# Técnicas Digitales II
# Guía de Trabajos Prácticos
# Parte 1

R4054 – 2023

## Application Binary Interface (ABI) en ARM

Se escribirá un programa en lenguaje “assembler” (llamado *suma_abi.s*) que realice un llamado a una función de suma escrita en lenguaje “C” (llamado *suma_c.c*). Para ello, debe utilizarse ABI de ARM a fin de poder pasarle los argumentos necesarios. Dichos argumentos serán los 2 sumandos y la función en C debe devolver la suma.
Los registros a utilizar serán R0 y R1 para enviar los argumentos, y R0 para recibir el dato. Al finalizar el programa, este debe quedar en un bucle infinito sencillo. Se le dará el linker script mínimo para poder realizar el trabajo (*memmap.ld*).

## Arranque baremetal de un procesador ARM Cortex-A8

Realiceramos el inicio de un procesador de 32 bits, en particular un Cortex-A8. Para ello, debemos crear algunos archivos fuentes, a saber:

- reset_vector.s
- startup.s
- handlers.s

La secuencia de arranque nos debe llevar primero al fuente *reset_vector.s*, el cual se encargará de enviarnos directo a nuestras rutinas de inicialización, ubicadas en principio, en *startup.s*. Lo primero que debemos hacer es armar la tabla de manejadores de excepciones en la dirección correspondiente: **0x00000000.** El problema es que el procesador que estaremos emulando en QEMU, “nos deja” en la posición: **0x70010000**. Por lo tanto, debemos realizar una rutina de copiado para resolver este problema (y evitar tener un binario de varios cientos de MB). Lo que deberemos copiar estará alojado en el fuente *handlers.s*, dónde simplemente colocaremos para este ejercicio las etiquetas que referenciaran posteriormente a los manejadores de excepciones. Podemos nuevamente, cada rutina, dejarlas en un bucle infinito haciendo: **B .** (no se enamoren de este recurso, dado que más adelante se convertira en nuestro enemigo número 2. El 1 siempre, siempre, va a ser querer trabajar con la MMU, que tienen suerte y lo conoceran en TD3).

Verifique que realizó la copia de forma apropiada de al menos, 2 maneras: a) inspección visual utilizando la herramienta de debugging y los archivos auxiliares que generamos para trabajar (los .lst por ejemplo), b) realice un salto a la posición de memoria **0x000000XX** lo cual debe llevarle al manejador asociado (XX representa la posición para la excepción).

## Inicialización del controlador genérico de interrupciones y timertick

Tomaremos el ejercicio desarrollado en el punto anterior para agregarle más funcionalidades. Para ello, debemos realizar una serie de pasos.

**Primero:** debemos inicializar las pilas (Stack Pointers, SP) de cada modo de operación del procesador. Esto es necesario dado que sin una pila, el procesador no puede (en principio) "pushear" nada y por ende, dejar al procesador en un estado desconocido (aunque siga funcionando, la realidad es que uno no tiene la más pálida idea de si va a seguir haciendolo o no en el corto plazo). Si se preguntan si realmente es necesario, revisen el primer ejercicio en el cual hemos tocado el SP antes de poder hacer un call a la rutina en C, se imaginarán por qué, ¿no?.

Para esto, vamos a repetir por cada modo que poseemos en el procesador, las siguientes líneas de código:

```
MSR cpsr_c,#(MODO_DE_OPERACION | I_BIT | F_BIT)
LDR SP,=__posicion_stack_top__
```

Dónde I_BIT y F_BIT deben colocar un "1" en el campo apropiado, ¿por qué querremos hacer eso en esta instancia?.

**Segundo:** primero vamos a inicializar el Controlador Genérico de Interrupciones (a.k.a. **GIC**) y luego el **Timertick** (o bueno, Timer 0 para el procesador en cuestión). Las rutinas de inicialización de estos dispositivos serán brindadas por la cátedra, tanto los fuentes (.c) como sus encabezados (.h). Usted debe escribir una rutina en C (que llamaremos **board_init**), que colecte las 2 rutinas en cuestión y las ejecute (además de incluirlas en su esquema de desarrollo y compilación). Finalmente, al retornar de board_init, debe habilitar las interrupciones (en startup.s, si). Si todo marcha bien, al ejecutar el binario con QEMU el Program Counter (PC) debe terminar frito, digo fijo, en el **B .** de la rutina de manejo de excepciones de IRQs.

**Tercero:** bueno, ahora si, vamos a armar un manejador de atención de interrupciones (IRQ) más serio (no el B . que pusimos en el ejercicio anterior). Este manejador más elegante, debe realizar algunas cosas, a saber:

1) tiene que corregir el Link Register (LR) que está apuntando a cualquier lado,

2) resguardar en la pila de IRQ (ven?, para esto también inicializamos los SP de cada modo de operación) el contexto actual, es decir, que debemos pushear desde R0 a R12, el LR, luego el SP y finalmente el SPSR.

3) ahora si nuestro manejador puede trabajar (leemos rápido y sin parar): debe identificar quién generó la interrupción, atender dicha interrupción, avisarle al dispositivo que ya atendió su solicitud, avisarle al GIC que ya atendimos la interrupción del dispositivo que nos interrumpió, y volver (ahora si podemos respirar),

4) Este paso demuestra lo crítico de **resguardar** y **recuperar** el contexto: debemos recuperar el contexto para poder volver correctamente. Es decir, recuperar de la pila en el órden inverso al que pusheamos: SPSR, el SP, el LR, y los registros R12 a R0. Si algo de esto falla, al momento en que el PC toma el valor de LR, podremos terminar con altas probabilidades con el procesador reiniciandose o "colgado" en algún lugar de memoria.

El paso 3) vamos a realizarlo en una rutina escrita en C.

Finalmente, nuestro startup ahora si está listo para terminar en una pequeña rutina que llamaremos *idle*:

```
idle:
    WFI
    B idle
```

Esto es posible dado que WFI deja al procesador suspendido hasta que llega una interrupción y podremos salir. De esta manera, el procesador pasa de estar trabajando al 100% a solamente cuando tenga que atender la interrupción. Si te preguntas por que querríamos algo así, pensá en tu celular y cuanto le duraría la batería si tuvieras el procesador del mismo funcionando al 100 % constantemente.

## Sistema multitareas básico bajo nivel

Ahora vamos a hacer algo divertido con nuestro pequeño sistema (por que que sentido tiene haber hecho lo anterior si no vamos a complejizarlo aunque sea un poco). Vamos agregarle algunas "tareas", y a aprovechar que tenemos un timertick interrupiendo periódicamente para usarlo para marcar los tiempos en que estas tareas van a ir conmutando entre ellas.

Las tareas no son más que rutinas que se ejecutan, y están administradas por "alguien". Ese alguien se llama "Scheduler" (bastante original, ¿no?), y si se preguntan: ¿dónde va a estar y que va a ser?, entonces comenzaron con las preguntas incorrectas. Sino, vuelvan a revisar el enunciado hasta este punto. De verdad, relean.

El Scheduler será, nada más, ni nada menos, ni nada más, ni nada menos que el Timertick, lógicamente. Cuando nos referimos a: es el Timertick, nos referimos a que es el manejador de interrupciones del Timertick. Cuando en el ejercicio anterior agregamos la rutina en C que nos identifica quién realizó la interrupción y lo único que hicimos fue atender la solicitud del Timer 0, esa única línea puede ser interpretada como el manejador en cuestión. Por lo tanto, ahora no solo haremos eso sino que vamos a agregar algunas cosas. Por lo tanto, lo ideal sería que de ese punto en realidad llamemos a (como ya se la ven venir): una rutina que podemos llamar, a falta de originalidad e inventiva: scheduler().

Este Scheduler va a tomar la decisión de si debemos cambiar de tarea o no, en función de un contador que llamaremos **ticks**. Por cada vez que el timer interrumpa, debemos incrementar dicho contador. Cada tarea tendrá asignada una determinada cantidad de **ticks**, que cuando se cumplan el Scheduler realizará lo necesario para que se efectúe el cambio de tarea.

Nuestro pequeño sistema operativo, que en definitiva, es lo que están construyendo constará de cuatro tareas, de las cuales una, será la que llamamos "idle" (u osciosa). La tarea idle es la tarea que se ejecuta cuando absolutamente nada más se está ejecutando en el sistema operativo.

Por ejemplo, definimos un Tiempo de Ejecución Total (**TET**) de **20 ticks**. Dentro de ese TET se deben ejecutar las 4 tareas. Si le asignamos **3 ticks** a cada tarea (excepto la idle), entonces a la idle le que dan: 20 – 3*3 = **11 ticks**.

Las tareas estarán escritas como rutinas en C, y todas deben poseer la instrucción WFI (nada de B .).

La Tarea 1 va a realizar el incremento de una variable local y de otra global. La Tarea 2, va a realizar el decremento de otra variable local y de otra global. La Tarea 3, como es la tercera en discordia, va a decrementar la variable global que la Tarea 1 incrementa, y a incrementar la variable global que la Tarea 2 decrementa. Las proporciones de los incrementos y decrementos, los definen ustedes, que les permita visualizar esta lucha.

Los ticks que asignarán para cada tarea serán:

- Tarea 1: 8 Ticks,
- Tarea 2: 12 Ticks,
- Tarea 3: 5 Ticks,
- Tarea Idle: calcular
- TET: 30 Ticks.

**A tener en cuenta:** conmutar de tarea implica que cuando querramos volver a la tarea que interrumpimos en algun momento, las condiciones deben estar dadas: el LR, el SPSR y los registros, deben ser los correctos. Por lo tanto, debe implementar un mecanismo que le permita resguardar el contexto de cada tarea para poder asegurarse volver apropiadamente.