stk[APP_TASK_LED2_STK_SIZE];
```

User data

- Optionally tasks can be given access to user data when they are created
- We will not use this feature in this module
- Advice: always specify this as `(void *)0` when creating a task

Task creation

- A task is created using the OS function

```
INT8U OSTaskCreate(
        void (*task)(void *pdata), /* function for the task    */
        void *pdata,               /* user data for function   */
        OS_STK *ptos,              /* pointer to top of stack  */
        INT8U priority             /* task priority            */
);
```

Task creation

- A task is created using the OS function

```
INT8U OSTaskCreate(
        void (*task)(void *pdata), /* function for the task   */
        void *pdata,               /* user data for function  */
        OS_STK *ptos,              /* pointer to top of stack */
        INT8U priority             /* task priority           */
);
```

Example

```
enum {APP_TASK_LED2_PRIO = 4};
enum {APP_TASK_LED2_STK_SIZE = 256};

static OS_STK appTaskLED2Stk[APP_TASK_LED2_STK_SIZE];

OSTaskCreate(appTaskLED2,
             (void *)0,
             (OS_STK *)&appTaskLED2Stk[APP_TASK_LED2_STK_SIZE - 1],
             APP_TASK_LED2_PRIO);
```

Task delay

- Often, a task will block itself by explicitly asking the OS to delay it for some period of time
- `void OSTimeDly(INT16U ticks);`
- Causes a context switch if `ticks` is between 1 and 65535
- If `ticks` is 0, `OSTimeDly()` returns immediately to caller
- On context switch uC/OS-II executes the next highest priority task
- Task that called `OSTimeDly()` will be made ready to run when the specified number of ticks elapses - actually runs when it becomes the highest priority ready task
- Resolution of the delay is between 0 and 1 tick
- Another task can cancel the delay by calling `OSTimeDlyResume()`

Task delay

- `OSTimeDly()` specifies delay in terms of a number of ticks
- Use `OSTimeDlyHMSM()` to specify delay in terms of Hours, Minutes, Seconds and Milliseconds
- Otherwise `OSTimeDlyHMSM()` behaves as `OSTimeDly()`

Complete example

```
#include <stdbool.h>
#include <ucos_ii.h>
#include "gpioPin.h"

typedef enum {
  APP_TASK_LED1_PRIO = 4,
  APP_TASK_LED2_PRIO
} taskPriorities_t;

typedef enum {
  APP_TASK_LED1_STK_SIZE = 256,
  APP_TASK_LED2_STK_SIZE = 256
} stackSizes_t;

static OS_STK appTaskLED1Stk[APP_TASK_LED1_STK_SIZE];
static OS_STK appTaskLED2Stk[APP_TASK_LED2_STK_SIZE];

static void appTaskLED1(void *pdata);
static void appTaskLED2(void *pdata);

typedef enum {
LED1 = 0,
LED2
} deviceNames_t;

gpioPin_t pin[2];
```

Complete example

```
int main() {

  /* Initialise the GPIO pins */
  gpioPinInit(&pin[LED1], 1, 18, OUTPUT_PIN);
  gpioPinInit(&pin[LED2], 0, 13, OUTPUT_PIN);

  /* Initialise the OS */
  OSInit();

  /* Create the tasks */
  OSTaskCreate(appTaskLED1,
               (void *)0,
               (OS_STK *)&appTaskLED1Stk[APP_TASK_LED1_STK_SIZE - 1],
               APP_TASK_LED1_PRIO);

  OSTaskCreate(appTaskLED2,
               (void *)0,
               (OS_STK *)&appTaskLED2Stk[APP_TASK_LED2_STK_SIZE - 1],
               APP_TASK_LED2_PRIO);


  /* Start the OS */
  OSStart();

  /* Should never arrive here */
  return 0;
}
```

Complete example

```c
static void appTaskLED1(void *pdata) {
  /* Start the OS ticker -- must be done in the highest priority task */
  SysTick_Config(SystemCoreClock / OS_TICKS_PER_SEC);

  /* Task main loop */
  while (true) {
    gpioPinToggle(&pin[LED1]);
    OSTimeDlyHMSM(0,0,0,500);
  }
}

static void appTaskLED2(void *pdata) {
  while (true) {
    gpioPinToggle(&pin[LED2]);
    OSTimeDlyHMSM(0,0,0,500);
  }
}
```

Acknowledgements

- Qing Li and Caroline Yao, Real-time concepts for embedded systems, CMP, 2003
- Jean Labrosse, MicroC/OS-II: The Real-Time Kernel, CMP, 2002