From blocking RTOS to event-driven with Quantum Leaps

...

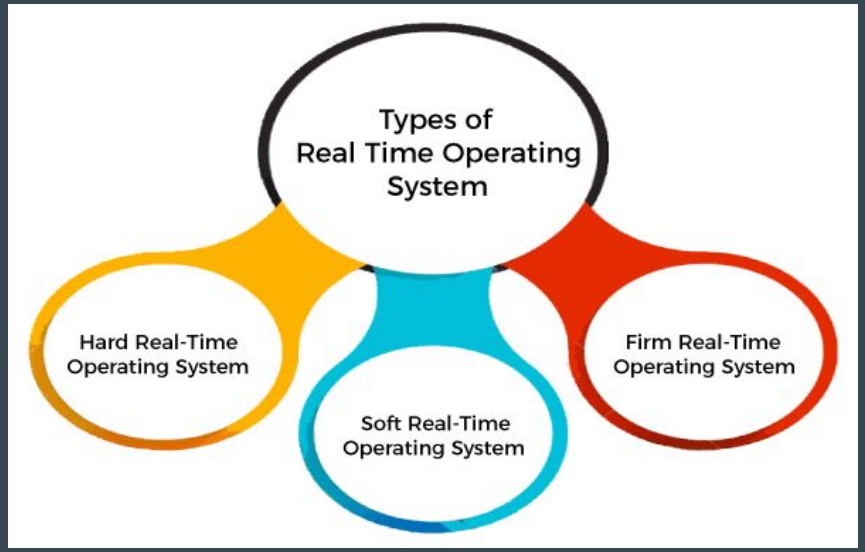
Mg.Ing. Pablo Slavkin
Est. Valentino Slavkin

Agenda

1. Conceptos básicos de un RTOS bloqueante
2. Conceptos básicos de un RTOS no bloqueante
3. Uso de QP en detalle y sus herramientas (SM visuales, Qspy)
4. Hands-on en un embebido compatible (STM32, TM4C, edu-ciaa)

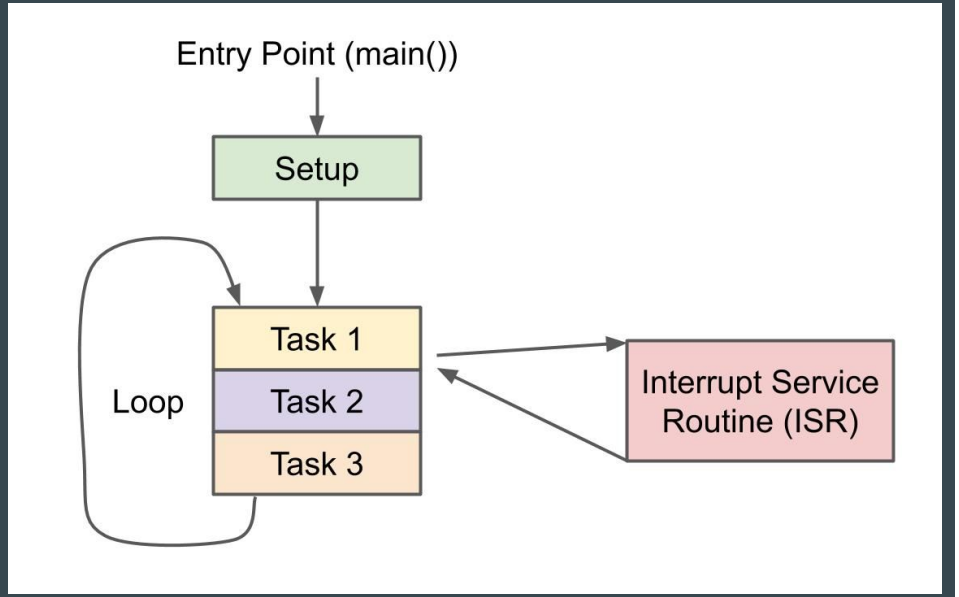
Porque RTOS?

Hard vs Firm vs soft



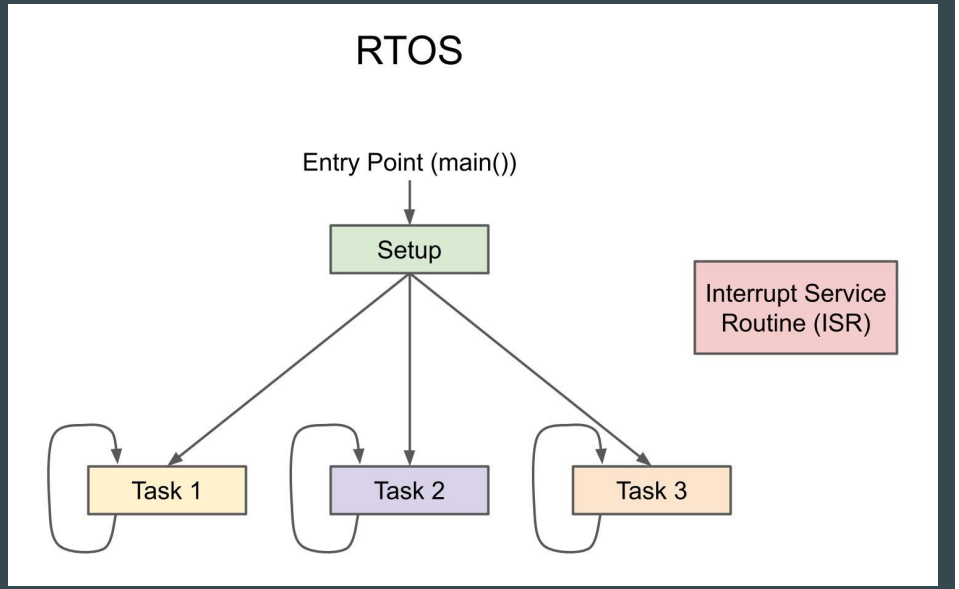
- **Soft**: no hay consecuencias si no se cumplen los requisitos temporales
  - Interfaces de usuario
  - Web server
- **Firm**: Se pretende cumpliir con requisitos de tiempo pero si eventualmente no se cumple, no hay consecuencias criticas
  - Video streaming (si se pierde un frame, aun se puede sguir viendo)
- **Hard**: Si no se cumplen los requisitos de tiempo, las consecuencias son catastróficas
  - Airbag
  - Sistema de apagado de un reactor nuclear

Baremetal



- Se ejecuta una función tras otra
- Generalmente se usan flags para determinar el momento de ejecutar una funcion
- Los flags se cambian en una ISR o desde otra función
- Facil de depurar
- Super simple y eficiente en cuanto a memoria y overhead de procesador
- No se puede garantizar el tiempo de atención a cada función
- No escala en proyectos grandes
- Difícil para trabajar en equipo

RTOS

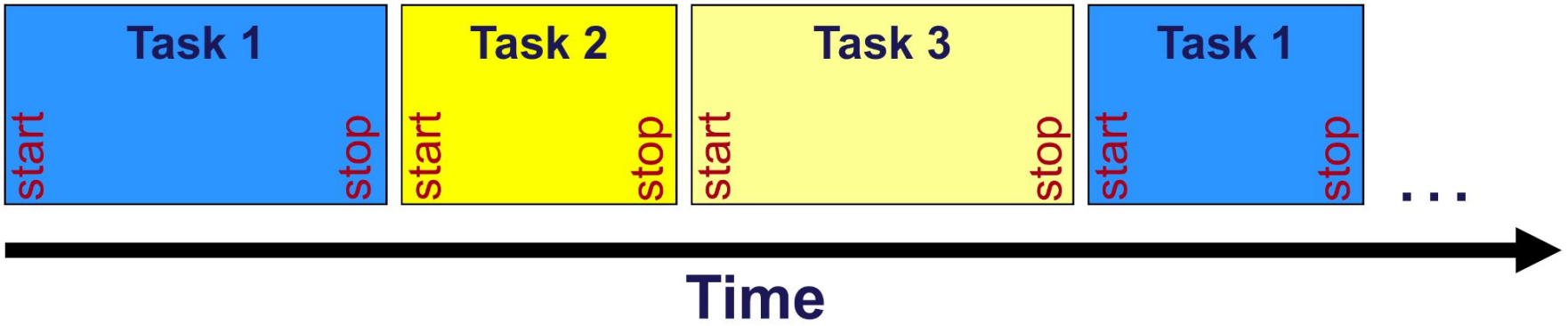


- Se ejecuta la tarea de mayor prioridad primero
- Escala mejor en proyectos grandes
- Fácil de repartir tareas entre miembros de un euipos
- Dificil de depurar
- Hay que tener consideraciones para compartir recursos entre tareas
- Menor eficiencia en el uso de memoria y procesador
- Requiere de un hardware particular para lograr una performance razonable
- Permite aplicaciones hard real time
- Dificil de portar entre RTOS de diferentes fabricantes

Schedulers: algoritmos de selección de tareas

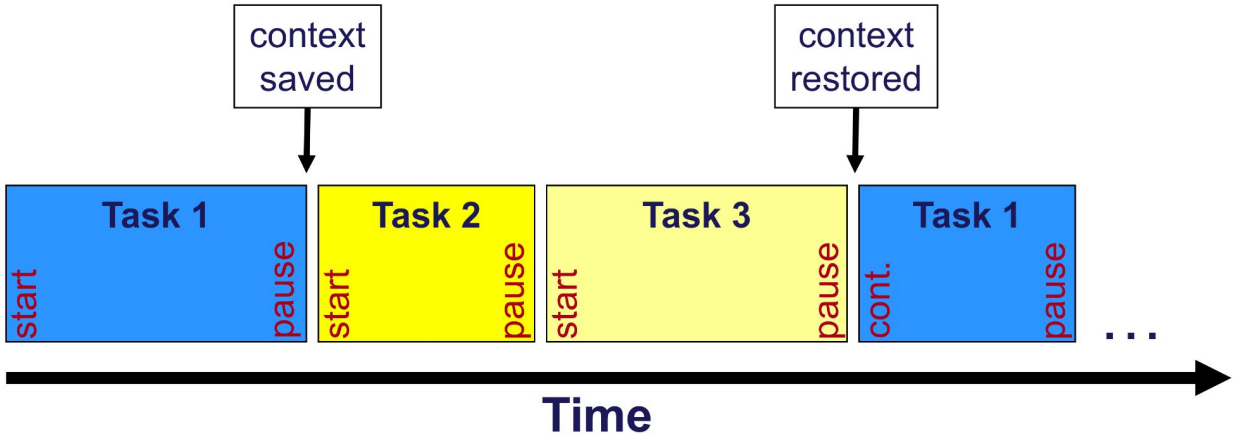
Run To Completion (RTC) sin OS

- El periodo entre una llamada y la siguiente, depende del resto de tareas.
- Simple y eficaz. Superloop
- Requiere cooperación entre las tareas, caso contrario, una tarea podría nunca recibir tiempo de procesador. (Inanicion, starvation).



Round Robin cooperativo

- Las tareas pueden no terminar, pero deben relegar el CPU cooperando asi con las demás tareas.
- El sistema operativo deberá mantener el contexto de cada tarea para continuar desde donde se la dejo.



Round Robin

```c
#include <pt.h>

static struct pt pt1, pt2;

PT_THREAD(task1(struct pt *pt)) {
    PT_BEGIN(pt);

    printf("Task 1 started\n");
    PT_YIELD(pt);
    printf("Task 1 resumed\n");

    PT_END(pt);
}

PT_THREAD(task2(struct pt *pt)) {
    PT_BEGIN(pt);

    printf("Task 2 started\n");
    PT_YIELD(pt);
    printf("Task 2 resumed\n");

    PT_END(pt);
}
```

- Protothreads es un ejemplo clasico de round robin cooperativo

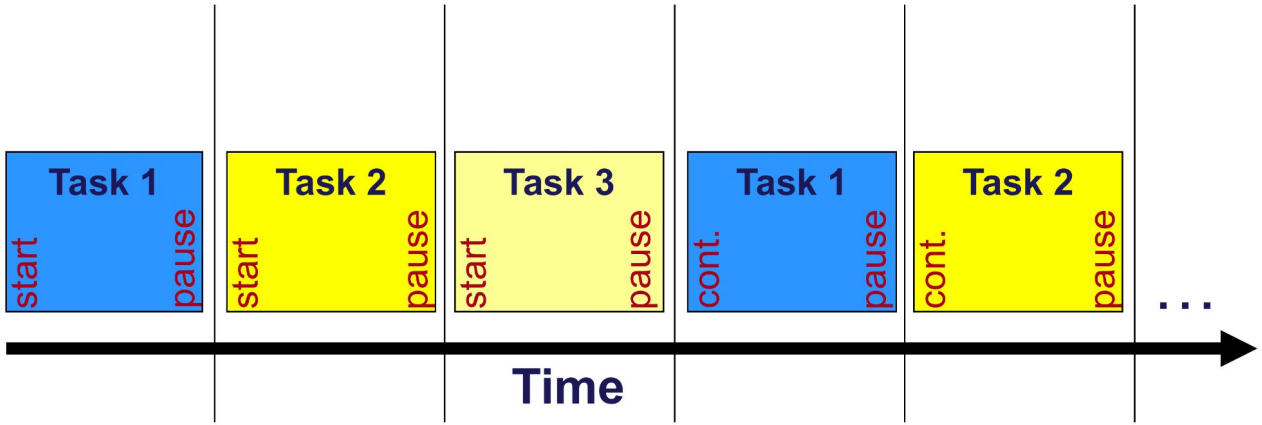
```c
int main() {
    PT_INIT(&pt1);
    PT_INIT(&pt2);

    while (1) {
        task1(&pt1);
        task2(&pt2);
    }

    return 0;
}
```

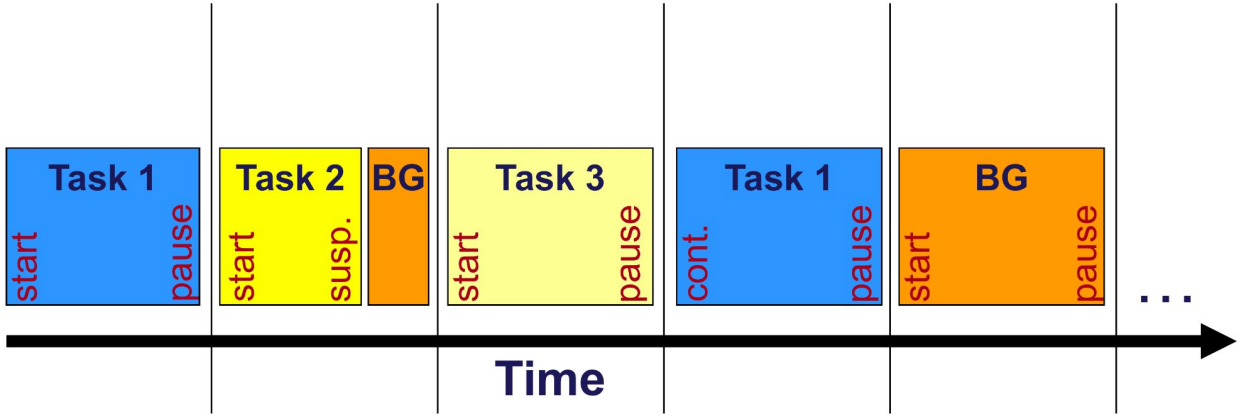
Time slice preemptivo

- Las tareas tienen un tiempo asignado ‘slot’ en las cuales son llamadas
- Si una tarea no tiene trabajo, igualmente desperdicia CPU
- Preemptivo, deterministico y full real time



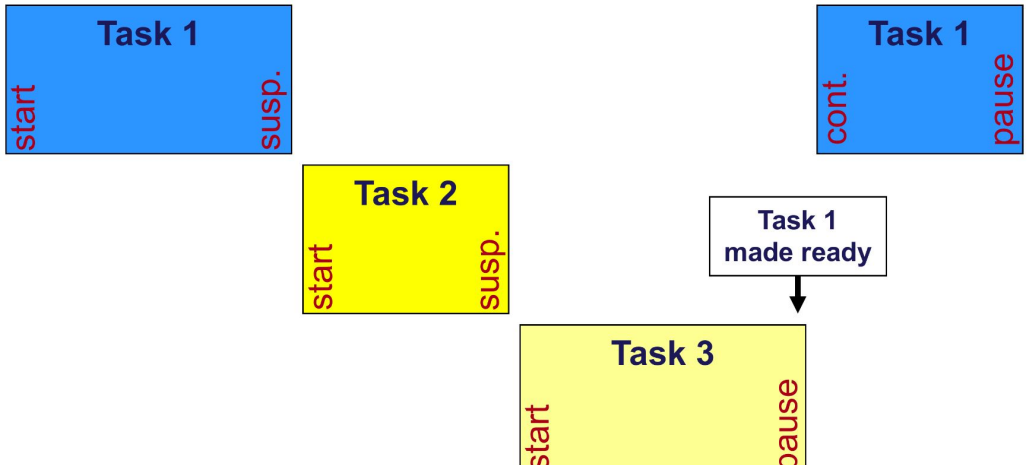
Time slice with background task

- Si alguna tarea no tiene trabajo que hacer se llama a una tarea de background
- Pude hacer tareas de diagnóstico, dormir, trace, loop



Time slice with priorities

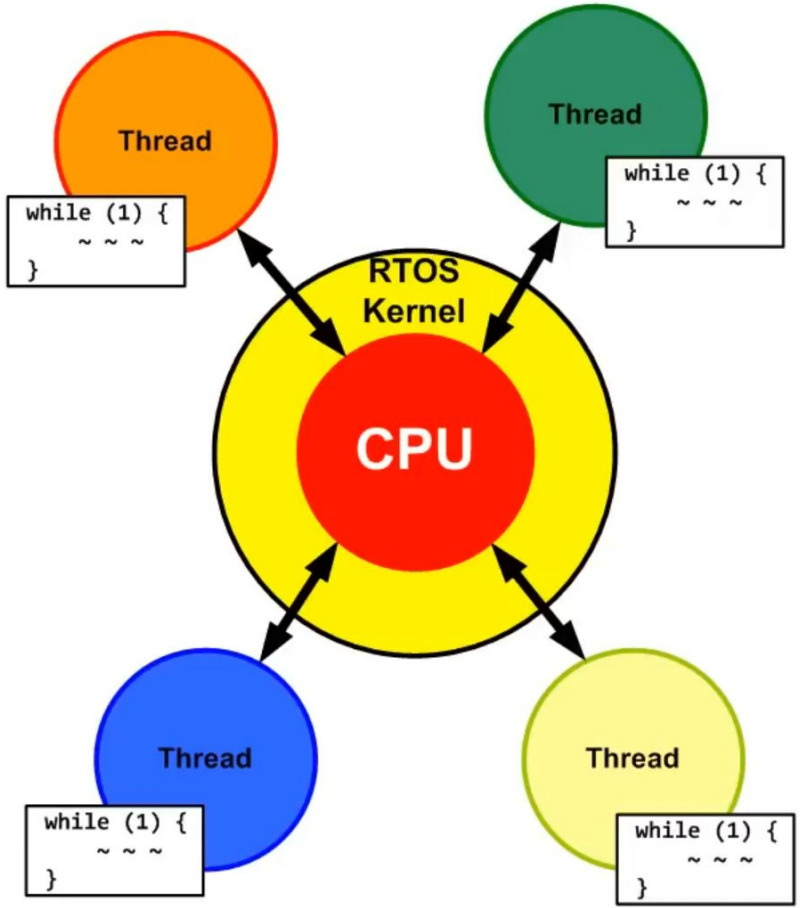
- El scheduler siempre ejecuta la tarea de mayor prioridad que esté pendiente de ejecución.
- Si más de una tarea tienen la misma prioridad se utiliza Time slice entre ellas



RTOS bloqueante:
Paradigma secuencial
FreeRTOS

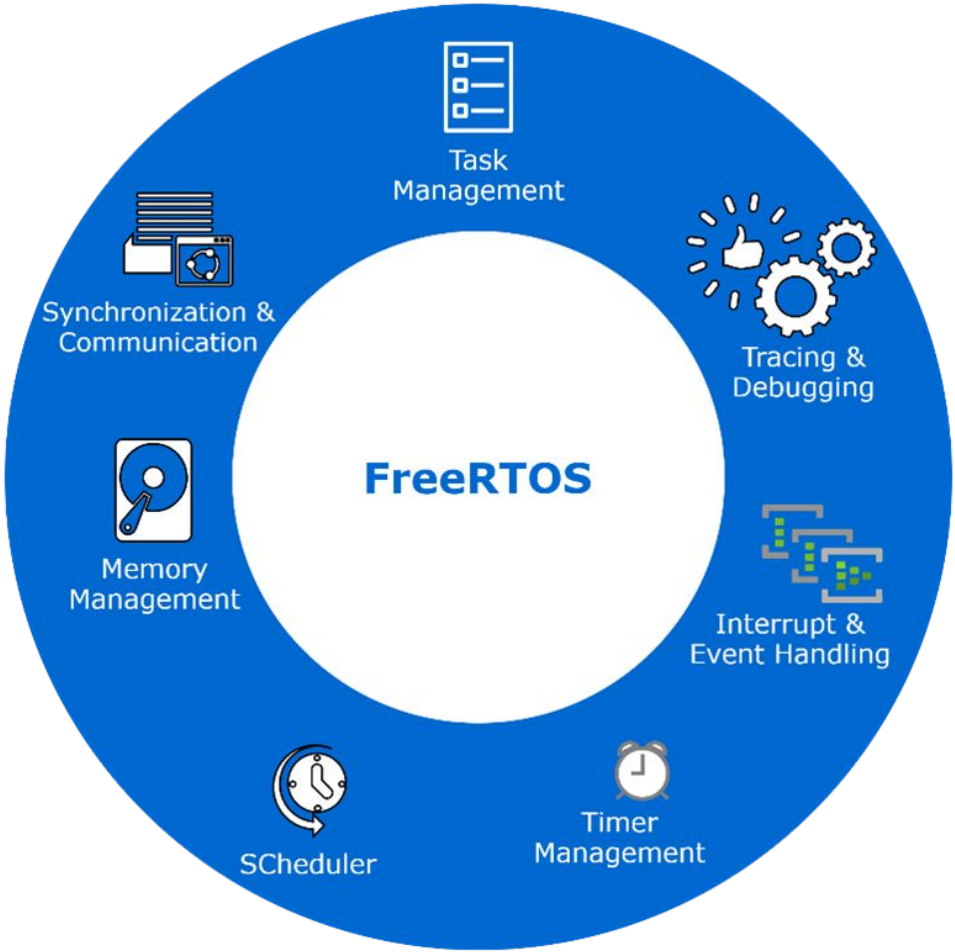
Preemtivo

- Programación secuencial, como si el resto de las tareas no existieran
- El kernel decide que tarea corre en cada momento, en función de sus prioridades



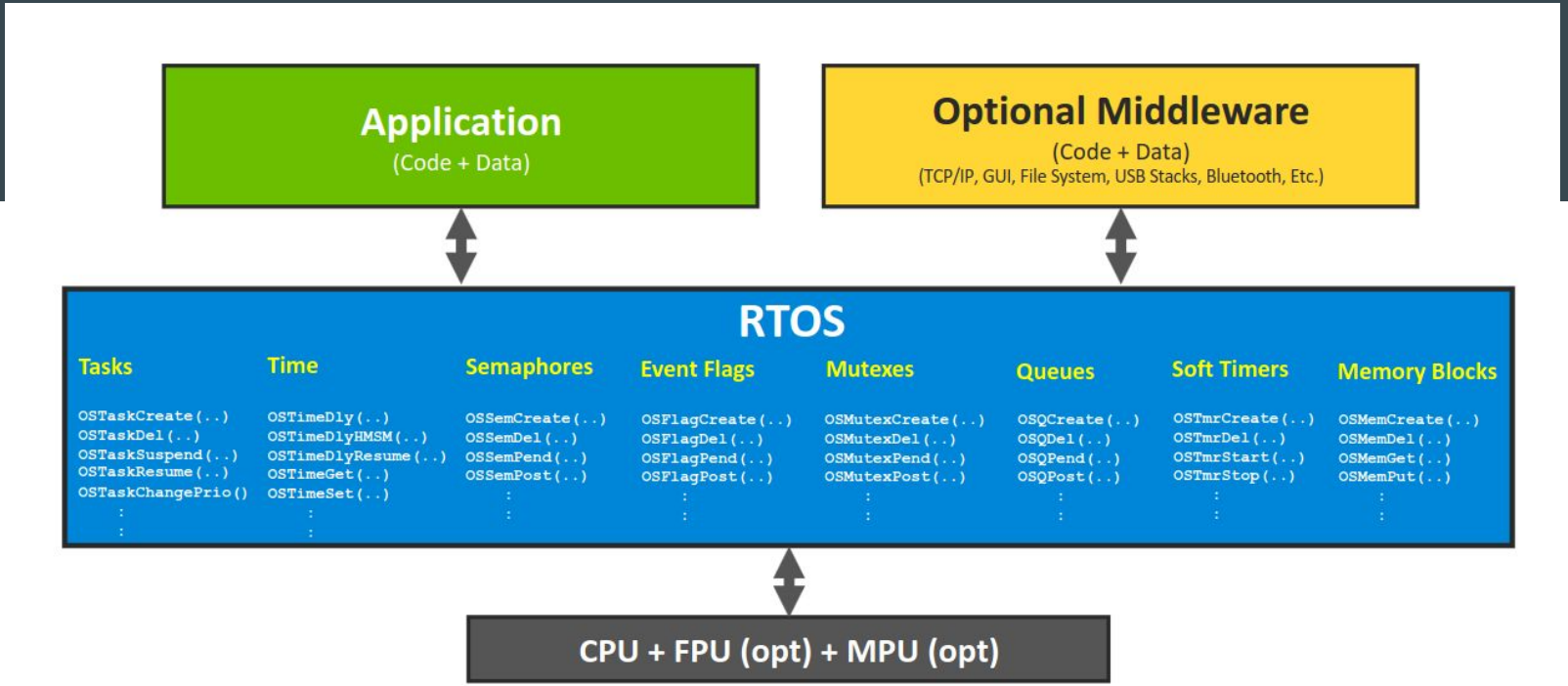
FreeRTOS

- Es uno de los RTOS más populares
- Además del kernel agrega otras utilidades
- Esta portado a muchas arquitecturas
- Tiene varias opciones de licenciamiento y certificados



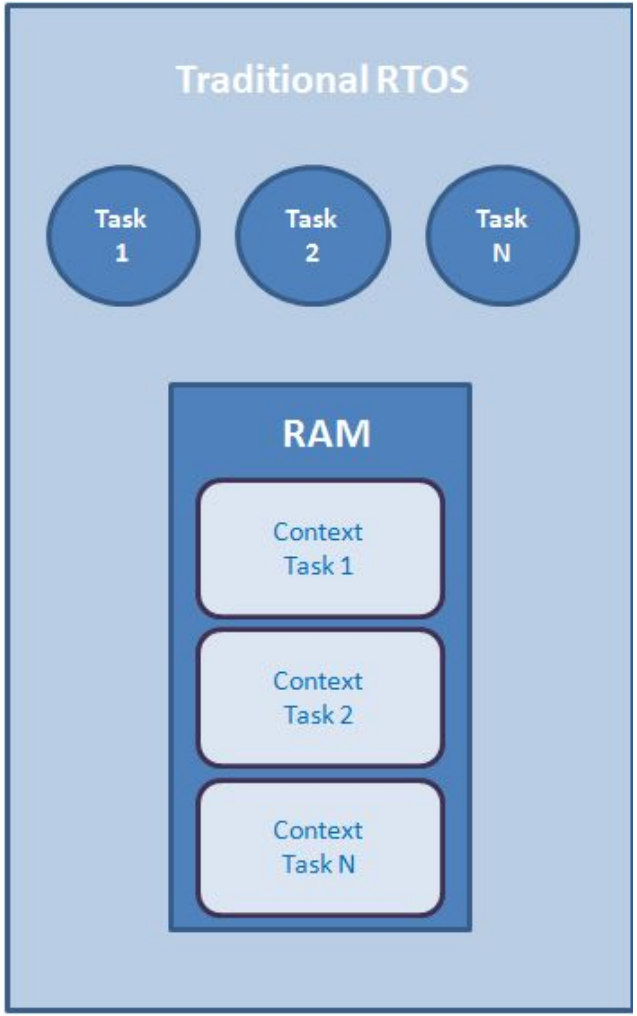
FreeRTOS

Algunas de las herramientas de FreeRTOS



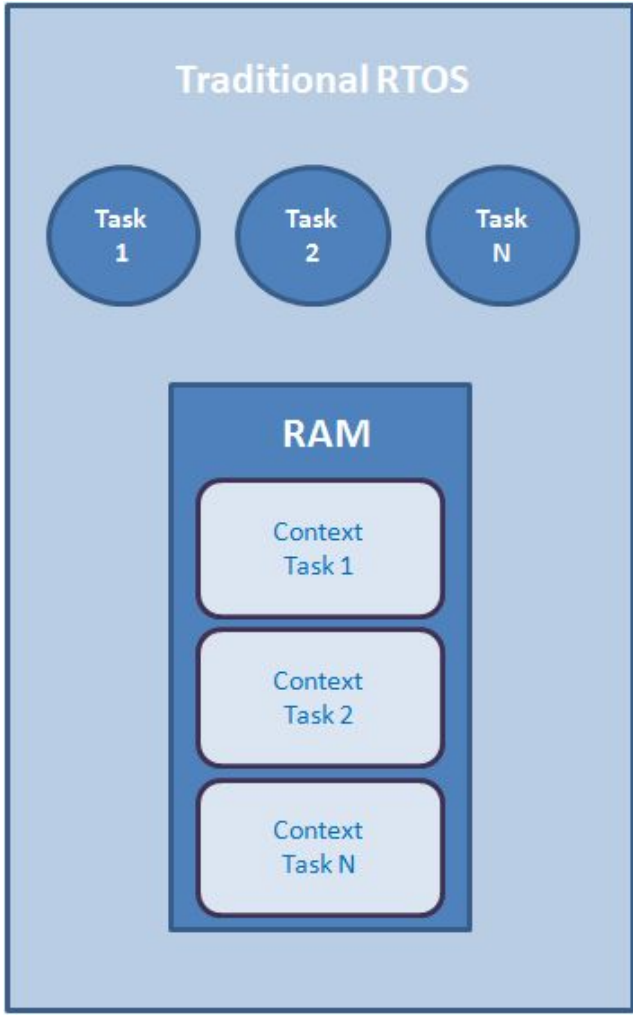
Uso de RAM para administrar tareas

- Como cada tarea requiere su propio stack, donde mantiene las variables, locales, no es tan eficiente en el uso de la RAM.



Stack para Interrupciones

- Donde se pushean los datos cuando llega una Interrupción?
  - Si la arquitectura soporta doble stack, en el stack del main
  - En otro caso se usará el stack de la tarea que esté corriendo en ese momento.
    - Habra que contemplar el stack de los handlers



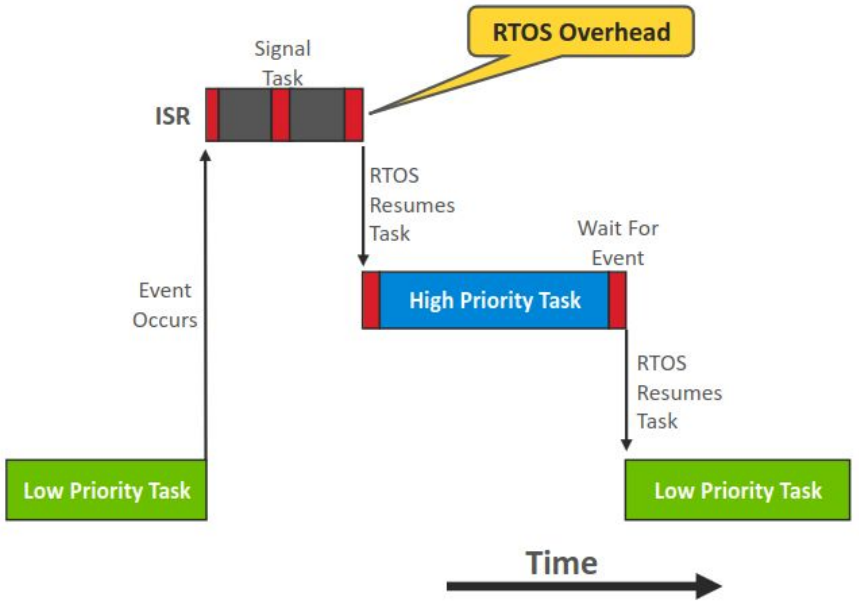
Cambios de contexto

Se almacenan los registros del uC de la tarea en curso para cambiar al contexto de otra
Se restablecen los registros del uC de la siguiente tarea a correr

```
void Low_Prio_Task (void)
{
  Task initialization;
  while (1) {
    Setup to wait for event;
    Wait for event to occur;
    Perform task operation;
  }
}
```

```
void ISR (void)
{
  Entering ISR;
  Perform Work;
  Signal or Send Message to Task;
  Perform Work;  // Optional
  Leaving ISR;
}
```

```
void High_Prio_Task (void)
{
  Task initialization;
  while (1) {
    Setup to wait for event;
    Wait for event to occur;
    Perform task operation;
  }
}
```

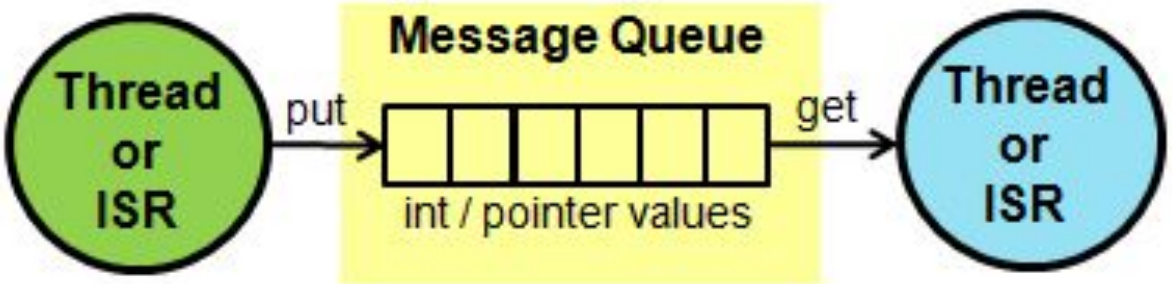


Semaforos

- Sincronización entre tareas
- Evitar condiciones de carrera

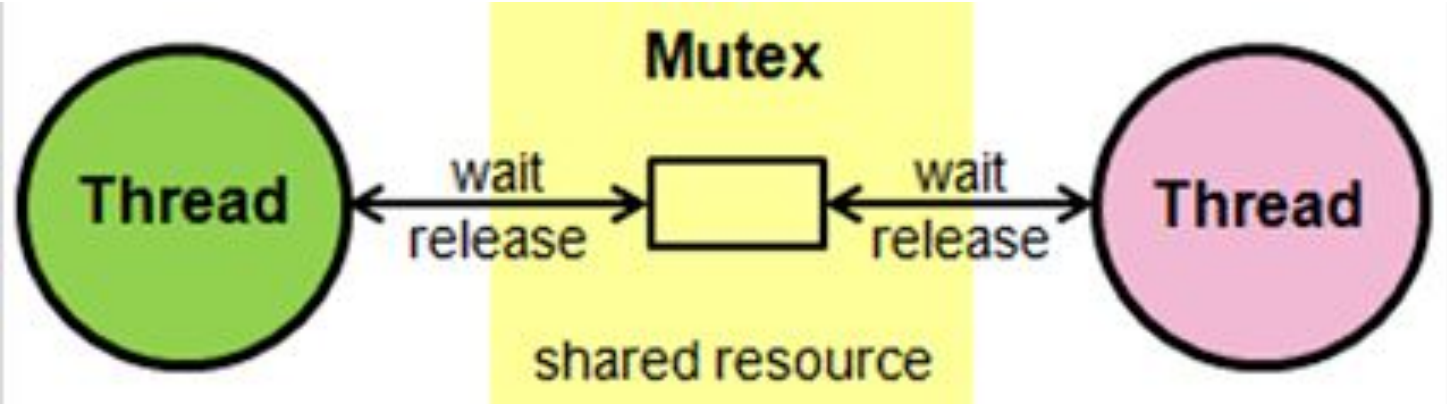
Queues

- Envío de datos de una tarea a otra
- Sincronización entre tareas
- Generalmente en FreeRTOS los msg se pasan por copia => doble memcpy => mas flexible



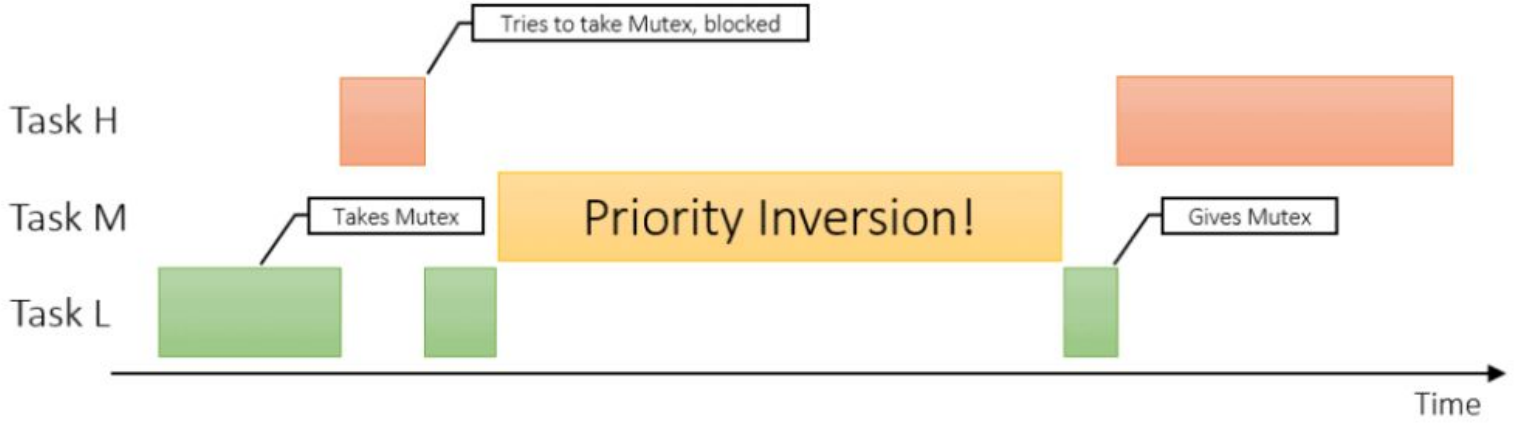
Mutex

- Protección de acceso concurrente a recursos
- Problema de inversión de prioridades

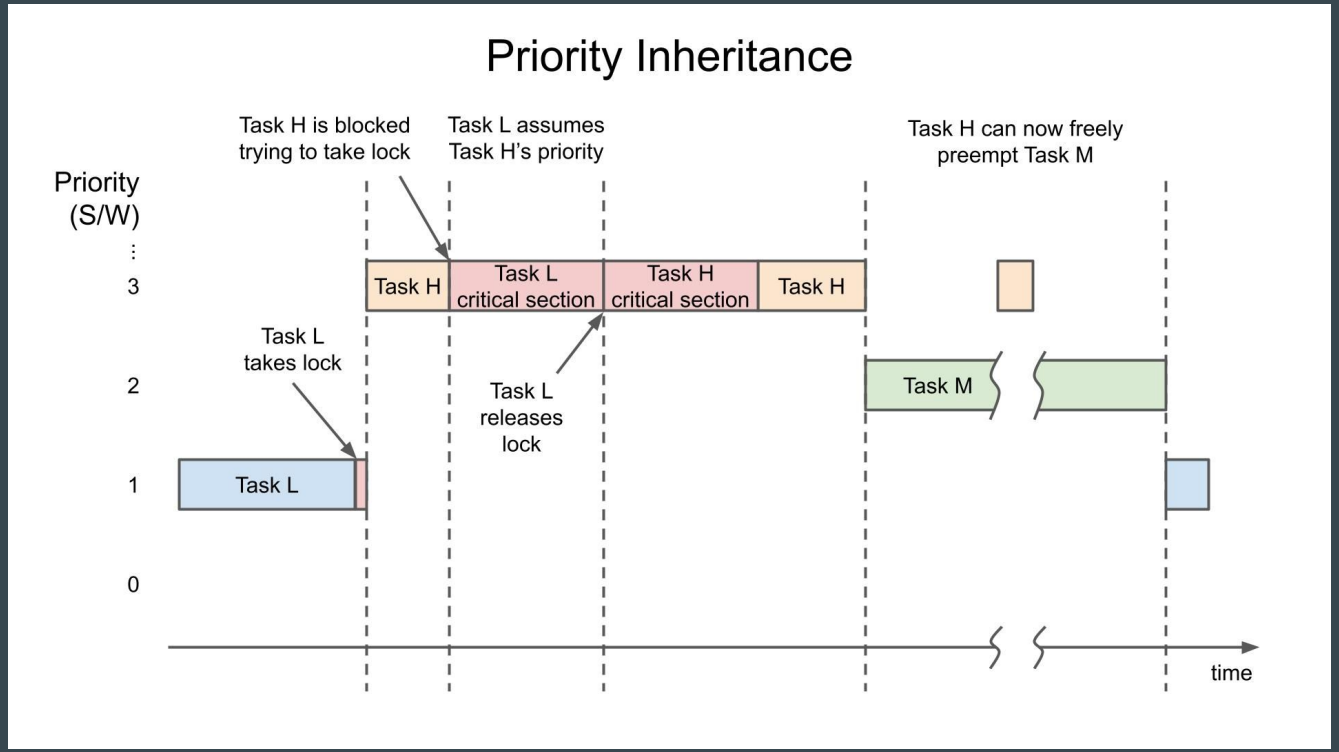


Inversion de prioridades

- Situación en la que una tarea de baja prioridad bloquea temporalmente a una tarea de mayor prioridad
- La tarea de baja prioridad monopoliza el recurso, impidiendo que la tarea de mayor prioridad se ejecute



Inversion de prioridades mitigacion



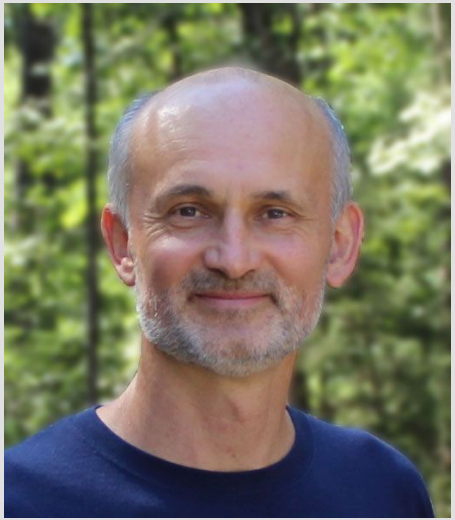
Cuando una tarea de baja prioridad adquiere un mutex, hereda temporalmente la prioridad de la tarea de mayor prioridad que está bloqueada

Hand on TM4C

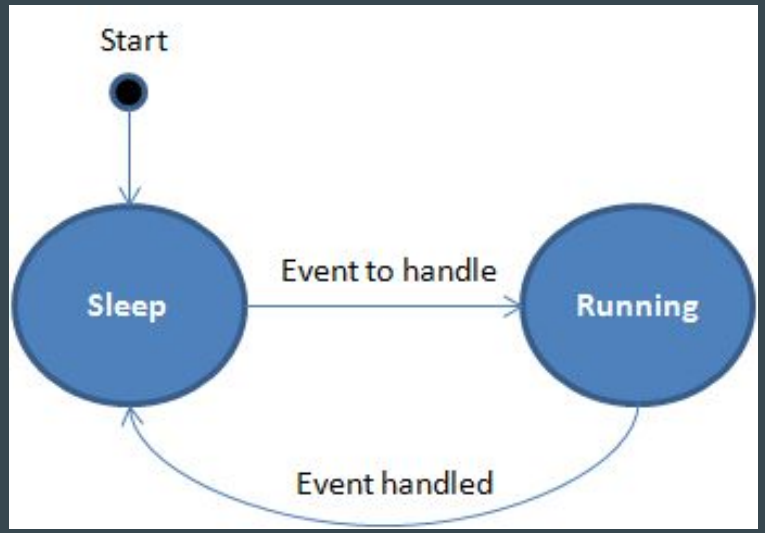
Blinky over RTOS

```
static void blink1(void* params)
{
    (void)params;
    while(true) {
        vTaskDelay(pdMS_TO_TICKS(500));
        led_on ( LED_GREEN );
        uart_printStr(APP_DEBUG_UART,  "led1 on\r\n");
        vTaskDelay(pdMS_TO_TICKS(500));
        led_off ( LED_GREEN );
        uart_printStr(APP_DEBUG_UART,  "led1 off\r\n");
    }
}
```

RTOS no bloqueante:
Paradigma RTC
Quantum Leaps



Run to completion



- No se permite bloquear
- Las tareas deberán ser relativamente ‘cortas’
- Si una tarea requiere de mucho tiempo de procesamiento, se puede dividir en tareas más cortas
- El RTC más largo de un sistema determina la responsividad de peor caso

QEP - RTC con cola de eventos y prioridades

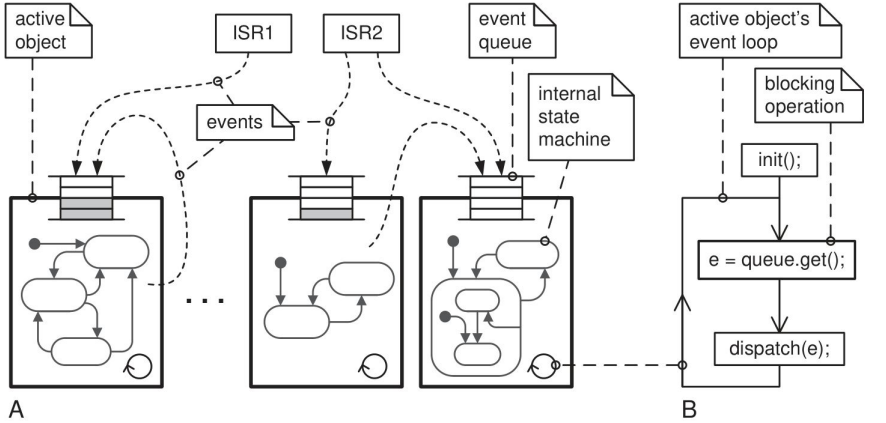
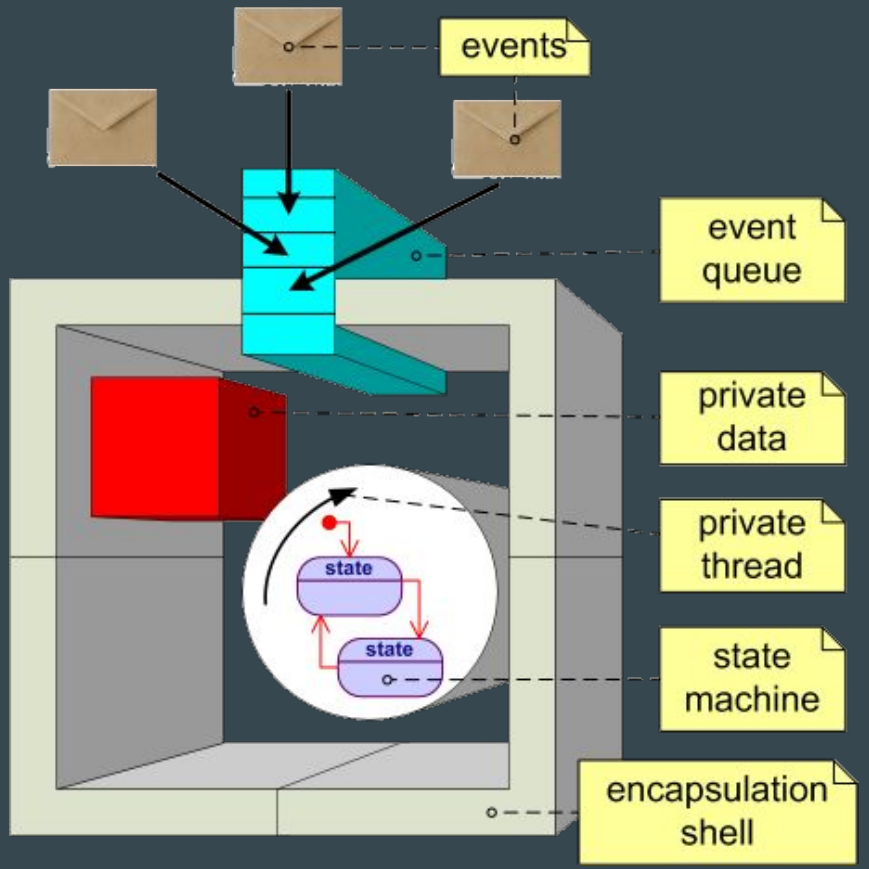


Figure 6.5: Active-object system (A) and active object's event loop (B).

- Una cola de eventos para cada tarea
- Un procesador de eventos decide qué tarea procesar en base a la prioridad de cada tarea y si tiene o no eventos
- Los eventos se pueden encolar desde una tarea a otra o desde una ISR o desde servicios del framework como timers
- Inversion de control

AO - Active Objects

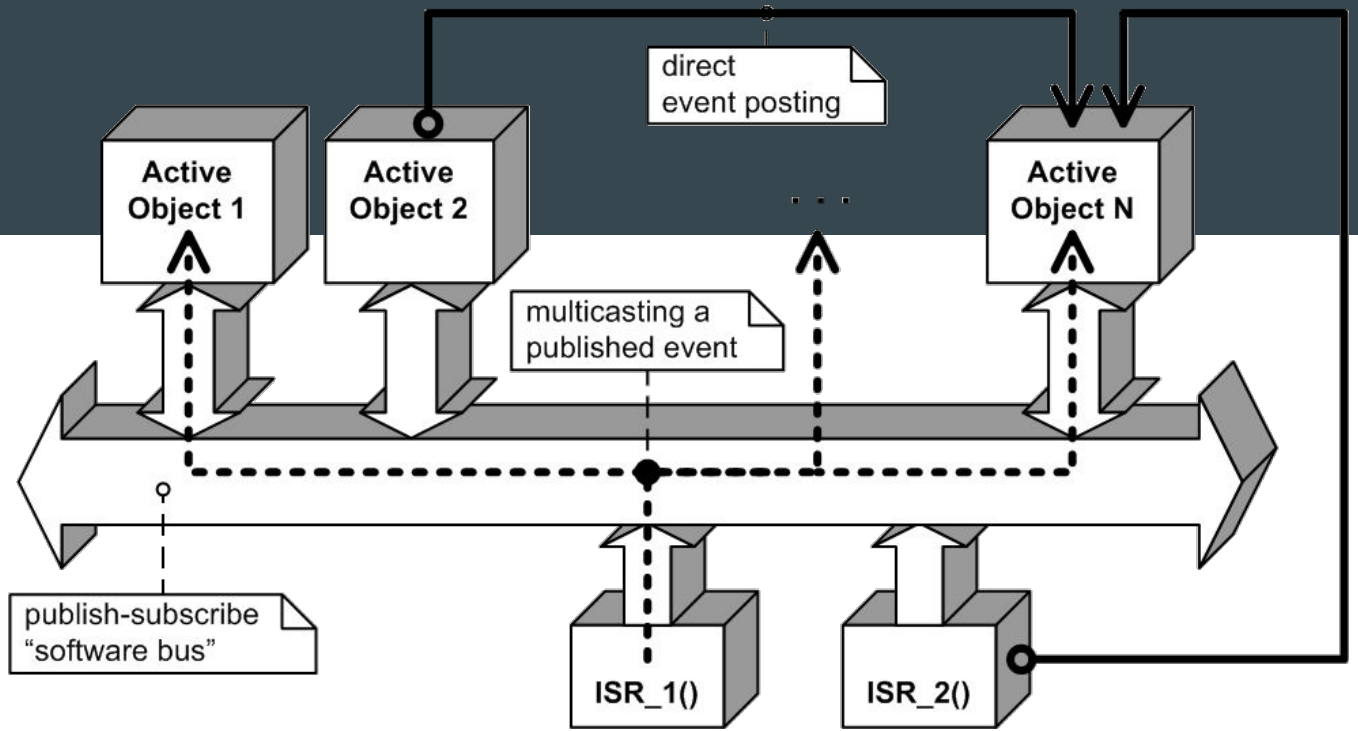
AO



- Es similar a una tarea en un RTOS. Limite de 31
- Tiene una cola de mensajes por la cual se comunica
- Tiene asignada una única SM, aunque a partir de esta puede manejar otras ortogonales
- Puede tener variables privadas
- Se debería poder instanciar más de una vez
- Tiene una ‘única’ prioridad o identificador. Aunque se pueden agrupar varios AO en un grupo de prioridades equivalentes

AO

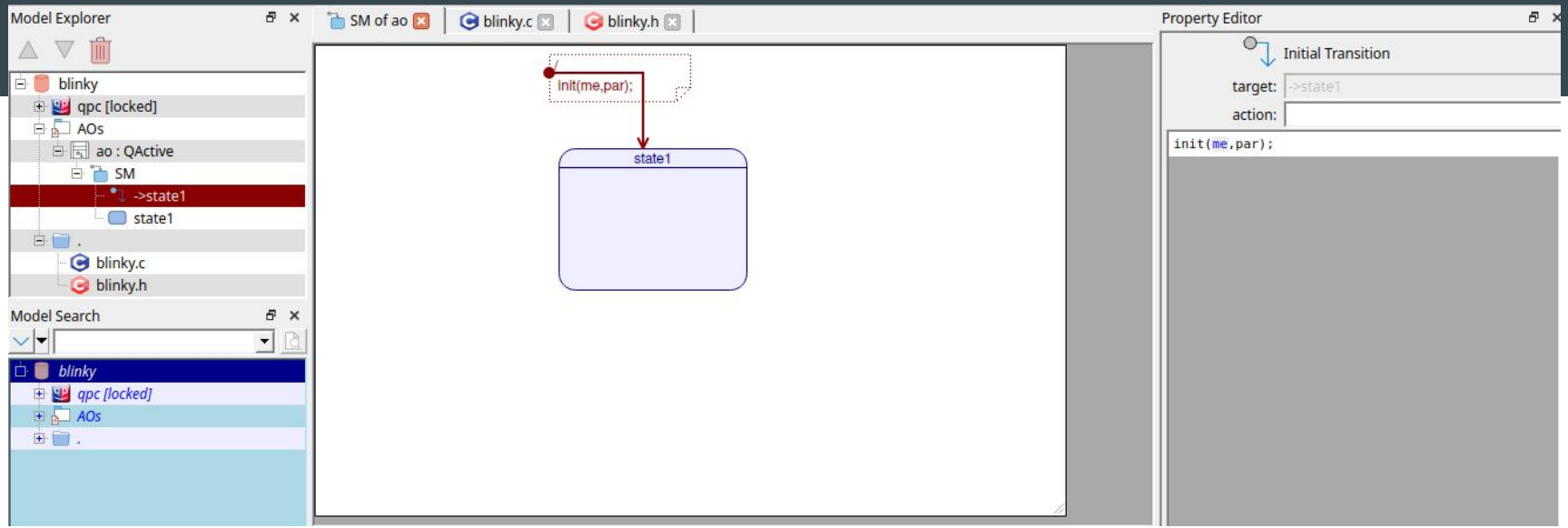
- Los AO se pueden comunicar con POST / PUSBLISH
- Las SM se comunican con DISPATCH



HSM - Máquinas de estado jerárquicas

QM

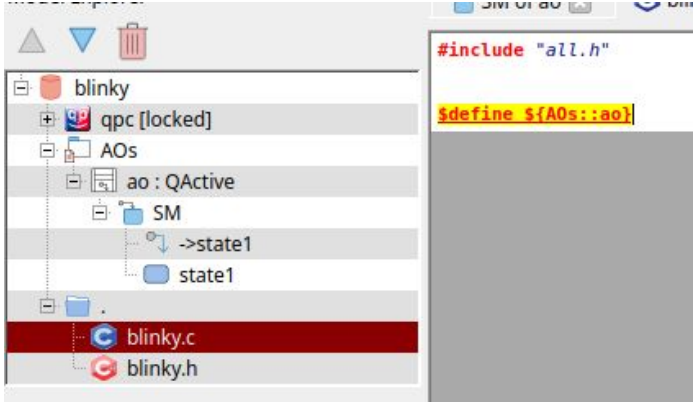
- Agregar un package
  - Agregar una clase de tipo QActive
    - Agregar una SM
      - Indicar el punto de inicio
        - Asociar files y generar código



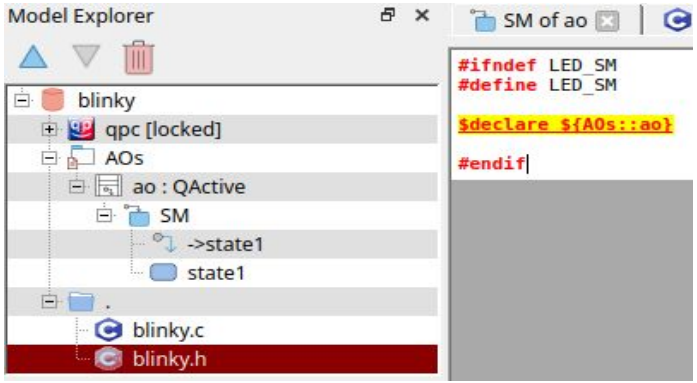
QM

Generar código y veo definiciones en .c, declaraciones en .h

```
18 //${AOs::ao::SM} .....
17 QState ao_initial(ao * const me, void const * const par) {
16     //${AOs::ao::SM::initial}
15     init(me,par);
14     return Q_TRAN(&ao_state1);
13 }
12
11 //${AOs::ao::SM::state1} .....
10 QState ao_state1(ao * const me, QEvt const * const e) {
9      QState status_;
8      switch (e->sig) {
7          default: {
6              status_ = Q_SUPER(&QHsm_top);
5              break;
4          }
3      }
2      return status_;
1  }
```

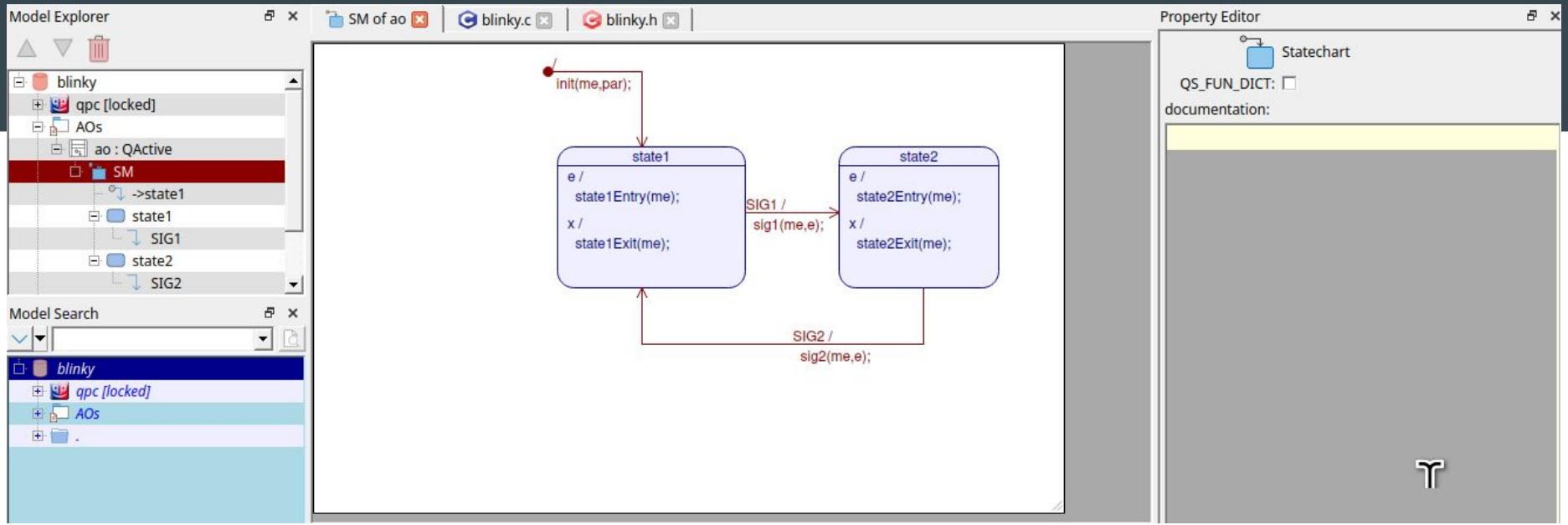



```
16 #ifndef LED_SM
15 #define LED_SM
14
13 //$declare${AOs::ao} vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
12
11 //${AOs::ao} .....
10 typedef struct {
9  // protected:
8      QActive super;
7  } ao;
6
5  // protected:
4  QState ao_initial(ao * const me, void const * const par);
3  QState ao_state1(ao * const me, QEvt const * const e);
2  //$enddecl${AOs::ao} ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
1
```



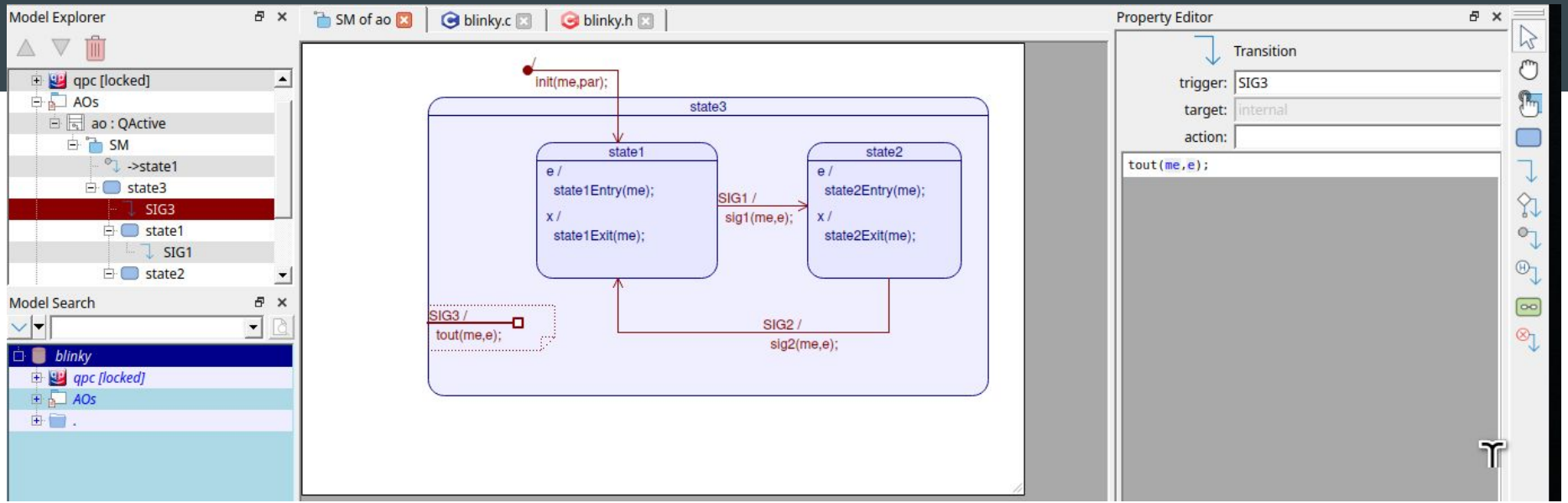
QM

- Conecto estados con signals/eventos
- Puedo usar entry/exit actions



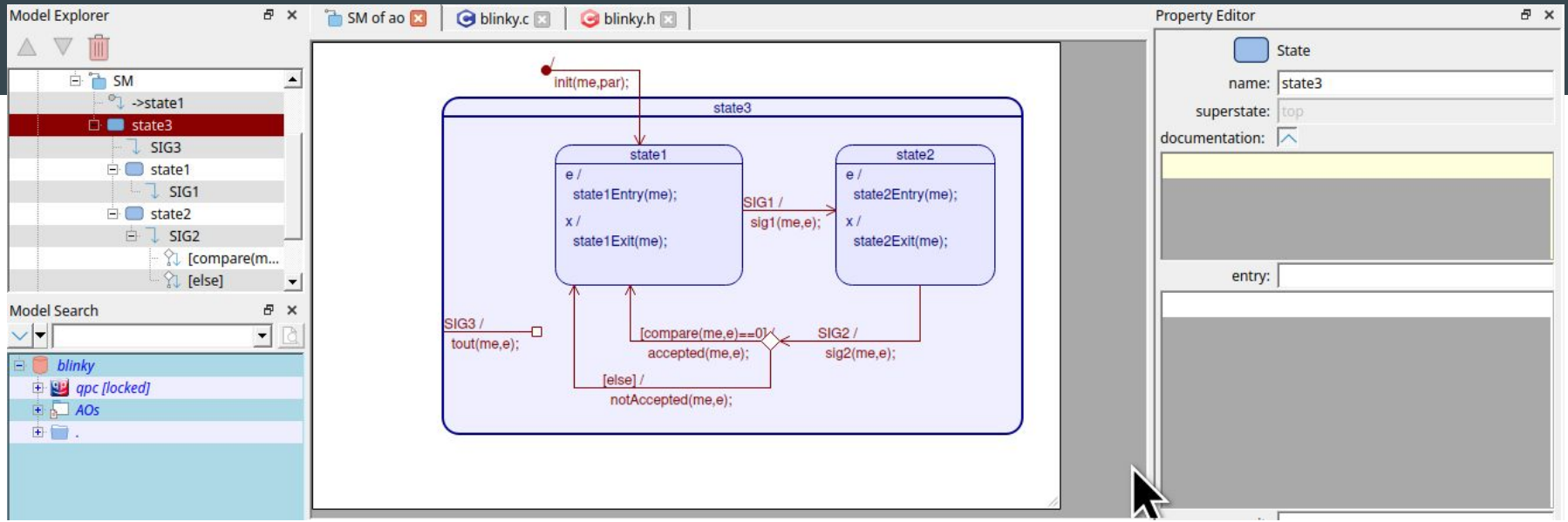
QM

- Estado jerarquico
- Evita tener que replicar la atención de las signals en todos los subestados
- Permite definir estados ‘macro’ que definen al sistema
- Es un buen lugar para manejar SM ortogonales relacionadas



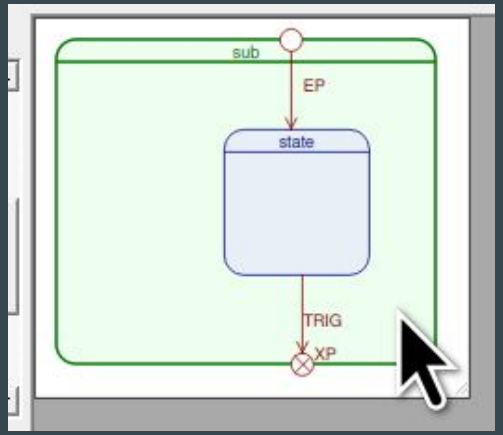
QM

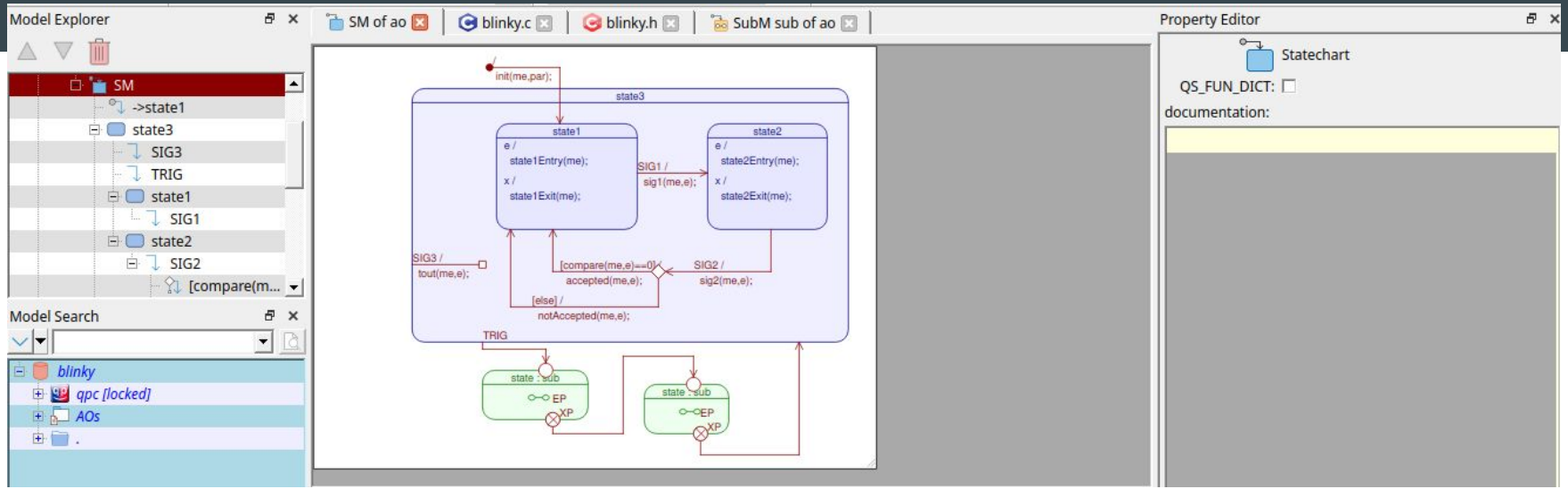
- Guard conditions:
  - Permiten tomar decisiones directamente en el evento
  - Es eficiente y útil para bifurcaciones simples



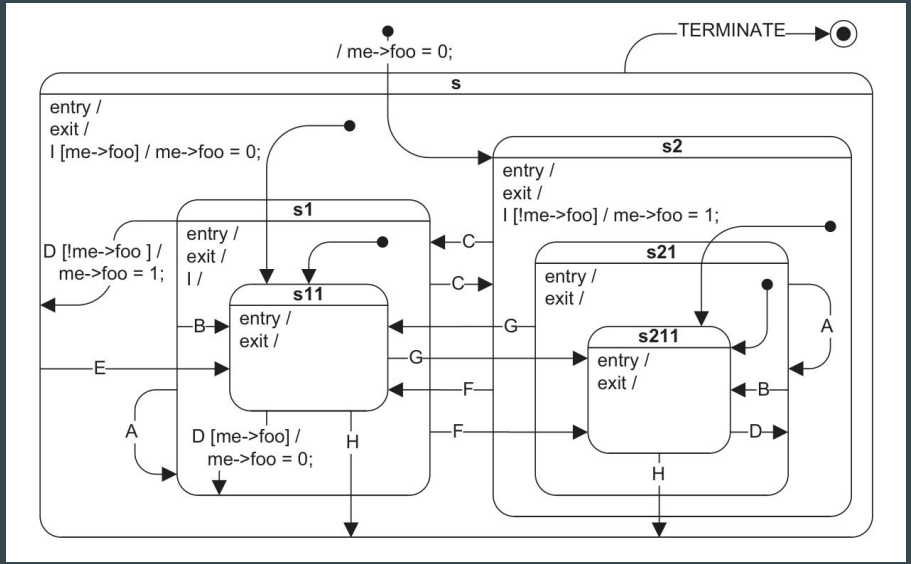
MSM

- Submachines-> reuse
- Solo funciona con MSM
- Se pueden instanciar varias submachines
- Se pueden exportar y usar desde otras SM
- Se puede armar una biblioteca de SM



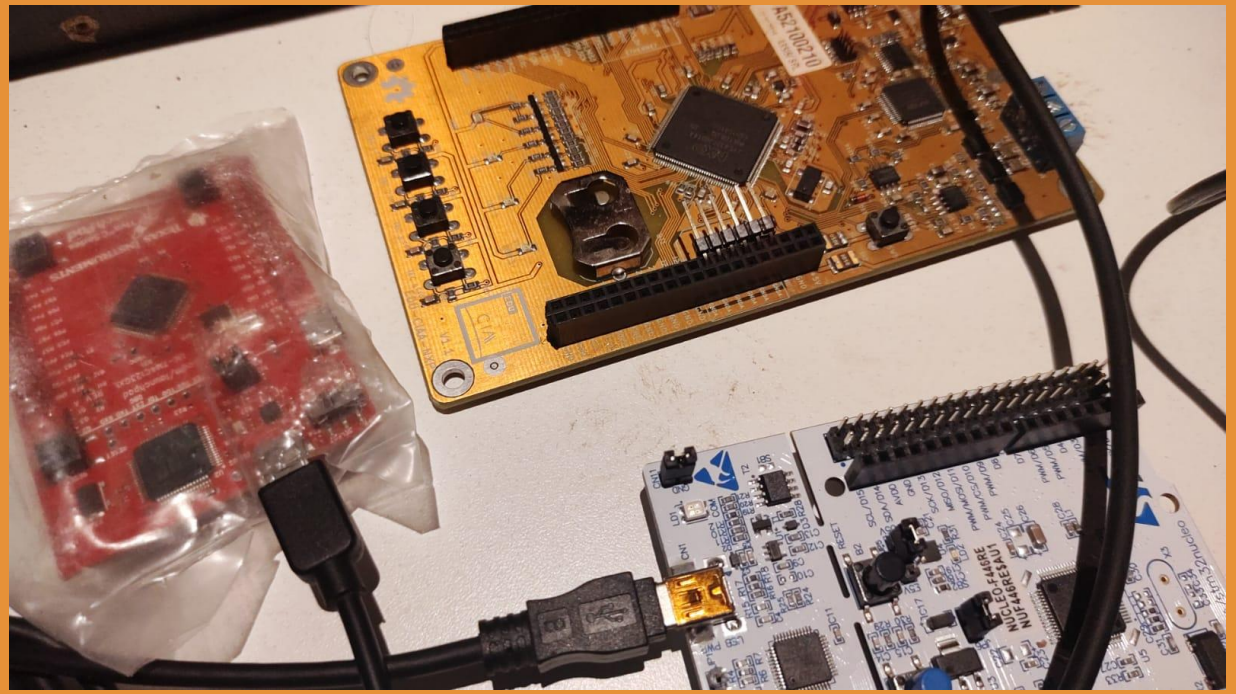


QM



- Combinando todas las opciones se pueden lograr SM muy complejas y versátiles
- Permite utilizar SM’s como si fueran funciones
- Si está bien implementado, permite ‘ver’ la logica del código.
- Ayuda a entender como funciona un determinado software
- Ayuda a detectar puntos ciegos.

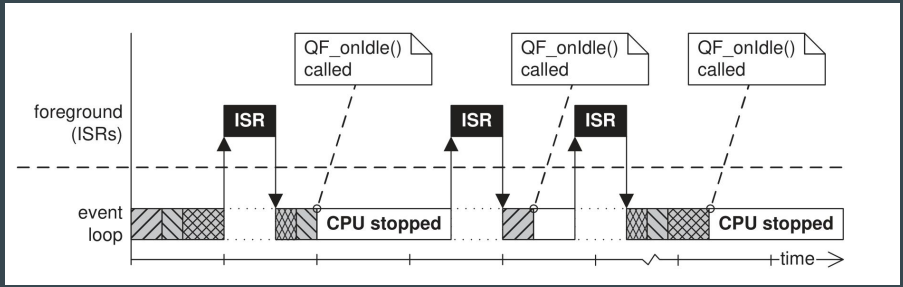
Hands on



- Probar posix
- Qspy
- Debug
- edu-ciaa o tmc
- Qspy
- Sin Qspy
- Modificar SM’s y probar

QP Kernels

Vanilla kernel cooperativo



- No hay conflicto para compartir recursos
- El kernel atiende las tareas que tienen eventos comenzando por la de mayor prioridad
- Atención con sleep-mode en la tarea idle
- Entre verificar que no quedan msg pendientes y entrar en modo sleep podria ocurrir una isr

QK - Full preemptive RTC kernel

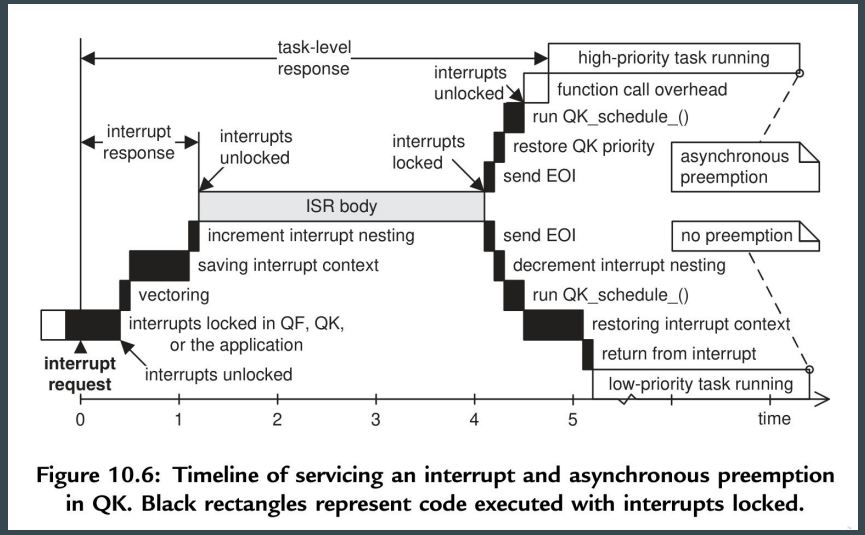
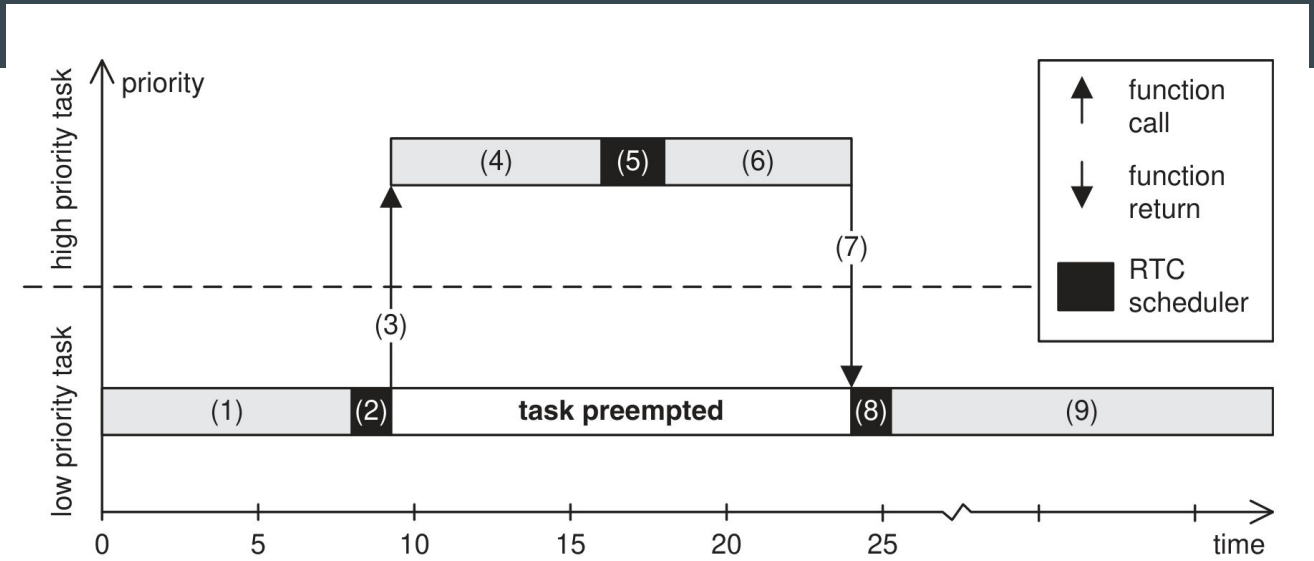


Figure 10.6: Timeline of servicing an interrupt and asynchronous preemption in QK. Black rectangles represent code executed with interrupts locked.

- Preemptivo NO bloqueate
- El cambio de contexto se puede hacer al terminar una ISR
- O también cuando un AO envia un mensaje a otra de mayor prioridad
- Full responsive
- Hay conflicto para compartir recursos
- El kernel atiende las tareas que tienen eventos comenzando por la de mayor prioridad
- No hay problemas con sleep mode

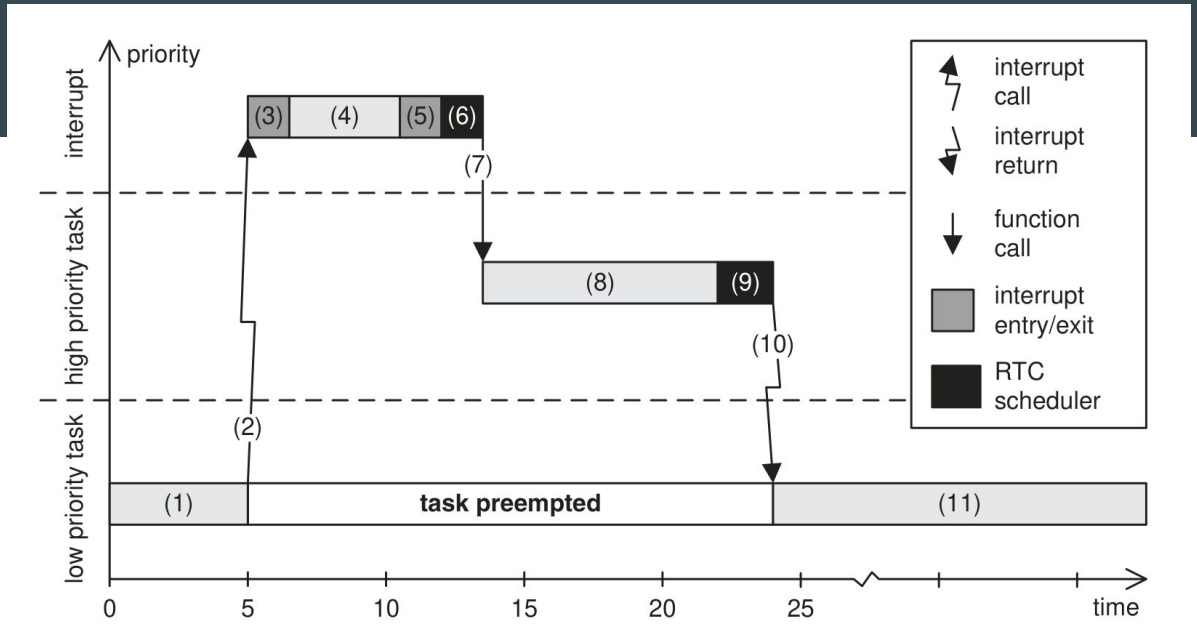
Preempcion sincronica

- Cuando una tarea postea un msg a una tarea de mayor prioridad, simplemente se ‘llama’ a dicha tarea
- Cuando la tarea de mayor prioridad termina, retorna a la de menor prioridad



Preemption asincronica

- Si salta una ISR que envía un mensaje a una tarea de mayor prioridad, el kernel 'llama' a dicha tarea.
- Luego que termina se retorna a la tarea que fue suspendida



Prioridades

Prioridades

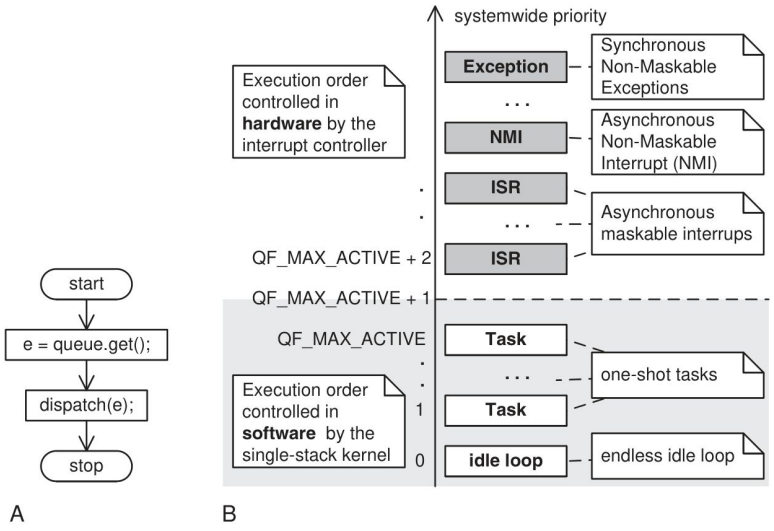
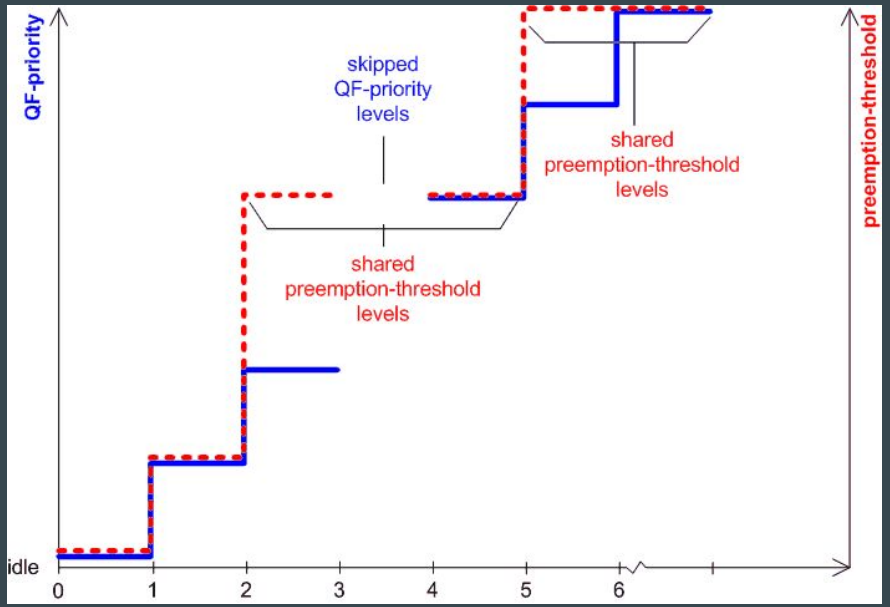


Figure 6.9: Systemwide priority of a single-stack kernel (A) and task structure (B).

- Un solo stack para todo el sistema, inclusive ISR's
- Hasta 32/64 prioridades max
- Permite implementar mutex con ceiling
  - Esto evita el efecto de inversión de prioridades

Grupos de prioridades



- Grupo de prioridades
- Siempre en orden creciente
- I.e.
  - 1 -> group 3
  - 2 -> group 3
  - 3 -> group 3
- Se comportan como si tuvieran la misma prioridad
  - Si hay mas de una tarea lista para correr, se turnan

Eventos

Eventos

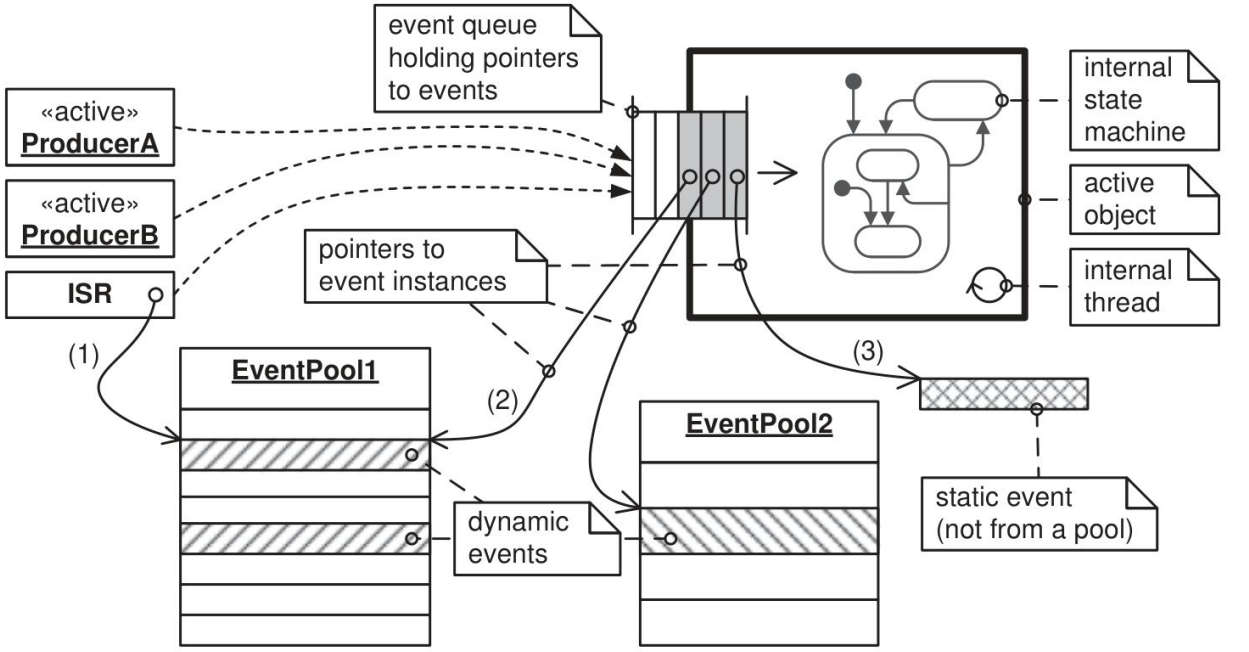


Figure 6.13: Passing events without copying them (zero-copy event delivery).

- Los eventos tienen una signal y un contador
- Opcionalmente se extienden para agregar puntero a otros datos
- Los eventos se pasan por referencia desde un eventPool y no por copia
- Los eventos se descartan cuando ya no se usan

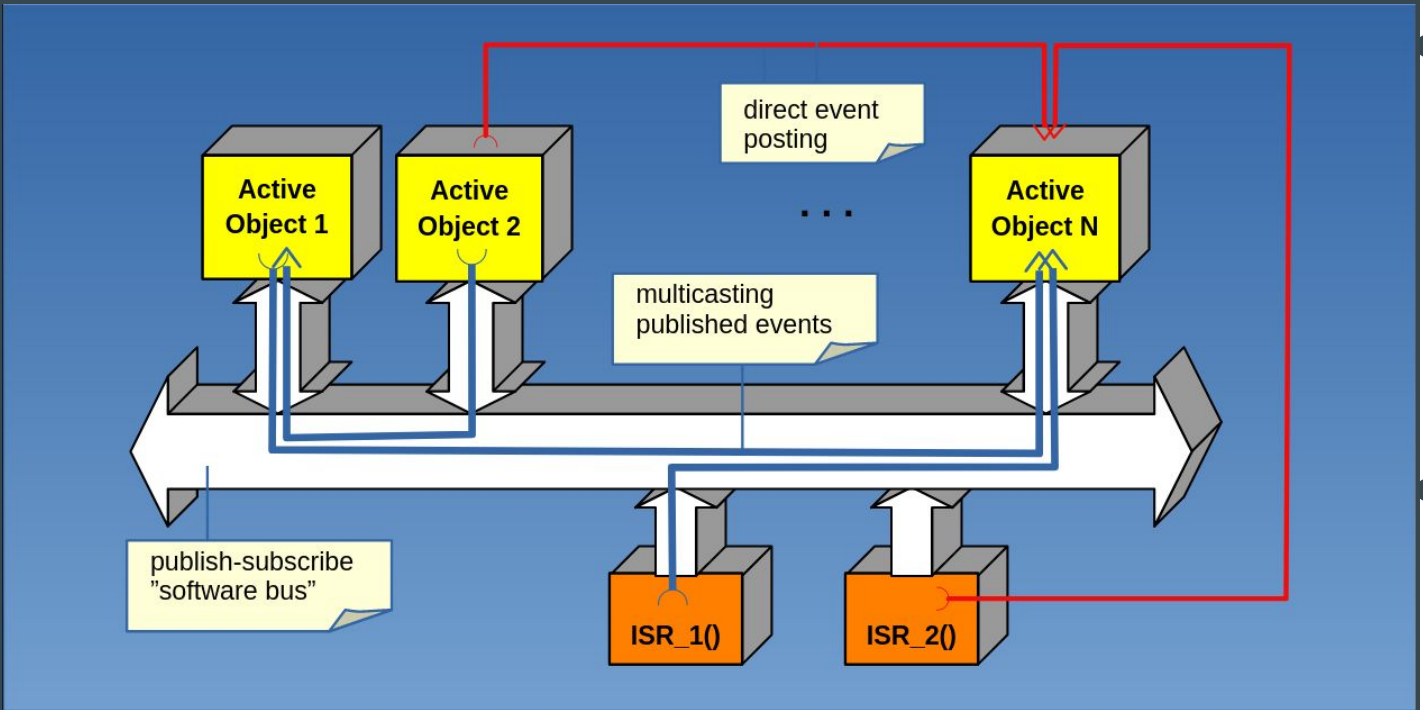
Hands on

- Enviar un evento con post desde un nodo a otro

```
7 void btn1Entry(btn * const me ,const void* par)
6 {
5     struct evtString_t *s = Q_NEW(struct evtString_t, PRINT_SIG);
4     strcpy(s->str, "btn1 \r\n");
3     QACTIVE_POST(uartAo() , &s->super, 0U);
2 }
```

Publish / Subscribe

Pub/Sub



- Los AO se registran en una lista que mantiene el framework
- Cuando un AO publica un evento, todos los registrados reciben una copia del mismo evento
- Permite desacoplar los nodos

Hands on

Mostrar en blinky como publicar una signal y que la reciban mas de un nodo a la vez

```
2  void btn1Entry(btn * const me ,const void* par)
1  {
18     struct evtString_t *s = Q_NEW(struct evtString_t, PRINT_SIG);
1      strcpy(s->str, "btn1 \r\n");
2      QACTIVE_PUBLISH( &s->super, &me->super );
3      //QACTIVE_POST(uartAo() , &s->super, 0U);
4  }
```

QSpy

Trace Qspy

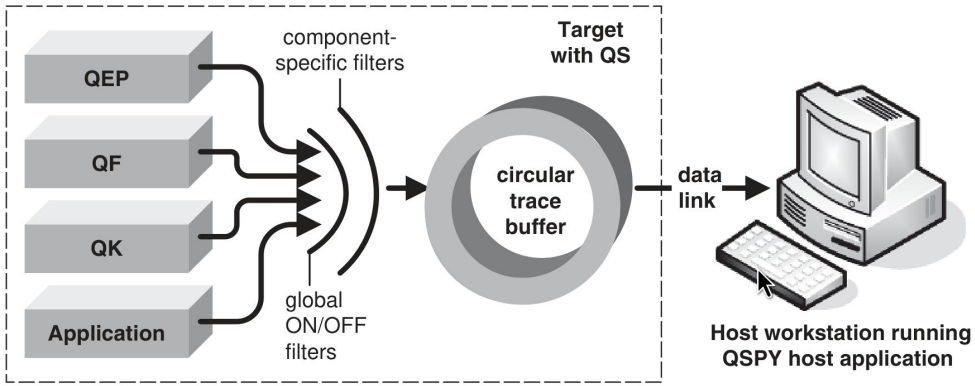
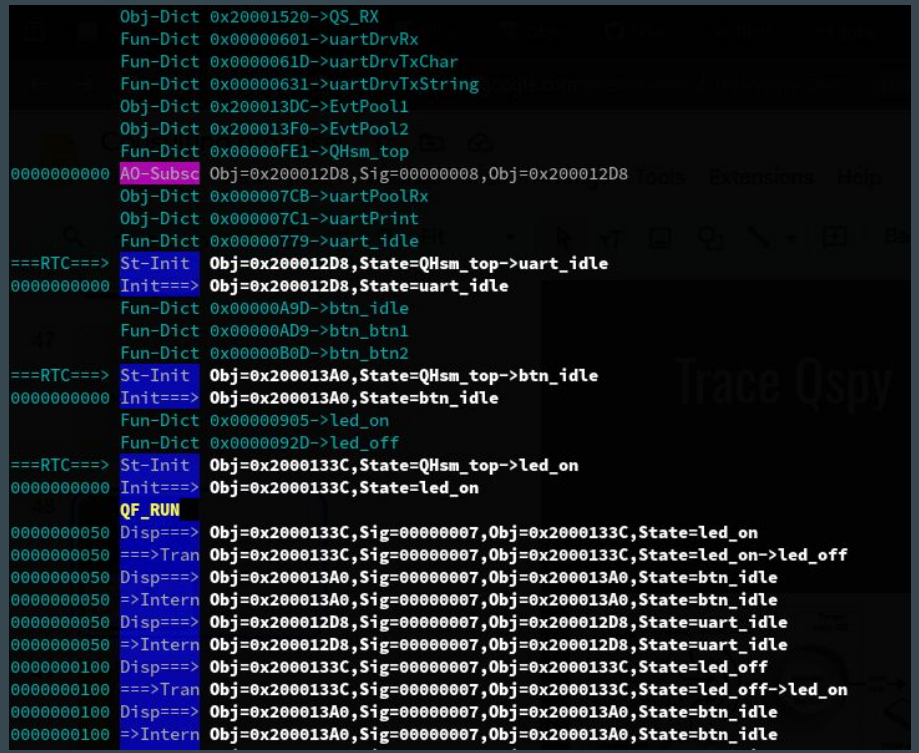


Figure 11.3: Structure of the QS target component.

- Integrado en el framework
- Utiliza un protocolo específico
- Almacena diccionarios en el host
- Permite monitorear todo el sistema
  - Estados
  - Memoria
  - Eventos
  - User data
- Cuenta con filtros globales por tipo de mensaje
- Filtrado fino para subtipos y datos específico

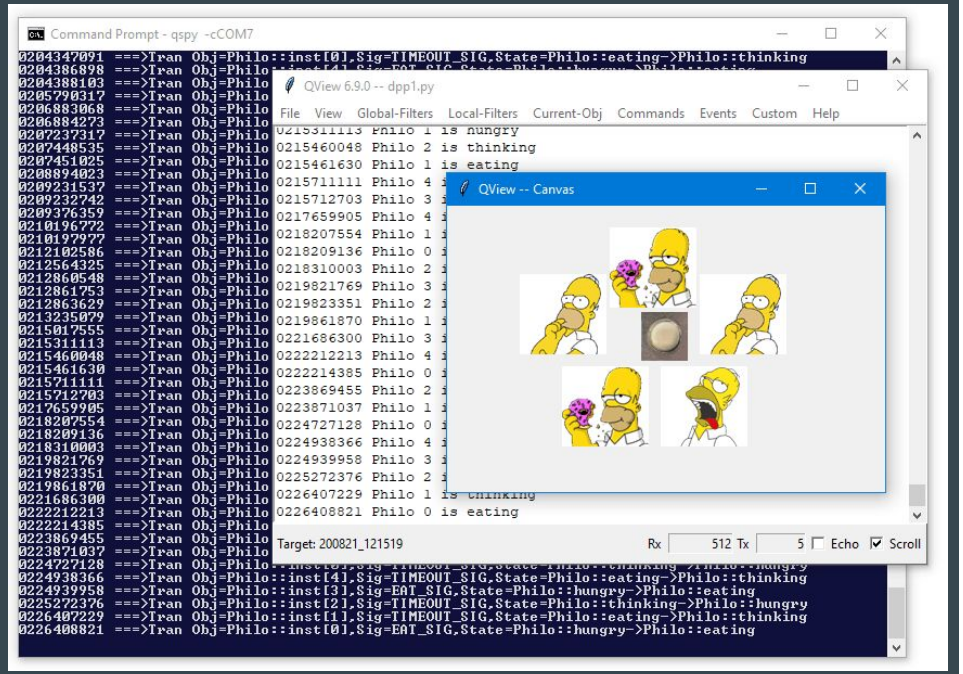
Qspy <-> UART



- Se puede conectar usando
  - Uart
  - Udp
  - Tcp
  - custom
- Permite loguear a un archivo de log
- También ofrece conexión con matlab y un simple ascii state transitions

QView

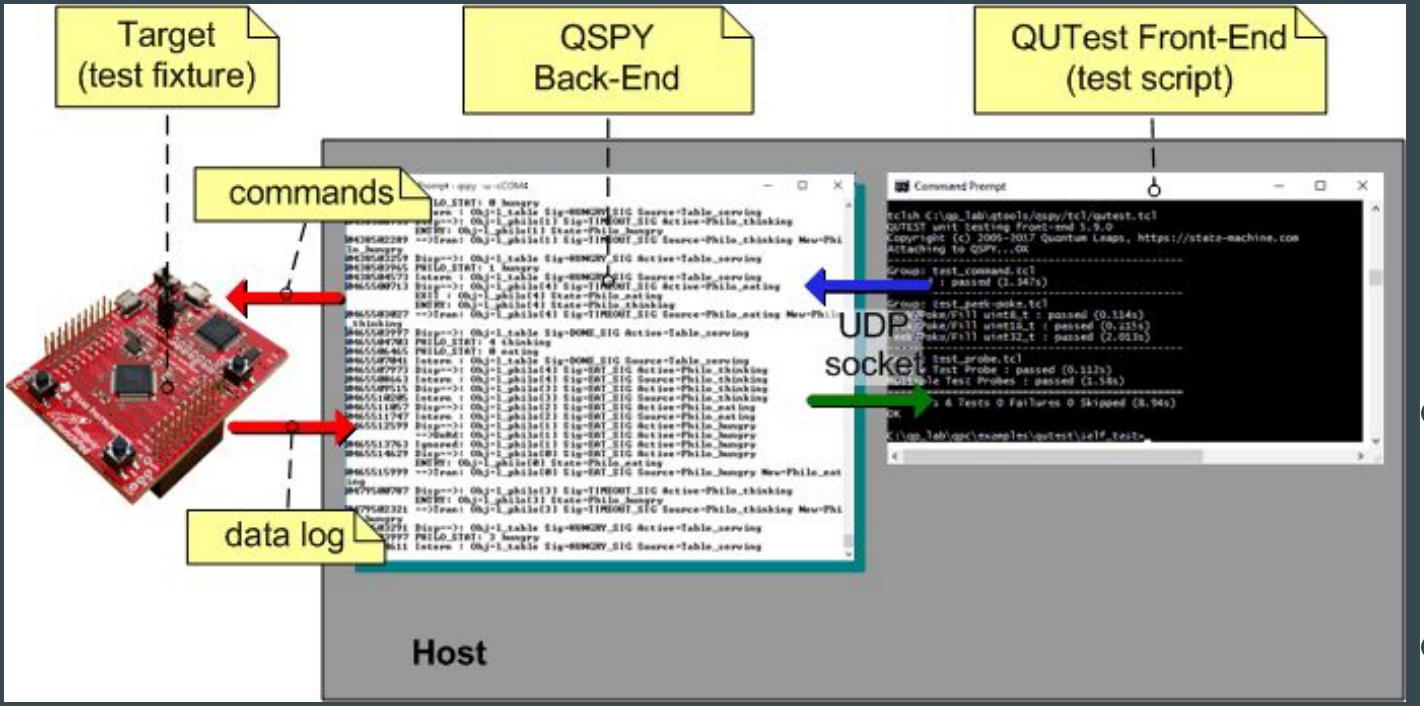
Qview



- Es una aplicación en python que se conecta a qspy por udp
- Qspy hace de bridge para enviar datos al embebido
- Esto permite enviar
  - Eventos
  - Filtros
  - Custom messages
  - Montorear
  - Etc
- También permite lanzar ventanas visuales que reaccionan con estados del sistema

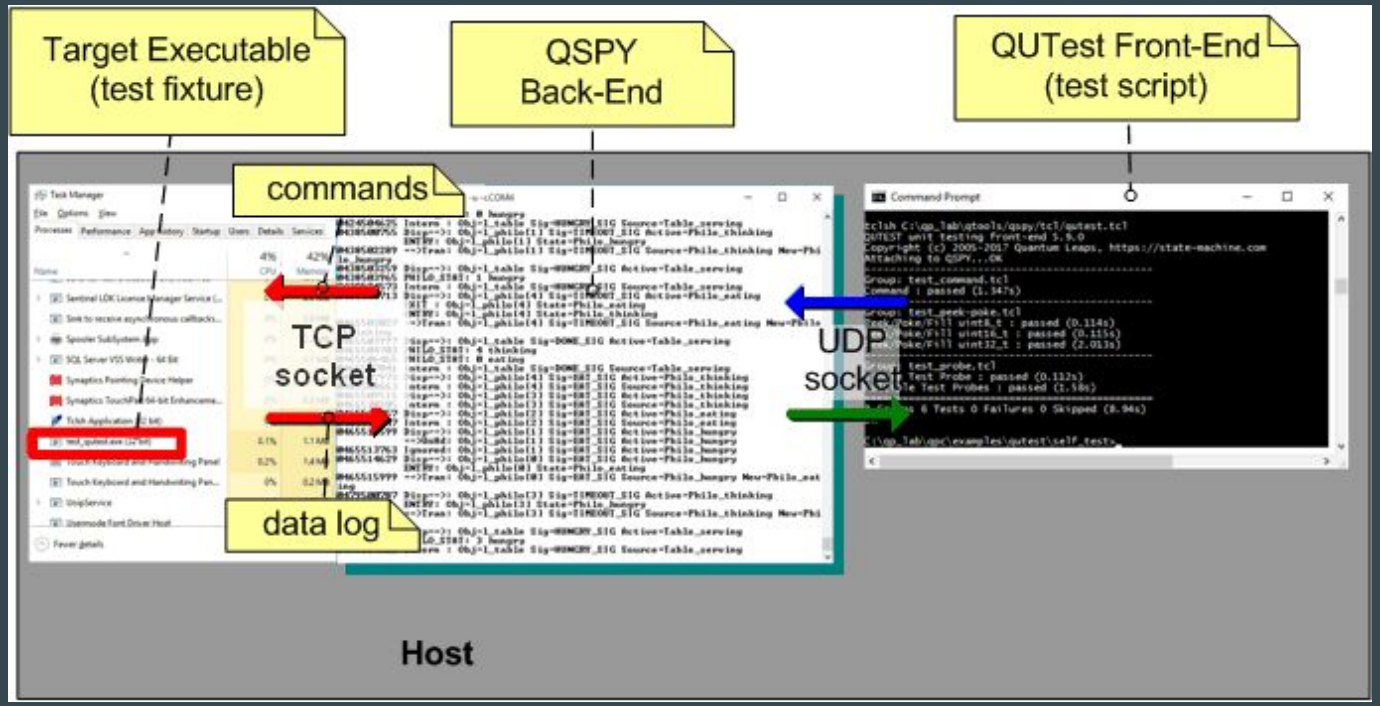
Qutest

Qutest



- Framework de testing que soporta
  - Unit tresting convencional
  - Testing funcional
  - Automatizaciones
- Conecta con Qspy y este con el dev
- Puede testear contra el host

Qutest



- En el caso del host, la misma aplicación se corre y testea en el host.
- Util para CI/CD

Conclusiones

Comparativa

Caracteristica	**Bloqueante**	**No bloqueante**
Complejidad de logica simple	simple	compleja
Recepcion de multiples estimulos	No escala bien	Escala sin problemas
Overhead para compartir recursos	Media	Baja o nula
Simple de razonar	Muy simple	Requiere un poco mas de analisis
Inversion de prioridad	Si	No

Preguntas

Referencias

https://www.youtube.com/watch?v=4DhFmL-6SDA