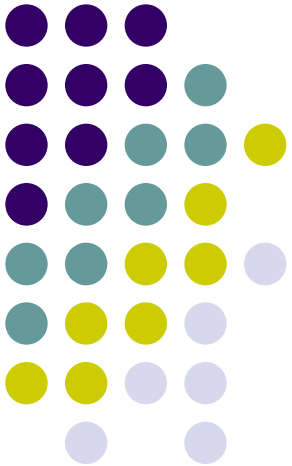
Chapter 7:Interrupts, Timers and Tasks

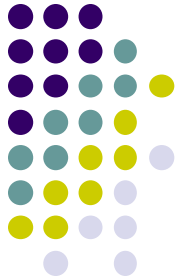
EE2405

嵌入式系統與實驗

Embedded System Lab



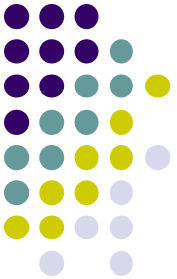
Tasks: Event-Triggered and Time-Triggered



- An embedded program has to undertake a number of different activities --- **task**.

- Task categories:
  - **Event**-triggered: occur when a particular external event happens, at a time which is not predictable (**asynchronously**)
  - **Time**-triggered happen periodically, at a time determined by the microcontroller.

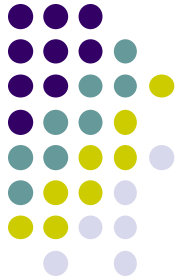
Examples of Events



- As a simple example, a room temperature controller may have the tasks shown below.

Task	Event or time-triggered
Measure temperature	Time (every minute)
Compute and implement heater and fan settings	Time (every minute)
Respond to user control	Event
Record and display temperature	Time (every minute)
Switch to battery backup in case of power loss	Event

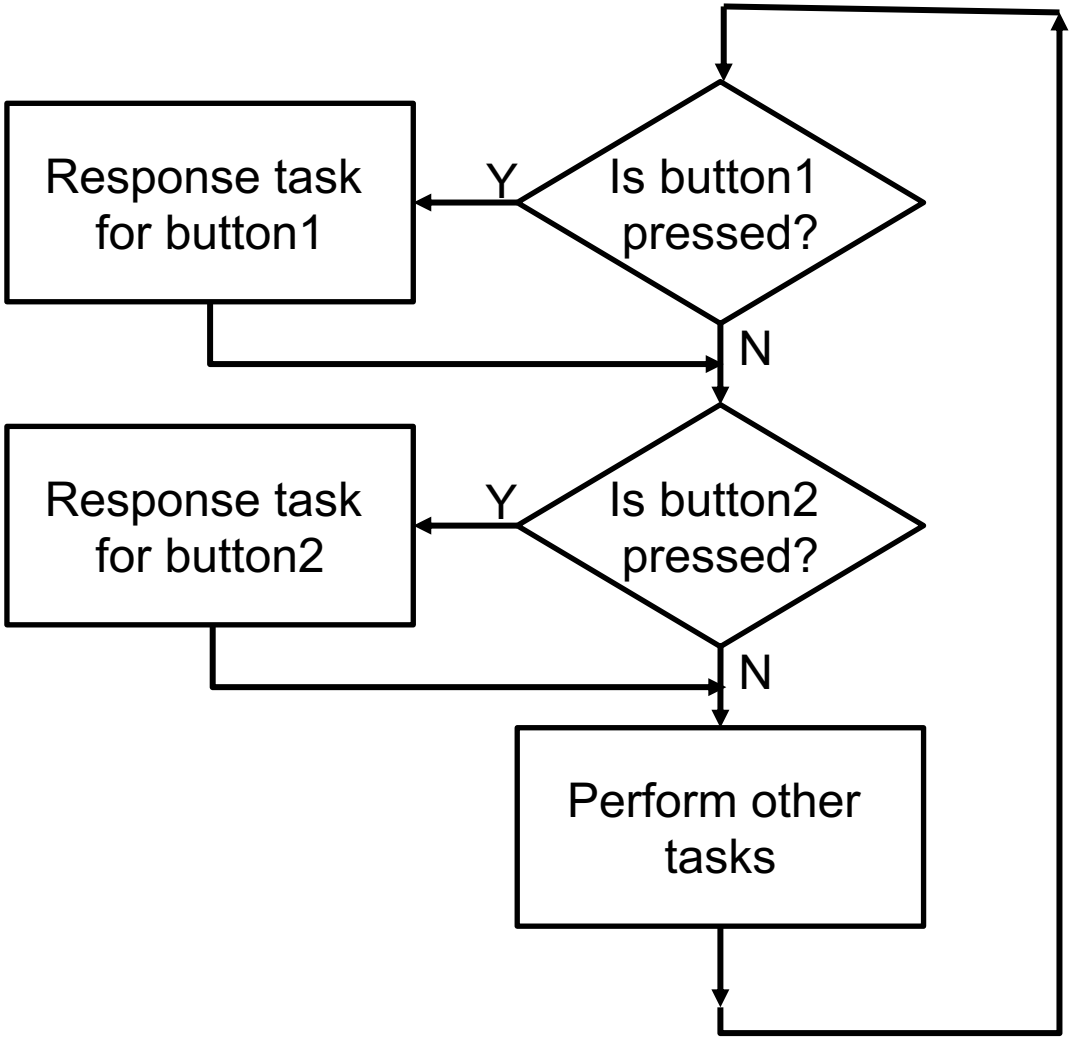
Polling



- One way of programming for an event-triggered activity, like a button push, is to continuously test that external input by polling.
  - As illustrated, a program is structured as a continuous loop

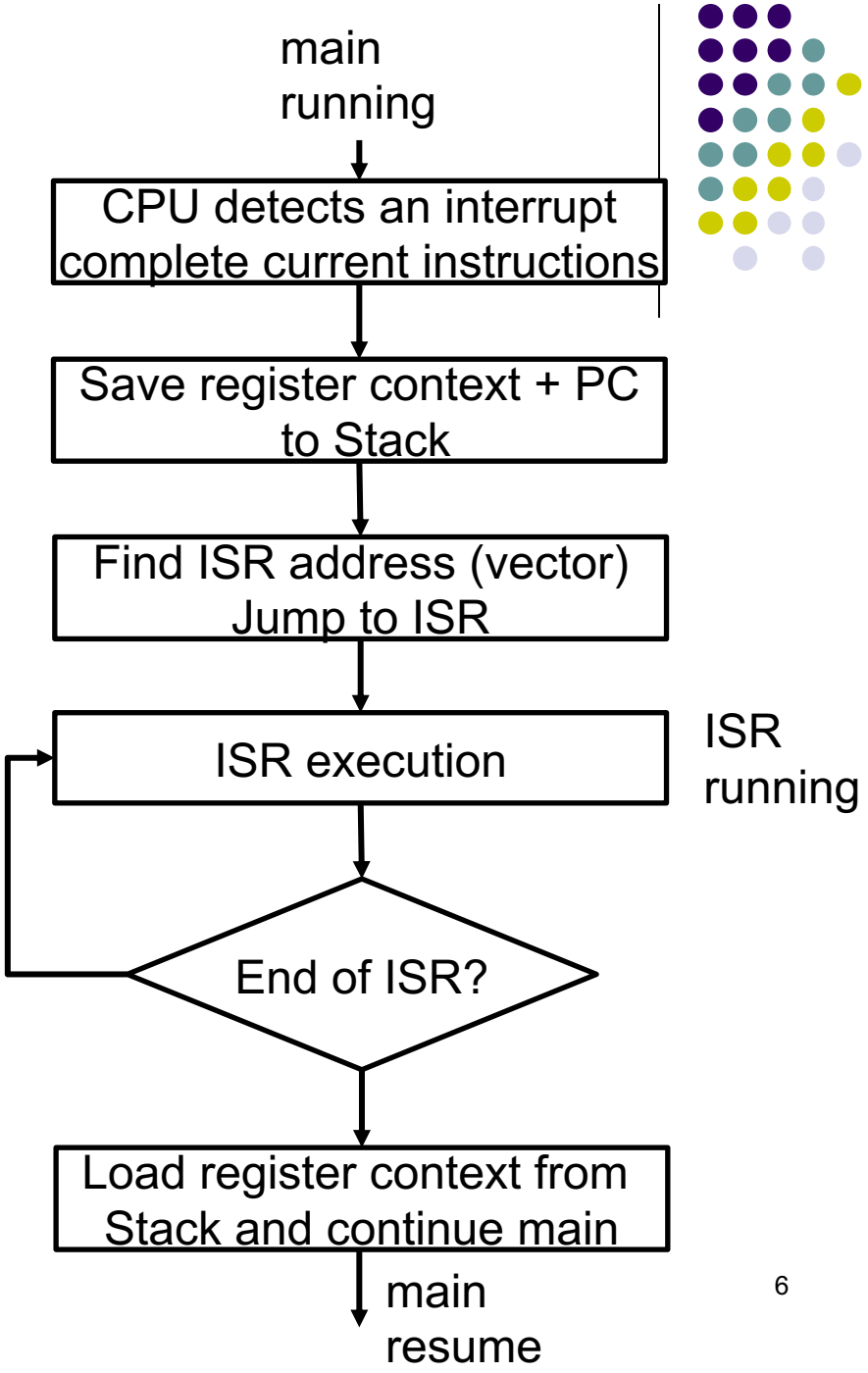
- Two main problems with polling:
  - The processor can't perform any other operations during a polling routine
  - All inputs are treated as equal
    - An urgent task has to wait its turn before it's processed by the computer.

Checking Two Buttons by Polling



Interrupt

- An interrupt is an hardware signal external to CPU.
- When an interrupt is triggered, CPU will stop the current job to run ISR (interrupt service routine)
- ISR is a predefined task located at a fixed memory address.



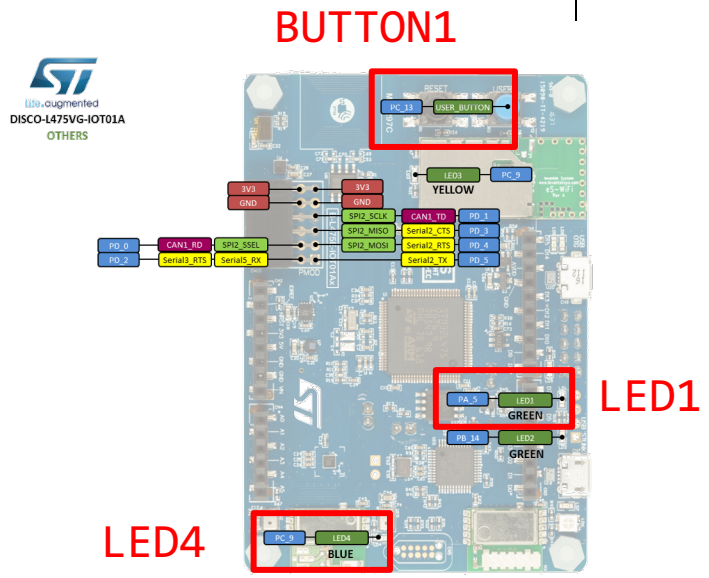
InterruptIn

- InterruptIn API

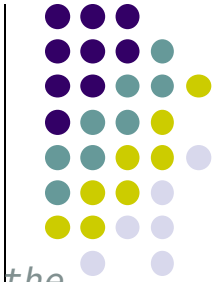
Function	Usage
InterruptIn	Create an InterruptIn connected to the specified pin
rise	Attach a function to call when a rising edge occurs on the input
fall	Attach a function to call when a falling edge occurs on the input
mode	Set the input pin mode: PullUp, PullDown, PullNone, PullDefault

Example of InterruptIn

```
#include "mbed.h"

InterruptIn button(BUTTON1);
DigitalOut led(LED1);
DigitalOut flash(LED4);

void flip()
{
    led = !led;
}

int main()
{
    button.rise(&flip); // attach the address of the flip function to the rising edge
    while (1)
    { // wait around, interrupts will interrupt this!
        flash = !flash;
        ThisThread::sleep_for(250ms);
    }
}
```

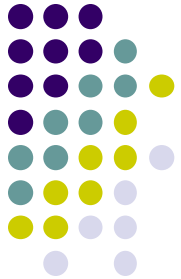


Example 2 of InterruptIn



```
#include "mbed.h"
class Counter {
public:
    Counter(PinName pin) : _interrupt(pin) { // create the InterruptIn on the
pin specified to Counter
        _interrupt.rise(callback(this, &Counter::increment)); // attach
increment function of this counter instance
    }

    void increment() {
        _count++;
    }

    int read() {
        return _count;
    }

private:
    InterruptIn _interrupt;
    volatile int _count;
};

Counter counter(BUTTON1);

int main() {
    while (1) {
        printf("Count so far: %d\n", counter.read());
        ThisThread::sleep_for(2s);
    }
}
```

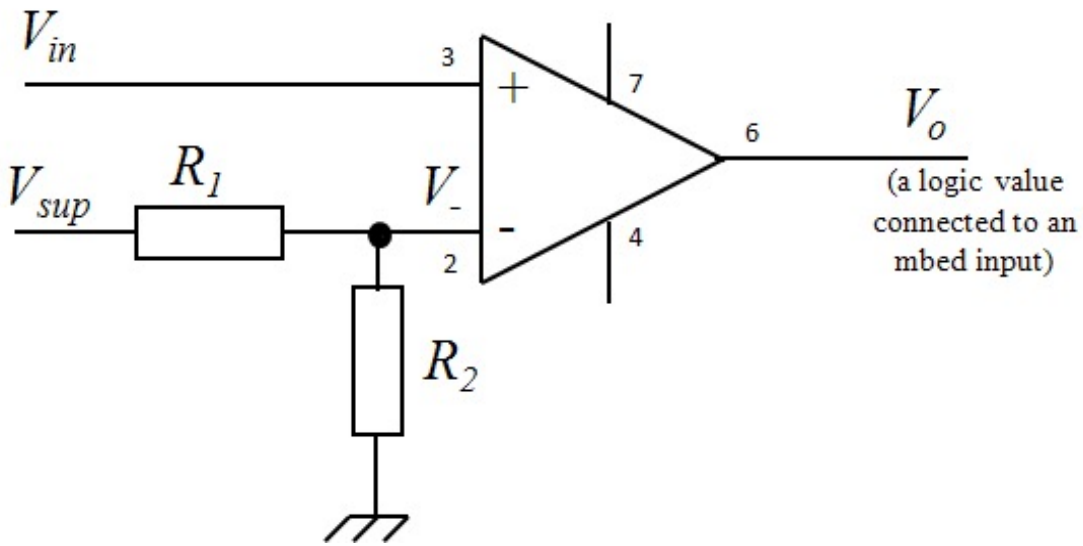
More on interrupt



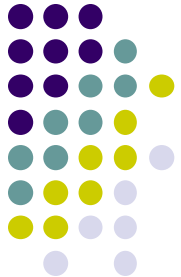
- **Prioritized** - some can be defined as more important than others. If two occur at the same time, then the higher priority one executes first.
- **Masked** - switched off, if they are not needed, or are likely to get in the way of more important activity. This masking could be just for a short period, for example while a critical program section completes.
- **Nested -** a higher priority interrupt can interrupt one of lower priority. The is strictly for advanced players only.
- The location of the ISR in memory can be selected, to suit the memory map and programmer wishes.
- Usually a specialized interrupt controller is implemented in hardware to assist above mechanisms.

Interrupts from analog inputs

- Aside from digital inputs, it is useful to generate interrupts when analog signals change
- For example, if an analog temperature sensor exceeds a certain threshold. Threshold detection can be done by a comparator, as shown.
- More effective than ADC with software detection.

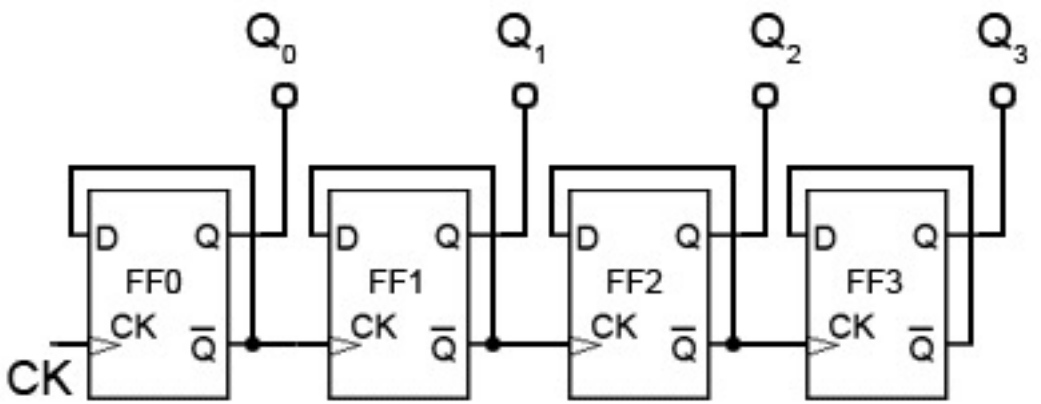


Digital Counter



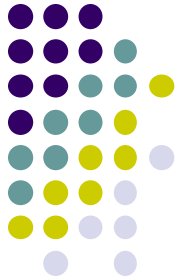
- Counters are useful in counting events. We can use a counter to set a threshold of an event count, e.g. 10 people passing a gate, and generate an interrupt.
- Counters can count up or down.
- We can also design a counter to read or load the current value (and reset to zero).
- An n-bit counter can count from 0 to $(2^n - 1)$. For example, an 8-bit counter can count from 0000 0000 to 1111 1111, or 0 to 255 in decimal.
  - If a counter reaches its maximum value, and the input clock pulses keep on coming, then it overflows back to zero, and starts counting up all over again.

Four-bit Asynchronous Up Counter



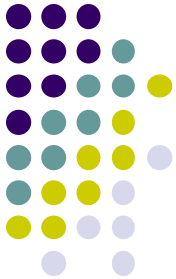
- More counter designs can be found at http://www.learnabout-electronics.org/Digital/dig56.php

Timer



- Assume we have a clock of 10MHz as input to a counter.
- Assume also we set the event count to 1024.
- We can calculate the time duration to count from 0 to 1024 = 1/10MHz * 1024 = 0.1us * 1024 =0.1024ms
- Therefore, a timer can be designed with a counter + a known clock source.

Timer

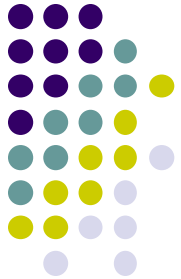


- mbed `Timer` uses timer hardware in micro-controller SoC to track times.

Function	Usage
Timer	Create a Timer object
start	Start the timer
stop	Stop the timer
reset	Reset the timer to 0
~~**read**~~	~~Get the time in seconds~~ (deprecated in mbed OS 6.0)
~~**read_ms**~~	~~Get the time in milli-seconds~~(deprecated in mbed OS 6.0)
~~**read_us**~~	~~Get the time in micro-seconds~~(deprecated in mbed OS 6.0)

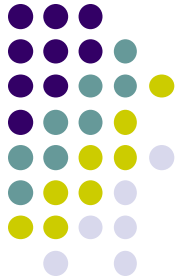
To read timer results, please use std::chrono API.
Though verbose, it is clear in units and only operates with integers.

Example of `Timer`



```
#include "mbed.h"

using namespace std::chrono;

Timer t;

int main()
{
    t.start();
    printf("Hello World!\n");
    t.stop();
    auto s =
chrono::duration_cast<chrono::seconds>(t.elapsed_time()).count();
    auto ms =
chrono::duration_cast<chrono::milliseconds>(t.elapsed_time()).count();
    auto us = t.elapsed_time().count();
    printf ("Timer time: %llu s\n", s);
    printf ("Timer time: %llu ms\n", ms);
    printf ("Timer time: %llu us\n", us);
}
```

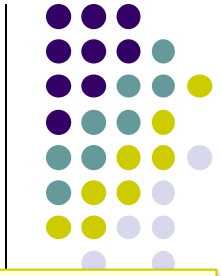
Timeout



- Timeout sets up an interrupt to call a function after a specified delay.

Function	Usage
Timeout	Create a Timeout object
attach	Attach a function or callback to be called by the Timeout, specifying the delay in chrono time (s, ms, us) in integer
attach_us	Attach a function to be called by the Timeout, specifying the delay in micro seconds
detach	Detach the function

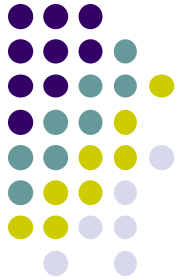
Example of Timeout



```
#include "mbed.h"
using namespace std::chrono;

Timeout flipper;
DigitalOut led1(LED1);
DigitalOut led2(LED4);

void flip()
{
    led2 = !led2;
}

int main()
{
    led2 = 1;
    flipper.attach(&flip, 2s); // setup flipper to call flip after 2
seconds

    // spin in a main loop. flipper will interrupt it to call flip
    while (1)
    {
        led1 = !led1;
        ThisThread::sleep_for(200ms);
    }
}
```

This example causes LED4 to turn off after 2 sec once. In the mean time, LED1 will blink for every 0.2 sec.

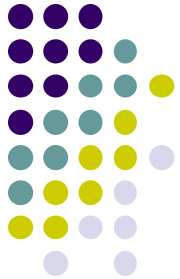
Ticker



- The mbed Ticker sets up a recurring interrupt, which can be used to call a function periodically, at a rate specified by the programmer.

Function	Usage
Ticker	Create a Ticker object
attach	Attach a function or callback to be called by the Ticker, specifying the **interval** in chrono time (s, ms, us) in integer
attach_us	Attach a function to be called by the Ticker, specifying the **interval** in microseconds
detach	Detach the function

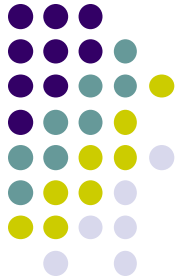
Example of Ticker



```
#include "mbed.h"
using namespace std::chrono;

Ticker flipper;
DigitalOut led1(LED1);
DigitalOut led2(LED4);

void flip()
{
    led2 = !led2;
}

int main()
{
    led2 = 1;
    flipper.attach(&flip, 2s); // the address of the function to be
attached (flip) and the interval (2 seconds)

    // spin in a main loop. flipper will interrupt it to call flip
    while (1)
    {
        led1 = !led1;
        ThisThread::sleep_for(200ms);
    }
}
```

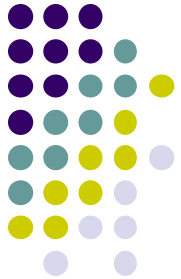
This example switches LED4 every 2 sec with a Ticker. In the mean time, LED1 will blink for every 0.2 sec.

Notes on Callback Function in Interrupt, Timeout and Ticker



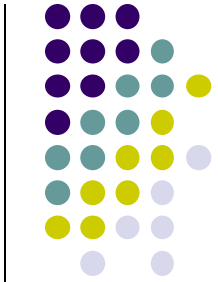
- No blocking code in ISR
  - Avoid any call to wait, infinite while loop or blocking calls in general.
- No printf, malloc or new in ISR
  - Avoid any call to bulky library functions. In particular, certain library functions (such as printf, malloc and new) are not re-entrant, and their behavior could be corrupted when called from an ISR.
- If you don't need microsecond precision, consider using the LowPowerTimeout class instead because this does not block deep sleep mode.

The real time clock



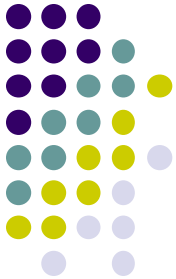
- The Real Time Clock (RTC) is an ultra-low-power peripheral on the micro-controller SoC.
- The RTC is a timing/counting system which is used to maintain a calendar and time-of-day clock, with registers for seconds, minutes, hours, day, month, year, day of month and day of year.
- It can also generate an alarm for a specific date and time.
- It runs from its own 32 kHz crystal oscillator, and can have its own independent battery power supply. It can thus be powered, and continue in operation, even if the rest of the microcontroller is powered down.
- The mbed API doesn’t create any C++ objects, but just implements standard functions from the standard C library.

RTC API



Function	Usage
time()	Get the current time
set_time	Set the current time
mktime	Converts a tm structure (a format for a time record) to a timestamp
localtime	Converts a timestamp to a tm structure
ctime	Converts a timestamp to a human-readable string
strftime	Converts a tm structure to a custom format human- readable string

Example of Real Time Clock



```
#include "mbed.h"

int main()
{
    set_time(1256729737); // Set RTC time to Wed, 28 Oct 2009 11:35:37

    while (true)
    {
        time_t seconds = time(NULL);

        printf("Time as seconds since January 1, 1970 = %u\n", (unsigned
int)seconds);

        printf("Time as a basic string = %s", ctime(&seconds));

        char buffer[32];
        strftime(buffer, 32, "%I:%M %p\n", localtime(&seconds));
        printf("Time as a custom formatted string = %s", buffer);

        ThisThread::sleep_for(1s);
    }
}
```

An application --- debouncing

- The mechanical contacts of a switch literally bounce together, as the switch closes. This lasts for a few milliseconds, and can cause a digital input to swing wildly between Logic 0 and Logic 1 for a short time after a switch closes, as illustrated:

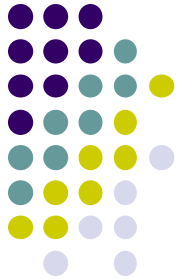
Demonstrating switch bounce

LED Switching with Debounce

This program solves the switch bounce issue by starting a timer on a switch button event, and ensuring that 1s has elapsed before allowing a second event to be processed.

```
/*Event driven LED switching with switch debounce */
#include "mbed.h"
using namespace std::chrono;

Timer debounce;                    //define debounce timer
InterruptIn button(USER_BUTTON); //Interrupt on pushbutton
DigitalOut led1(LED1);

void toggle()
{
    if (duration_cast<milliseconds>(debounce.elapsed_time()).count() > 1000)
    {
        //only allow toggle if debounce timer has passed 1s
        led1 = !led1;
        debounce.reset(); //restart timer when the toggle is performed
    }
}
int main()
{
    debounce.start();
    button.rise(&toggle); // attach the address of the toggle
    while (1)
        ;
}
```

Tasks in Real Time OS

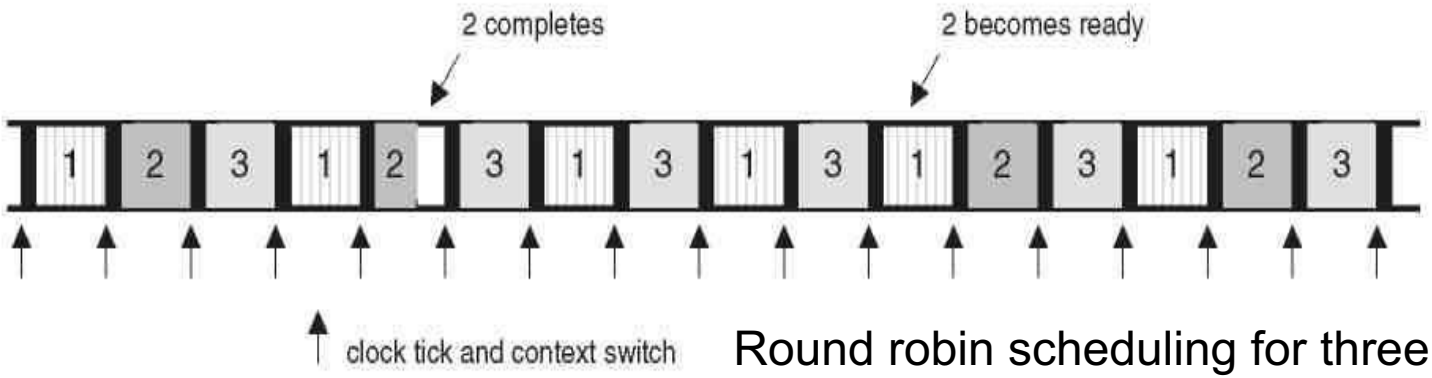


- A program written for an RTOS is structured into tasks or threads.
  - Each task is written as a self-contained program module.
  - The tasks can be prioritized.

- The RTOS performs three main functions for tasks:
  - It decides which task/thread should run and for how long.
  - It provides communication and synchronization between tasks.
  - It controls the use of resources shared between the tasks, for example, memory and hardware peripherals.

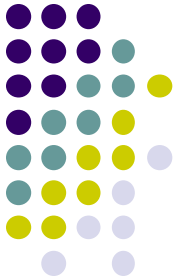
RTOS Scheduling

- An important part of the RTOS is its scheduler, which decides which task runs and for how long.
- A simple example is the Round Robin scheduler, as illustrated.
  - The scheduler synchronizes its activity to a clock tick.
- At every clock tick, the scheduler determines if a different task should be given CPU time.
  - In Round Robin scheduling, the task is switched in order.



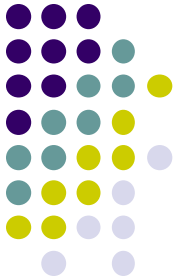
Round robin scheduling for three tasks

mbed Thread



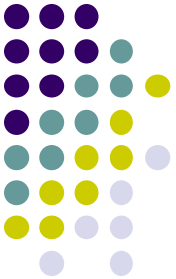
- Thread a single sequential flow of control within a program.
  - Multiple threads can simultaneously perform different tasks in a single program.
  - To avoid long super-loops.
- Thread can be set to different priorities.

Example of Thread 1



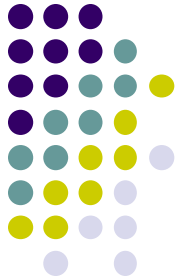
```
#include "mbed.h"

DigitalOut led1(LED1);
DigitalOut led2(LED);
Thread thread;

void led2_thread()
{
    while (true)
    {
        led2 = !led2;
        ThisThread::sleep_for(1000ms);
    }
}

int main()
{
    thread.start(led2_thread);

    while (true)
    {
        led1 = !led1;
        ThisThread::sleep_for(500ms);
    }
}
```

Example of Thread 2 (with Arguments)



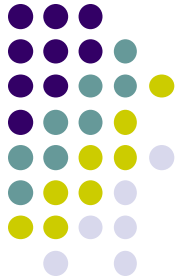
```
#include "mbed.h"

Thread thread;
DigitalOut led1(LED1);
volatile bool running = true;

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

// Spawns a thread to run blink for 5 seconds
int main()
{
    thread.start(callback(blink, &led1));
    ThisThread::sleep_for(5s);
    running = false;
    thread.join();
}
```

mbed EventQueue



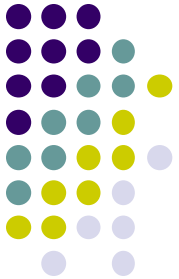
- A flexible queue for scheduling events
  - Thread and ISR safe.
- Note that some I/O classes are not allowed in ISR, e.g., printf(), DAC/ADC, etc. (Not ISR safe)
- When we get an interrupt,
  - (1) Run a simple ISR to handle the interrupt
  - (2) Schedule an task in an EventQueue
  - (3) Return the ISR.
- For different priorities, setup queues in threads with different priority.

Example of EventQueue Call



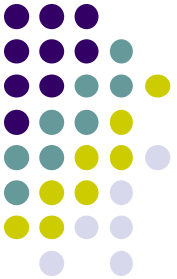
```
#include "mbed.h"
using namespace std::chrono;

int main()
{
    // creates a queue with the default size
    EventQueue queue;

    // printf will be put into queue and execute immediately
    queue.call(printf, "called immediately\r\n");
    // Replace Timeout
    queue.call_in(2s, printf, "called in 2 seconds\r\n");
    // Replace Ticker
    queue.call_every(1s, printf, "called every 1 seconds\r\n");

    // events are executed by the dispatch method
    queue.dispatch();
}
```

EventQueue with a Thread 1



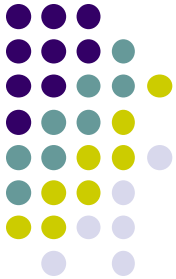
```
#include "mbed.h"

DigitalOut led1(LED1);
InterruptIn sw2(USER_BUTTON);
EventQueue queue(32 * EVENTS_EVENT_SIZE);
Thread t;

void led1_info() {
    // Note that printf is deferred with a call in the queue
    // It is not executed in the interrupt context
    printf("led1 is triggered! \r\n");
}

void Trig_led1() {
    // Execute the time critical part first
    led1 = !led1;
    // The rest can execute later in user context
    queue.call(led1_info);
}

int main() {
    // t is a thread to process tasks in an EventQueue
    // call in queue is now executed in the context of the thread
    // t call queue.dispatch_forever() to start the scheduler of the EventQueue
    t.start(callback(&queue, &EventQueue::dispatch_forever));

    // 'Trig_led1' will execute in IRQ context
    sw2.rise(Trig_led1);
}
```

EventQueue with a Deferred Call 2



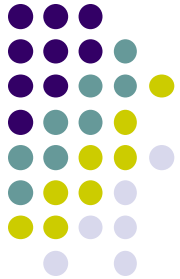
```
#include "mbed.h"

DigitalOut led2(LED4);
InterruptIn sw3(USER_BUTTON);
EventQueue queue(32 * EVENTS_EVENT_SIZE);

Thread t;

void Trig_led2()  {
    led2 = !led2;
    // Safe to use 'printf' in context of thread 't', while IRQ is not.
    printf("led2 is triggered! \r\n");
}


int main() {

    t.start(callback(&queue, &EventQueue::dispatch_forever));

    // 'Trig_led2' is put directly into the queue (same as queue.call)
    // 'Trig_led2' will execute in context of thread 't'
    // The interrupt event simply processes the queue.call(), instead of 'Trig_led2'
    sw3.rise(queue.event(Trig_led2));
}
```

EventQueue with Priority 1/2



```
#include "mbed.h"
#include "mbed_events.h”
using namespace std::chrono;

DigitalOut led1(LED1);
DigitalOut led2(LED4);
InterruptIn btn(USER_BUTTON);

EventQueue printfQueue;
EventQueue eventQueue;

void blink_led2() {
  // this runs in the normal priority thread
  led2 = !led2;
}

void print_toggle_led() {
  // this runs in the lower priority thread
  printf("Toggle LED!\r\n");
}

void btn_fall_irq() {
  led1 = !led1;
  // defer the printf call to the low priority thread
  printfQueue.call(&print_toggle_led);
}
```

EventQueue with Priority 2/2



```
int main() {

  // low priority thread for calling printf()
  Thread printfThread(osPriorityLow);
  printfThread.start(callback(&printfQueue,
&EventQueue::dispatch_forever));

  // normal priority thread for other events
  Thread eventThread(osPriorityNormal);
  eventThread.start(callback(&eventQueue, &EventQueue::dispatch_forever));

  // call blink_led2 every second, automatically defering to the
eventThread
  Ticker ledTicker;
  ledTicker.attach(eventQueue.event(&blink_led2), 1s);

  // button fall still runs in the ISR
  btn.fall(&btn_fall_irq);

  while (1) {ThisThread::sleep_for(1s);}
}
```