

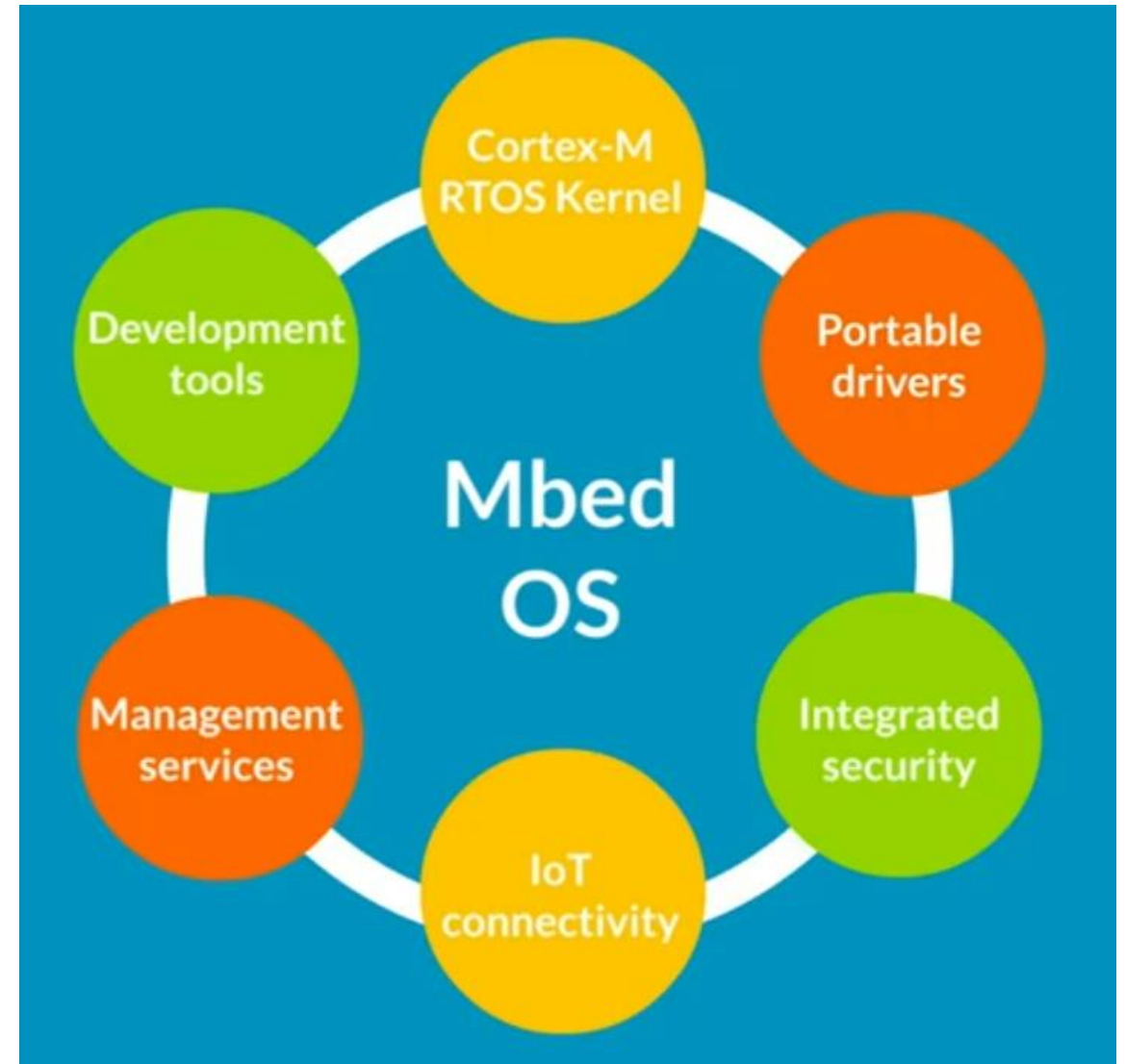
Mbed OS

The Things Network Madrid

Juan Félix Mateos
Octubre 2021
juanfelixmateos@gmail.com

¿Qué es un RTOS?

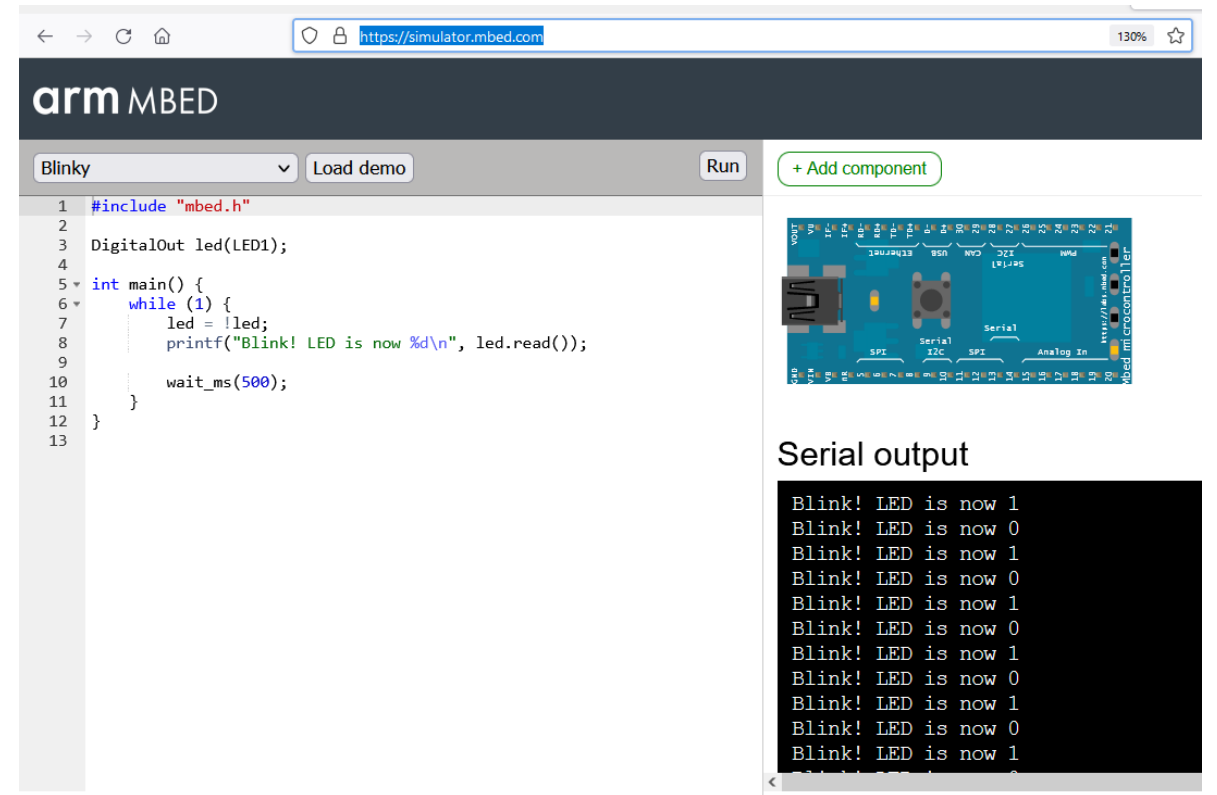
- Sistema operativo de tiempo real
- Capa de abstracción entre el hardware y el programador
- **Determinista:** Cada operación tiene un tiempo fijo asignado para ejecutarse (o fallar).
- Conceptos:
 - Semáforos
 - Locks/Mutexes
 - Multi-Treading



Mbed simulator

<https://simulator.mbed.com/>
Experimental
Mbed OS 5
No admite threads

Si falla, insistir pulsando
nuevamente el botón download
que hay a la derecha de Add
component



GPIO

Table 19. STM32WLE5/E4xx pin definition (continued)

Pin number			Pin name (function after reset)	Pin type	I/O structure	Notes	Alternate functions	Additional functions
UFQFPN48	WLCSP59	UFBGA73						
11	K11	H5	VDD	S	-	-	-	-
12	J10	J1	PA4	I/O	FT	-	RTC_OUT2, LPTIM1_OUT, SPI1_NSS, USART2_CK, DEBUG_SUBGHZSPI_ NSSOUT, LPTIM2_OUT, CM4_EVENTOUT	-
13	H9	J2	PA5	I/O	FT	-	TIM2_CH1, TIM2_ETR, SPI2_MISO, SPI1_SCK, DEBUG_SUBGHZSPI_ SCKOUT, LPTIM2_ETR, CM4_EVENTOUT	-
14	G8	F4	PA6	I/O	FT	-	TIM1_BKIN, I2C2_SMBA, SPI1_MISO, LPUART1_CTS, DEBUG_SUBGHZSPI_ MISOOUT, TIM16_CH1, CM4_EVENTOUT	-
15	E8	H3	PA7	I/O	FT_fa	-	TIM1_CH1N, I2C3_SCL, SPI1_MOSI, COMP2_OUT, DEBUG_SUBGHZSPI_ MOSIOUT, TIM17_CH1, CM4_EVENTOUT	-

GPIO

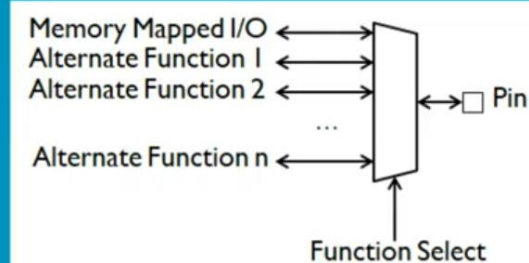
General Purpose **Input** **Output**

Configurable for a range of signals

Advantages

Saves space

Improves flexibility



API: Clases y métodos

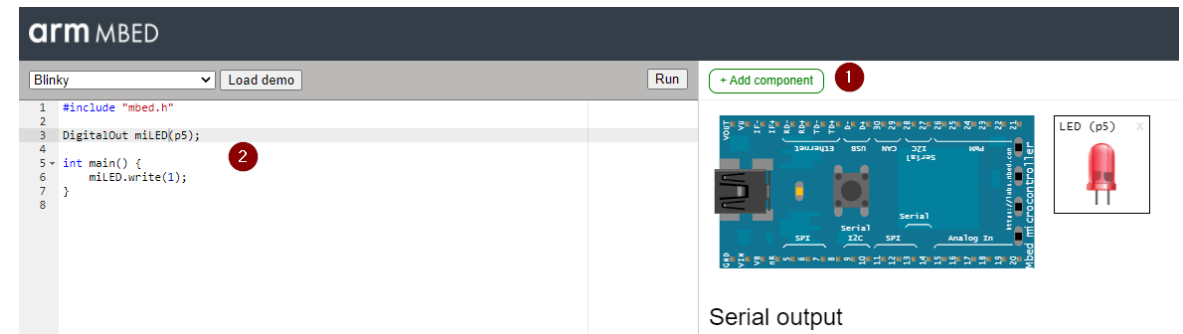
<https://os.mbed.com/docs/mbed-os/v6.15/apis/digitalout.html>

DigitalOut class reference

DigitalOut Class Reference

Public Member Functions

	DigitalOut (PinName pin)
	Create a DigitalOut connected to the specified pin. More...
	DigitalOut (PinName pin, int value)
	Create a DigitalOut connected to the specified pin. More...
void	write (int value)
	Set the output, specified as 0 or 1 (int) More...



The screenshot shows the mbed IDE interface. On the left, the code editor displays the following code:

```
1 #include "mbed.h"
2
3 DigitalOut mLED(p5);
4
5 int main() {
6     mLED.write(1);
7 }
8
```

Red circles with numbers 1 and 2 highlight the '+ Add component' button and the line 'DigitalOut mLED(p5);' respectively.

On the right, the hardware schematic shows a blue mbed microcontroller board with an LED connected to pin p5. Below the schematic, the text 'Serial output' is visible.

Nomenclatura de los pines

- El emulador está basado en el NXP LPC1768
- https://github.com/ARMmbed/mbed-os/blob/master/targets/TARGET_NXP/TARGET_LPC176X/TARGET_MBED_LPC1768/PinNames.h
- Podríamos cambiar en el código anterior p5 por P0_9 y funcionaría igual

```
30     PIN_INPUT,  
31     PIN_OUTPUT  
32 } PinDirection;  
33  
34 #define PORT_SHIFT 5  
35  
36 typedef enum {  
37     // LPC Pin Names  
38     P0_0 = LPC_GPIO0_BASE,  
39     P0_1, P0_2, P0_3, P0_4, P0_5, P0_6, P0_7, P0_8, P0_9, P0_10, P0_11, P0_12, P0_13, P0_14, P0_15, P0_16, P0_17, P0_18, P0_19, P0_20,  
40     P1_0, P1_1, P1_2, P1_3, P1_4, P1_5, P1_6, P1_7, P1_8, P1_9, P1_10, P1_11, P1_12, P1_13, P1_14, P1_15, P1_16, P1_17, P1_18, P1_19, P1_20,  
41     P2_0, P2_1, P2_2, P2_3, P2_4, P2_5, P2_6, P2_7, P2_8, P2_9, P2_10, P2_11, P2_12, P2_13, P2_14, P2_15, P2_16, P2_17, P2_18, P2_19, P2_20,  
42     P3_0, P3_1, P3_2, P3_3, P3_4, P3_5, P3_6, P3_7, P3_8, P3_9, P3_10, P3_11, P3_12, P3_13, P3_14, P3_15, P3_16, P3_17, P3_18, P3_19, P3_20,  
43     P4_0, P4_1, P4_2, P4_3, P4_4, P4_5, P4_6, P4_7, P4_8, P4_9, P4_10, P4_11, P4_12, P4_13, P4_14, P4_15, P4_16, P4_17, P4_18, P4_19, P4_20,  
44  
45     // MBED DXP Pin Names  
46     p5 = P0_9,  
47     p6 = P0_8,  
48     p7 = P0_7,  
49     p8 = P0_6,  
50     p9 = P0_5,  
51     p10 = P0_4,  
52     p11 = P0_3,  
53     p12 = P0_2
```

DigitalIn

DigitalIn (PinName pin)

Create a **DigitalIn** connected to the specified pin. [More...](#)

DigitalIn (PinName pin, PinMode mode)

Create a **DigitalIn** connected to the specified pin. [More...](#)

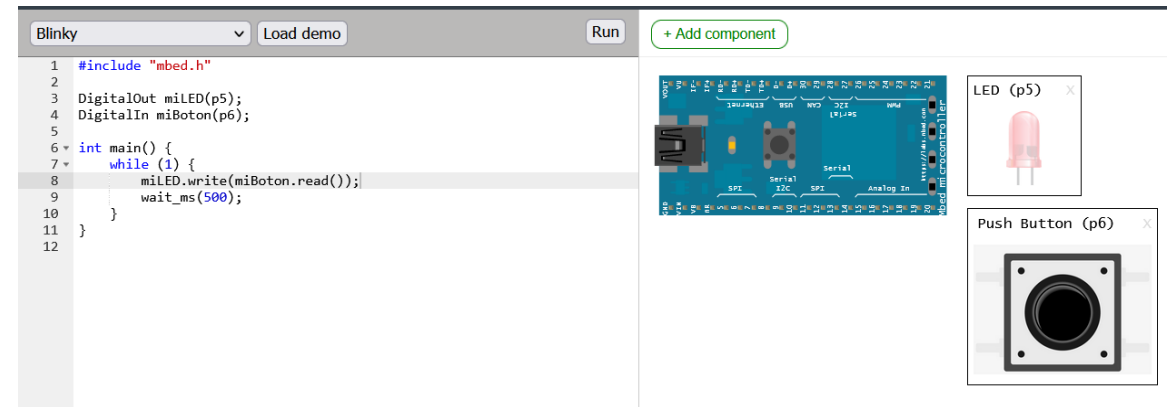
~DigitalIn ()

Class destructor, deinitialize the pin. [More...](#)

int **read** ()

Read the input, represented as 0 or 1 (int) [More...](#)

void **mode** (PinMode mode)



Los modos pull son:

- PullUp, PullDown, PullNone, OpenDrain

En el emulador hay que poner siempre wait en los bucles infinitos

PWMOut

```
1  #include "mbed.h"
2
3  PwmOut miLED(p5);
4
5  int main() {
6      while(1) {
7          for(float i=0;i<1;i=i+0.1){
8              miLED.write(i);
9              wait(0.5);
10         }
11         for(float i=1;i>0;i=i-0.1){
12             miLED.write(i);
13             wait(0.5);
14         }
15     }
16 }
```

PwmOut Class Reference

Public Member Functions

	PwmOut (PinName pin)
	Create a PwmOut connected to the specified pin. More...
	PwmOut (const PinMap &pinmap)
	Create a PwmOut connected to the specified pin. More...
void	write (float value)
	Set the output duty-cycle, specified as a percentage (float) More...

AnalogIn

FL 14 AnalogIn Class Reference

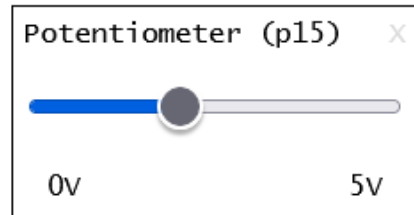
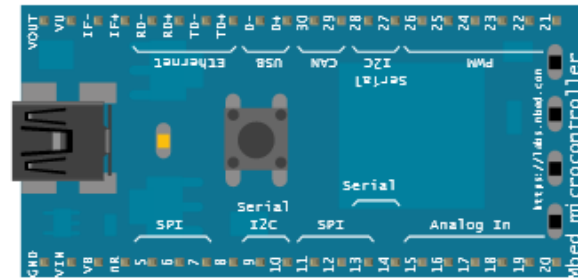
Public Member Functions	
	AnalogIn (const PinMap &pinmap, float vref=MBED_CONF_TARGET_DEFAULT_ADC_VREF)
	Create an AnalogIn , connected to the specified pin. More...
	AnalogIn (PinName pin, float vref=MBED_CONF_TARGET_DEFAULT_ADC_VREF)
	Create an AnalogIn , connected to the specified pin. More...
float	read ()
	Read the input voltage, represented as a float in the range [0.0, 1.0]. More...

Symbol	Pin/ball					
	LQFP100	TFBGA100	WLCSP100			
P0[23]/AD0[0]/I2SRX_CLK/CAP3[0]	9	E5	D5	[2]	I/O	P0[23] — General purpose digital input/output pin.
					I	AD0[0] — A/D converter 0, input 0.
					I/O	I2SRX_CLK — Receive Clock. It is driven by the master and received by the slave. Corresponds to the signal SCK in the <i>I²S-bus</i> specification. (LPC1769/68/67/66/65/63 only).
					I	CAP3[0] — Capture input for Timer 3, channel 0.
P0[23]/AD0[0]	9	E5	D5	[2]	I/O	P0[23] — General purpose digital input/output pin.

No todos los pines tienen funcionalidad ADC. El p15 del LPC1768 es el P0_23, que es la entrada 0 del ACD 0.

AnalogIn y PwmOut

```
1 #include "mbed.h"
2
3 PwmOut miLED(p5);
4 AnalogIn miPot(p15);
5
6 int main() {
7     while (1) {
8         miLED.write(miPot.read());
9         printf("Intensidad: %.2f\n", miLED.read());
10        wait_ms(500);
11    }
12 }
13
```



Serial output

```
Intensidad: 0.39
Intensidad: 0.39
Intensidad: 0.39
```

Interrupciones externas

InterruptIn Class Reference

Public Member Functions

	InterruptIn (PinName pin)
	Create an InterruptIn connected to the specified pin. More...
	InterruptIn (PinName pin, PinMode mode)
	Create an InterruptIn connected to the specified pin, and the pin configured to the specified mode. More...
int	read ()
	Read the input, represented as 0 or 1 (int) More...
	operator int ()
	An operator shorthand for read() More...
void	rise (Callback< void()> func)
	Attach a function to call when a rising edge occurs on the input. More...
void	fall (Callback< void()> func)
	Attach a function to call when a falling edge occurs on the input. More...

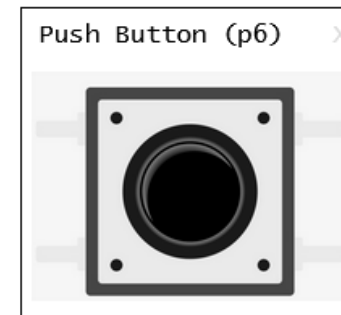
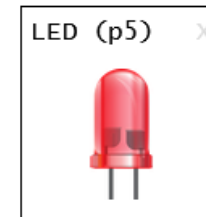
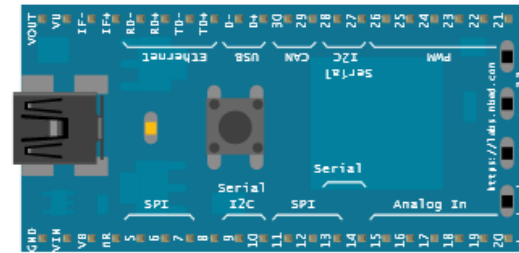
8.7.2 Interrupt sources

Each peripheral device has one interrupt line connected to the NVIC but may have several interrupt flags. Individual interrupt flags may also represent more than one interrupt source.

Any pin on Port 0 and Port 2 (total of 42 pins) regardless of the selected function, can be programmed to generate an interrupt on a rising edge, a falling edge, or both.

Interrupciones externas

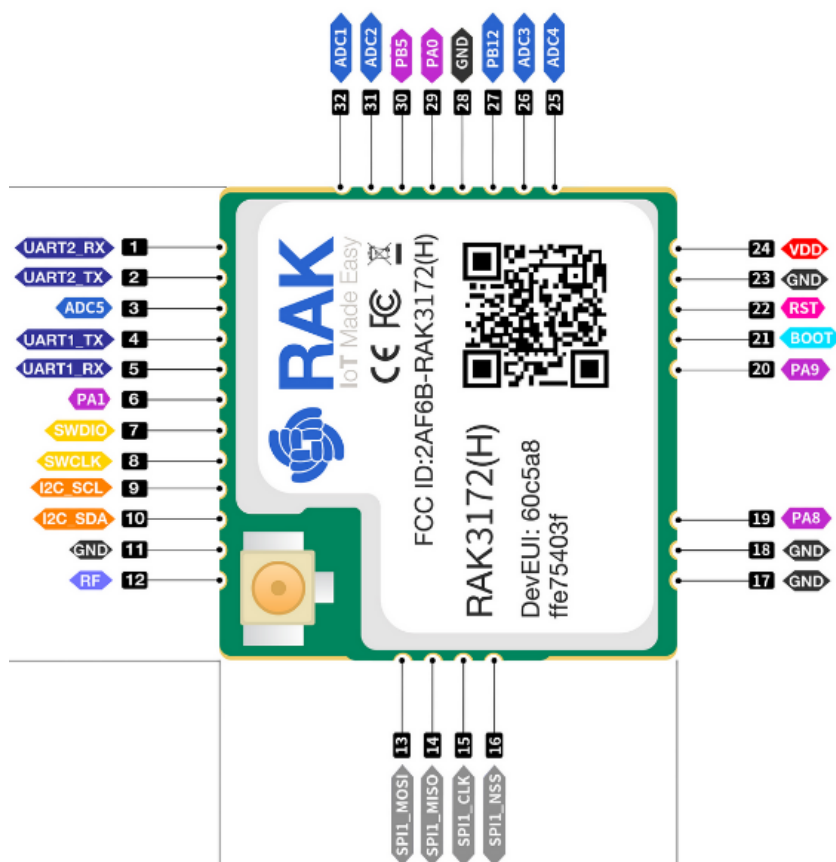
```
1  #include "mbed.h"
2
3  DigitalOut miLED(p5);
4  InterruptIn miBoton(p6);
5
6
7  void alternarLED() {
8      printf("LED alternado\n");
9      miLED.write(!miLED.read());
10 }
11
12
13
14 int main() {
15     miBoton.fall(&alternarLED);
16
17     while(1){
18         wait(1);
19     }
20 }
21
```



Serial output

LED alternado

RAK3172



Features

- Based on STM32WLE5CCU6
- LoRaWAN 1.0.3 specification compliant
- Supported bands: EU433, CN470, IN865, EU868, AU915, US915, KR920, RU864, and AS923-1/2/3/4
- LoRaWAN Activation by OTAA/ABP
- LoRa Point to Point (P2P) communication
- Easy to use AT Command Set via UART interface
- Long-range - greater than 15 km with optimized antenna
- Arm Cortex-M4 32-bit
- 256 kbytes flash memory with ECC
- 64 kbytes RAM
- Ultra-Low Power Consumption of 1.69 μ A in sleep mode
- Supply Voltage: 2.0 V ~ 3.6 V
- Temperature Range: -40° C ~ 85° C

RAK3272

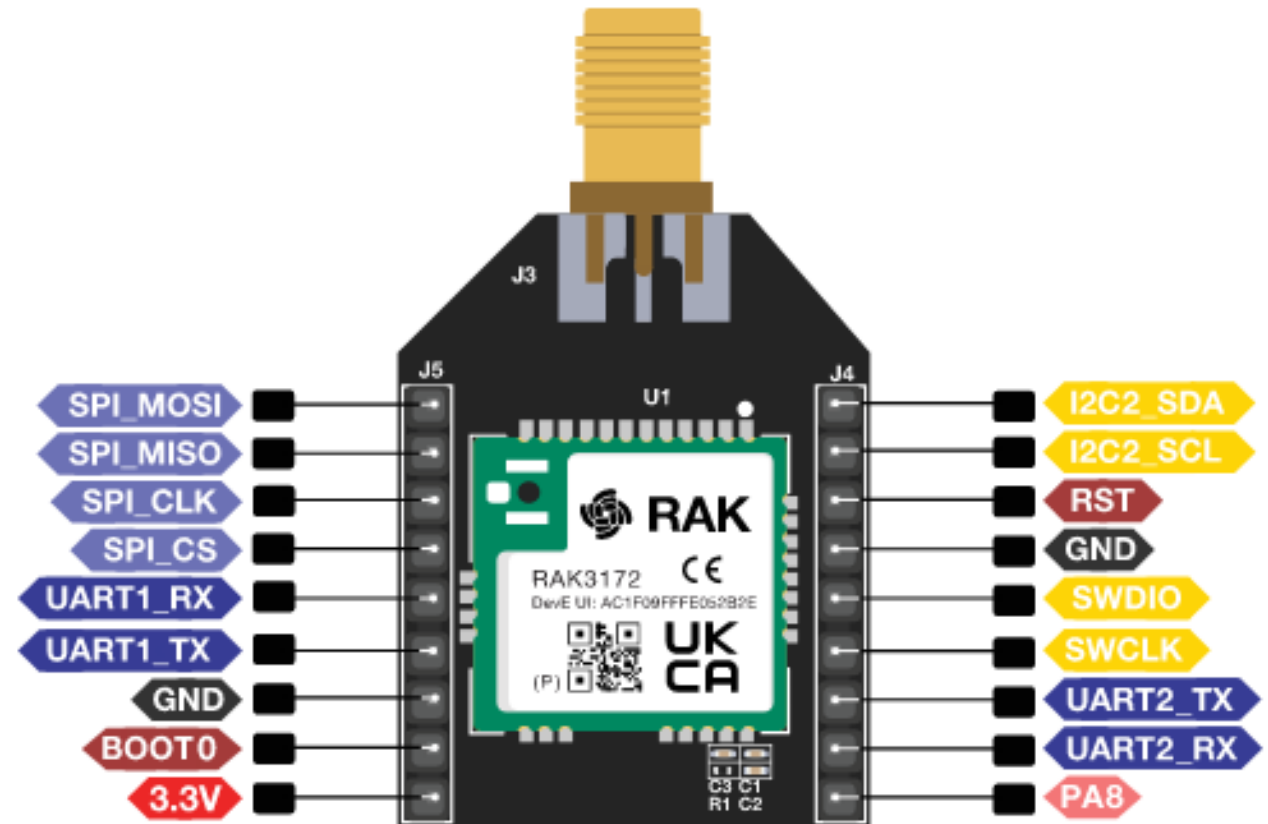
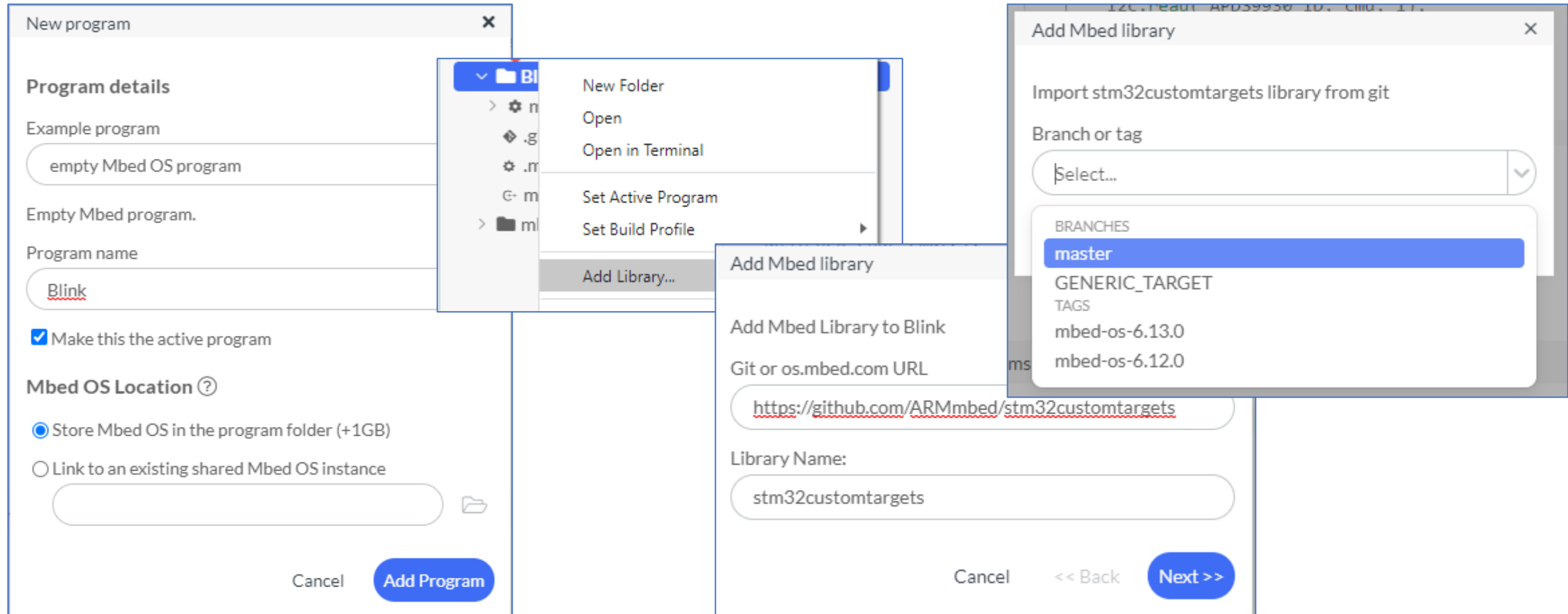


Figure 2: RAK3272S Breakout Board Pinout

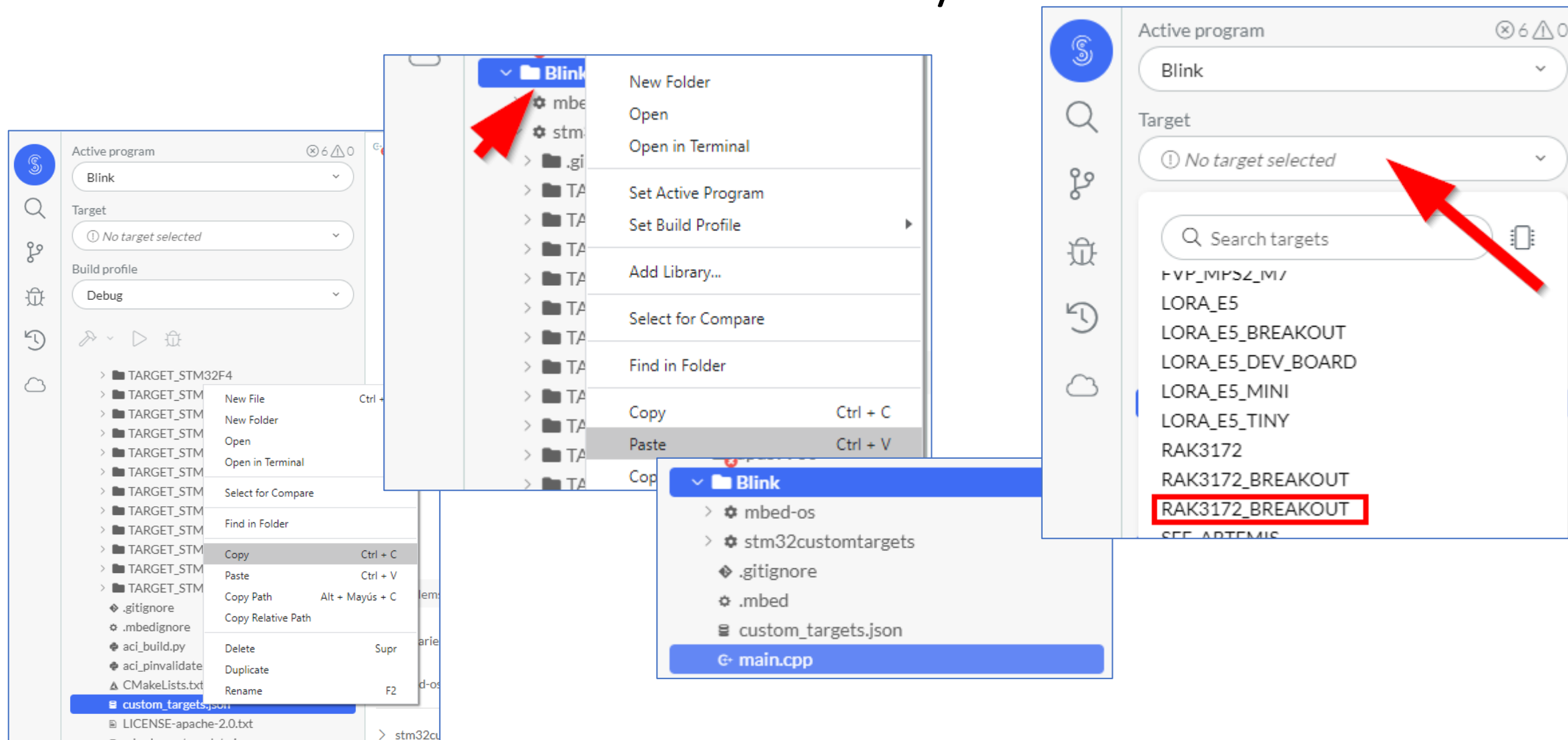
RAK3172 en Mbed Studio 1/3

- Mbed Studio no incluye aún el módulo RAK3172
- Afortunadamente Charles Hallard ha incluido una en la librería de "Custom Targets" de Mbed
 - <https://github.com/ARMmbed/stm32customtargets>
 - Procedimiento:
 1. Crear un programa nuevo
 2. Importar la librería stm32customtargets
 3. Copiar el archivo custom_targets.json de la librería anterior a la carpeta raíz del proyecto
 4. Seleccionar el nuevo target RAK3172_BREAKBOARD

RAK3172 en Mbed Studio 2/3

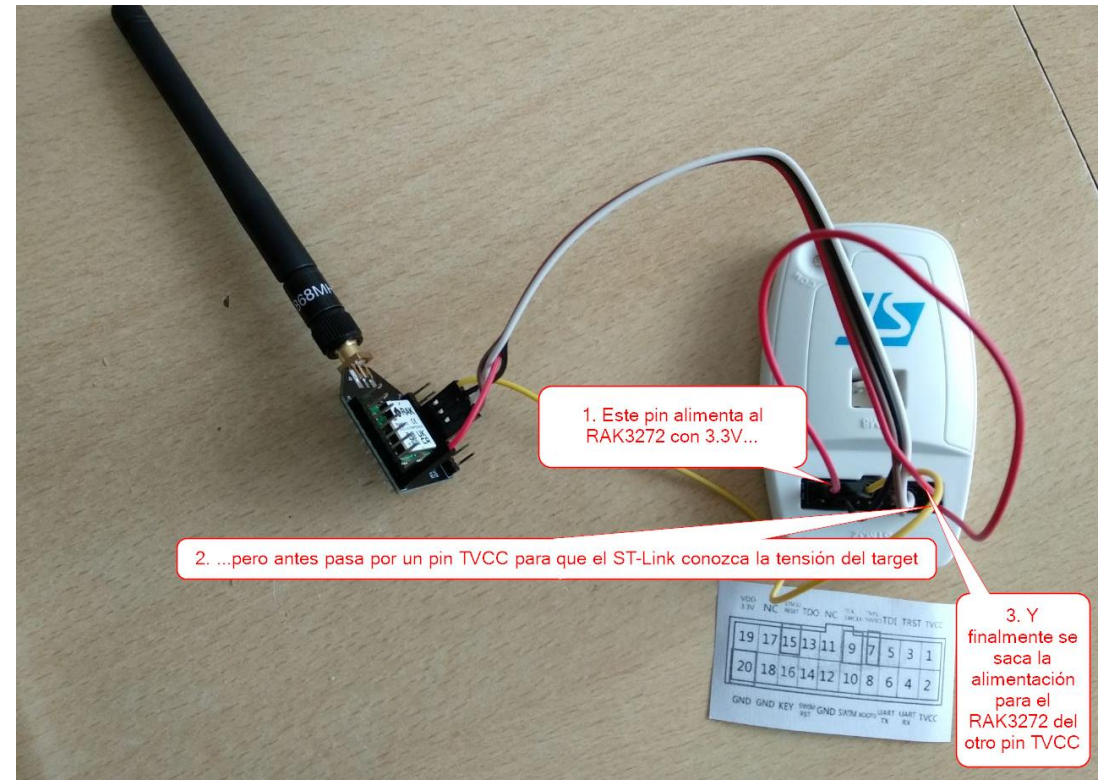


RAK3172 en Mbed Studio 3/3



Conexión del ST-Link v2

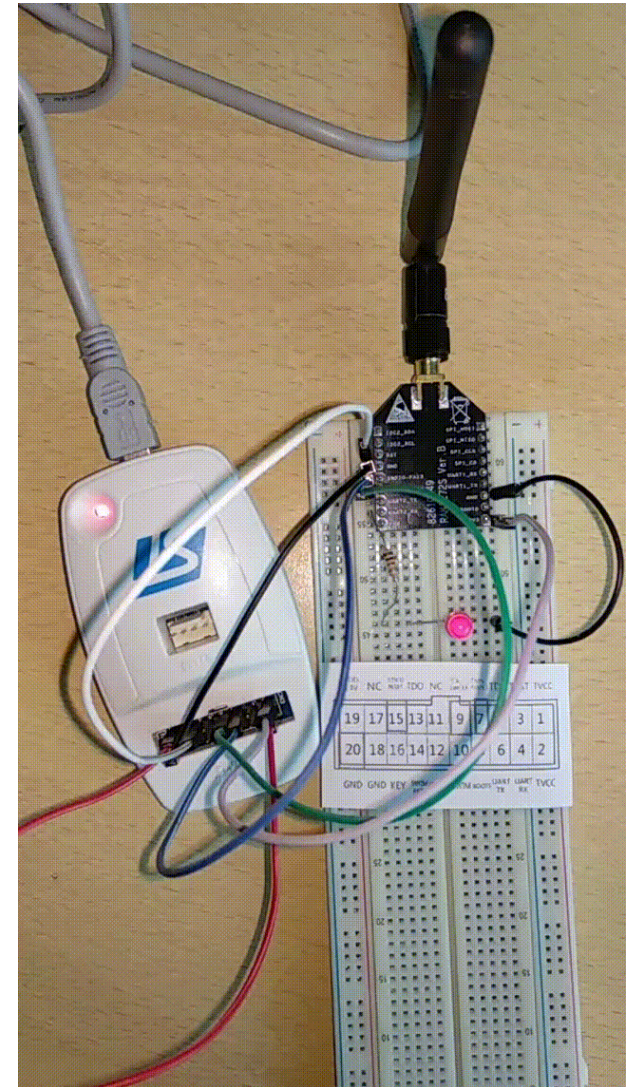
Se utiliza el pin 3.3V del ST-Link para alimentar el RAK3272, pero además tiene que conectarse a los pines TVCC para que el ST-Link "sepa" cuál es la tensión de alimentación del target.



Blink en el RAK3272

```
4  int main()
5  {
6      while (true) {
7          miLED.write(1);
8          //wait() ha sido deprecado en Mbed OS6
9          ThisThread::sleep_for(500ms);
10         miLED.write(0);
11         ThisThread::sleep_for(500ms);
12     }
13 }
14
```

Para que printf funcione por defecto, es necesario estar en el Build profile Debug; no Release.



RTOS: Conceptos básicos

Thread

SysTick timer → RTOS ticker

Scheduler → Quantum time

Delays

Signals o EventFlags

Mutex

Semaphore

Queue y Memory pool

EventQueue

Threads o Tasks: Hilos o tareas

Un proyecto puede requerir diversas tareas o threads.

Por ejemplo, en un reproductor de música podríamos definir 2 tareas:

- Atender la interfaz de usuario (botones para cambiar de canción, y pantalla para mostrar la información de la canción que se está reproduciendo).
- Reproducir el audio de la canción seleccionada.

No podemos esperar a que una tarea termine para ejecutar la otra; ambas tienen que ejecutarse **concurrentemente**.

Pero los microcontroladores de un único núcleo sólo pueden ejecutar una tarea a la vez, por lo que se utiliza un gestor (**Scheduler**) para asignar intervalos de ejecución (**time quantum**) a cada tarea y saltar de una a otra, creando la "ilusión" de que el sistema es capaz de ejecutar varias tareas simultáneamente (**multitasking**).

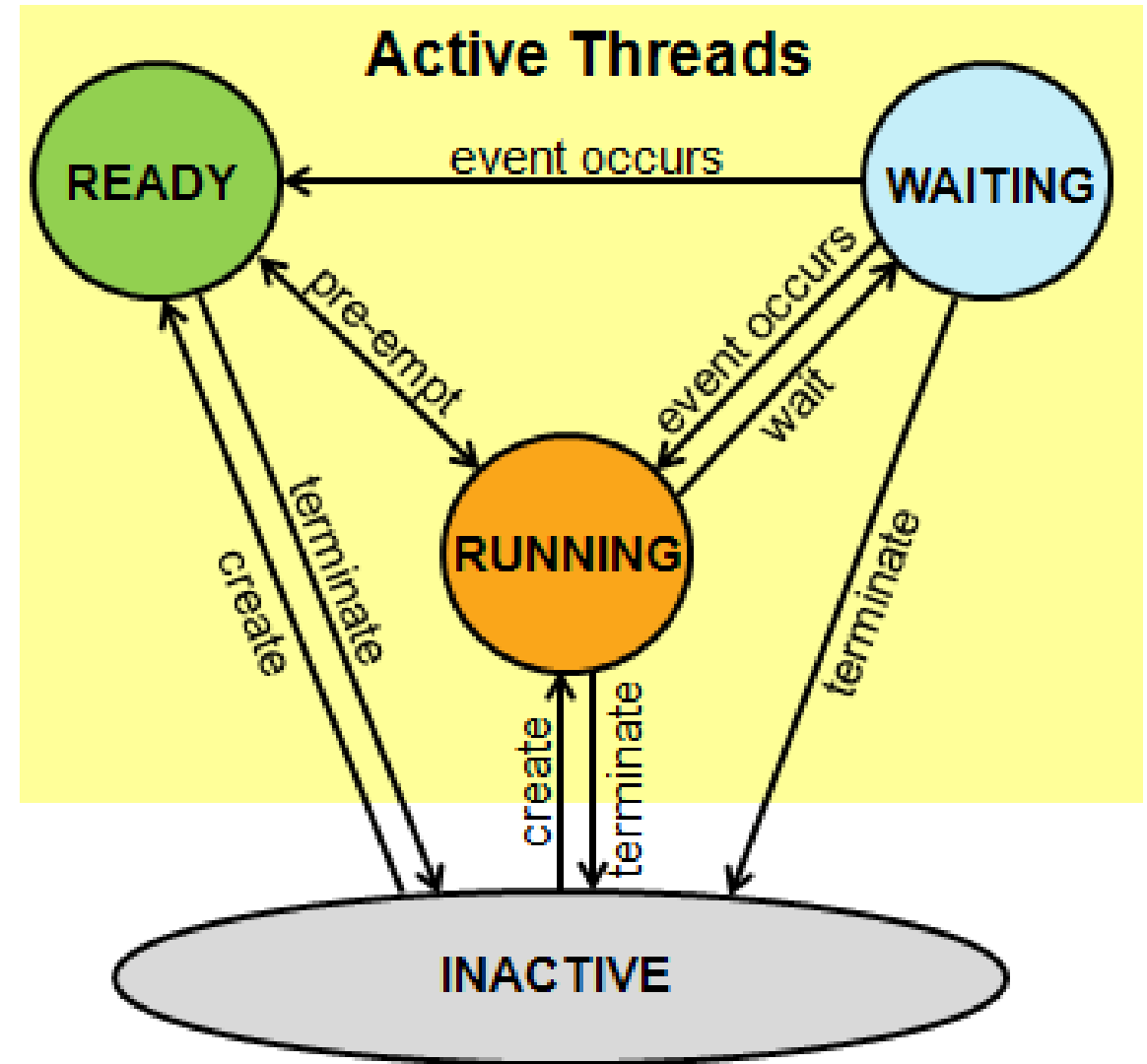
Mbed OS utiliza intervalos de ejecución de **1 ms** de duración.

La rutina **main** es un thread.

Además, existe un thread llamado **idle** que se ejecuta cuando no hay ningún thread en ejecución o están todos detenidos.

Estados de un thread

- **RUNNING**: Es el thread que se está ejecutando. Sólo puede haber un thread en este estado.
- **READY**: Son threads que están esperando que el Scheduler les otorgue un turno de ejecución.
- **WAITING**: Threads que están esperando que se produzca un evento; cuando se produzca pasarán a estado READY.
- **INACTIVE**: Threads que aún no se han iniciado o han terminado.



Priority based round robin Scheduler

Mbed utiliza (por defecto) un Scheduler de tipo **Round Robin**, que se caracteriza por ir saltando de un thread a otro dedicando exactamente el mismo tiempo de ejecución a cada uno de ellos, pero...

...después de cada time quantum, elige a qué thread saltar en función de la prioridad con la que hayan sido definidos.

Si hay un thread con más prioridad que los demás, no consentirá que la ejecución pase a otro thread hasta que él termine o pase a modo wait.

PROCESS	ARRIVAL TIME	BURST TIME		PRIORITY
		TOTAL	REMAINING	
P1	0	4	4	4
P2	1	3	3	3
P3	3	4	4	1
P4	6	2	2	5
P5	8	4	4	2

GANTT CHART:



Modos de ejecución: Thread o Handler

El microcontrolador puede adoptar 2 modos de funcionamiento:

- **Handler o Interrupt:** Es un modo privilegiado en el que el código tiene acceso a todos los recursos del microcontrolador, y que se activa en respuesta a un evento, como puede ser el cambio de estado de un pin (o eventos internos que puede generar el propio RTOS). Generalmente se usa este modo cuando necesitamos responder lo más rápido posible al evento (reducir la **latencia** al mínimo). En general, pero especialmente al usar un RTOS, es esencial que el tiempo que el microcontrolador pasa en modo Handler sea el mínimo posible, porque durante este periodo el RTOS no puede atender otras tareas, pudiendo incluso **comprometer su funcionamiento**.
- **Thread o segundo plano:** El código que se ejecuta en este modo no requiere tanta "velocidad de reacción". La mayoría del código se ejecuta en este modo. De hecho el código que se ejecuta en modo Handler, que se denomina **Interrupt Service Routine (ISR)** generalmente sólo activa una bandera (flag) para que posteriormente un thread se encargue de realizar las tareas de respuesta a ese evento.

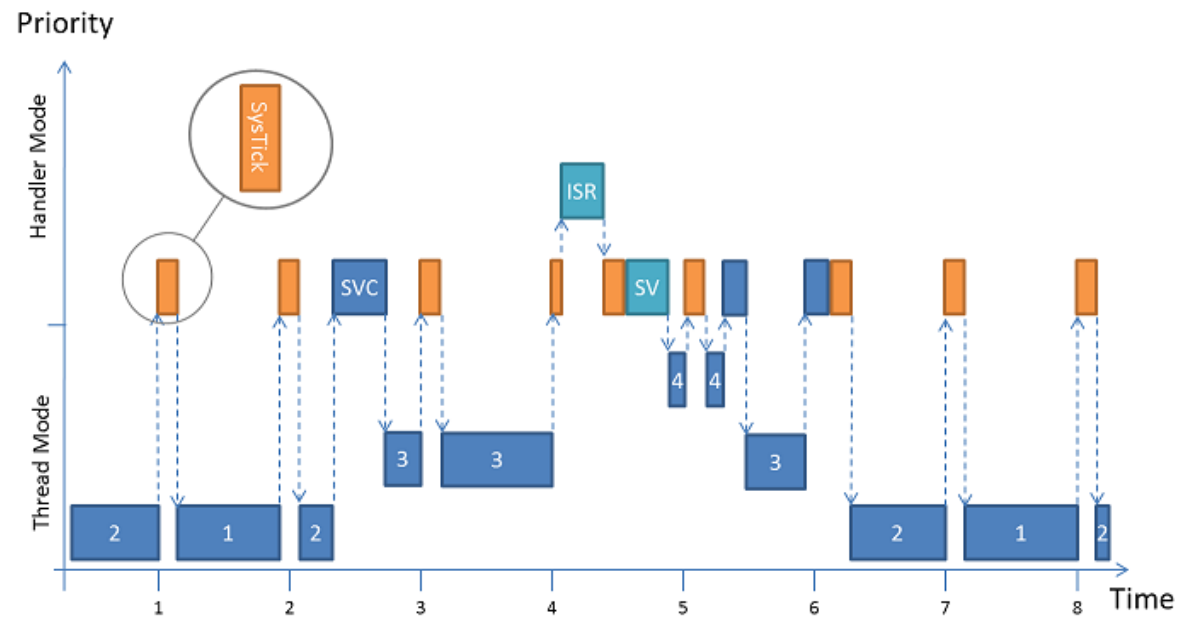
El difícil equilibrio entre el RTOS y las interrupciones

Si damos prioridad a las interrupciones, el RTOS puede quedar comprometido, porque hay operaciones mínimas que tiene que realizar periódicamente.

Si damos prioridad a los thread, las interrupciones aumentarán su latencia.

¿Solución?

El RTOS ejecuta en cada quantum las tareas mínimas de mantenimiento, luego cede el control a las ISR si las hubiera, y luego realiza el resto de sus tareas. Las ISR deben ser cortas, para no superar el tiempo de quantum.



Diferencias entre Threads y funciones

- Los Thread siempre contienen un bucle infinito
- Las funciones siempre devuelven un valor

```
unsigned int procedure (void)
{
    .....
    return(ch);
}
```

```
void thread (void)
{
    while(1)
    {
        .....
    }
}
```

Thread & ThisThread

Thread

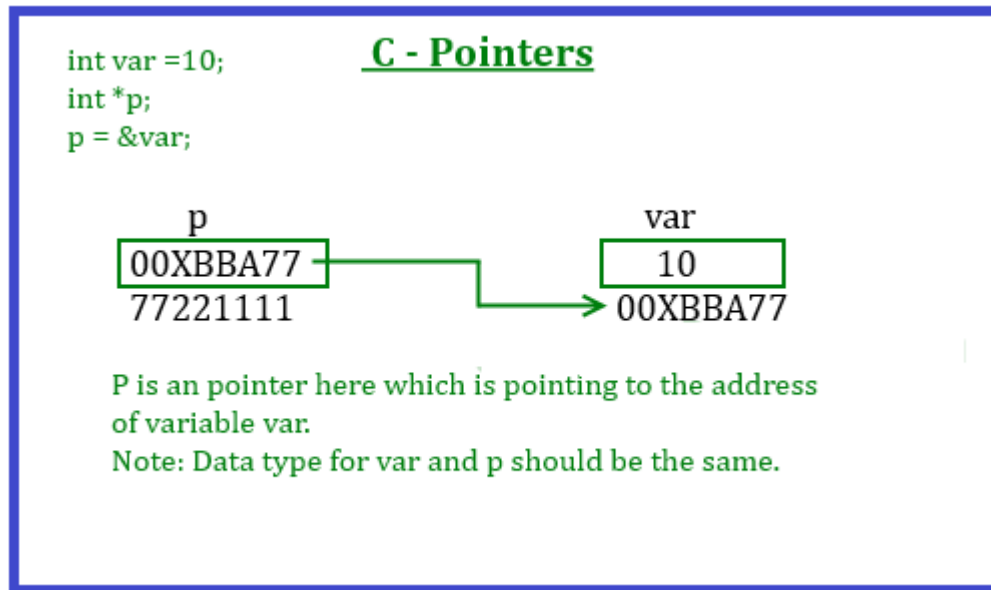
- Constructor: Thread (prioridad)
- start(nombre) → Inicia el Thread
- start(callback(nombre,argumentos))
- get_state()
- get_id()
- terminate()

ThisThread es una **espacio de nombres**, no una clase (no se pueden crear objetos suyos)

```
#include "mbed.h"
DigitalOut miLED(PA_8);
Thread thread;
void led_thread(){
    while (true) {
        miLED = !miLED;
        printf("Toggle LED!\r\n");
        ThisThread::sleep_for(1s);
    }
}
int main(){
    printf("ID main: %d\r\n",ThisThread::get_id());
    thread.start(led_thread);
    printf("ID led_thread: %d\r\n",thread.get_id());
    printf("State: %d\r\n",thread.get_state());
    ThisThread::sleep_for(10s);
    thread.terminate();
    printf("State: %d\r\n",thread.get_state());
}
```

Thread Callback

Para pasar argumentos a un thread tenemos que usar un callback



```
#include "mbed.h"
DigitalOut miLED(PA_8);
Thread thread;
void led_thread(DigitalOut *led){
    while (true) {
        *led = !*led;
        printf("Toggle LED!\r\n");
        ThisThread::sleep_for(1s);
    }
}
int main(){
    printf("ID main: %d\r\n",ThisThread::get_id());
    thread.start(callback(led_thread,&miLED));
    printf("ID led_thread: %d\r\n",thread.get_id());
    printf("State: %d\r\n",thread.get_state());
    ThisThread::sleep_for(10s);
    thread.terminate();
    printf("State: %d\r\n",thread.get_state());
}
```

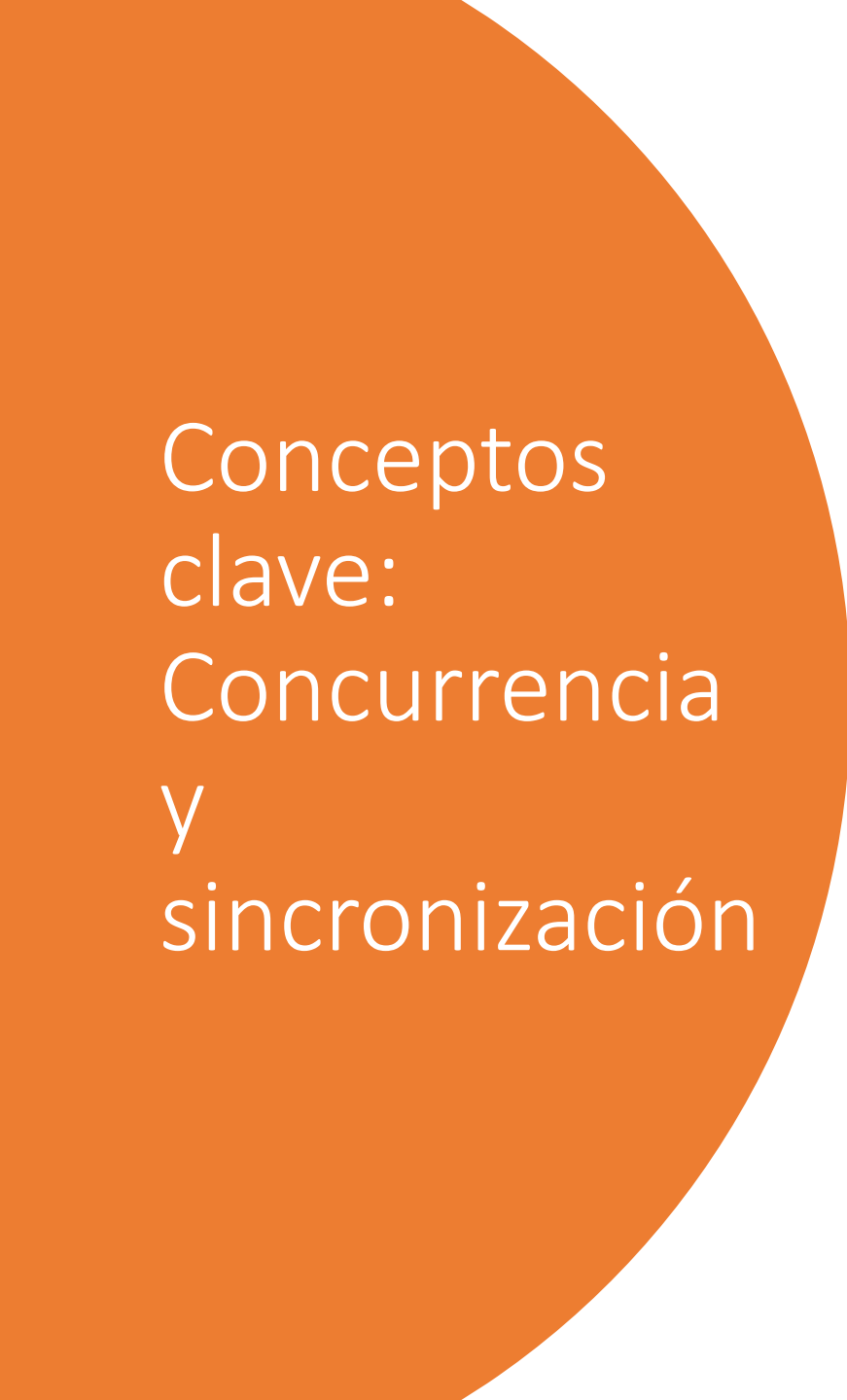
¿Qué ocurrirá en este programa con 2 threads?

```
#include "mbed.h"

Thread alternar_led;
DigitalOut led1(PA_8);
volatile bool running = true;
void blink(DigitalOut *led) {
    while (true){
        if(running) {
            *led = 1;
            ThisThread::sleep_for(2s);
            *led = 0;
            ThisThread::sleep_for(2s);
        }
    }
}
```

```
int main() {
    alternar_led.start(callback(blink, &led1));
    while(true){
        ThisThread::sleep_for(10s);
        running = false;
        ThisThread::sleep_for(6s);
        running = true;
    }
}
```

1. Se ejecuta el thread main, que a su vez inicia la ejecución del thread alternar_led y, a continuación, entra en un bucle infinito en el que alterna el valor de la variable booleana running a intervalos de 10 y 6 segundos.
2. El thread alternar_led, si la variable running es true, hace parpadear un LED una vez cada 4 segundos.
3. El LED se encenderá 3 veces durante 2 segundos, seguidos de 2 segundos de apagado, salvo la tercera vez, que el apagado durará 6 segundos... y se continuará ejecutando este patrón indefinidamente.

A large orange circle on the left side of the slide, partially cut off by the edge.

Conceptos clave: Concurrencia y sincronización

Para que los threads puedan "cooperar" entre sí, no podemos encomendárselo todo a las esperas (`sleep_for`), necesitamos que puedan intercambiar información entre sí.



EventFlags: Señalización entre threads 1/2

Las EventFlags no permiten detener la ejecución de cualquier hilo hasta que se activen ciertas banderas (flags).

Una bandera es simplemente un bit, que estará activado cuando adquiera el valor 1.

Cada EventFlags puede contener hasta 31 banderas.

EventFlags: Señalización entre threads 2/2

```
#include "mbed.h"

#define SAMPLE_FLAG1 (1UL << 0)
#define SAMPLE_FLAG2 (1UL << 9)

EventFlags event_flags;

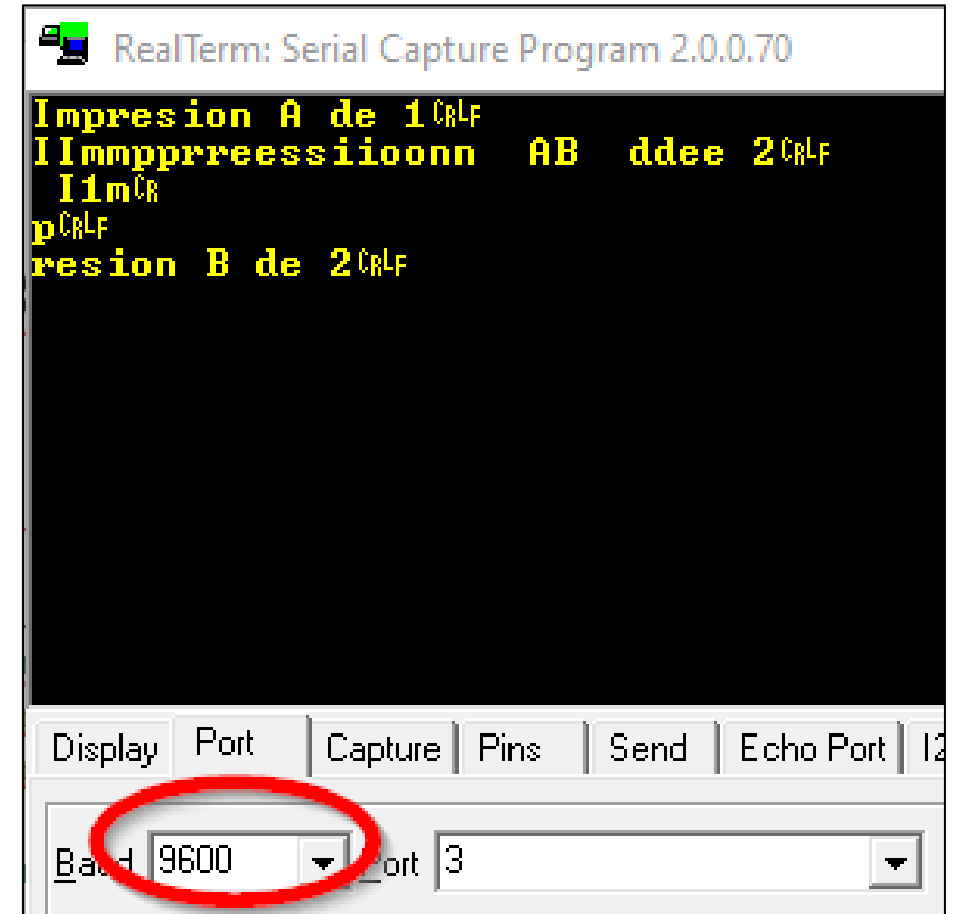
void worker_thread_fun(){
    printf("Waiting for any flag from 0x%08lx.\n", SAMPLE_FLAG1 | SAMPLE_FLAG2);
    uint32_t flags_read = 0;
    while (true) {
        flags_read = event_flags.wait_any(SAMPLE_FLAG1 | SAMPLE_FLAG2);
        printf("Got: 0x%08lx\n", flags_read);
    }
}

int main(){
    Thread worker_thread;
    worker_thread.start(mbed::callback(worker_thread_fun));
    while (true) {
        ThisThread::sleep_for(1000);
        event_flags.set(SAMPLE_FLAG1);
        ThisThread::sleep_for(500);
        event_flags.set(SAMPLE_FLAG2);
    }
}
```


Mutex: Mutual Exclusion 1/2

```
#include "mbed.h"

Thread thread1, thread2;
void doble_impresion (int i){
    printf("Impresion A de %d\r\n",i);
    printf("Impresion B de %d\r\n",i);
}
void codigo_thread1(){
    doble_impresion(1);
}
void codigo_thread2(){
    doble_impresion(2);
}
int main() {
    thread1.start(codigo_thread1);
    thread2.start(codigo_thread2);
    thread1.join();
    thread2.join();
}
```



Mutex: Mutual Exclusion 2/2

```
#include "mbed.h"

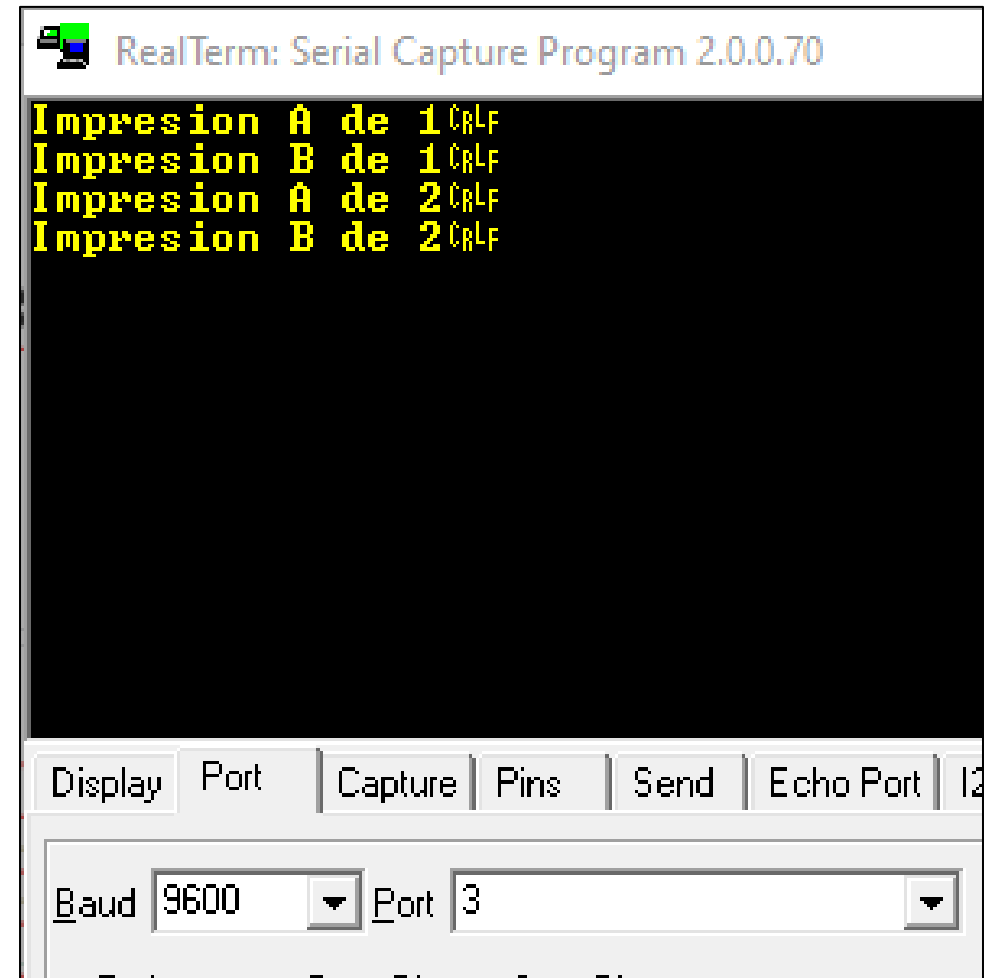
Thread thread1, thread2;
Mutex compartir;

void doble_impresion (int i){
    compartir.lock();
    printf("Impresion A de %d\r\n",i);
    printf("Impresion B de %d\r\n",i);
    compartir.unlock();
}

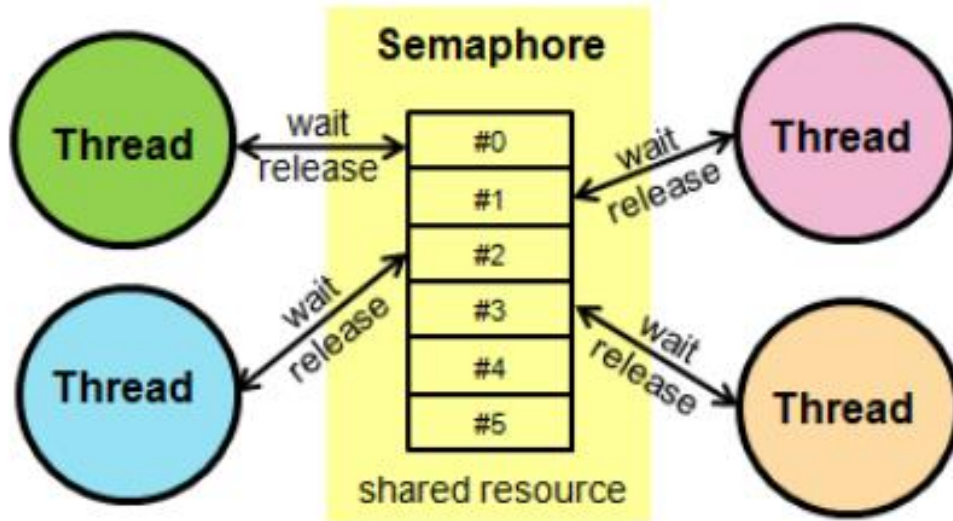
void codigo_thread1(){
    doble_impresion(1);
}

void codigo_thread2(){
    doble_impresion(2);
}

int main() {
    thread1.start(codigo_thread1);
    thread2.start(codigo_thread2);
    thread1.join();
    thread2.join();
}
```



Semaphore 1/3



- Un semáforo es un **repartidor de tokens**, al que los threads le pueden solicitar tokens (acquire) o añadirselos (reléase).
- Si un thread desea adquirir un token, pero no hay ninguno disponible en el semáforo, tendrá que esperar.

Semaphore 2/3

```
#include "mbed.h"

Semaphore continuar(0);
DigitalOut led(PA_8);
Thread t1;
Thread t2;

void codigo_t1(){
    while (true) {
        printf("Voy a dejar que t2 parpadee el LED una vez dentro de 10 segundos\r\n");
        ThisThread::sleep_for(10s);
        continuar.release();
    }
}
```

```
void codigo_t2(){
    while (true) {
        continuar.acquire();
        led=1;
        ThisThread::sleep_for(500ms);
        led=0;
        ThisThread::sleep_for(500ms);
    }
}

int main(void){
    t1.start(codigo_t1);
    t2.start(codigo_t2);
}
```

Semaphore 3/3

Cuándo utilizar semáforos:

- Enviar **señales** entre threads (diapositiva anterior)
- **Multiplexar** el acceso de varios threads a un conjunto limitado de recursos (por ejemplo, a una memoria con doble puerto)
- **Redezvous**: Es un caso particular de envío de señales en el que 2 threads se envían mensajes entre sí al alcanzar cierto punto crítico, para garantizar que ninguno de ellos sigue adelante sin que el otro haya alcanzado también su punto crítico.
- **Barrera**: Es un caso más general de Redezvous que sirve para cualquier número de threads (ninguno continuará hasta que todos hayan alcanzado su punto crítico). Se usan 2 semáforos (contador y barrera) y una variable global (terminado). Inicialmente contador tiene un token y barrera ninguno. Cada vez que un hilo alcanza su punto crítico, intenta adquirir un token de contador, incrementa el valor de terminado, libera el token de contador para que lo pueda usar otro thread, comprueba si terminado es igual al número total de threads, en cuyo caso añade tanto tokens a barrera como threads, o en caso contrario espera hasta poder adquirir un token de barrera.