

TW rev. 12.03.2024

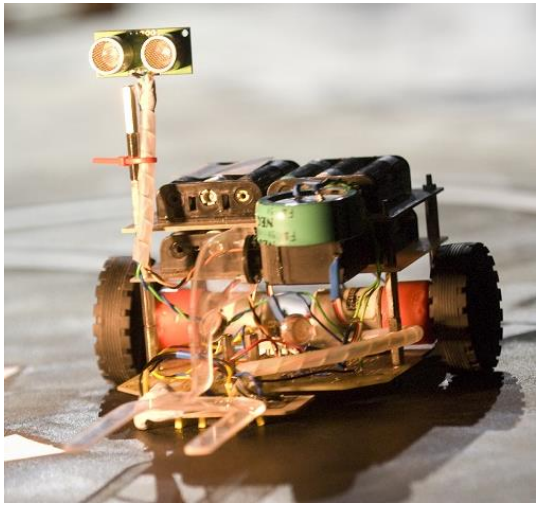
Chapter 9: Programming in Real Time

If you use or reference these slides or the associated textbook, please cite the original authors' work as follows:

Wilmshurst, T., Spink, T. & Toulson, R. (2024). Fast and Effective Embedded Systems Design – From Bits and Bytes to IoT, with the ARM Mbed (3rd edition), Newnes, Oxford, ISBN: 978-0323951975.

www.embedded-knowhow.co.uk

Multitasking and Real Time



Almost every embedded system has more than one thing that it needs to do.

A program for a simple autonomous vehicle may need to sense its environment through proximity and light sensors, measure the distance it has moved, keep track of its battery charge, move its manipulator arm, and calculate and implement drive values for its motors.

As a system becomes more complicated, it becomes more difficult to balance the needs of the different things it does. Each will compete for CPU time and may therefore cause delays in other areas of the system.

Systems which have to meet critical deadlines in order to function properly are said to be operating in *real time.* A simple but completely effective definition of real time, often used by veteran real-time programmer David Kalinsky, is as follows:

***A system operating in real time must be able to provide the correct results at the required time deadlines*.**

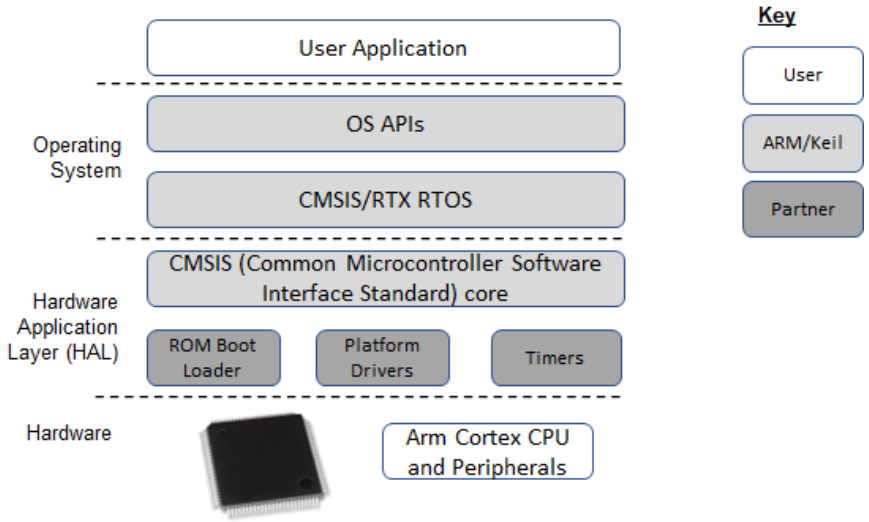
The Real Time Operating System (RTOS)

The RTOS offers a mechanism for programming in a multitasking, real time environment. It requires a different approach to program development, and takes us far from the assumptions of traditional sequential programming.

With the RTOS, we hand over control of the CPU and system resources to the OS. It is the OS which now determines which section of the program is to run and for how long, and how it accesses system resources. The application program is subservient to the OS, and is written to recognise the demands of the OS.

An RTOS itself is a general-purpose program framework. It is adapted for a particular application by the programmer writing threads for it and by customising it in other ways.

While you can write your own RTOS, it is pretty much a specialist activity and generally best done by specialists.



Multitasking, Tasks and Threads

Multitasking describes a situation where there are many tasks which need to be performed, ideally simultaneously. A program written for an RTOS is structured into tasks or *threads.*

While there are differences between the use of the words task and thread, let's simplify by saying that the task is the thing that needs to be done, and the thread is the code that delivers it.

A thread is *a program strand or section which has a clear and distinct purpose and outcome.*

With the RTOS, each thread is written as a self-contained program module, a bit like a function. The threads can be prioritised, though this is not always the case.

The RTOS performs three main functions:

- It decides which thread should run and for how long, this is called *scheduling*,
- It provides communication and synchronisation between threads,
- It controls the use of resources shared between the threads, for example memory and hardware peripherals.

Scheduling, Cyclic Scheduling

A central part of the RTOS is the scheduler. This determines which thread is allowed to run at any particular moment. The scheduler must be aware of what threads are ready to run and any priorities. There are a number of different scheduling strategies.

Cyclic scheduling is the simplest scheduling form, and reflects the superloop program structure. Each thread runs to completion before it hands over to the next, and cannot be discontinued as it runs. The threads may take a different duration each time they come, so the following thread may have to wait an unpredictable time before it can execute.

In the diagram the horizontal strand represents program execution; the thread being executed at any one time is indicated by a shaded, numbered box.



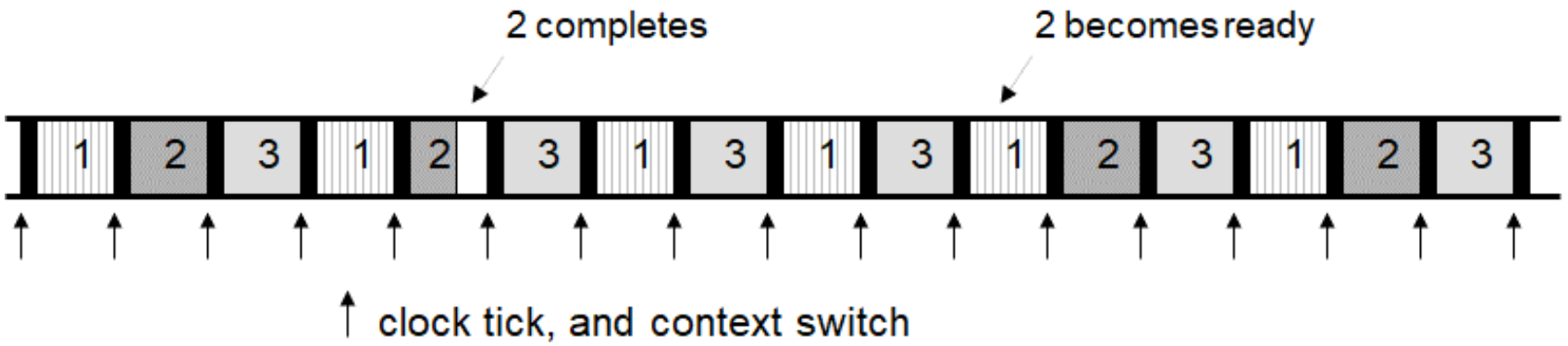
Round Robin Scheduling, Pre-emption, Context Saving

In round-robin scheduling the OS is driven by a regular interrupt (the ‘clock tick’). Threads are selected in a fixed sequence for execution. On each clock tick, the current thread is discontinued and the next is allowed to start execution.

All threads are treated as being of equal importance and wait in turn for their slot, or “time slice”, of CPU time. Threads are not allowed to run to completion, but are pre-empted, i.e. their execution is discontinued mid-flight.

When the thread is allowed to run again, it must be able to pick up operation seamlessly, with no side-effect from the pre-emption.

Therefore, complete context saving (all flags, registers, relevant memory locations) must be undertaken as the thread switches. This places demands on execution time and memory.



Prioritised Pre-emptive Scheduling

The scheduler is still run by a clock tick, but threads are prioritised.

High priority threads are allowed to complete before any time is given to lower priority threads. On every tick the OS checks which ready thread has the highest priority, that then gets access to the CPU.

An executing thread which still needs CPU time, and is highest priority, keeps the CPU. A low priority thread which is executing is replaced by one of higher priority, if that has become ready.

This can be considered “classic” RTOS action and is worth understanding well.

Prioritised Pre-emptive Scheduling

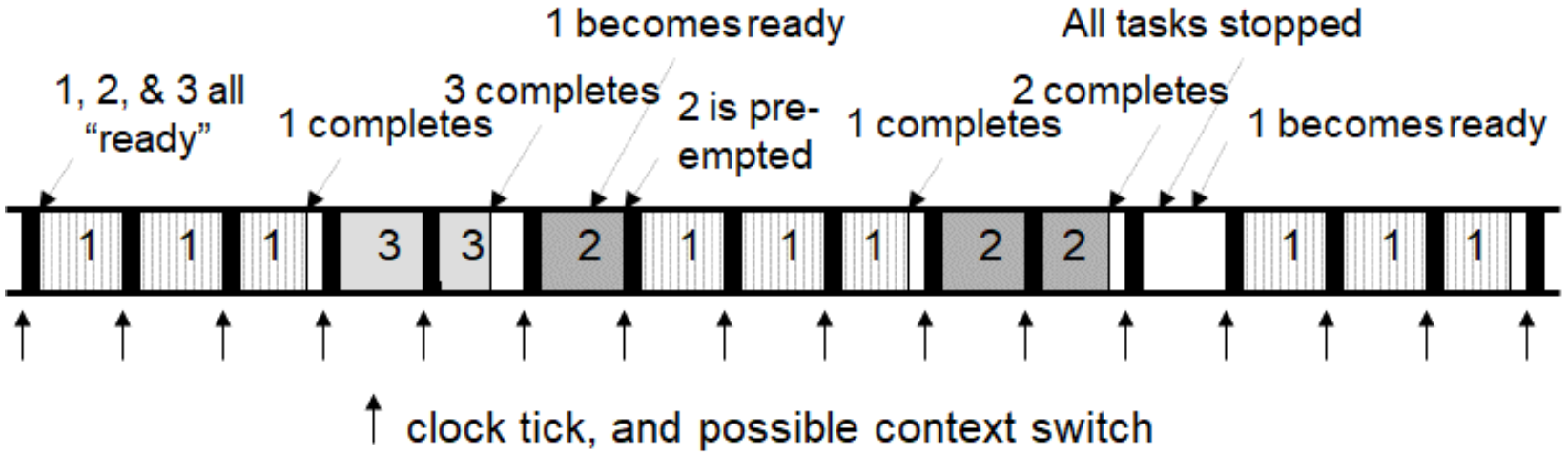
The diagram shows three threads, each of different priority and execution duration. At the beginning, all are ready to run.

Because Thread 1 has the highest priority, the scheduler selects it to run. At the next clock tick, Thread 1 still needs time, so it is allowed to continue. The same happens at the next clock tick, the thread completes during the next time slice. Thread 1 does not now need CPU time and becomes suspended.

At the next clock tick the scheduler selects the ready thread which has the highest priority, i.e. Thread 3. This also runs to completion.

Thread 2 then finally gets CPU time, only to be pre-empted.

Thread	Priority	Duration (in time slices)
1	1 (highest)	2.7
2	3	2.8
3	2	1.5



Cooperative Scheduling

While pre-emptive scheduling is very effective, it does create demands on time and memory when context switching.

In cooperative scheduling the tasks themselves relinquish CPU access, and thus manage their own context saving, if needed.

Cooperative scheduling is unlikely to be as responsive to tight deadlines as prioritised pre-emptive, but for a small system of limited resources it may prove to be the most effective option.

Thread States

As the RTOS runs, threads move between different states. A possible state diagram is shown. Note that the terminology used and the way the state is managed vary to some extent from one RTOS to another.

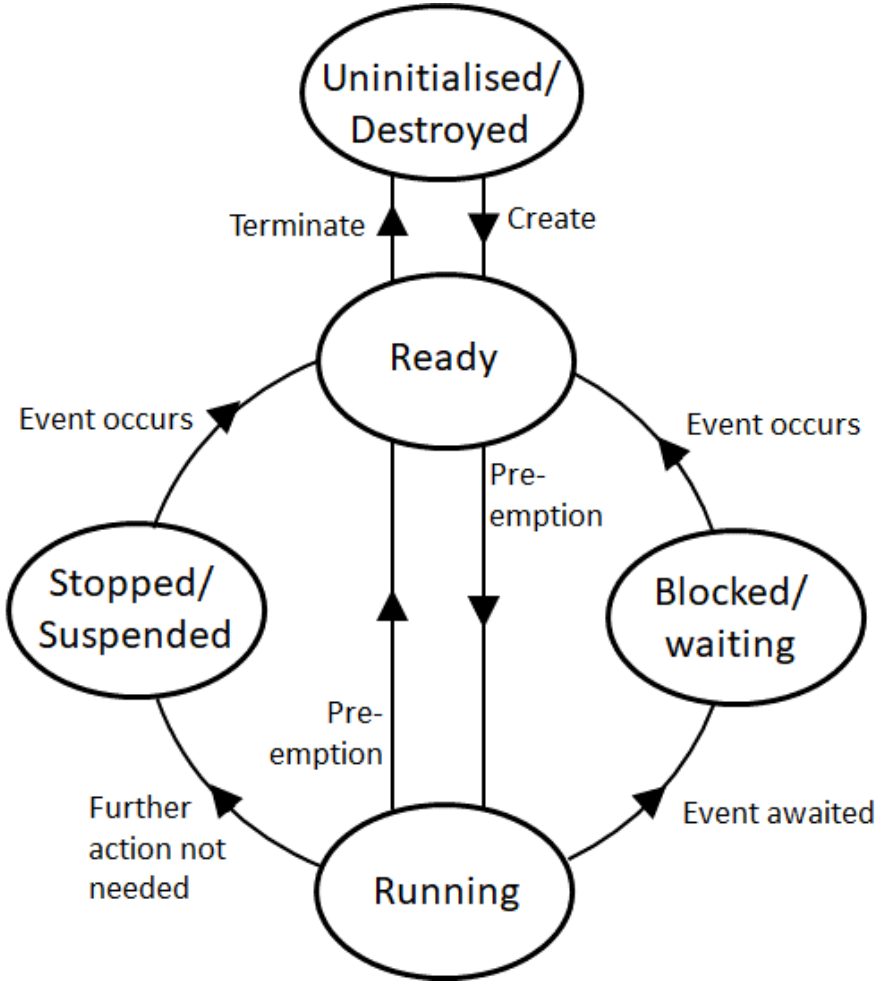
Ready (or eligible). The thread is ready to run and will do so as soon as it is allocated CPU time.

Running (or active). The thread has been allocated CPU time and is executing.

Blocked/waiting. The thread is ready to run, but for one reason or another is not allowed to.

Stopped/suspended. The thread does not at present need CPU time.

Uninitialised/destroyed. The thread no longer exists as far as the RTOS execution is concerned.



The Mbed Real Time Operating System

The Mbed RTOS applies the principles which just introduced. Its underlying scheduling strategy is prioritised pre-emptive scheduling; for threads of equal priority it applies round-robin scheduling. The default time slice is 1 ms.

The scheduling capabilities of the Mbed OS give it its RTOS qualities. The APIs fall into the categories shown (in reverse alphabetical order).

API Classes/features	Summary
Thread	Defines, creates and controls parallel threads.
ThisThread	Provides features for control of the current thread.
Semaphore	Manages thread access to a pool of shared resources of a certain type.
Queue	Enables queueing of pointers to data from producer threads to consumer threads.
Mutex	Protects access to a shared resource, and synchronizes the execution of threads.
Mail	Provides a queue combined with a memory pool for allocating messages.
MemoryPool	Defines and manages memory pools of fixed size.
Kernel interface functions	Namespace implements functions to control or read RTOS information, such as tick count.
IdleLoop	Background system thread, executed when no other threads are ready to run.
EventFlags	An event channel that provides a way of notifying other threads about condition changes or events.
ConditionVariable	Provides a mechanism to wait for or signal state changes, for example a mutex change.

The Thread API

Fundamental to successful use of the RTOS is the creation and control of threads. This is done using the API summarised here.

Name/Format	Description
Constructor	
`Thread (osPriority priority, uint32_t stack_size, unsigned char *stack_mem, const char *name)`	Allocate a new thread without starting execution, setting (as needed) priority, stack size, stack pointer and thread name.
Member Functions	
`uint_32 flags_set(uint_32 flags)`	Set the specified flags for the thread, as indicated by flags.
`osPriority get_priority()`	Get priority of an active thread.
`osStatus set_priority (osPriority priority)`	Set priority of an active thread.
`osStatus join()`	Wait for thread to terminate (then calling function can continue).
`osStatus start (mbed::Callback<void()> thread)`	Start a thread executing the function specified by thread. Returns status code that indicates the execution status of the function.
`State get_state()`	Get the state of this thread. Returns code indicating thread status.
`osStatus terminate()`	Terminate execution of a thread and remove it from active threads.

ThisThread Namespace

It will also be necessary to start using functions from the **ThisThread** namespace, with examples shown in the Table. Here at last we see the **ThisThread::sleep_for( )** function, that we've been using extensively.

Name/Format	Description
`uint32_t` **`flags_clear`** `(uint32_t` *`flags`*`)`	Clears the specified thread flags of the currently running thread.
`uint32_t` **`flags_get`** `()`	Returns the thread flags currently set for the currently running thread.
`uint32_t flag`**`s_wait_all`** `(uint32_t` *`flags`*`, bool` *`clear`*`)`	Wait for all of the specified thread flags to become signalled for the current thread, clear the flags if *clear* = 1.
`uint32_t` **`flags_wait_any`** `(uint32_t` *`flags`*`, bool` *`clear`*`)`	Wait for any of the specified thread flags to become signalled for the current thread, clear the flags if *clear* = 1.
`void` **`sleep_for`** `(chrono duration)`	Sleep for the specified time period.
`void` **`sleep_until`** `(chrono duration)`	Sleep until the specified time.
`void` **`yield`** `()`	Pass control to next equal-priority thread that is in ready state.

Writing Multi-threaded Programs: Defining and Writing Threads

Threads should be written as if they are to run continuously, as self-contained functions.

They cannot call on a section of another's code, but can access common code, for example C libraries. They may depend on services provided by each other and may need to be synchronised with each other.

The number of threads created should not be too many. More threads imply more programming complexity, and for every thread switch there is a time and memory overhead.

A set of activities which are closely related in time are likely to serve similar deadlines and should therefore be grouped together into a single thread.

A set of activities which are closely related in function and interchange a large amount of data may also be grouped into a single thread.

A Two-Threaded Program

This is a simple example program. The first thread is **main( ).** A second thread, **thread2**, is created with the **Thread** constructor. It is immediately started within **main( ),** using the **start( )** member function. The two switch their LEDs indefinitely.

```
/* Program Example 9.1. Two threads, of which one is main(), run indefinitely.
Runs on LPC1768 or F401RE (with LEDs shown in Fig 3.1) */

#include "mbed.h"

DigitalOut led1(LED1); //Use A0 for F401RE
DigitalOut led2(LED2); //Use A1 for F401RE

Thread thread2;    //Create thread2

void led2_thread(){
  while (true) {
    led1 = !led1;
    ThisThread::sleep_for(1100ms);
  }
}
int main(){
  thread2.start(led2_thread);    //Launch thread2.
  while (true) {
    led2 = !led2;
    ThisThread::sleep_for(500ms);
  }
}
```

One Thread Controlling Another

This example shows how one thread can both start and terminate another, with **thread2** being started from the **main( )** thread. However, after a wait of 10 s **thread2** is stopped by the **main( )** thread, through setting the **running** variable to false.

```
/* Program Example 9.2. A program in which one thread starts and stops another.
Runs on LPC1768 or F401RE */

#include "mbed.h"
Thread thread2;   //name thread2 chosen for this thread
DigitalOut led1(LED1);
volatile bool running = true;

// Blink function toggles the led in a long running loop
void blink() {
  while (running) {
    led1 = !led1;
    ThisThread::sleep_for(1s); //set blink rate
  }
}

int main() {                // Launches a thread which then runs for 10 seconds
  thread2.start(blink);          //Starts thread2 executing.
  //Set how long thread2 will be able to run
  ThisThread::sleep_for(10s);
  running = false;
  thread2.join();       //Wait for thread2 to terminate.
}
```

A Multi-threaded Program (1)

The program in the next three slides is a more practical example. It uses features of the Mbed app board (standalone components can also be used), notably the temperature sensor, the speaker, two potentiometers, the LCD and the multicolored LED. The program contains three threads:

main( ), which calls a diagnostic function, launches the other threads, and then sleeps;

temp_read_Th, which reads the on-board temperature sensor;

display_Th, which displays values read, and controls the LED and speaker.

It should not be difficult to follow the program through. Each thread is moderately self-contained in structure. A striking thing about the program is that it contains no less than three **while(true)** loops, one per thread; in conventional programming this would not be possible.

A Multi-threaded Program (2)

```
/* Program Example 9.3. An RTOS-based program applying the Mbed App Board (or
separate
components).
Works with LPC1768 or F401RE, with pin connections
shown.                        */

#include "mbed.h"
#include "C12832.h"
#include "LM75B.h"    //note: more than one version of LM75V available on Mbed
web

C12832 lcd(p5, p7, p6, p8, p11); //use D11,D13,D12,D10,D9 for F401RE
LM75B tmp(p28,p27);   //I2C Temperature Sensor. Use D8,D2 for F401RE
PwmOut red(p23);      //Use D2 for F401RE
PwmOut grn(p24);      //Use D3 for F401RE
PwmOut blu(p25);      //Use D4 for F401RE
PwmOut speaker(p26);  //Speaker with PWM driver. Use D7 for F401RE
AnalogIn pot1(p19);   //Pot 1 (near LCD. Sets alarm threshold. Use A4 for
F401RE
AnalogIn pot2(p20);   //Pot 2 (near RGB LED. Sets LED intensity. Use A5 for
F401RE

float board_temp;
float alarm_temp;

//Function prototypes
void diag(void);
void temp_read (void);
void display_fn (void);
//continued over
```

A Multi-threaded Program (3)

```
//Threads
Thread Temp_read_Th;
void temp_read (void){
  while(true){
    board_temp = tmp.temp();     //read temperature from sensor
  }
}
//This thread deals with both display and alarm
Thread Display_Th;
void display_fn (void){
  speaker.period(1.0/800.0); // 800 Hz period
  red = grn = blu = 1.0; //PWM 100% duty cycle, all LEDs off. Blue remains off throughout
  while(true){
    lcd.cls();       //clear LCD
    lcd.locate(0,0);
    lcd.printf("Board Temperature = %0.2f\n\r", board_temp);
    alarm_temp = 50.0 * pot1;
    lcd.printf("Temp Alarm Setting = %0.2f\n\r", alarm_temp);
    if(board_temp > alarm_temp) {   //check temp for alarm
      red = 1.0 - pot2;             //RGB LED red
      grn = 1.0;
      speaker = 0.5;                // PWM is 50%, alarm tone sounds
    }
    else{
      grn = 1.0 - pot2;             //RGB LED green
      red = 1.0;
      speaker = 0.0;
    }
    ThisThread::sleep_for(2s);
  }
}
//continued over
```

A Multi-threaded Program (4)

```
int main(){
  diag();
  Temp_read_Th.start(temp_read);  //initialise threads
  Display_Th.start(display_fn);
  while (true) {
    ThisThread::sleep_for(2s);
  }
}
//Diagnostic; flash LED colors to indicate successful program launch
void diag(void){
  red = 0, grn = 1, blu = 1;
  ThisThread::sleep_for(500ms);
  red = 1, grn = 0, blu = 1;
  ThisThread::sleep_for(500ms);
  red = 1, grn = 1, blu = 0;
  ThisThread::sleep_for(500ms);
  red = 0, grn = 1, blu = 0;
  ThisThread::sleep_for(500ms);
}
```

a) the main.cpp file

```
{
    "target_overrides":{
        "*": {
            "target.printf_lib": "std"
        }
    }
}
```

b) the mbed_app.json file

Setting Thread Priority

In most programs, threads are not all of equal importance. A high priority thread has the right to execute before a low priority thread. Each thread also has a deadline, or could have one estimated for it.

Generally an RTOS allows the programmer to set thread priorities.

In the case of *static* priority, priorities are fixed. In the case of *dynamic* priority, priorities may be changed as the program runs.

One way of determining priority is to consider how important a thread is to the operation and well-being of the system, its user and environment. A simple three-level distinction is:

- Highest priority: threads essential for system survival
- Middle priority: threads essential for correct system operation
- Low priority: threads needed for adequate system operation – these threads might occasionally be expendable or a delay in their completion might be accepted.

Setting Thread Priority

The **Thread** API Table has shown that there are two mechanisms in the Mbed OS to set a priority, either as the thread is created, or by using the **set_priority( )** member function.

This Table shows a selection of priority levels available. A name and value is defined for each, with the higher value implying higher priority. The default **osPriorityNormal,** unless specified otherwise, is applied to every new thread. It is also the priority of the **main( )** thread.

Priority Level	Value
osPriorityIdle	1
osPriorityLow	8
osPriorityBelowNormal	16
osPriorityNormal	24
osPriorityAboveNormal	32
osPriorityHigh	40
osPriorityRealtime	48
osPriorityError	-1

Programming with Thread Priorities (1)

The following program example explores the behavior of three threads as their priorities are changed (hence these are dynamic priorities).

Each thread simply blinks an LED. The blink rate of **thread3** is determined by invoking a **sleep_for()** function.

The blink rates of **thread1** and **thread2** are determined by coded timing loops within the thread. Thus each of these threads are permanently “busy”.

The **main( )** function first launches the three threads, which run for 10 s. Priorities are then changed, and the program executes for a further 10 s, and so on.

Each thread is promoted to top priority for one 10 s period. These are kept equal to or below Normal.

In this program, if a priority is set to higher than Normal, then the high priority task may execute continuously, not allowing **main( )** to execute again and reset the priorities.

Programming with Thread Priorities (2)

```
/* Program Example 9.4. Demonstrates the setting of thread priorities.
Runs on LPC1768 or F401RE (with LEDs shown in Fig 3.1)  */

 #include "mbed.h"
DigitalOut led1(LED1);
DigitalOut led2(LED2);     //Use A0 for F401RE
DigitalOut led3(LED3);     //Use A1 for F401RE
DigitalOut led4(LED4);     // Use A2 for F401RE, the LED is not essential, but useful to
                                  //indicate timing loops are running

Thread thread1;   //create three threads
Thread thread2;
Thread thread3;

void thread1_fn(){
  while(true){
    led1 = !led1;
    for (int j = 0;j < 1000000;j++) {
    led4 = !led4;  //do something, or compiler can "optimise out" this loop
    }
  }
}
void thread2_fn(){
  while(true){
    led2 = !led2;
    for (int i = 0;i < 1000000;i++) {
    led4 = !led4;  //do something, or compiler can "optimise out" this loop
    }
  }
}
//continued over
```

Programming with Thread Priorities (3)

```
void thread3_fn(){
  while (true){
    led3 = !led3;
    ThisThread::sleep_for(1s);
  }
}
int main(){
  thread1.start(thread1_fn); //Starts thread1 executing.
  thread2.start(thread2_fn); //Starts thread2 executing.
  thread3.start(thread3_fn); //Starts thread3 executing.
  //run for 10s equal priorities
  ThisThread::sleep_for(10s);
  while(true){            //changes thread priorities every 10 seconds
    thread1.set_priority(osPriorityNormal);
    thread2.set_priority(osPriorityBelowNormal);
    thread3.set_priority(osPriorityLow);
    ThisThread::sleep_for(10s); //run for 10s new priorities
    thread1.set_priority(osPriorityLow);
    thread2.set_priority(osPriorityNormal);
    thread3.set_priority(osPriorityBelowNormal);
    ThisThread::sleep_for(10s); //run for 10s new priorities
    thread1.set_priority(osPriorityLow);
    thread2.set_priority(osPriorityBelowNormal);
    thread3.set_priority(osPriorityNormal);
    ThisThread::sleep_for(10s);
  }
}
```

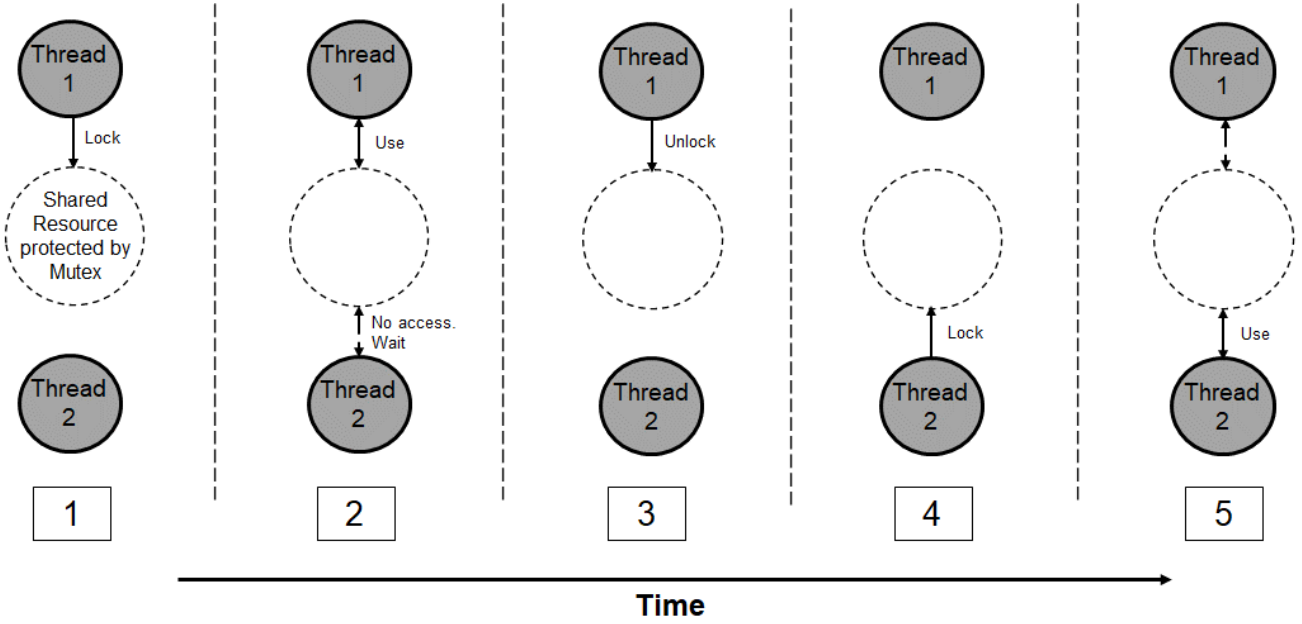
The Mutex

With the thread switching going on in an RTOS, new issues arise. One thread, using a certain resource, may be pre-empted before that use is complete. The next thread may then try to use the same resource, before the first has finished with it. This could be damaging to overall program performance.

The Mutex (from *mutual exclusion*) allows controlled access to shared resources, as illustrated below, showing a program having two threads and a mutex.

A single resource is shared by the threads. Thread 1 uses it first so locks the Mutex, denying the other thread the chance to access. When it has completed, it unlocks the Mutex. Thread 2 can then use the resource, now locking the Mutex itself.

Only the thread which has locked the Mutex can unlock it.



The Mutex API

Selections from the Mutex API are shown in the Table.

Name/Format	Description
Constructor	
`Mutex (const char *name)`	Create and Initialize a Mutex object.
Member Functions	
`void lock()`	Wait until a Mutex becomes available.
`bool trylock()`	Try to lock the mutex, and return immediately, returns true if mutex acquired.
`void unlock()`	Unlock the mutex that has previously been locked by the same thread.

A Mutex Program (1)

It would be simple to demonstrate the need for a mutex by using **printf( ),** but **printf( )** is already protected in the Mbed OS by an inbuilt mutex.

The following program example simulates a simple control system, in which two processes are being monitored, with status information output to a display.

It uses **printf( ),** placing it in a function used by two threads, i.e. a shared resource. However each access to **printf( )** prints only a single character, making the overall message printing vulnerable to corruption due to shared access.

The program has three threads. One is **main( ).** Two further threads are called **Process1** and **Process2.** Each of these accesses a display function called **printing( ).**

A mutex, called **print_mutex,** is available to protect the **printing( )** function. The mutex is locked and then unlocked by each thread before and after access to **printing( ).**

A Mutex Program (2)

```
/* Program Example 9.5. Mutex applied to a printing application.
Works on LPC1768 and F401RE with no external connections
                                        */
#include "mbed.h"

DigitalOut led1(LED1);
Thread Process1;
Thread Process2;
Mutex print_mutex;    //Create and name Mutex

void printing (int);
//2 possible messages for display
char messages[] = "System operational  Boiler overheating";

void thread1_fn(){         //Thread for Process1
  while(true){
    print_mutex.lock();
    printing(0);           //pass pointer to "System operational" message
    print_mutex.unlock();
    ThisThread::sleep_for(800ms);
  }
}
//continued over
```

A Mutex Program (3)

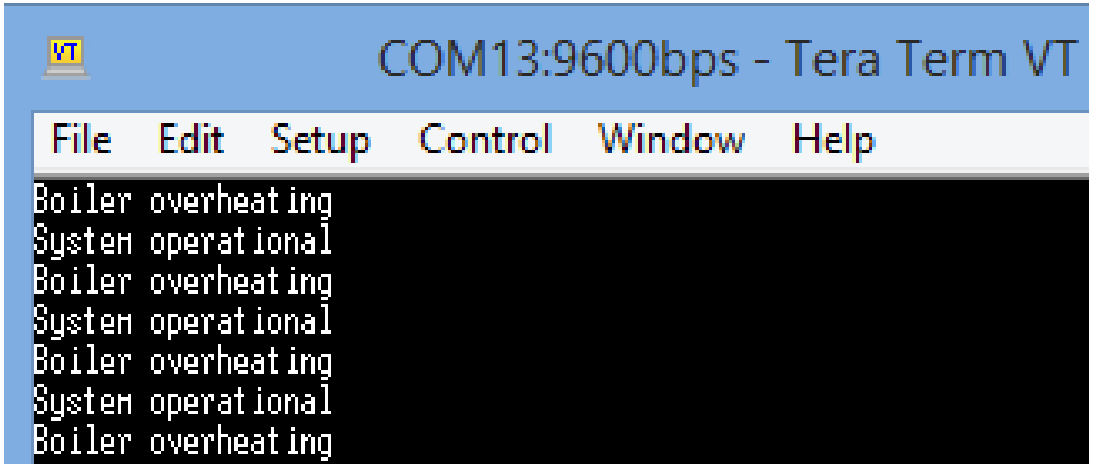
```
void thread2_fn(){        //Thread for Process2
  while(true){
    print_mutex.lock();
    printing(20);         //pass pointer to "Boiler overheating" message
    print_mutex.unlock();
    ThisThread::sleep_for(800ms);
  }
}

void printing (int msg_ptr){    //print the message passed with msg_ptr
  for (char i = 0; i < 18; i++){
    printf("%c",messages[msg_ptr+i]);//print a single character from the string
    wait_us(2000);                   //represents other processing commitments
  }
  printf("\n");
}

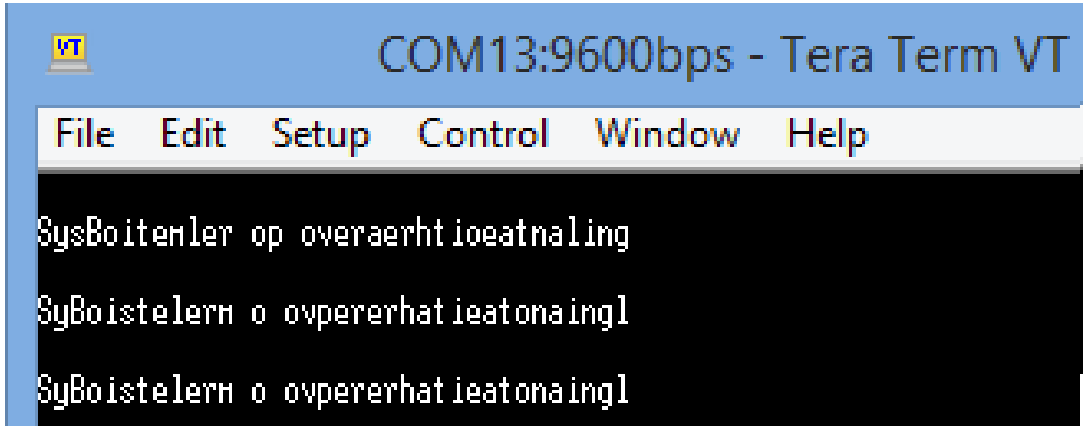
int main(){
  Process1.start(thread1_fn); //Start Process1 executing.
  Process2.start(thread2_fn); //Start Process2 executing.
  while(true){
    led1 = !led1;
    ThisThread::sleep_for(1s);
  }
}
```

A Mutex Program (4)

With Mutex in place the screen output is as shown here.



With Mutex removed scrambling of the messages occurs.



The Semaphore

An RTOS Semaphore extends the Mutex concept. It is an integer created in the RTOS, with an initial value determined by the programmer. This may be used to manage access to a certain shared resource, or group of resources.

The semaphore integer can be incremented (but not above its initial value) or decremented (but not below zero).

The number can be thought of as being a number of tokens, taken or returned within the program, as a resource is accessed.

If a thread tries to decrement the semaphore and the result would be negative, then that thread enters a blocked state. It needs another thread to increment the semaphore, whereupon it can proceed.

The value of the semaphore hence represents the number of threads that can access the semaphore, without being blocked.

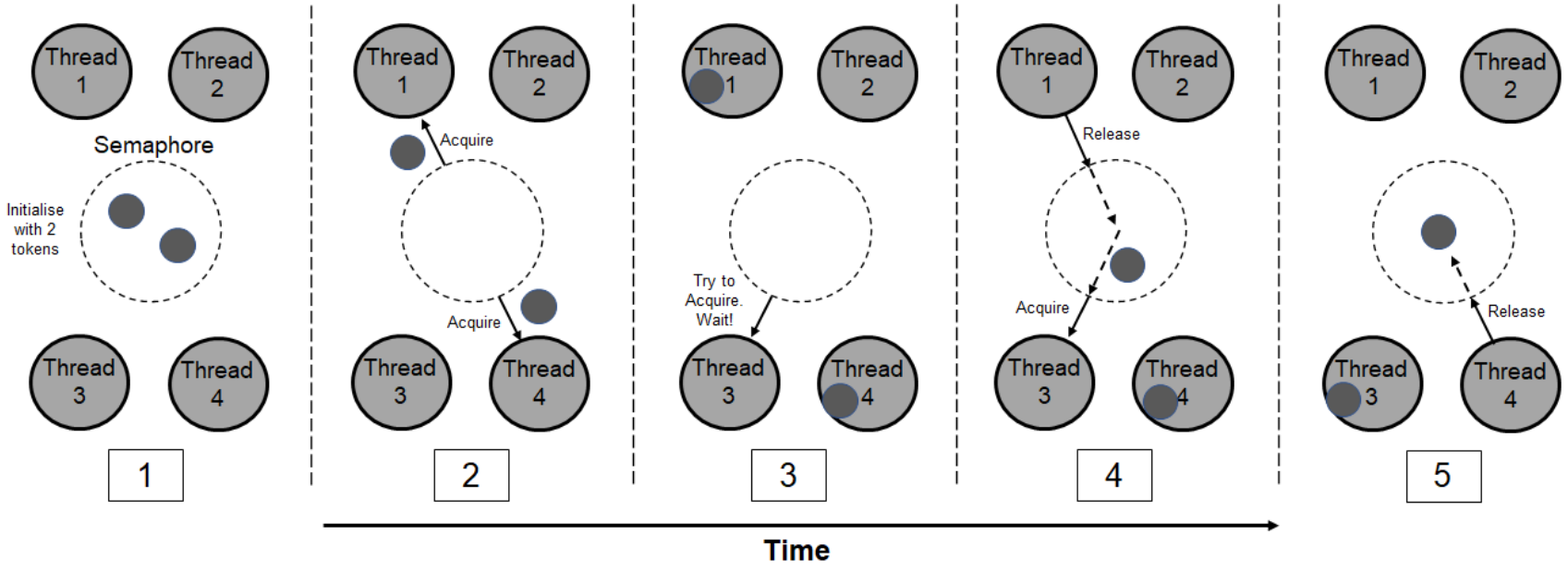
The value is not generally known as the program executes, so a thread accessing the semaphore will not “know” if it’s about to be blocked or not. Nor does it know if by incrementing the semaphore it will release another thread from the blocked state.

A zero value of the semaphore implies that the next thread to decrement it will become blocked.

The Semaphore

The concept is illustrated here. A semaphore is initialised with two tokens in a program which has four threads running.

Early in program execution, Threads 1 and 4 each take a token, leaving none. A little later, Thread 3 attempts to take a token. As none are there, the thread is required to wait, entering a blocked state. Thread 1 then releases a token. This is acquired by the waiting Thread 3, which can then proceed. In the final phase shown, Thread 4 releases a token, available to the next thread which needs it.



Programming with Semaphores

The main features of the Mbed Semaphore API are shown in the Table.

Name/Format	Description
Constructor	
`Semaphore (int32_t count, uint16_t max_count)`	Create and Initialize a Semaphore object used for managing resources, where count is number of available resources.
Member Functions	
`void acquire()`	Acquire a Semaphore resource, wait if needed till available.
`bool try_acquire()`	Try to acquire a Semaphore resource, and return immediately. Returns true if a resource was acquired, false otherwise.
`osStatus release (void)`	Release a Semaphore resource that was obtained with acquire().

Programming with Semaphores

The following program example demonstrates simple semaphore action.

It simulates three similar processes running in an industrial system; from time to time each requires use of a certain resource, of which there are two.

Three threads, including **main( ),** are created. They have near identical, but non-synchronous activity. Each flashes an LED, first slowly, and then fast; this should run continuously.

To proceed to the fast flashing phase, where the shared resource is required, they must acquire a semaphore token.

A semaphore, **demo_sem**, loaded with two tokens, is created. The token is released by the thread, when the fast flashing phase is complete.

As there are only two items of resource and hence two tokens, there can only be two threads fast flashing at any one time. If a third wants to enter this phase, then it must wait till one of the others has completed and returned its token.

Programming with Semaphores

```
/* Program Example 9.6. Demonstrate Semaphore principles.
Works on LPC1768 and F401RE (with 2 extra LEDs)
                                        */
#include "mbed.h"

DigitalOut led1(LED1);
DigitalOut led2(LED2);
DigitalOut led3(LED3);

Thread Thread1;
Thread Thread2;

Semaphore demo_sem(2);    //Create Semaphore with 2 tokens

void thread1_fn(){        //Thread 1
  while(true){
    for (char i = 0;i < 8;i++){  //Phase 1 - Slow flashing
      led1 = !led1;
      ThisThread::sleep_for(1s);
    }
    demo_sem.acquire();  //Acquire a token
    for (char i = 0;i < 120;i++){  //Phase 2 - Use resource, fast flashing
      led1 = !led1;
      ThisThread::sleep_for(150ms);
    }
    demo_sem.release();  //Release the token
  }
}
//continued over
```

Programming with Semaphores

```
void thread2_fn(){   //Thread 2
  while(true){
    for (char j = 0;j < 12;j++){  //Phase 1 - Slow flashing
      led2 = !led2;
      ThisThread::sleep_for(750ms);
    }
    demo_sem.acquire();  //Acquire a token
      for (int j = 0;j < 400;j++){ //Phase 2 - Use resource, fast flashing
      led2 = !led2;
      ThisThread::sleep_for(120ms);
    }
    demo_sem.release();  //Release the token
  }
}

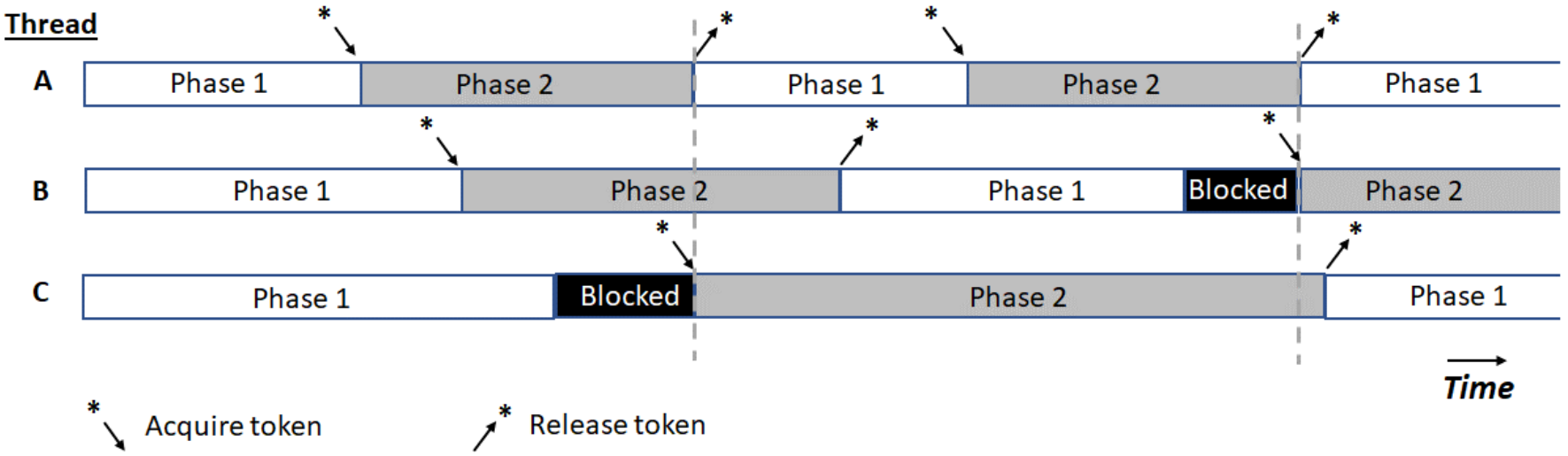
int main(){                      //Let's call this Thread 3
  Thread1.start(thread1_fn); //Start Thread 1 executing.
  Thread2.start(thread2_fn); //Start Thread 2 executing.
  while(true){
    for (char k = 0;k < 20;k++){ //Phase 1 - Slow flashing
      led3 = !led3;
      ThisThread::sleep_for(500ms);
    }
    demo_sem.acquire();   //Acquire a token
    for (char k = 0;k < 100;k++){  //Phase 2 - Use resource, fast flashing
      led3 = !led3;
      ThisThread::sleep_for(200ms);
    }
    demo_sem.release();  //Release the token
  }
}
```

Programming with Semaphores

The principle of this program example is further illustrated in the Timing Diagram below.

Three threads are labelled A, B and C, and each has a Phase 1 and Phase 2 of operation. To execute Phase 2 the thread must acquire a semaphore token, of which there are two. Tokens are symbolised by asterisks, with arrow directions indicating acquiring and releasing.

Task C is the first to be blocked, with both tokens being taken by the time it needs to execute Phase 2. Having acquired a token it can proceed. Later Thread B becomes blocked, for a similar reason.



Using Interrupts with the RTOS

Interrupts form an essential part of RTOS programming, but they must be used with care.

The first use of interrupts, unseen to the programmer, is almost always to provide the clock tick, provided by a timer interrupt on overflow. Incorrectly used interrupt service routines (ISRs) can hence upset the correct timing of the RTOS scheduling.

The ISR should not disturb the flow of the RTOS, so should execute in fastest possible time. Hence there are greater restrictions when working within an RTOS environment.

The best use of the ISR is to supply urgent information to the threads or scheduler. It could, for example, be set to signal that a certain event has occurred, thereby releasing a thread from a blocked state. The ISRs themselves are not normally used as threads.

Any function which is the least bit time-consuming should be avoided, including wait/delay loops, lengthy while loop, or calls to time-hungry library functions.

Using Interrupts with the RTOS

Many RTOS features simply aren't available in the ISR. The Table gives examples. Unsurprisingly, no constructors listed are usable.

The functions which can be used tend to be those which can compete simply and fast. For example, in the semaphore class, **acquire( )** is not available, as it might force a delay. However **try_acquire( )** can be used, as it returns immediately.

Function Name	Can it be used in an ISR?	Function Name	Can it be used in an ISR?
Thread		**Semaphore**	
constructor	X	constructor	X
get_priority()	X	acquire()	X
set_priority()	X	try_acquire()	√
start()	X	release()	√
join()	X	**Queue**	
Mutex		constructor	X
constructor	X	empty()	√
member functions	X	full()	√

Using Interrupts with the RTOS

In this example there are two threads, each of which simply flashes an LED. One however is dependent on the status of variable **running**, which the ISR reverses every time it is called. Reflecting good practice, the ISR is as short as possible, with all the action within the threads.

```
/* Program Example 9.7. An interrupt controls the action of a thread.
Runs on LPC1768 or F401RE (with LEDs shown in Fig 3.1) */

#include "mbed.h"
Thread thread2;   //name thread2 chosen for this thread
DigitalOut led1(LED1);
DigitalOut led2(A0);          //Replace A0 with LED2 for LPC1768
InterruptIn button(BUTTON1); //Replace BUTTON1 with p5 for LPC1768
volatile bool running = true;

// Blink function toggles the led in a long running loop
void blink(void) {
  while (true){
    if (running) {
      led1 = !led1;
      ThisThread::sleep_for(1s); //set blink rate
    }
  }
}
//continued over
```

Using Interrupts with the RTOS

```
void ISR1(void){
  running = !running;
}

int main() {                  // Launches a thread which runs for 10 seconds
  thread2.start(blink);       //Starts thread2 executing.
  button.rise(&ISR1);
  while(true){
    ThisThread::sleep_for(800ms);
    led2 = !led2;
  }
}
```

Queues

The queue acts as a data pipeline, with buffering, between two threads. It queues pointers to data, for example to pass data from threads which produce to threads which apply the data.

The data items can be prioritised, retrieved in order of descending priority. Equal priorities are withdrawn on a FIFO (first in, first out) basis.

Data stored can be integer, pointer, or a type specified by a user template.

An API summary is shown.

Name/Format	Description
Constructor	
`Queue <T, queue_sz>`	Create and initialize a message queue of objects, of data type T and maximum capacity queue_sz.
Member Functions	
`bool empty() const`	Check if the queue is empty. Returns True if queue is empty, False otherwise.
`bool full() const`	Check if the queue is full. Returns True if queue is full, False otherwise.
`bool try_get(T** dataout)`	Retrieve a message from the queue, storing message in location pointed to by dataout. Returns immediately, with True for success.
`bool try_put(T* data, uint8_t prio)`	Inserts the message pointed to by data into the end of the queue, with priority prio (higher number for higher priority). Returns immediately, with True for success.

Memory Pools

Queues are also used with the MemoryPool class, which allows you to define and manage fixed-size memory pools of objects of a given type.

An API summary is shown.

Name/Format	Description
Constructor	
`MemoryPool``<`*`T, pool_sz`*`>`	Create and Initialize a memory pool, of data type *T* and number of objects *pool_sz*
Member Functions	
`T* try_`**`alloc`**`()`	Allocate a memory block from a memory pool, without blocking. Returns address of allocated memory block. or **nullptr** if no memory available.
`osStatus` **`free`**`(T*` *`block`*`)`	Free a memory block. Returns **osOK** for success.

Program Example with Queues and Memory Pools

This program example demonstrates use of a queue and memory pool. A set of data, containing simulated voltage and current readings, and a counter, is transferred from one thread to another using a queue.

There are two user-defined threads running, **main( )** and **thread2.** This latter thread, which generates the data and puts it in the queue, runs every second.

The **main( )** thread extracts data from the queue and displays it. However, it is set to run only once every four seconds. Thus the queue is required to fill, before it is emptied.

Using **typedef**, a **struct** data block is defined as a new data type **message_t.** In **thread2**, the simulated data is calculated, based on a cycle counter i. The memory pool is populated with these calculated values, and the data placed in the queue. To illustrate application of priority, every third data item is prioritised.

In the **main( )** thread the values in the queue are read back and printed. Higher priority items are automatically read first, followed by the others (of equal priority), thereafter. Because we are printing floating point numbers, the usual addition of the little **mbed_app.json** file is needed.

Program Example with Queues and Memory Pools

```
/* Program Example 9.8. Demonstrates memory pools and queue, including
prioritisation.
   Developed from https://os.mbed.com/docs/mbed-os/v5.15/apis/memorypool.html
   Runs on LPC1768 or F401RE with no external connection needed */

#include "mbed.h"

// Define this struct as new type, named message.
typedef struct {
  float voltage;    // ADC result of measured voltage
  float current;    // ADC result of measured current
  uint32_t counter; // A counter value
} message_t;

// Declare a MemoryPool object of 16 elements, with type message_t.
MemoryPool<message_t, 16> mpool;

// Create queue, with same capacity as the memory pool.
Queue<message_t, 16> queue;

Thread thread2;           // Declare a thread object for the send_thread.
```

Program Example with Queues and Memory Pools

```
void send_thread(void) {
  uint32_t i = 0;
  while (true) {
    i++;                    // cycle counter
    // Allocate a message object
    message_t *message = mpool.try_alloc();
    if (message == nullptr) {
      printf("error: unable to allocate memory\n");
      continue;
    }
    // Use i to calculate simulated data
    message->voltage = (i * 3.3);
    message->current = (i * 1.1);
    message->counter = i;
    // Set priority to 5, whenever i is divisible by 3
    int prio = (i % 3) ? 0 : 5;
    if (!queue.try_put(message, prio)) {
      printf("error: unable to insert message into queue\n");
      mpool.free(message);
      continue;
    }
    ThisThread::sleep_for(1s);
  }
}
```

Program Example with Queues and Memory Pools

```
int main(void) {                          // viewed as Thread 1
  thread2.start(callback(send_thread)); // launch thread2
  printf("New set of readings:\n");     //Indicate new readings
  printf("--------------------\n");
  while (true) {
    message_t *message;
    if (queue.try_get(&message)) {
      // Print the data
      printf("Voltage: %.2f V\n", message->voltage);
      printf("Current: %.2f A\n", message->current);
      printf("Number of cycles: %u\n\n", message->counter);
      mpool.free(message);             // Free the message object
    } else {
      printf("Waiting for readings...\n");  //indicate queue is now empty
      ThisThread::sleep_for(4s);
      printf("\nNew set of readings:\n");   //Indicate new readings
      printf("--------------------\n");
    }
  }
```

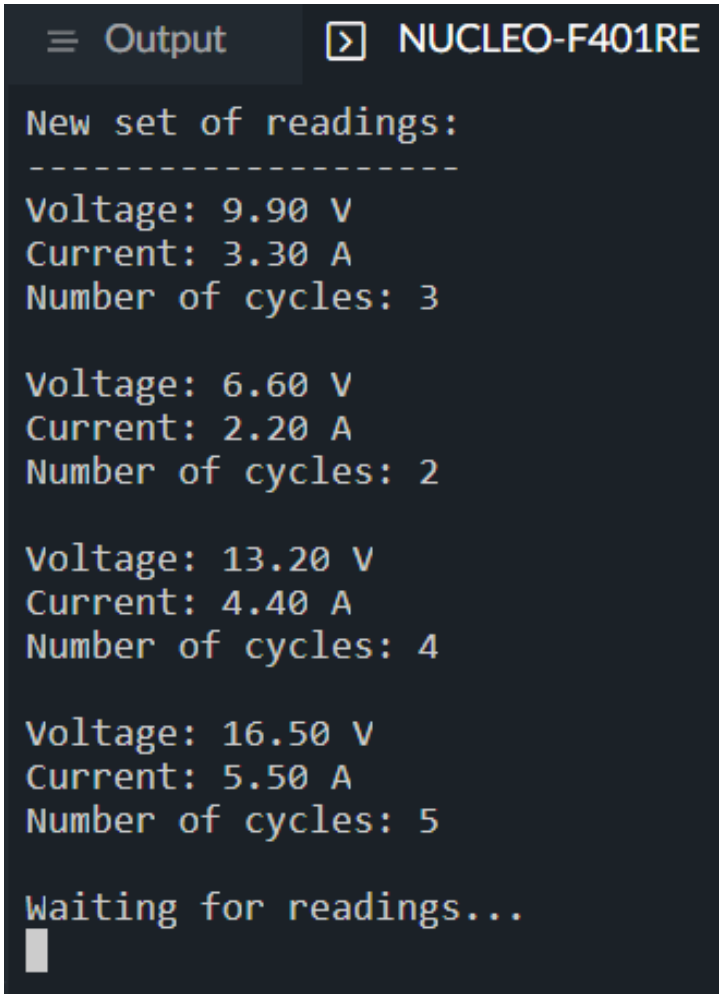
a) the main.cpp file

```
{
    "target_overrides":{
        "*": {
            "target.printf_lib": "std"
        }
    }
}
```

b) the mbed_app.json file

Programming with Queues and Memory Pools

A running program gives a display as shown. Every batch of readings is started with a new heading. In each batch, notice how any cycle which is divisible by 3 is prioritised, and appears first in the list.



The “Bare Metal” Profile

For simple programs, or in resource-constrained situations, it’s not worth using the RTOS; an alternative is the “Bare Metal” profile.

This is a subset of the Mbed OS, which results in efficient and compact code. Only the APIs that applications require are used.

Only a subset of the OS APIs are used. Examples are shown here. Drivers, like **DigitalIn** or **AnalogOut** are mostly available, but many of the RTOS APIs are not. Note that you can manually add missing APIs if your application requires.

A project must have its **mbed_app.json** file correctly configured for this profile. The simplest way to achieve this is to select the bare metal template in Mbed Studio or Studio Cloud. In this the **mbed_app.json** file has already been configured.

API	Available in Bare Metal
Thread	No
ThisThread	Yes
Semaphore	Yes
Queue	No
Mutex	Yes
Mail	No
MemoryPool	No
Kernel interface functions	limited
IdleLoop	No
EventFlags	Yes
ConditionVariable	No

Chapter Review

- A system operating in real time is one that is able to meet its deadlines.
- The requirement of multitasking, common to almost every embedded system, carries with it the concepts of threads, deadlines and priorities.
- Use of a Real Time Operating System requires that programs are structured in a different way, with the programmer clearly understanding the underlying principles of the operating system.
- The Real Time Operating System schedules threads, with cyclic, round robin, prioritized pre-emptive, or cooperative scheduling.
- Threads can interact with each other, applying carefully managed techniques.
- Access to shared resources, and synchronisation, can be provided by the mutex and the semaphore.
- Transfer of data between threads can be managed with queues and memory pools.
- Interrupts remain available in the RTOS environment, but must be used with particular care.