

Manual de Laboratorio de
Estructura de Sistemas Operativos (ESO)

Fase 1

Garlic_OS (versión 1.0)

Sistema Operativo para NDS

3º curso de Grado en Informática
Universitat Rovira i Virgili

Profesor responsable:
Santiago Romaní (santiago.romani@urv.cat)

Índice de contenido

1 Introducción.....	5
1.1 Objetivos de la práctica.....	5
1.2 Resumen de la práctica.....	5
1.3 Conocimientos previos.....	5
1.4 Organización del trabajo.....	6
1.5 Sistema de gestión de versiones (git).....	8
1.6 Calendario de trabajo.....	11
1.7 Evaluación de la práctica.....	12
 2 Estructura de la plataforma de trabajo.....	 14
2.1 Modos de ejecución del procesador ARM9.....	14
2.2 Gestión de excepciones ARM en la plataforma NDS.....	17
2.3 Gestión de interrupciones mediante libnds9.....	20
2.4 Ocupación de las zonas de memoria NDS.....	24
2.4.1 Data Tightly Coupled Memory.....	25
2.4.2 Instruction Tightly Coupled Memory.....	26
2.4.3 Main Memory.....	27
2.4.4 BIOS ARM9.....	29
2.4.5 Zona de trabajo para el ARM7.....	30
2.4.6 BIOS ARM7.....	31
2.4.7 GBA ROM.....	32
2.5 Comunicación entre los procesadores ARM7 y ARM9.....	33
 3 Especificaciones generales de la práctica.....	 35
3.1 Requisitos generales.....	35

3.2 Requisitos restringidos para la primera fase.....	39
3.3 Organización del micro-kernel empotrado en la NDS.....	40
3.4 Proyectos involucrados en la fase 1.....	42
3.5 El interfaz de funciones para los programas (API).....	43
3.6 Estructura de un programa para GARLIC.....	46
3.7 Estructuras de datos de GARLIC.....	50
3.8 Estructura de funciones y rutinas de GARLIC.....	52
3.9 Distribución en memoria de los componentes del sistema. .	55
3.9.1 Data Tightly Coupled Memory.....	56
3.9.2 Instruction Tightly Coupled Memory.....	57
3.9.3 Main Memory.....	58

4 Tareas de gestión del procesador (progP).....59

4.1 Rutina principal de gestión de interrupciones de GARLIC....	59
4.2 Rutina de Servicio de la Interrupción IRQ_VBL.....	61
4.3 Rutinas de salvar y restaurar contexto.....	62
4.4 Rutina de crear proceso.....	66
4.5 Otras rutinas de gestión de procesos.....	68
4.6 Programa principal para progP.....	69

5 Tareas de gestión de la memoria (progM).....75

5.1 Formato de fichero ejecutable ELF.....	75
5.2 Tipos de datos ELF.....	76
5.3 Cabecera de un fichero ELF.....	77
5.4 Tabla de segmentos.....	79
5.5 El problema de la reubicación de direcciones.....	81
5.6 Tabla de secciones.....	84

5.7 Estructura de los reubicadores.....	86
5.8 Algoritmo de carga y reubicación.....	88
5.9 Programa principal para progM.....	92
6 Tareas de gestión de los gráficos (progG).....	96
6.1 La fuente de letras Garlic.....	96
6.2 Inicialización del entorno gráfico.....	97
6.3 Dibujo de los marcos de las ventanas.....	98
6.4 La escritura de texto.....	100
6.5 Programa principal para progG.....	104
7 Tareas de gestión de teclado (progT).....	108
8 Tareas de integración del código (master).....	112
9 Consejos para la depuración del código.....	114

1 Introducción

1.1 Objetivos de la práctica

El **objetivo principal** de cualquier práctica es consolidar los conceptos teóricos mediante su aplicación a la resolución de un problema real. Concretamente, en esta práctica de Estructura de Sistemas Operativos se implementará un sistema operativo pedagógico capaz de realizar **carga dinámica de programas en memoria y ejecución concurrente de los procesos correspondientes**.

Aparte del marco teórico de la asignatura, será necesario profundizar en los conocimientos propios de la plataforma *hardware* sobre la que se tiene que ejecutar este sistema operativo, es decir, procesadores ARM, zonas de memoria, procesadores gráficos y otros controladores de E/S de la **NDS**.

Como **objetivo metodológico** se establece un modelo de **trabajo en equipo** basado en un sistema de control de versiones (**Git**), en el que los componentes de cada grupo de prácticas tienen que distribuirse las tareas y aprender a fusionar las distintas partes del proyecto en un programa único.

1.2 Resumen de la práctica

La práctica consistirá en realizar un *microkernel* de sistema operativo para la plataforma NDS que permita cargar y ejecutar concurrentemente hasta 15 procesos de usuario, más un proceso específico de control del propio sistema operativo.

Dichos procesos podrán realizar cálculos y escribir información en una ventana de texto dedicada al proceso (16 ventanas de 24 filas por 32 columnas cada una). También podrán bloquearse durante un cierto tiempo.

Adicionalmente, se prevé la posibilidad de realizar entrada de información (lectura de texto) por medio de un controlador de entrada/salida que permita simular un teclado.

1.3 Conocimientos previos

Para la realización de esta práctica es necesario tener bien adquiridos los conceptos impartidos en las asignaturas de *Fundamentos de Programación*,

Fundamentos de Computadores, Computadores y Fundamentos de Sistemas Operativos. Concretamente, se requiere un buen nivel de:

- programación en general (algorítmica)
- programación en lenguaje C
- programación en lenguaje ensamblador GAS/ARM
- programación a bajo nivel de la plataforma NDS
- programación de rutinas de servicio de interrupciones (RSIs)
- uso del entorno de desarrollo devkitPro + DeSmuMe
- uso de las funciones de librería *libnds*
- uso del sistema de control de versiones Git

1.4 Organización del trabajo

Debido a la complejidad de la práctica, es fundamental distribuir bien las diferentes tareas a realizar entre los miembros del grupo de prácticas. Para ello se proponen 4 roles:

- Gestión del procesador (**progP**): gestionar la ejecución concurrente de los procesos (cambio de contexto de los procesos, colas de procesos preparados y bloqueados, creación de nuevos procesos, etc.)
- Gestión de la memoria (**progM**): leer un fichero ejecutable en formato .ELF del disco (memoria ROM), cargar sus segmentos en memoria RAM, reubicar sus referencias y gestionar la memoria libre y ocupada.
- Gestión de los gráficos (**progG**): controlar un entorno gráfico de 16 ventanas de 24 filas x 32 columnas de caracteres cada una, permitiendo a los procesos enviar texto a partir de la posición actual del cursor o bien en una posición concreta de su ventana.
- Gestión del teclado (**progT**): permitir la introducción de *strings* de texto a través de una interfaz de teclado virtual, de modo que los procesos puedan leer información introducida por el usuario.

Los grupos de prácticas podrán ser de 1, 2, 3 o 4 personas. Lógicamente, en el caso de grupos de 4 personas los roles se asignarán uno por persona.

En el caso de grupos de menos de 4 personas, se permitirá la entrega de un sistema operativo que incorpore solo las funcionalidades de los roles asignados a las personas que formen el grupo, es decir, se permitirá la entrega de **prácticas parciales**, tanto en la primera como en la segunda fase, y tanto en primera como en segunda convocatoria.

Sin embargo, con el fin de promover la realización del trabajo en equipo, la nota máxima que se podrá obtener vendrá regulada por el número de **roles integrados**. Esto significa que habrá que fusionar el código de las ramas de cada programador sobre la rama **master**, obteniendo **un único programa** que integre todas las funcionalidades de los distintos roles; se considerará como versión definitiva de la práctica al programa contenido en el último *commit* de la rama **master**, subido al servidor **Git** del departamento dentro de las fechas límite establecidas.

A continuación se expone una tabla con la nota máxima (sobre 10) que se podrá llegar a obtener según el número de roles integrados, además de potenciar la nota del rol **progT** con dos puntos adicionales para compensar la mayor dificultad de sus tareas, aunque se requiere fusionarse con **progP** para poder optar a dichos puntos adicionales:

num. roles integrados	progP, progM, progG	progT
1 rol	max. 7 puntos	max. 7 puntos
2 roles	max. 9 puntos	max. 11 puntos
3 roles	max. 10 puntos	max. 12 puntos
4 roles	max. 11 puntos	max. 13 puntos

Sin embargo, **no se contabilizarán los roles acabados pero no integrados**. Es decir, si se presentan 3 roles acabados, 2 fusionados y uno no, los alumnos que hayan fusionado sus roles podrán llegar a 9 puntos, mientras que el alumno que presente su rol no integrado con el resto del proyecto solo podrá llegar a 7 puntos.

Los grupos de prácticas compuestos por menos de 4 alumnos también pueden optar a máximos superiores a su correspondiente número de roles, si completan (e integran) roles adicionales. Por ejemplo, un grupo de 2 alumnos que presente 3 roles integrados puede optar al máximo de 10 puntos.

Así mismo, si alguno de los componentes de un grupo no cumple con su trabajo, el resto de los componentes podrán asumir sus tareas, con lo cual podrán optar a la nota máxima correspondiente de los roles integrados, quedando sin nota el alumno que no trabaje.

1.5 Sistema de gestión de versiones (git)

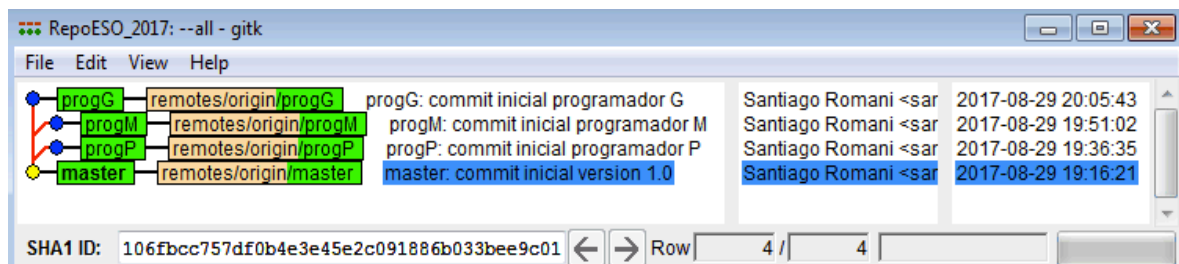
En la práctica de la asignatura de *Computadores* ya se introdujo la utilización del sistema de control de versiones **Git**. En esta práctica se continuará con el uso de este sistema para reforzar su manejo, especialmente con el **trabajo con ramas**.

Para entender cómo funcionan las ramas se recomienda el manual de *git* disponible en el espacio *Moodle* de la asignatura. A continuación se repasan los conceptos más relevantes.

```
$ git checkout nombre_rama
```

El comando `git checkout` permite, entre otras cosas, cambiar rápidamente de rama. Los nombres de las ramas serán **progP**, **progM**, **progG**, **progT** y **master**.

Estas ramas (excepto **progT**) ya están creadas en el repositorio del proyecto ubicado en el servidor del departamento. También se proporciona una copia en el espacio *Moodle* de la asignatura (archivo **Garlic_OS_1.zip**), cuya estructura de *commits* iniciales es la siguiente:



El programador de entrada por teclado **progT** tendrá que crear su propia rama con uno de los siguientes comandos:

```
$ git checkout -b progT
$ git branch progT
```

Hay que recordar que el cambio de rama implica el **cambio de los archivos del directorio** de trabajo, puesto que dichos archivos corresponden al contenido del *commit* seleccionado actualmente.

De hecho, **Git** almacena toda la **historia del proyecto** en su **base de datos local** (directorio oculto *.git*), lo cual permite a los programadores acceder fácilmente a **versiones anteriores** del proyecto. Además, las ramas permiten trabajar con **versiones paralelas** del proyecto.

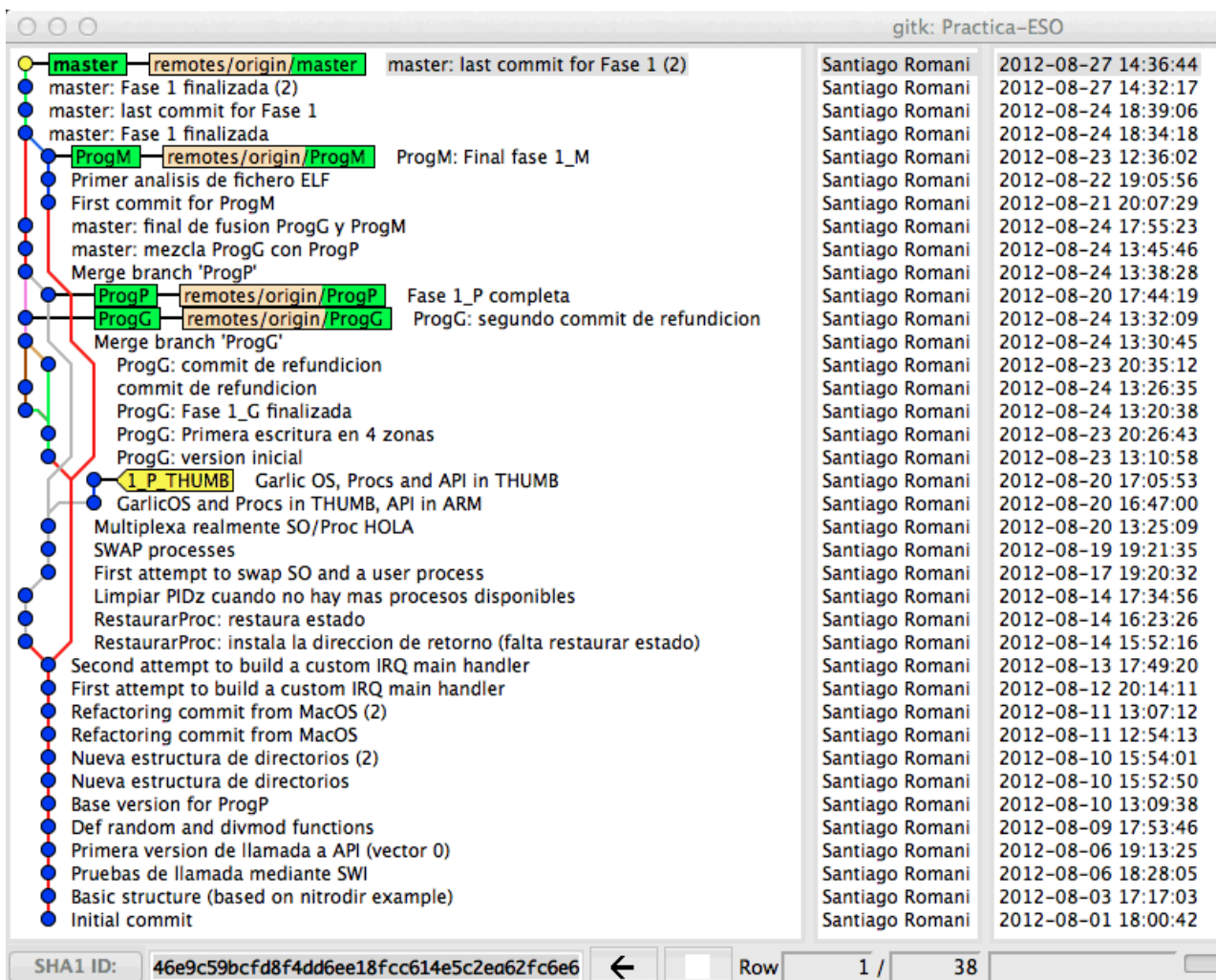
Cuando se tengan que fusionar los trozos de código de los diversos programadores, habrá que utilizar el siguiente comando:

```
$ git merge --no-ff nombre_rama
```

Con el comando `git merge` se fusiona el contenido de la rama actual (típicamente la rama `master`) con la rama especificada como argumento. Por ejemplo, `git merge --no-ff progP` tratará de fusionar el contenido de los ficheros de la rama actual con los ficheros de la rama `progP`.

Sin embargo, puede ocurrir que *Git* **no** pueda realizar la fusión de los ficheros automáticamente, lo cual comunicará por pantalla. En este caso, los programadores tendrán que revisar manualmente todos los ficheros en los que se hayan producido conflictos, con el fin de establecer manualmente cual será el contenido definitivo de cada fichero.

El siguiente gráfico muestra un ejemplo de realización de la fase 1 de la práctica:



Como se puede observar, este proyecto empezó con el *commit* inicial el día 1 y terminó el día 27 de Agosto de 2012. Aunque al principio la evolución es lineal, el día 14 aparecen bifurcaciones que corresponden a las variaciones del proyecto debidas a las distintas tareas de los roles. Al final, sin embargo, todas las bifurcaciones convergen en la rama **master**.

Las ramas se bifurcan y se fusionan diversas veces, pero el sistema *Git* **no** indica a qué rama perteneció cada *commit*, simplemente señala el último *commit* de la rama. De hecho, **una rama solo es un puntero a un *commit***. En la figura anterior se pueden observar los punteros de las ramas **progP**, **progM**, **progG**, **master**.

Para que los programadores y el profesor de prácticas puedan realizar un correcto seguimiento de los *commits* de cada rol, se pide que se añada un prefijo indicativo del rol en el mensaje de cada *commit*. Por ejemplo, "progG: Primera escritura en cuatro ventana" es un mensaje marcado con el prefijo "**progG:** ", que indica en qué rama se ha realizado el *commit*.

Por último, se recomienda trabajar con *Git* de forma incremental. Es decir, **no** realizar un único *commit* (o dos) con la versión final del rol, sino que hay que ir guardando en sucesivos *commits* los avances que se vayan haciendo al proyecto. Por ejemplo, se puede realizar un *commit* por cada rutina finalizada, aunque también se pueden hacer *commits* de fragmentos de rutinas. El objetivo último de un sistema de control de versiones es permitir registrar la evolución del proyecto, de modo que sea posible volver a versiones anteriores en caso de necesidad, por ejemplo, para comprobar en qué versión se introdujo un determinado *bug*.

1.6 Calendario de trabajo

El calendario de trabajo típico para 14 semanas es el siguiente, aunque puede variar en función de la distribución de días festivos en cada curso académico concreto:

- sesión 1: explicación general de la práctica
- sesión 2: explicación fase 1 (tareas de cada rol)
- sesión 3: repaso del sistema git
- sesión 4: trabajo fase 1
- sesión 5: trabajo fase 1
- sesión 6: trabajo fase 1
- sesión 7: evaluación fase 1
- sesión 8: explicación fase 2 (tareas de cada rol)
- sesión 9: trabajo fase 2
- sesión 10: trabajo fase 2
- sesión 11: trabajo fase 2
- sesión 12: trabajo fase 2
- sesión 13: evaluación fase 2
- sesión 14: evaluación fase 2 (continuación)

En las **semanas de explicación**, el profesor de prácticas explica qué hay que hacer en cada fase.

En las **semanas de trabajo**, los alumnos realizan la práctica en horas de laboratorio (y también en casa), y pueden preguntar dudas al profesor.

En las **semanas de evaluación**, el profesor realizará una entrevista individual a los miembros de los grupos de prácticas, con el fin de evaluar los conocimientos de cada alumno en particular.

Cada fase de la práctica también se podrá presentar antes de la correspondiente semana de evaluación, aunque la tiene que presentar todo el grupo al completo o, al menos, todos los roles que se pretendan integrar.

Para la segunda convocatoria, los alumnos que no hayan aprobado o no se hayan presentado a la evaluación de alguna de las dos fases de la práctica se tendrán que presentar en una fecha determinada (se concretará por *Moodle*) para evaluarse de las fases pendientes.

1.7 Evaluación de la práctica

En este apartado se listan los criterios generales para evaluar las dos fases de la práctica:

- La nota de la práctica es individual por alumno, y vendrá determinada por las respuestas a las preguntas del profesor realizadas durante la entrevista.
- La entrega del trabajo consiste en los *commits* enviados al servidor **Git** del departamento DEIM, dentro de las fechas límite correspondientes.
- No se requiere presentar informe.
- Las entregas de las dos fases de la práctica en primera convocatoria son **independientes**, es decir, no se evaluará la fase 1 en el momento de evaluar la fase 2.
- La nota máxima que podrá obtener cada alumno vendrá determinada por el número de roles integrados y por el tipo de rol asumido, según la tabla expuesta en el apartado 1.4 de este manual.
- Además de las tareas propias de implementación del sistema operativo, también será **requisito indispensable para aprobar** la presentación de un **programa específico** para el sistema operativo implementado; dicho programa debe ser sencillo pero debe servir para comprobar las funcionalidades implementadas. Además, cada alumno debe proponer su propio programa, que el profesor de prácticas validará al inicio de curso.
- Si un alumno no cumple con sus obligaciones dentro de unos límites de tiempo razonables, sus compañeros podrán asumir las tareas de su rol; esto supondrá la expulsión del grupo del alumno que no trabaje.
- La nota final de la práctica es la **media aritmética** de las notas de cada fase. Para poder aprobar la asignatura, se exigirá **un 4 sobre 10** en la nota de cada fase, y **un 5 sobre 10** en la nota media de las dos fases, tanto en primera como en segunda convocatoria.
- Se guardarán las notas de cada fase entre convocatorias.

- Se conservarán las notas de las fases aprobadas durante un máximo de dos cursos académicos, pero reduciendo el valor obtenido (o conservado) en el curso anterior según las siguientes reglas:
 - nota anterior ≥ 6 ☐ nota conservada = 5
 - nota anterior $\geq 5 < 6$ ☐ nota conservada = 4
 - nota anterior < 5 ☐ no se conserva nota.

2 Estructura de la plataforma de trabajo

2.1 Modos de ejecución del procesador ARM9

Los procesadores ARM, además de la capacidad de ejecutar dos tipos de juego de instrucciones (ARM: 32 bits / THUMB: 16 bits), tienen diversos modos de ejecución. Concretamente, en el ARM9 de la NDS (ARMv5) son los siguientes:

Binario	Hex	Modo
10000b	10h	User (non-privileged)
10001b	11h	FIQ (Fast Interrupt reQuest)
10010b	12h	IRQ (normal Interrupt ReQuest)
10011b	13h	Supervisor (SWI)
10111b	17h	Abort
11011b	1Bh	Undefined
11111b	1Fh	System (privileged User mode)

El significado de dichos modos es el siguiente:

- **User / System:** modos de ejecución “normal” de los programas, con la diferencia de que el modo **System** permite realizar ciertas operaciones privilegiadas que no se permiten en modo **User** (ver más adelante).
- **FIQ:** modo de ejecución de una petición de interrupción crítica, por ejemplo, detección de un fallo en la alimentación del sistema. Este tipo de interrupciones tienen más prioridad que las IRQ y se supone que se tienen que atender más rápidamente (*fast*).
- **IRQ:** modo de ejecución de una petición de interrupción habitual, por ejemplo, retroceso vertical (*Vertical Blank*), pulsación de un botón, finalización de un *timer*, etc.
- **Supervisor:** modo de ejecución de rutinas de sistema activadas por una instrucción `swi` (*SoftWare Interrupt*); normalmente son rutinas de la BIOS (*Basic Input Output System*), que es el código *firmware* almacenado en una memoria *flash* ROM y que proporciona rutinas de control del *Hardware*, como la rutina `swiWaitForVBlank()` (esperar inicio de retroceso vertical).

- **Abort:** modo de ejecución para cuando se produce un acceso no permitido a memoria (fallo de acceso a instrucciones o datos).
- **Undefined:** modo de ejecución para cuando el procesador intenta ejecutar un código de instrucción no definido en el juego de instrucciones actual (ARM o THUMB).

El modo actual del procesador está codificado en el registro de estado actual del procesador, llamado **CPSR** (*Current Processor Status Register*), cuyo contenido en el ARM9 es el siguiente:

Bits	Nombre	Significado
31	N	Flag Negative (0=positivo, 1=negativo)
30	Z	Flag Zero (0=no cero, 1=cero)
29	C	Flag Carry (0=no acarreo, 1=acarreo)
28	V	Flag Overflow (0=no overflow, 1=overflow)
27-8	---	Reservados
7	I	Flag IRQ (0=permitidas, 1=no permitidas)
6	F	Flag FIQ (0=permitidas, 1=no permitidas)
5	T	Tipo de juego de instrucciones (0=ARM, 1=THUMB)
4-0	M4-M0	Bits de modo

Como se puede observar, el registro **CPSR** codifica el modo de ejecución actual del procesador, además del tipo de juego de instrucciones actual y el estado de los *flags* como resultado de la última operación realizada que los haya modificado (una `cmp`, por ejemplo).

La cuestión de los modos de ejecución es muy importante para esta práctica, ya que existen diferentes variantes de los registros de lenguaje máquina según el modo actual.

Como se puede ver en la figura de la página siguiente, el modo **Sistema / Usuario** presenta el conjunto de 16 registros básicos (**R0-R15**) más el registro de estado actual del procesador (**CPSR**).

Sin embargo, los otros 5 modos presentan registros **SP / LR** propios. Por ejemplo, en el modo **IRQ** encontramos el `R13_irq` y el `R14_irq`. Esto significa que, cuando el procesador cambia de modo, en este caso por activación de una IRQ, el procesador utilizará el valor de estos nuevos registros cuando haya que acceder a la pila (instrucciones `push` / `pop`) o cuando haya que almacenar la dirección de retorno en una llamada a rutina (instrucción `bl`).

System/User	FIQ	Supervisor	Abort	IRQ	Undefined
R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7
R8	R8_fiq	R8	R8	R8	R8
R9	R9_fiq	R9	R9	R9	R9
R10	R10_fiq	R10	R10	R10	R10
R11	R11_fiq	R11	R11	R11	R11
R12	R12_fiq	R12	R12	R12	R12
R13 (SP)	R13_fiq	R13_svc	R13_abt	R13_irq	R13_und
R14 (LR)	R14_fiq	R14_svc	R14_abt	R14_irq	R14_und
R15 (PC)	R15	R15	R15	R15	R15
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
--	SPSR_fiq	SPSR_svc	SPSR_abt	SPSR_irq	SPSR_und

(fuente: <http://problemkaputt.de/gbatek.htm#armcpuregisterset>)

Esto permite **no** modificar los registros R13 y R14 del modo anterior (habitualmente, el modo **Usuario** o **Sistema**) cuando se ejecuta código especializado como el de una RSI.

Esta cualidad, sin embargo, también supondrá una mayor complejidad de la práctica, puesto que hay que tener en cuenta que cada modo de ejecución tiene **su propia pila** y **su propia dirección de retorno**.

Además, los modos excepcionales disponen de un registro específico llamado **SPSR** (*Saved Processor Status Register*), que almacena el contenido del **CPSR** del modo anterior.

Esto permite **salvar** el estado del procesador antes de cambiar de modo y restaurar dicho estado cuando se vuelve al modo anterior (al final de una RSI, por ejemplo).

Una vez más, esta característica de los procesadores ARM se tendrá que tener en cuenta a la hora de programar el cambio de proceso activo, puesto que dicho cambio se realizará habitualmente dentro de una RSI (modo **IRQ**), pero tendrá que gestionar el contenido de los registros del modo **Sistema**.

Por último, destacar que la diferencia fundamental entre el modo **Usuario** y el modo **Sistema** es que el segundo puede modificar el contenido del registro **CPSR** y el primero no, lo cual permite cambiar el modo de ejecución o activar / desactivar las interrupciones IRQ y FIQ con instrucciones de lenguaje máquina.

2.2 Gestión de excepciones ARM en la plataforma NDS

Las excepciones están muy relacionadas con los modos de ejecución explicados anteriormente. Una excepción es **una señal** que indica al procesador que ha ocurrido algún evento que requiere **una atención especial**, por ejemplo, una IRQ.

Cuando esto ocurre, el procesador **guarda** su estado y la dirección de ejecución actual (**PC**) en los registros **SPSR** y **LR** del modo de ejecución correspondiente a la excepción, **salta** a una determinada posición de memoria para ejecutar el código de gestión de la excepción y, cuando dicho código termina, se **restaura** el contenido de los registros **CPSR** y **PC** con los valores almacenados en los registros **SPSR** y **LR** del modo actual.

Nótese que, al restaurar el contenido anterior del **CPSR**, se restaura también el modo de ejecución anterior. Esto también afecta al tipo de juego de instrucciones anterior, ya que, cuando el procesador salta a ejecutar el código de gestión de una excepción, siempre lo hace en modo ARM, aunque puede que el código interrumpido fuese THUMB; al restaurar el contenido del bit **T**, el procesador regresará al código interrumpido con el tipo de juego de instrucciones correcto, fuese THUMB o ARM.

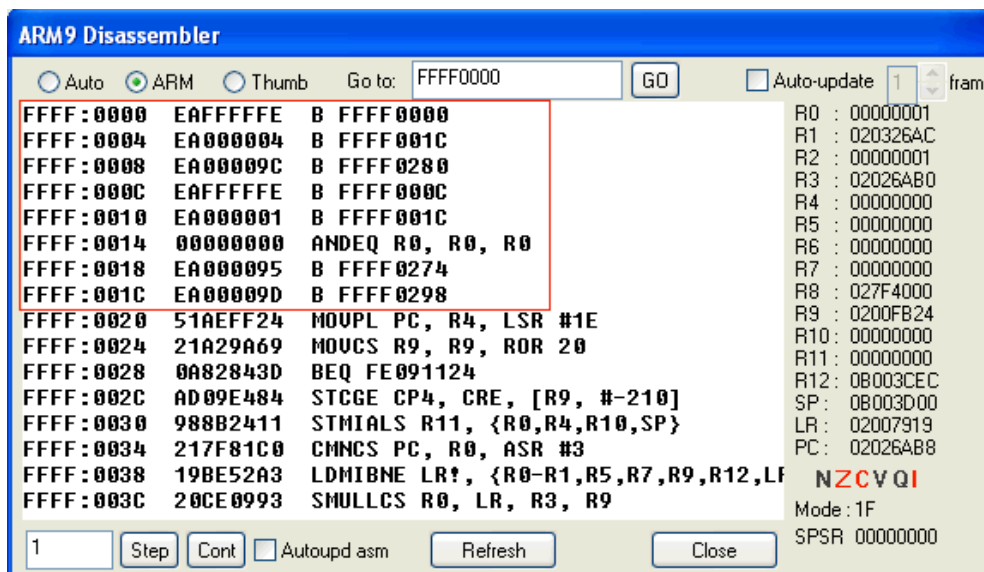
Por último, es importante recordar que el valor de los *flags* en el momento de la interrupción tiene que ser restaurado obligatoriamente, puesto que, de otro modo, se podrían producir fallos en la ejecución del programa si la interrupción ocurriese, por ejemplo, entre una instrucción **cmp** y una instrucción de salto condicional **beq**, ya que la segunda instrucción consulta el estado de los *flags* que ha establecido la primera instrucción.

Para determinar la posición de salto de una excepción, el procesador utiliza lo que se denomina el **vector de excepciones**, el cual se estructura del siguiente modo:

Address	Exception	Mode on Entry	Interrupt Flags
BASE+00h	Reset	Supervisor	I=1, F=1
BASE+04h	Undefined Instruction	Undefined	I=1, F=unchanged
BASE+08h	Software Interrupt (SWI)	Supervisor	I=1, F=unchanged
BASE+0Ch	Prefetch Abort	Abort	I=1, F=unchanged
BASE+10h	Data Abort	Abort	I=1, F=unchanged
BASE+14h	(Reserved)	---	--
BASE+18h	Normal Interrupt	IRQ	I=1, F=unchanged
BASE+1Ch	Fast Interrupt	FIQ	I=1, F=1

La dirección BASE se puede alternar entre 0x00000000 y 0xFFFF0000. La configuración inicial de la NDS utiliza el segundo valor, que corresponde a la dirección inicial de la memoria ROM que alberga el código de la BIOS para el procesador ARM9.

Cada posición del vector de excepciones solo puede albergar una instrucción ARM, que habitualmente es una instrucción de salto **b** hacia la posición de inicio de la rutina de gestión de la excepción. Cuando se inicia una simulación con **DeSmuME**, el contenido del vector de excepciones es el siguiente:



La ventana de desensamblado de **DeSmuME** muestra las direcciones de memoria en la primera columna, el contenido de cada posición de memoria (*words*) en la segunda columna y la interpretación de dicho contenido como instrucciones de lenguaje máquina en la tercera columna, según el tipo de juego de instrucciones seleccionado.

El rectángulo rojo se ha añadido manualmente para señalar la zona del vector de excepciones. Como se puede observar, en la columna de instrucciones de lenguaje máquina todo son saltos a rutinas de gestión de excepciones de la BIOS (direcciones 0xFFFF0xxx), salvo la entrada BASE+14h que contiene un cero (la instrucción resultante no tiene sentido).

De todas las excepciones disponibles, la que nos interesa para esta práctica es la gestión de la IRQ. Como se ve en la dirección 0xFFFF0018, cada vez que se provoca una interrupción y el *flag* **I** es igual a 0, el procesador salta a la dirección 0xFFFF0274, que presenta el siguiente código:

```
0xFFFF0274 push {r0-r3, r12, lr}      @; salvar registros
0xFFFF0278 mrc cp15, 0, r0, c9, c1, 0
0xFFFF027C mov r0, r0, lsr #0xC
0xFFFF0280 mov r0, r0, lsl #0xC      @; R0 = dir. Base DTCM
0xFFFF0284 add r0, r0, #0x4000      @; R0 = final DTCM (+16Kb)
0xFFFF0288 add lr, pc, #0           @; guardar dir. retorno
0xFFFF028C ldr pc, [r0, #-4]        @; saltar a __irq_vector
0xFFFF0290 pop {r0-r3, r12, lr}     @; recuperar registros
0xFFFF0294 subs pc, lr, #4          @; retorno de excepción IRQ
```

Básicamente, esta rutina de la BIOS salta a la **rutina principal de gestión de interrupciones IRQ** (*Main Interrupt handler*), cuya dirección se debe haber guardado previamente en la posición de memoria que se define simbólicamente como `__irq_vector`, y que se encuentra en el último *word* de la memoria DTCM (*Data Tightly Coupled Memory*), que corresponde a la posición 0x0B003FFC de la NDS.

2.3 Gestión de interrupciones mediante libnds9

Cuando se genera un proyecto para la plataforma NDS con el entorno de trabajo **DevkitPro**, habitualmente se programa el procesador ARM9 en lenguaje C y se compila y enlaza con la librería de funciones estándar de C (`printf()`, `exit()`, `malloc()`, etc.) y con la librería de funciones **libnds9**.

La librería **libnds9** ofrece un conjunto muy amplio de funciones, definiciones, estructuras de datos y servicios de gestión del *Hardware* de la NDS, descritas en ficheros de cabecera del lenguaje C (ficheros '.h').

Uno de dichos servicios es la rutina principal de gestión de interrupciones IRQ, que se llama `IntrMain()`, y cuya dirección inicial se copia en la dirección `__irq_vector` cuando se inicializa el entorno del programa, antes de llamar a la función `main()` de nuestro proyecto.

El código fuente de `IntrMain()` se puede consultar en los ficheros fuente de la librería **libnds**, ya que son código abierto. Por ejemplo, la rutina se define en el fichero `libnds-src-1.5.4/source/common/interruptDispatcher.s`.

De todos modos, su funcionamiento es bastante complejo para explicarlo en este manual, pero sí resulta interesante entender la infraestructura que define para gestionar las RSIs específicas para distintos dispositivos, ya que en esta práctica se pretende aprovechar las funciones de la librería **libnds** para instalar nuestra propia rutina principal de gestión de interrupciones, además de nuestras RSIs para IRQs específicas.

La estructura de datos básica es `IntTable`:

```
struct IntTable{IntFn handler; u32 mask};
```

Esta estructura define una entrada del vector de interrupciones que comentaremos después. Sus campos son el `handler`, que es la dirección de memoria de la primera instrucción de la RSI, y la `mask`, que es una máscara que indica qué IRQ atiende dicha RSI.

Para instalar una RSI solo hay que llamar a la siguiente función de **libnds**:

```
void irqSet(u32 irq, VoidFn handler);
```

donde `irq` es la máscara y `handler` es la dirección inicial de la RSI.

Las máscaras disponibles vienen también definidas por los ficheros de cabecera (.h) de **libnds**. Concretamente, en `interrupts.h` se definen las siguientes constantes:

```
enum IRQ_MASKS {
    IRQ_VBLANK      =    BIT(0),    // vertical blank
    IRQ_HBLANK      =    BIT(1),    // horizontal blank
    IRQ_VCOUNT     =    BIT(2),    // vcount match
    IRQ_TIMER0      =    BIT(3),    // end of timer 0
    IRQ_TIMER1      =    BIT(4),    // end of timer 1
    IRQ_TIMER2      =    BIT(5),    // end of timer 2
    IRQ_TIMER3      =    BIT(6),    // end of timer 3
    IRQ_NETWORK     =    BIT(7),    // serial interrupt
    IRQ_DMA0        =    BIT(8),    // end of DMA 0
    IRQ_DMA1        =    BIT(9),    // end of DMA 1
    IRQ_DMA2        =    BIT(10),   // end of DMA 2
    IRQ_DMA3        =    BIT(11),   // end of DMA 3
    IRQ_KEYS        =    BIT(12),   // Keypad activity
    IRQ_CART        =    BIT(13),   // GBA cartridge
    IRQ_IPC_SYNC    =    BIT(16),   // IPC sync
    IRQ_FIFO_EMPTY  =    BIT(17),   // Send FIFO empty
    IRQ_FIFO_NOT_EMPTY = BIT(18),   // Receive FIFO not empty
    IRQ_CARD        =    BIT(19),   // DS Card Slot
    IRQ_CARD_LINE   =    BIT(20),
    IRQ_GEOMETRY_FIFO = BIT(21),   // geometry FIFO (3D)
    IRQ_LID         =    BIT(22),   // DS hinge activity
    IRQ_SPI         =    BIT(23),   // SPI interrupt
    IRQ_WIFI        =    BIT(24),   // WIFI interrupt (ARM7)
    IRQ_ALL         =    (~0) // 'mask' for all interrupts
};
```

Nota: la definición `BIT(n)` construye una máscara de 32 bits, donde el bit `n` se pone a uno y el resto a ceros.

Cuando se llama a la función `irqSet()`, esta almacena una entrada en el vector `irqTable[]`, que se define de la siguiente forma:

```
struct IntTable irqTable[MAX_INTERRUPTS] INT_TABLE_SECTION;
```

Es decir, se trata de un vector con `MAX_INTERRUPTS` entradas (25) de tipo `IntTable`, que se define en una sección de memoria específica (ITCM) que se detallará en el siguiente apartado.

Las entradas se almacenan consecutivamente en el vector. La primera entrada con máscara igual a cero se considera como final del vector (elemento centinela). Por ejemplo, si se ejecuta el siguiente código:

```
irqSet(IRQ_VBLANK, __timeoutvbl);
irqSet(IRQ_FIFO_EMPTY, fifoInternalSendInterrupt);
irqSet(IRQ_FIFO_NOT_EMPTY, fifoInternalRecvInterrupt);
```

el vector de RSIs queda con el siguiente contenido:

index	handler	mask
0	@__timeoutvbl	0x00000001
1	@fifoInternalSendInterrupt	0x00020000
2	@fifoInternalRecvInterrupt	0x00040000
3	---	0x00000000
4	---	0x00000000
...
24	---	0x00000000

Además de la función `irqSet()`, también utilizaremos las funciones de **libnds** siguientes:

```
void irqEnable(u32 irq);
void irqDisable(u32 irq);
```

Estas funciones permiten indicar al controlador de interrupciones (vía el registro REG_IE) qué interrupciones específicas tiene que atender (*enable*) o ignorar (*disable*), según la máscara que se pasa con el parámetro `irq`.

En el ejemplo de las tres RSIs que se han instalado en el vector `irqTable[]` se tendría que invocar la siguiente llamada para que el controlador de interrupciones pudiera atender interrupciones de esos tipos:

```
irqEnable(IRQ_VBLANK | IRQ_FIFO_NOT_EMPTY | IRQ_FIFO_EMPTY);
```

En este caso se activan los tres tipos de interrupción con una sola llamada, puesto que las máscaras de cada tipo se han agregado mediante la función lógica OR (`|`).

Cabe recordar que, para que las RSIs funcionen, también puede ser necesario activar algún bit en el controlador del dispositivo involucrado. Por ejemplo, en el caso de la interrupción por retroceso vertical es necesario ejecutar el siguiente código:

```
REG_DISPSTAT |= DISP_VBLANK_IRQ;
```

Sin embargo, la función `irqEnable()` ya ejecuta el código anterior, si se activan este tipo de interrupciones. También efectúa operaciones similares

para las interrupciones IRQ_HBLANK, IRQ_VCOUNT y IRQ_IPC_SYNC, pero no para el resto, o sea que la activación de las interrupciones en el controlador de dispositivo para los otros casos se tendrá que realizar explícitamente.

La librería **libnds** nos ofrece otra función que utilizaremos en la inicialización de la práctica:

```
void irqInitHandler(VoidFn handler);
```

Con esta función podremos instalar nuestra propia rutina principal de gestión de interrupciones en la dirección `__irq_vector`, para que la BIOS le pase el control cuando se produzca cualquier interrupción IRQ.

Por último, es posible que sea necesario activar explícitamente el bit 0 del registro principal del controlador de interrupciones REG_IME, para que las IRQ de los dispositivos puedan activar efectivamente las interrupciones correspondientes. Par esta activación, desde lenguaje C se puede especificar el acceso directo al registro del siguiente modo:

```
*((vuint32 *) 0x4000208) = 1;
```

Alternativamente, podemos utilizar las definiciones de la librería **libnds** para expresarlo de modo más simbólico:

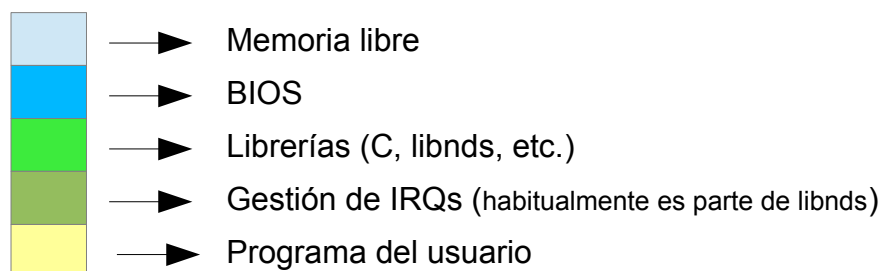
```
REG_IME = IME_ENABLE;
```

2.4 Ocupación de las zonas de memoria NDS

Los mapas de memoria de este apartado muestran dónde se ubican las diferentes partes de un proyecto basado en **libnds** dentro de las zonas de memoria disponibles en la plataforma NDS.

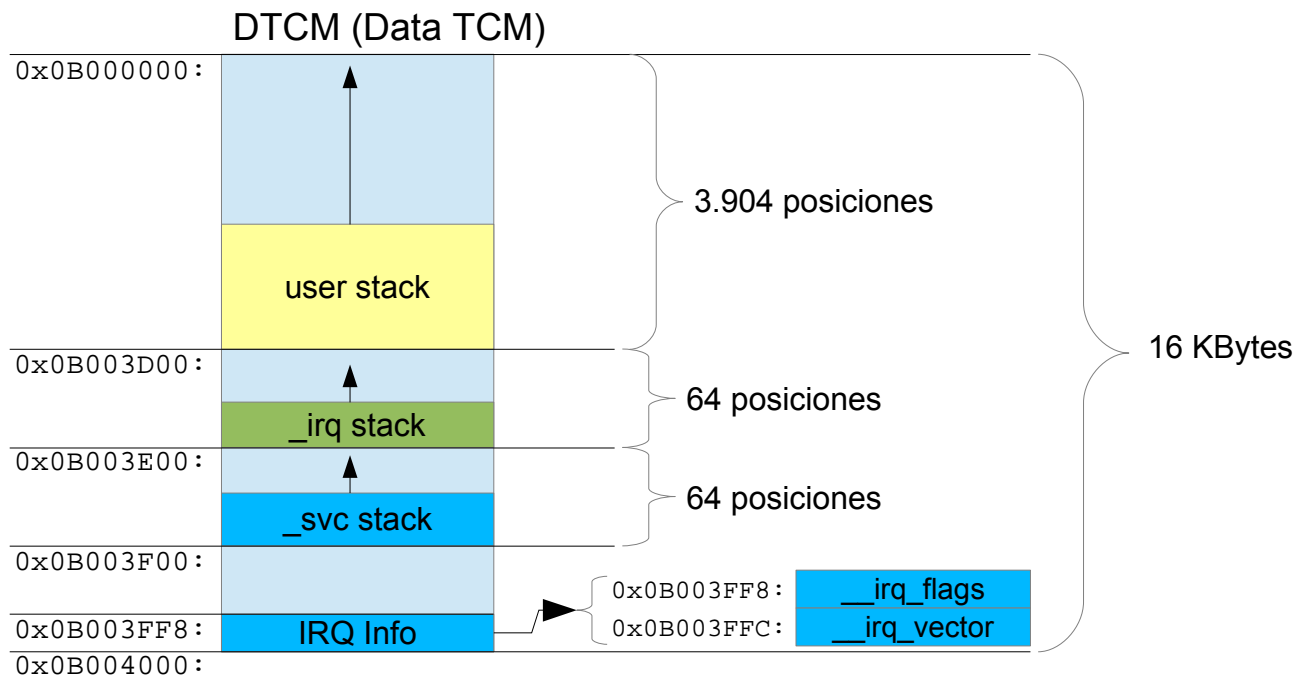
El siguiente código de colores permitirá distinguir distintos tipos de *software* que ocupa cada zona de memoria:

Código de colores:



2.4.1 Data Tightly Coupled Memory

Esta zona específica permite al ARM9 leer y escribir datos rápidamente, mediante un bus de datos dedicado de 32 bits, una frecuencia de 66 Mhz y con posibles accesos paralelos con otras zonas de memoria.

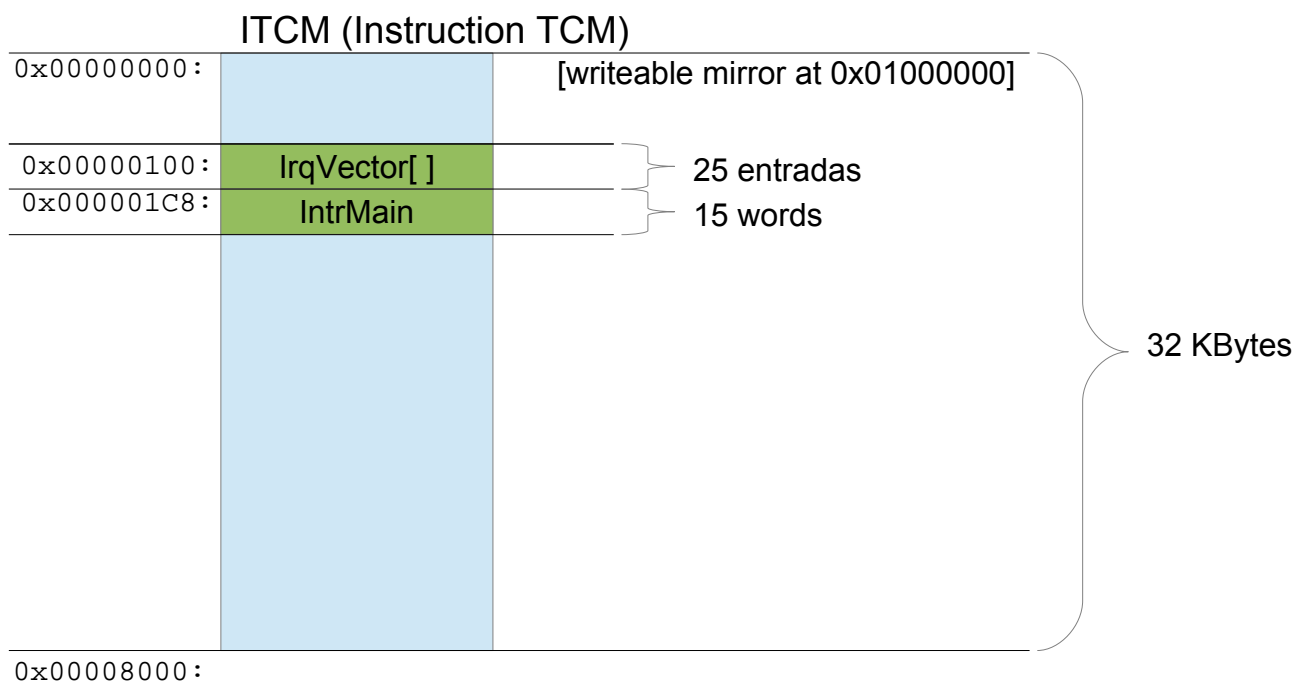


En esta zona, **libnds** coloca las pilas para los modos de ejecución **Usuario**, **IRQ** y **Supervisor**. Hay que observar que las pilas crecen hacia posiciones bajas de memoria (hacia "arriba"), y que cada pila tiene un cierto límite que, en caso de sobrepasarlo, provocaría el bloqueo de todo el sistema.

Además, la BIOS utiliza los dos últimos *words* para saltar a la rutina principal de gestión de las interrupciones IRQ (**__irq_vector**) y para detectar qué tipo de interrupción se ha producido (**__irq_flags**) dentro de las rutinas de espera de la BIOS, como la `swiWaitForVBlank()`.

2.4.2 *Instruction Tightly Coupled Memory*

Esta zona específica permite al ARM9 leer instrucciones rápidamente, mediante un bus de instrucciones dedicado de 32 bits, una frecuencia de 66 Mhz y con posibles accesos paralelos con otras zonas de memoria.

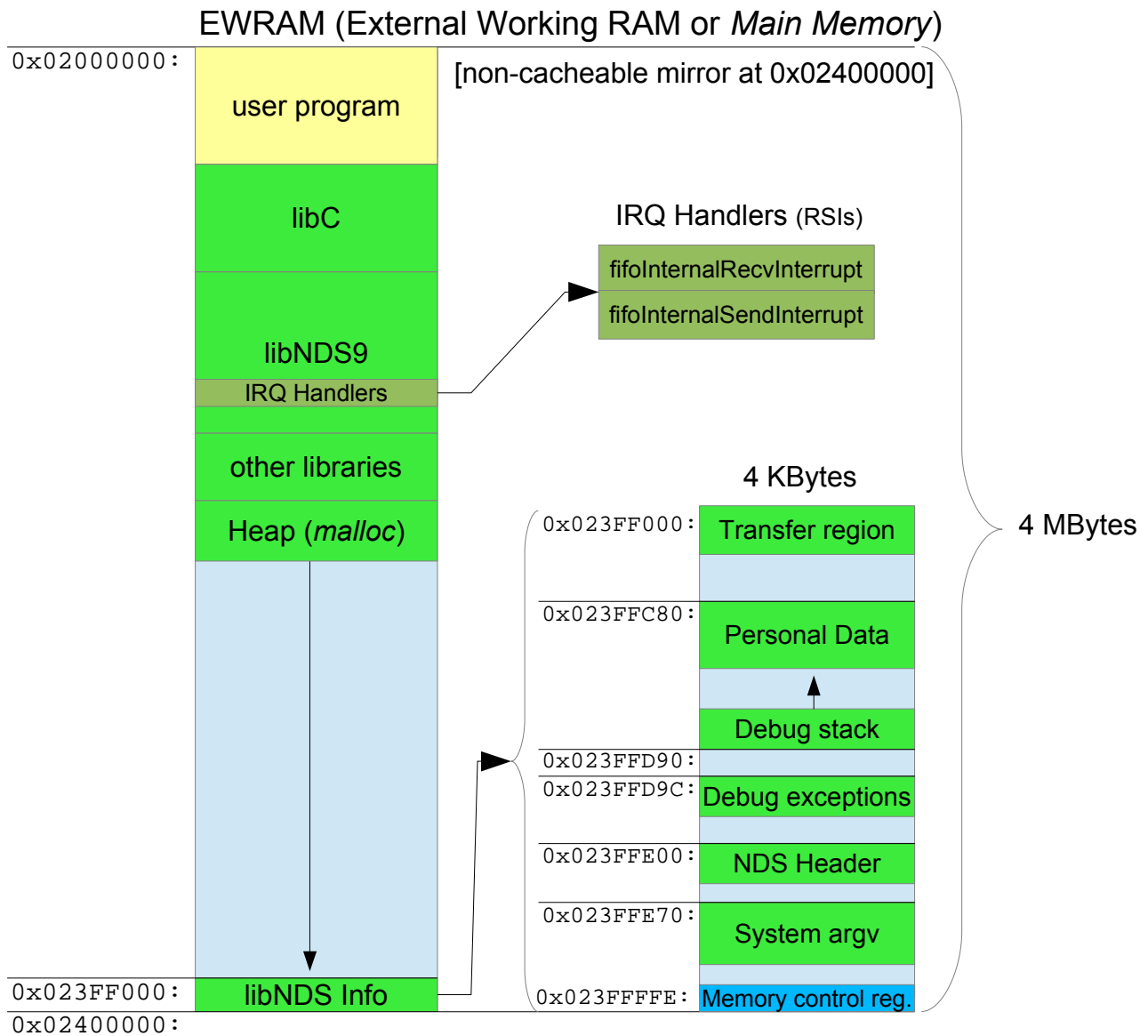


La librería **libnds** solo utiliza esta zona para almacenar el vector de las RSIs `IrqVector[]` y la rutina principal de gestión de interrupciones `IntrMain()`.

Existe, además, un *mirror* o “reflejo” del contenido de esta zona a partir de la posición 0x01000000, donde el procesador también puede escribir información, lo cual permite realizar modificaciones en el vector de las RSIs.

2.4.3 Main Memory

La memoria principal es la que alberga la mayor parte del proyecto. Aunque dispone de mucha capacidad (4 Mbytes), es una memoria de acceso lento (16 bits / 8 Mhz).



El bloque “*user program*” se refiere al código escrito explícitamente por los programadores del proyecto.

Los bloques en verde se adjuntan por el compilador y el *linkador*. La librería **libC** implementa las funciones estándar de C, como `printf()`, `exit()` o `malloc()`.

La librería **libnds9** implementa las funciones **libnds** para el procesador ARM9. Entre otras cosas, incorpora dos RSIs de uso interno para la gestión de la comunicación de información con el ARM7 (ver apartado 2.5).

Pueden haber otras librerías adjuntas, como por ejemplo la librería **libfilesystem**, que permite el uso de funciones relacionadas con el acceso a ficheros y directorios.

Como última parte del programa se encuentra el bloque de *Heap*, que es una estructura de datos gestionada por la librería de C para reservar y liberar memoria dinámicamente, mediante las funciones estándar `malloc()` y `free()`. Esta estructura puede crecer hacia zonas altas de la memoria (hacia “abajo”) a medida que el programa va requiriendo más memoria dinámica, mientras no libere la memoria reservada anteriormente.

Los últimos 4 Kbytes están reservados por la librería **libnds** para almacenar diversas estructuras de información, como la `TransferRegion`, que permite registrar la pulsación de los botones **X** e **Y**, o bien las coordenadas de la pantalla táctil donde se presiona con el lápiz, todo lo cual no es accesible directamente desde el ARM9 y se transfiere desde el ARM7 mediante un protocolo específico.

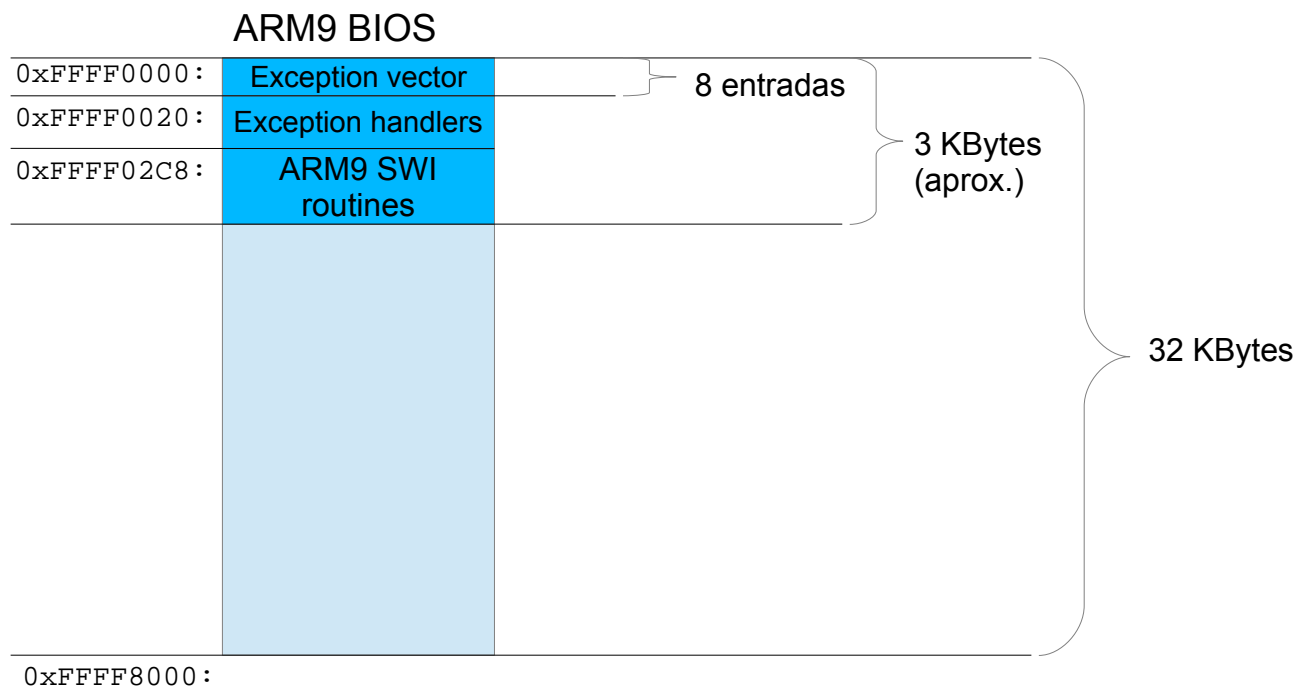
Los últimos 2 bytes de la memoria son, en realidad, un registro de configuración de la memoria, o sea, que se comportan como un registro de Entrada/Salida. Hemos utilizado el color azul para indicar que su uso se suele realizar desde la BIOS, aunque técnicamente se tendría que pintar de otro color porque no forma parte de la BIOS sino del *hardware* de la memoria principal.

Existen, *mirrors* del contenido de esta zona a partir de las posiciones 0x02400000, 0x02800000 y 0x02C00000, por lo que, en diversas fuentes de información o en el propio código fuente de **libnds**, se puede observar direcciones como 0x027FF000 en vez de 0x023FF000, aunque en realidad se refieren al mismo contenido.

Sin embargo, los *mirrors* a partir de 0x02400000 y 0x02C00000 son no “cacheables,” es decir, el procesador ARM9 no utiliza la caché cuando se producen escrituras en los respectivos rangos de memoria. Esto es necesario cuando el contenido de la memoria se deba actualizar inmediatamente, en vez de registrar el cambio solo en la memoria caché, por ejemplo para compartirlo con el ARM7.

2.4.4 BIOS ARM9

Esta zona es una memoria ROM que alberga el *firmware* de la BIOS para el ARM9. Es una memoria con acceso de 32 bits, pero **no** se puede escribir en ella.



Al principio se encuentra el vector de excepciones, seguido por el código básico para manejar dichas excepciones.

A continuación está almacenado el código de las rutinas SWI para el ARM9, como la `swiWaitForVBlank()`.

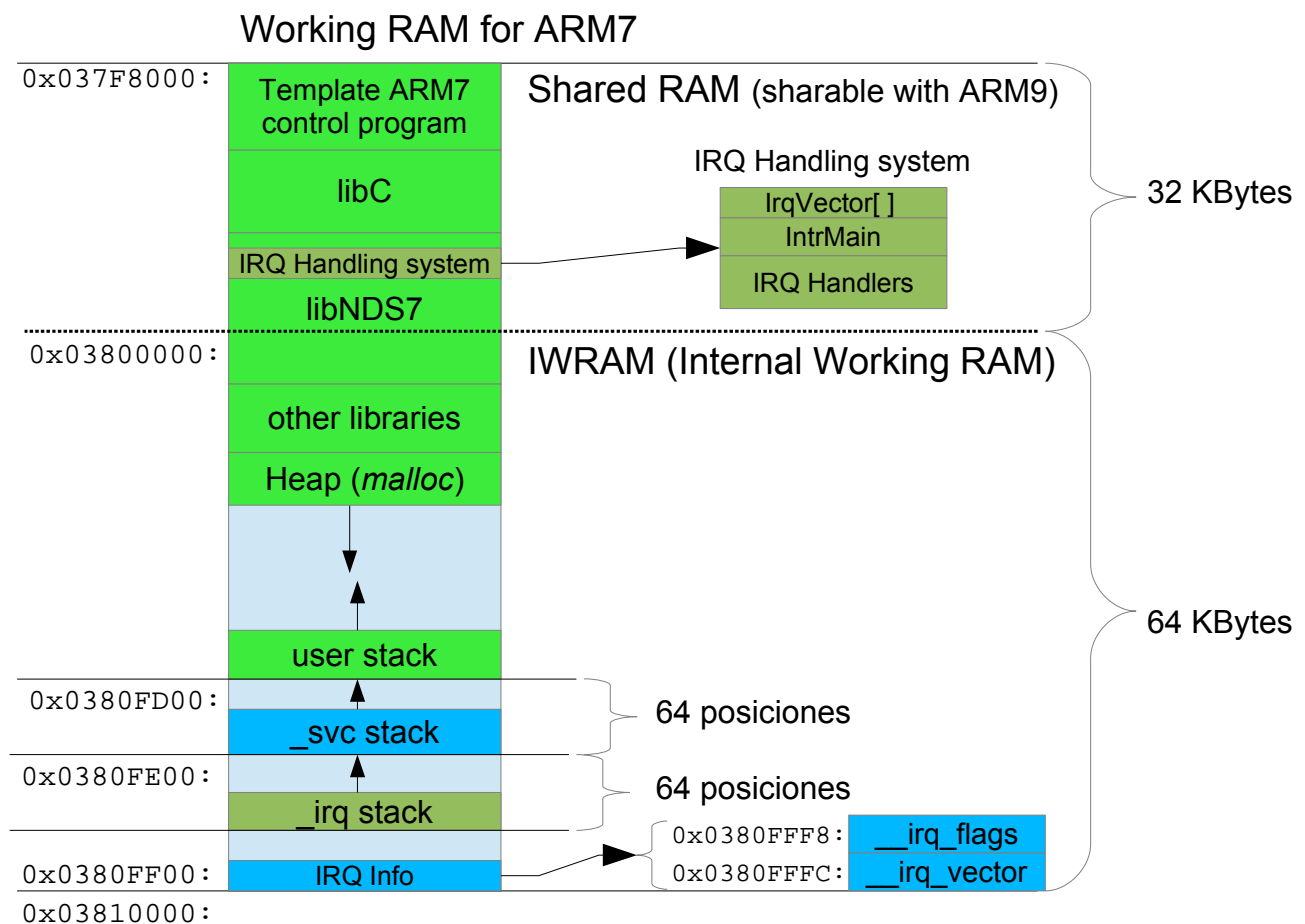
Solo se aprovechan unos 3 Kbytes del espacio total disponible.

2.4.5 Zona de trabajo para el ARM7

Aunque el procesador ARM7 también tiene acceso a la zona de memoria principal (apartado 2.4.3), la configuración **libnds** reserva dos zonas de memoria específicas de la NDS para albergar su código.

La zona *Shared Memory* (memoria compartida) son dos bloques de 16 Kbytes (acceso de 32 bits) que se pueden asignar independientemente a uno de los dos procesadores de la NDS, aunque **libnds** asigna ambos al ARM7.

La zona *Internal Working RAM* es una memoria específica de trabajo destinada al ARM7, con acceso de 32 bits y 33MHz de frecuencia. Se utiliza el último *mirror* de la *shared memory* (el primero empieza en 0x03000000) para conseguir un espacio contiguo de 96 Kbytes.



En este caso vemos que el bloque "*user program*" está representado por otro bloque llamado "*Template ARM7 control program*", lo cual significa que el programador **no** escribe el código para el ARM7 (habitualmente), sino que se utiliza una plantilla (*template*) para controlar este procesador. Como este código pertenece en realidad a la librería **libnds**, se ha utilizado el color verde en vez del amarillo.

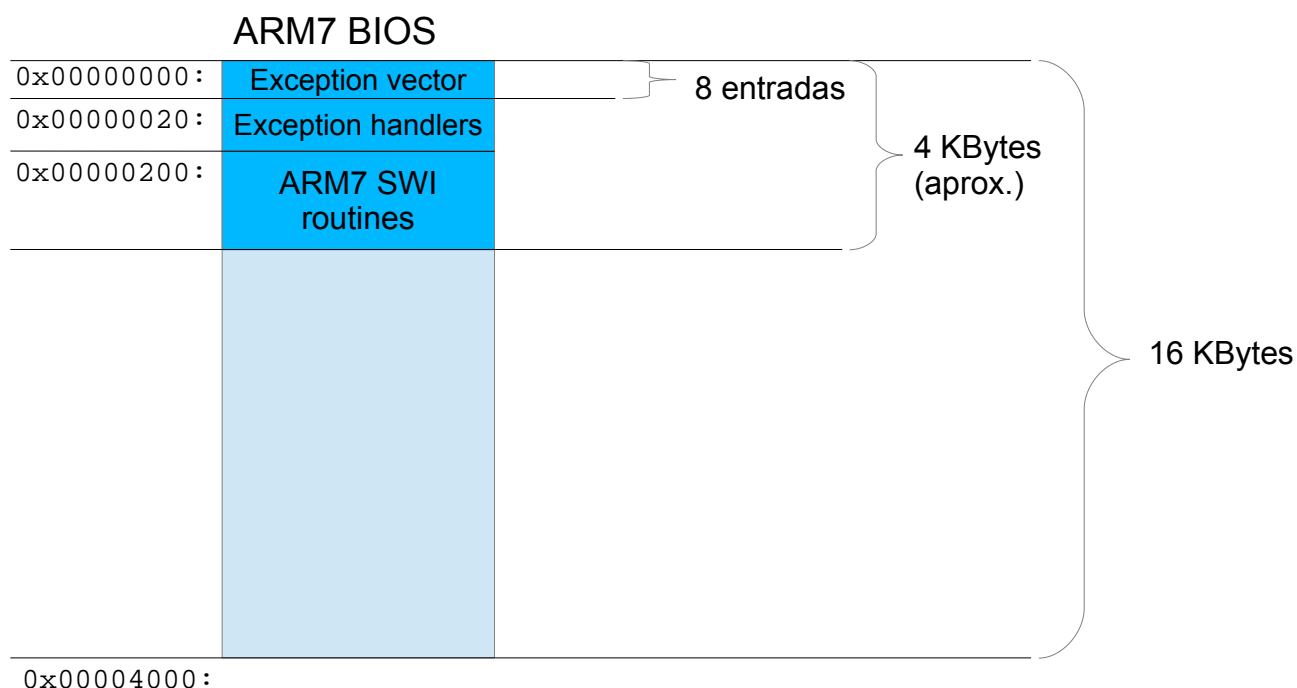
Existe también el código de la librería estándar de C, la versión de la librería **libnds** para el ARM7, otras librerías específicas y el *Heap* propio del programa de control.

Además, la parte de la librería **libnds7** alberga toda la información relativa al manejo de las interrupciones del procesador ARM7, es decir, su propio `IrqVector[]`, su propia rutina principal de gestión de interrupciones IRQ y sus RSIs particulares.

En las últimas posiciones de toda la zona de trabajo del ARM7 están ubicadas las pilas de usuario, supervisor e IRQ correspondientes, además de las variables `__irq_flags` e `__irq_vector` propias del ARM7.

2.4.6 BIOS ARM7

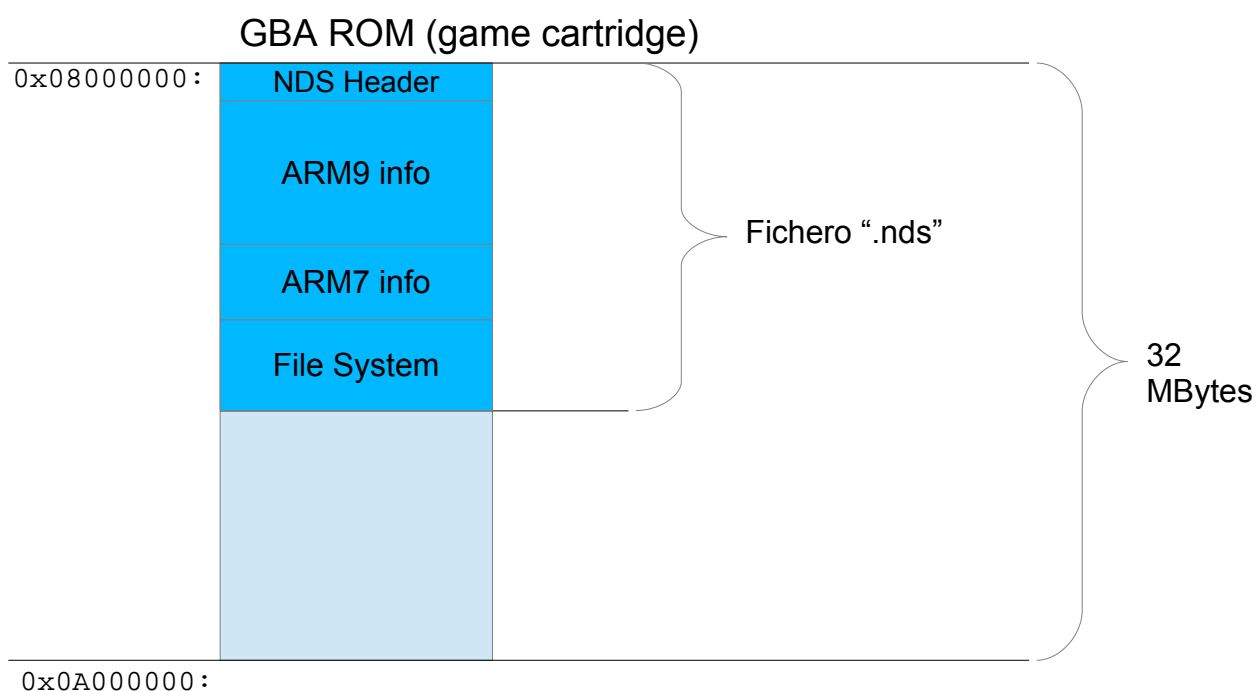
Esta zona es una memoria ROM que alberga el *firmware* de la BIOS para el ARM7. Es una memoria con acceso de 32 bits, pero **no** se puede escribir en ella.



Su disposición es muy similar a la BIOS para el ARM9, aunque se diferencia en la dirección base y en el número y contenido de las rutinas SWI.

2.4.7 GBA ROM

Los cartuchos de juegos (o aplicaciones), que se insertan en el *slot* GBA o en el *slot* NDS, tienen que presentar una cierta organización para que el *firmware* de la consola los pueda reconocer. Una vez admitido, su contenido se puede leer a partir de la posición 0x08000000, aunque **no** se puede escribir.



El *firmware* de la NDS se encarga de cargar la información para cada procesador en las zonas de memoria de trabajo principales, para que los procesadores puedan pasar a ejecutar el código correspondiente.

El bloque "File System" **no** se copia en memoria RAM, sino que se accede mediante las funciones del sistema de ficheros (**Nitro** o **Fat**). Por lo tanto, los sistemas de ficheros son de **solo lectura**, pero son útiles porque podemos albergar hasta ocho veces más información de la que cabe en los 4 Mbytes de RAM principal.

El entorno de desarrollo **DevkitPro** se configura habitualmente para generar un fichero '**.nds**'. Este fichero es el que se tiene que copiar en una tarjeta de memoria compatible para poder ejecutar el proyecto en una consola NDS real.

Si no se dispone de la consola, el simulador **DeSmuME** (u otro similar) permite leer directamente el fichero '**.nds**' generado y simular su "carga" en memoria, como si se tratara de una tarjeta insertada en el zócalo correspondiente.

2.5 Comunicación entre los procesadores ARM7 y ARM9

Para terminar con esta descripción interna de la plataforma NDS, hay que recordar que existe un sistema de comunicación *hardware* entre los dos procesadores (ver sistema FIFO en los apuntes de *Computadores*) que la librería **libnds** utiliza intensivamente.

El sistema de comunicación definido por **libnds** se basa en canales. En el fichero **fifocommon.h** se definen los tipos de canales disponibles:

```
typedef enum {  
    FIFO_PM           = 0,          // power management  
    FIFO_SOUND        = 1,          // sound access  
    FIFO_SYSTEM       = 2,          // system functions  
    FIFO_MAXMOD       = 3,          // maxmod library  
    FIFO_DSWIFI       = 4,          // dswifi library  
    FIFO_SDMMC        = 5,          // dsi sdmmc control  
    FIFO_RSVD_01      = 6,          // reserved for future use  
    FIFO_RSVD_02      = 7,          // reserved for future use  
    FIFO_USER_01      = 8,          // channels available for users  
    FIFO_USER_02      = 9,  
    FIFO_USER_03      = 10,  
    FIFO_USER_04      = 11,  
    FIFO_USER_05      = 12,  
    FIFO_USER_06      = 13,  
    FIFO_USER_07      = 14,  
    FIFO_USER_08      = 15,  
} FifoChannels;
```

En este manual **no** se explicarán los diferentes comandos y paquetes de información que se envían por cada canal, pero es necesario insistir en que algunos de los servicios de la librería **libnds** se apoyan en este sistema de comunicaciones para poder funcionar, como por ejemplo, la transmisión de la información relativa al estado de los botones **X** e **Y**, junto con la posición del lápiz en la **pantalla táctil**, ya que el ARM9 no tiene acceso al *hardware* involucrado.

Más concretamente, este sistema de comunicaciones se basa en las RSIs `fifoInternalRecvInterrupt()` y `fifoInternalSendInterrupt()`, lo cual significa que los servicios indicados dependen de la correcta gestión de las interrupciones.

Sin embargo, a partir del momento en que instalemos nuestra propia rutina principal de gestión de interrupciones, dicho sistema de comunicaciones dejará de funcionar correctamente, puesto que requiere que las interrupciones sean **reentrantes**, es decir, que dentro de una interrupción se pueda procesar otra interrupción, lo cual incrementaría considerablemente la complejidad del sistema de multiplexación de procesos.

Por este motivo, si se necesita intercambiar información con el ARM7, habrá que definir un protocolo propio para realizar dicha comunicación, por ejemplo para obtener el estado de la pantalla táctil (segunda fase del control de teclado).

En general, será necesario conocer bien muchas de las particularidades del funcionamiento interno del *hardware* NDS y de la librería **libnds** para poder realizar esta práctica con garantías de éxito.

3 Especificaciones generales de la práctica

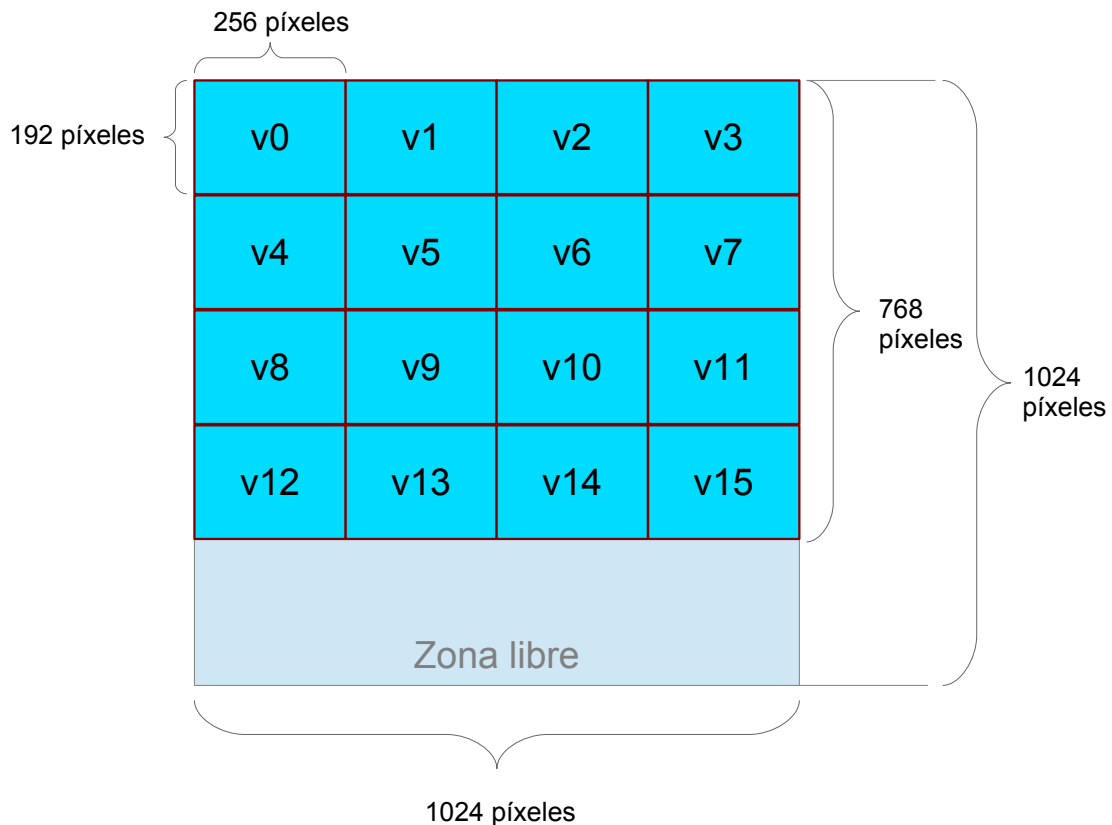
3.1 Requisitos generales

En general, el sistema operativo **GARLIC** tendrá que ser capaz de cargar programas compatibles desde ficheros ejecutables **ELF**, multiplexando los correspondientes procesos sobre el procesador ARM9 de la plataforma NDS.

Se exigen los siguientes requisitos:

- **Programas:** los programas estarán escritos en **lenguaje C**, pero solo podrán utilizar llamadas al **API** (*Application Program Interface*) del sistema operativo, es decir, **no** podrán invocar a las funciones estándar de la librería de C, como `printf()` o `fopen()`.
- **Sistema de ficheros:** se utilizará el sistema de ficheros **Nitro** para almacenar los programas, aunque solo el sistema operativo tendrá acceso a los ficheros (solo lectura).
- **Procesos:** se podrán ejecutar hasta **16** procesos concurrentemente (grado de multiprogramación 16), con intercambio de procesos por *Round Robin* (FIFO sin prioridad); uno de los procesos será de control del propio sistema operativo, mientras que los otros 15 procesos podrán ser cualquier programa compatible con GARLIC; un mismo programa se podrá cargar varias veces como procesos diferentes.
- **Prioridad:** los procesos tendrán todos la **misma prioridad**, incluyendo el proceso de control del sistema operativo.
- **Quantum:** los procesos tendrán un quantum de **16,67 milisegundos**, que corresponde al periodo de la interrupción de retroceso vertical (VBL), es decir, se producirán 60 cambios de proceso por segundo.
- **Contexto e identificación de los procesos:** cada proceso dispondrá de su propio contexto (registros, pila, etc.), que se gestionará con un **PCB** (*Process Control Block*) particular; para referenciar el PCB de un proceso, se usará un índice de la tabla de PCBs, es decir, un número del 0 al 15, que denominaremos **zócalo**; este índice se usará en determinadas partes del sistema operativo para obtener un acceso rápido al contexto del proceso, aunque cada proceso dispondrá de un identificador genérico **PID** (*Process Identifier*), que se incrementará con cada nuevo proceso creado.

- **Ventanas de texto:** se definen diversas ventanas en las que los procesos podrán escribir mensajes o caracteres de texto, según la posición actual de un cursor o en unas coordenadas de fila y columna determinadas; cada ventana tiene 24 filas por 32 columnas; se podrán generar hasta 16 ventanas, distribuidas en un mapa de vídeo de 1024x1024 píxeles, según el siguiente gráfico:



- **Sincronización de escritura:** cuando un proceso envía información a su ventana, el texto resultante de salida se almacenará en un vector de 32 caracteres, correspondientes a una fila entera de la ventana, es decir, **un buffer de una línea**; cuando el vector esté lleno o cuando se envíe un salto de línea ('\n'), la función de escritura del sistema transferirá dicho buffer a las posiciones del mapa de baldosas correspondientes a dicha ventana, pero antes tendrá que esperar el señal de retroceso vertical, lo que significa que el proceso que está escribiendo será **desbancado** para asegurar que se accede a la memoria de vídeo de forma sincronizada con la visualización de los píxeles de la pantalla.
- **Desplazamiento automático (scroll):** cuando la fila actual de escritura en una ventana sea la 23 (última fila), la nueva fila que se envíe provocará un desplazamiento automático del contenido de la ventana una fila hacia arriba, de modo que quede una nueva fila libre para escribir, a costa de eliminar la fila más antigua (primera) de la ventana.

- **Escritura directa:** en el caso de que el proceso envíe caracteres a posiciones concretas (fila, columna), se escribirán directamente en la memoria de vídeo, sin sincronización y sin realizar ningún tipo de desplazamiento.
- **Color del texto:** se podrán utilizar hasta 4 colores distintos de texto.
- **Zoom de las ventanas:** el sistema permitirá visualizar el contenido de cualquier ventana de texto utilizando todo el espacio disponible de la pantalla superior de la NDS (256x192 píxeles), pero también se podrá realizar un zoom de reducción de $\frac{1}{2}$ o de $\frac{1}{4}$ para visualizar 4 ventanas o las 16 ventanas simultáneamente.
- **Retardos:** los procesos podrán pedir al sistema operativo que se retarde su ejecución durante un cierto número de segundos, o ser desbancados hasta el próximo turno.
- **Pantalla de control:** en la pantalla inferior de la NDS se visualizará una tabla con el estado de todos los procesos cargados en el sistema; su contenido será similar al siguiente:

Z	PID	Prog	PCactual	Pi	E	Uso
0	0	GARL	010004BC		R	50%
1	----	----	-----		-	-%
2	----	----	-----		-	-%
3	1	HOLA	0100031C		Y	50%
4	----	----	-----		-	-%
5	----	----	-----		-	-%
6	----	----	-----		-	-%
7	----	----	-----		-	-%
8	----	----	-----		-	-%
9	----	----	-----		-	-%
10	----	----	-----		-	-%
11	----	----	-----		-	-%
12	----	----	-----		-	-%
13	----	----	-----		-	-%
14	----	----	-----		-	-%
15	----	----	-----		-	-%

(Nota: el significado de cada columna se explicará en el manual de la fase 2)

- **Estado de la memoria:** debajo de la tabla de control de procesos se visualizará el estado de la memoria como una secuencia de fragmentos ocupados por los distintos procesos cargados, utilizando un color diferente para cada proceso; la visualización será similar a la siguiente:

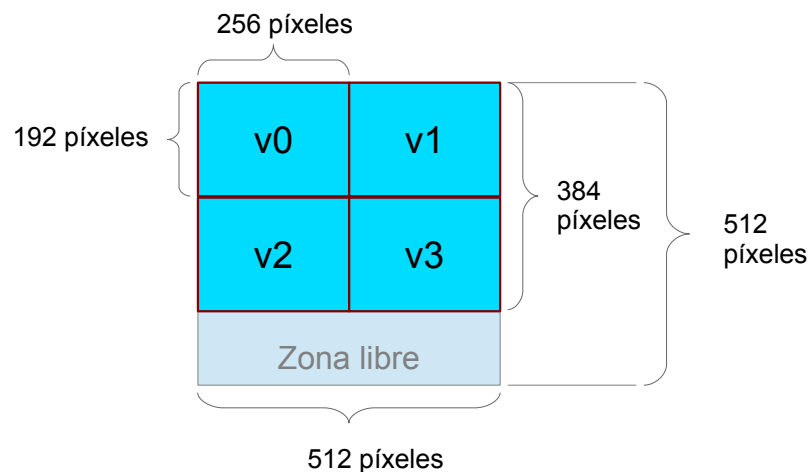


- **Teclas de control:** para controlar todo el sistema se utilizarán los botones de la NDS, según se explicará en el manual de la fase 2.
- **Introducción de texto:** cuando un proceso (o varios) requiera capturar información introducida por el usuario, se dibujará un teclado virtual sobre la pantalla de control y se detectarán las pulsaciones de las teclas con el sensor táctil, para luego transferir todo el *string* al primer proceso que haya pedido introducción de texto.

3.2 Requisitos restringidos para la primera fase

La fase 1 restringirá los requisitos generales, puesto que se pretende realizar una primera aproximación a la solución definitiva. Las restricciones son las siguientes:

- **Procesos:** la primera fase tiene que ser capaz de cargar y multiplexar en el tiempo hasta 16 procesos simultáneamente (sin restricción).
- **Ventanas de texto:** solo se mostrarán 4 ventanas; los procesos en ejecución enviarán el texto a la ventana cuyo índice corresponda al número de zócalo del proceso módulo 4.



- **Acceso concurrente a las ventanas:** para simplificar, supondremos que en cada ventana solo enviará información un único proceso.
- **Escritura directa:** no se implementará; solo se implementará la escritura de texto en la posición actual del cursor.
- **Color de texto:** no se implementará; solo habrá un único color del texto (blanco).
- **Zoom de las ventanas:** no se implementará; solo se mostrarán las cuatro ventanas simultáneamente.
- **Retardos:** no se implementarán.
- **Pantalla de control:** no se implementará.
- **Estado de la memoria:** no se implementará.

- **Entrada de texto:** la detección de pulsaciones no se realizará sobre la pantalla táctil, sino que se utilizarán algunos botones de la NDS para cambiar la letra actual (incrementar/decrementar código ASCII), para mover el cursor y para validar la introducción del *string* (*Carry Return*).

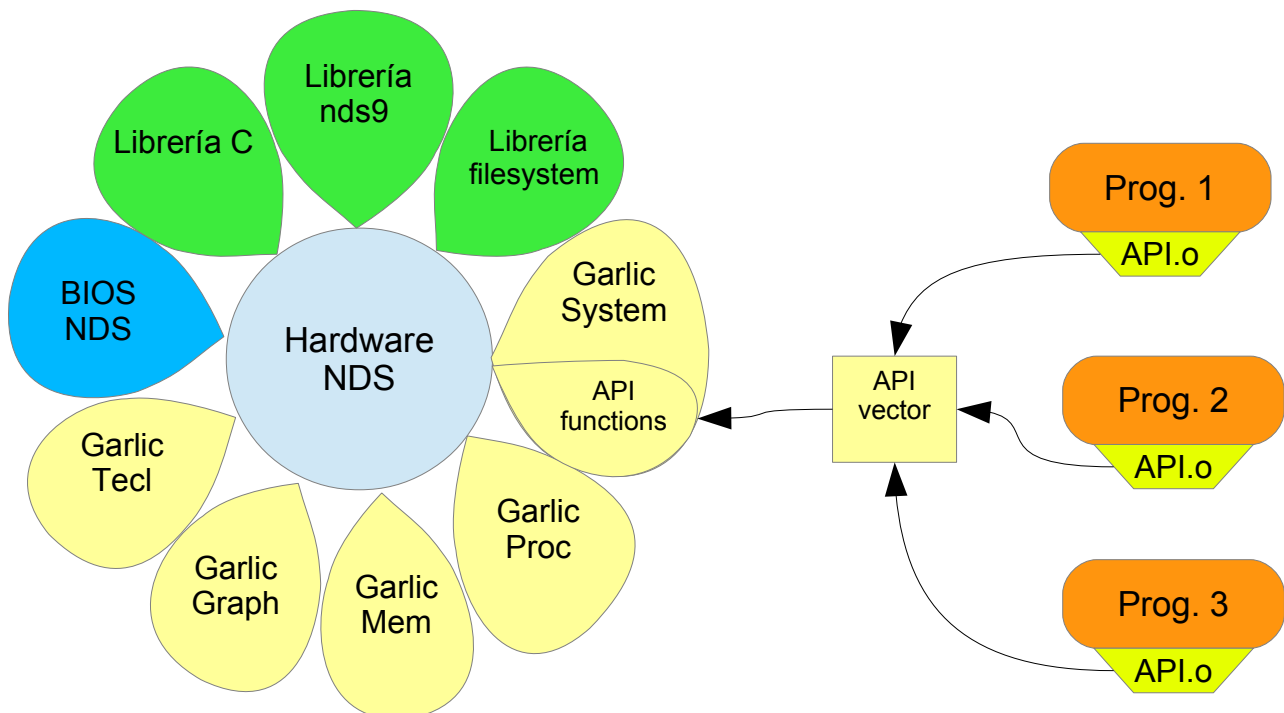
3.3 Organización del micro-kernel empotrado en la NDS

El sistema operativo a realizar se llama **GARLIC** en referencia a la organización de sus componentes dispuestos para funcionar "uno al lado del otro", como los dientes de una cabeza de ajo.

Se trata de un concepto opuesto a otro sistema operativo creado en un entorno universitario cuyo nombre es **ONION**, en referencia a una organización de sus componentes dispuestos para funcionar "uno encima del otro", como las capas de una cebolla.

El hecho de cambiar de estrategia no es fortuito: el sistema **ONION** estaba diseñado para ejecutarse sobre ordenadores personales genéricos (PCs de los años 90). El sistema **GARLIC**, sin embargo, está diseñado para ejecutarse sobre un computador móvil con muchas restricciones, es decir, sobre un **sistema empotrado**.

La organización de componentes que propone GARLIC es la siguiente:



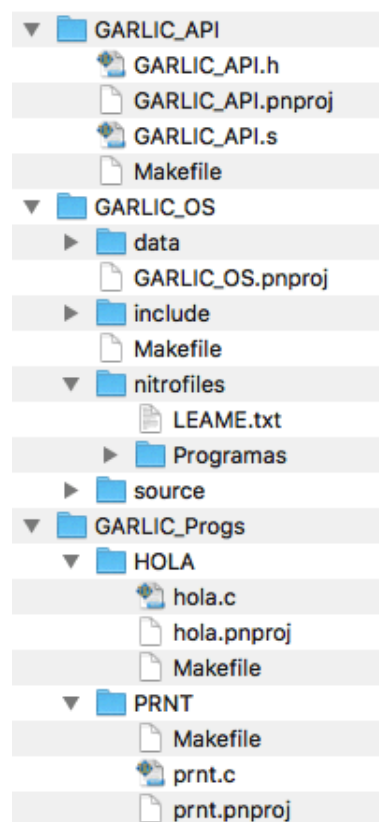
- **Hardware NDS:** es el *hardware* de la plataforma, incluyendo los procesadores ARM, la memoria, los controladores y dispositivos de Entrada/Salida y todos los demás elementos necesarios para la gestión del *hardware*.
- **BIOS NDS:** es el código almacenado en una memoria ROM interna de la NDS, el cual se ejecuta en determinadas tareas de bajo nivel, como por ejemplo, en la gestión de las excepciones.
- **Librería C:** son las funciones de la librería estándar del lenguaje C codificadas para la plataforma NDS, que el compilador y *linkador* enlazarán automáticamente al proyecto del sistema operativo GARLIC.
- **Librería nds9:** son las funciones definidas por la librería *libnds* para ejecutarse en el procesador ARM9 de la NDS, que se utilizarán en el proyecto del sistema operativo, por ejemplo, para la parte gráfica.
- **Librería filesystem:** son las funciones de la librería para gestionar el sistema de ficheros *Nitro*, que se utilizarán en el proyecto para la carga de ficheros ejecutables en formato ELF.
- **Garlic System/API:** representa todas las funciones y estructuras de datos del sistema operativo GARLIC que no pertenecen a los otros módulos, incluida la interfaz de las funciones del API.
- **Garlic Proc:** todas las funciones y estructuras de datos relacionadas con la gestión de procesos (creación y multiplexación de procesos, cola de procesos preparados y bloqueados, etc.).
- **Garlic Mem:** todas las funciones y estructuras de datos relacionadas con la carga en memoria de los programas ejecutables en formato ELF y la gestión de la memoria ocupada por los procesos resultantes.
- **Garlic Graph:** todas las funciones y estructuras de datos relacionadas con la gestión de las ventanas de texto.
- **Garlic Tecl:** todas las funciones y estructuras de datos relacionadas con la gestión de la entrada de texto por parte del usuario.
- **API vector:** vector con las direcciones de las funciones del API de GARLIC.
- **Programas:** son los programas que se podrán ejecutar sobre el sistema operativo, los cuales utilizan un vector de direcciones para invocar las rutinas del API de GARLIC.

Los programas no son parte del *microkernel*, puesto que se ejecutan **encima** de dicho *microkernel*. De hecho, los programas **no** pueden

interactuar con ningún otro módulo, ni siquiera con las funciones estándar de C, o sea, que no pueden llamar a la `printf()`, por ejemplo. En realidad, los programas no se pueden considerar parte del sistema operativo, sino que en realidad son la "información" de entrada de dicho sistema operativo, cuya tarea principal es ejecutarlos.

3.4 Proyectos involucrados en la fase 1

La estructura de directorios de la primera fase de la práctica es la siguiente:



Los directorios **GARLIC_API**, **GARLIC_OS**, **GARLIC_Progs/HOLA** y **GARLIC_Progs/PRNT** contienen ficheros `*.pnproj` que indican el contenido de los proyectos para el editor **Programmer's Notepad**. El propósito de cada proyecto es el siguiente:

- **GARLIC_API**: creación del fichero objeto `GARLIC_API.o`, que se enlazará con los programas para llamar al API de GARLIC.
- **GARLIC_OS**: definición del sistema operativo GARLIC, obteniendo como resultado el fichero para NDS `garlic_os.nds`; dentro del directorio

`nitrofiles/Programas` se guardaran los ficheros `*.elf` correspondientes a los programas que se ejecutarán sobre GARLIC.

- `GARLIC_Progs/HOLA`: primer programa de ejemplo para la fase 1; imprime el mensaje "Hello World!" un número aleatorio de veces, según el valor de un argumento,
- `GARLIC_Progs/PRNT`: segundo programa de ejemplo para la fase 1; imprime diversos mensajes para realizar un juego de pruebas intensivo de la función `GARLIC_printf()`.

3.5 El interfaz de funciones para los programas (API)

El **API** (*Application Program Interface*) son todas las funciones del sistema operativo que los programas pueden llamar para realizar diversas tareas comunes. En el fichero `GARLIC_API.h` del proyecto `GARLIC_API` se describen las 4 funciones para la primera fase (ver el fichero para más detalles):

```
/* GARLIC_pid: devuelve el identificador del proceso actual */
extern int GARLIC_pid();

/* GARLIC_random: devuelve un número aleatorio de 32 bits */
extern int GARLIC_random();

/* GARLIC_divmod: calcula la división num / den (numerador /
denominador) */
extern int GARLIC_divmod(unsigned int num, unsigned int den,
                        unsigned int * quo, unsigned int * mod);

/* GARLIC_printf: escribe string en la ventana del proceso actual
*/
extern void GARLIC_printf(char * format, ...);
```

La implementación de estas funciones se encuentra en el fichero **GARLIC_API.S**:

```
.text
    .arm
    .align 2

    .global GARLIC_pid
GARLIC_pid:
    push {r4, lr}
    mov r4, #0                @; vector base de rutinas API
    mov lr, pc                @; guardar dirección de retorno
    ldr pc, [r4]              @; llamada indirecta a rutina 0x00
    pop {r4, pc}

    .global GARLIC_random
GARLIC_random:
    push {r4, lr}
    mov r4, #0
    mov lr, pc
    ldr pc, [r4, #4]          @; llamada indirecta a rutina 0x01
    pop {r4, pc}

    .global GARLIC_divmod
GARLIC_divmod:
    push {r4, lr}
    mov r4, #0
    mov lr, pc
    ldr pc, [r4, #8]          @; llamada indirecta a rutina 0x02
    pop {r4, pc}

    .global GARLIC_printf
GARLIC_printf:
    push {r4, lr}
    mov r4, #0
    mov lr, pc
    ldr pc, [r4, #12]         @; llamada indirecta a rutina 0x03
    pop {r4, pc}
```

Todas las funciones se limitan a invocar una rutina a través de un **vector de direcciones**, que contendrá las direcciones definitivas de las rutinas del API de GARLIC. Esto permite llamar a dichas rutinas de forma independiente de su ubicación real en memoria, la cual **cambia** cada vez que se modifica el código de las rutinas.

Concretamente, este vector se alojará en las primeras posiciones de la memoria ITCM, y su definición se encuentra en el fichero `garlic_vectors.s` del proyecto `GARLIC_OS`:

```
.section .vectors,"a",%note

APIVector:                @; Vector de direcciones de rutinas del API
    .word _ga_pid          @; (código de rutinas en "garlic_itcm_api.s")
    .word _ga_random
    .word _ga_divmod
    .word _ga_printf
```

Dentro del fichero `garlic_itcm_api.s` se encuentra la implementación de las rutinas del API, de ahí el prefijo `"_ga_"`. Por ejemplo, la implementación de la rutina `_ga_printf()` es similar a la siguiente:

```
.global _ga_printf
@;Parámetros
@; R0: char * format,
@; R1: unsigned int val1 (opcional),
@; R2: unsigned int val2 (opcional)
_ga_printf:
    push {r4, lr}
    ldr r4, =_gd_pidz      @; R4 = dirección _gd_pidz
    ldr r3, [r4]
    and r3, #0x3           @; R3 = ventana de salida (zócalo actual MOD 4)
    bl _gg_escribir
    pop {r4, pc}
```

La rutina `_ga_printf()` añade un parámetro adicional sobre el registro `R3` (número de zócalo módulo 4) e invoca a la función `_gg_escribir()` de escritura de mensajes, que tendrá que programar el programador `progG`.

3.6 Estructura de un programa para GARLIC

Un programa compatible con el sistema GARLIC solo puede llamar a funciones del API de GARLIC, además de llamar a sus propias funciones. El programa `hola.c` del proyecto `GARLIC_Progs/HOLA` es uno de los ejemplos que se proporciona en la fase 1:

```
/*-----
   "HOLA.c" : primer programa de prueba para el sistema operativo
   GARLIC 1.0;

   Imprime el típico mensaje "Hello world!" por una ventana de
   GARLIC, un número aleatorio de veces, dentro de un rango entre 1 y 10
   elevado al argumento ([0..3]), es decir, hasta 1, 10, 100 o 1000
   iteraciones.
   -----*/
#include <GARLIC_API.h> /* definición de las funciones API de GARLIC */

int _start(int arg)      /* función de inicio : no se usa 'main' */
{
    unsigned int i, j, iter;

    if (arg < 0) arg = 0;           // limitar valor máximo y
    else if (arg > 3) arg = 3;      // mínimo del argumento
                                   // escribir mensaje inicial
    GARLIC_printf("-- Programa HOLA - PID (%d) --\n", GARLIC_pid());

    j = 1;                         // j = cálculo de 10 elevado a arg
    for (i = 0; i < arg; i++)
        j *= 10;
        // cálculo aleatorio del número de iteraciones 'iter'
    GARLIC_divmod(GARLIC_random(), j, &i, &iter);
    iter++;                       // asegurar que hay al menos una iteración

    for (i = 0; i < iter; i++)      // escribir mensajes
        GARLIC_printf("(%d)\t%d: Hello world!\n", GARLIC_pid(), i);

    return 0;
}
```

Este programa escrito en lenguaje C **no** puede utilizar las funciones típicas como la `printf()`, ya que no se puede compilar con la librería estándar de C.

Sí que puede (y debe) llamar a las funciones del API de GARLIC, como la `GARLIC_printf()`, para que realmente sea compatible con el sistema operativo.

Atención: la función `GARLIC_printf()` **no** es un simple cambio de nombre de la función de C estándar `printf()`, sino que es una versión específica e independiente para el sistema operativo que estamos diseñando. El programador de gráficos tendrá que implementar todo el código necesario para que la función `GARLIC_printf()` realice su cometido.

Un programa para GARLIC no puede empezar con la función `main()`, como es típico del lenguaje C, sino que debe empezar con la función `_start()`, que es la función de inicio de los programas ejecutables en formato ELF.

Además, los programas para GARLIC recibirán un argumento de tipo *int* por parámetro, a través del registro `R0`, el cuál permitirá modificar el comportamiento del programa según del valor de dicho argumento (entre 0 y 3).

Se debe incluir un único fichero de cabeceras, el `GARLIC_API.h`, que se ha descrito en el apartado anterior, para que el compilador reconozca las funciones del API de GARLIC.

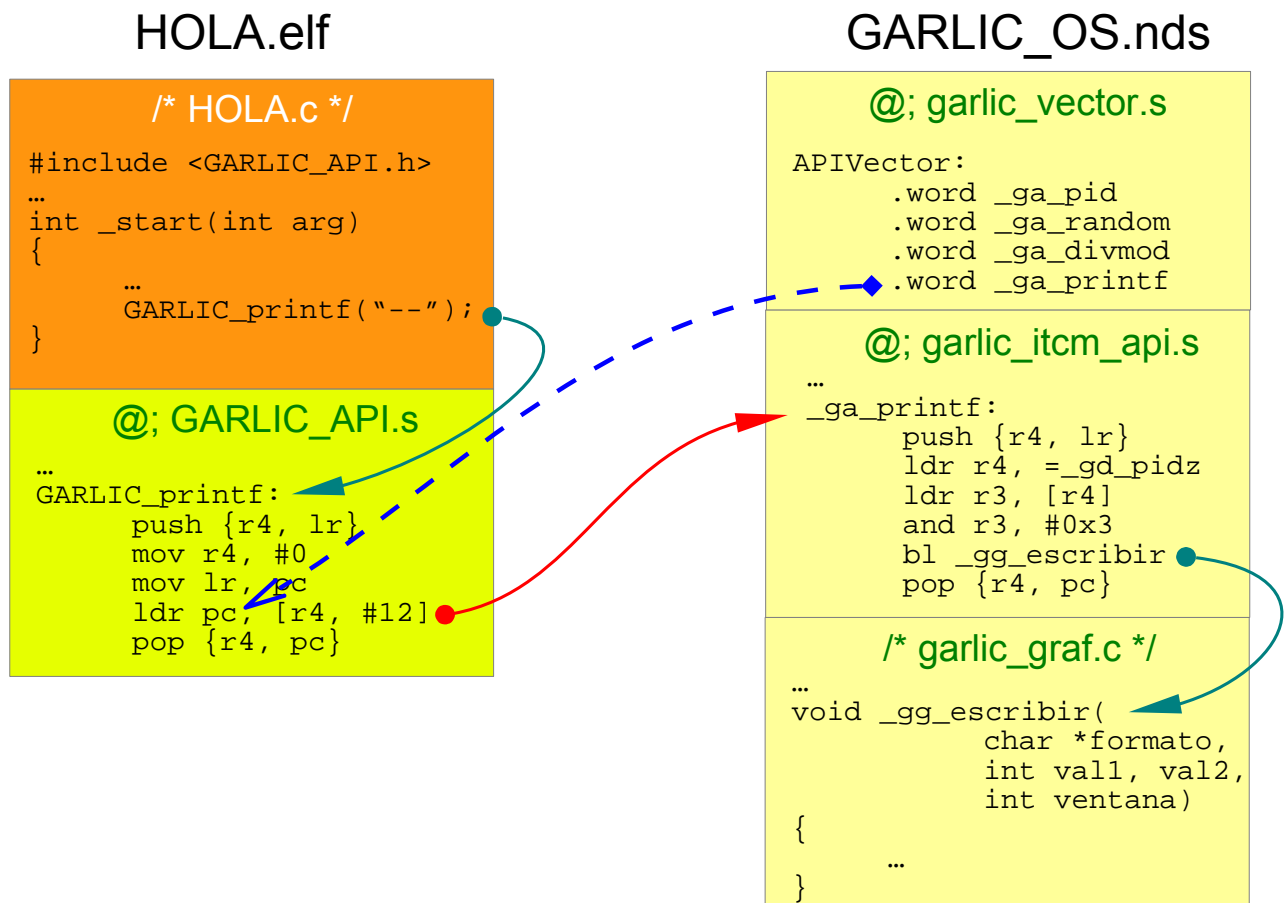
Para que el programa pueda llamar realmente a las rutinas del API de GARLIC, se debe enlazar con el fichero `GARLIC_API.o`, que se genera a partir del fichero `GARLIC_API.s`. De las tareas de enlazado (*linking*) se encargan las siguientes líneas del fichero `Makefile` del proyecto `GARLIC_Progs/HOLA`:

```
#-----  
# make commands  
#-----  
  
$(TARGET).elf : $(TARGET).o  
    @arm-none-eabi-ld $(LDFLAGS) $(TARGET).o  
    $(GARLICAPI)/GARLIC_API.o -o $(TARGET).elf
```

donde `$(TARGET)` se sustituye por el nombre del proyecto, en este caso "HOLA", y `$(GARLICAPI)` indica el directorio que contiene el fichero `GARLIC_API.o`, en este caso `../../GARLIC_API`, es decir, se supone que dos directorios hacia arriba del proyecto actual (directorios padre) se encuentra el directorio `GARLIC_API` que contiene el fichero objeto externo a enlazar.

De este modo, se enlazará el fichero **HOLA.o** con **GARLIC_API.o** para formar el fichero ejecutable **HOLA.elf**, aunque dicho ejecutable solo podrá funcionar sobre el sistema operativo GARLIC.

A continuación se muestra un esquema que ilustra todo el proceso de llamada de una función de sistema operativo GARLIC desde un programa:



La idea fundamental detrás del vector de direcciones es que un programa que se ha compilado y *linkado* “fuera” del proyecto del sistema operativo pueda llamar a una rutina de la cual **es imposible saber su dirección** inicial en el momento de generar dicho programa. El vector de direcciones es, pues, un lugar de referencia donde ir a buscar dicha dirección.

Un último detalle a tener en cuenta es que el fichero **Makefile** está preparado para generar la versión en ensamblador del programa (con la opción “-s” del compilador), es decir, primero se genera el fichero **HOLA.s** a partir del fichero **HOLA.c**, y después se ensambla el **HOLA.s** para obtener el fichero **HOLA.o**, que es el que se utiliza por el enlazador para generar el fichero ejecutable final **HOLA.elf**.

El propósito de esta secuencia es tener acceso al código en ensamblador de los programas para GARLIC, para poder realizar un seguimiento de su ejecución desde el depurador, ya que resulta imposible mostrar el código fuente en C de un programa que se ejecuta sobre GARLIC, puesto que el depurador solo reconocerá el código fuente del propio sistema operativo.

3.7 Estructuras de datos de GARLIC

Las estructuras de datos del sistema operativo de acceso más frecuente se almacenarán en la memoria DTCM, y comenzarán con el prefijo "_gd_". Estas estructuras de datos se encuentran ya declaradas en el fichero `garlic_dtcn.s` del proyecto `GARLIC_OS`:

```
.section .dtcm,"wa",%progbits

    .global _gd_pidz    @; Identificador de proceso + zócalo actual
_gd_pidz:    .word 0

    .global _gd_pidCount    @; Contador global de PIDs
_gd_pidCount:    .word 0

    .global _gd_tickCount    @; Contador global de tics
_gd_tickCount:    .word 0

    .global _gd_seed    @; Semilla para números aleatorios
_gd_seed:    .word 0xFFFFFFFF

    .global _gd_nReady    @; Número de procesos en cola de READY
_gd_nReady:    .word 0

    .global _gd_qReady    @; Cola de READY
_gd_qReady:    .space 16

    .global _gd_pcbs    @; Vector de PCBs de procesos activos
_gd_pcbs:    .space 16 * 6 * 4

    .global _gd_wbfs    @; Vector de WBUFs de las ventanas
_gd_wbfs:    .space 4 * (4 + 32)

    .global _gd_stacks    @; Pilas de los procesos activos
_gd_stacks:    .space 15 * 128 * 4
```

Para obtener una descripción más detallada del propósito de cada variable se puede consultar el fichero **garlic_system.h**:

```
//-----
//   Variables globales (garlic_dtcn.s)
//-----

extern int _gd_pidz;    // Identificador de proceso (PID) + zócalo
                        // (PID en 28 bits altos, zócalo en 4 bits bajos,
                        // cero si se trata del propio sistema operativo)

extern int _gd_pidCount;    // Contador de PIDs

extern int _gd_tickCount;  // Contador de tics

extern int _gd_seed;       // Semilla para números aleatorios

extern int _gd_nReady;     // Número de procesos en cola de READY

extern char _gd_qReady[16]; // Cola de READY (procesos preparados)

typedef struct            // Estructura del bloque de control de un proceso
{
    int PID;              // (PCB: Process Control Block)
                        // identificador del proceso
    int PC;               // contador de programa
    int SP;               // puntero al top de pila
    int Status;           // estado del procesador (CPSR)
    int keyName;          // nombre en clave del proceso
    int workTicks;        // contador de ciclos de trabajo (24
                        // bits bajos), % Uso CPU (8 bits altos)
} PACKED garlicPCB;

extern garlicPCB _gd_pcbs[16];    // vector con los PCBs

typedef struct                // Estructura del buffer de una ventana
{
    int pControl;             // (WBUF: Window BUffer)
                        // control de escritura en ventana
                        // 16 bits altos: número de línea (0-23)
                        // 16 bits bajos: caracteres pendientes (0-32)
    char pChars[32];          // vector de 32 caracteres pendientes
                        // indicando el código ASCII de cada posición
} PACKED garlicWBUF;

extern garlicWBUF _gd_wbfs[4];    // buffers de 4 ventanas

extern int _gd_stacks[15*128];    // vector con las pilas
```

La variable más referenciada será la `_gd_pidz`, que contiene el identificador de proceso y el número de zócalo asociado al proceso que se está ejecutando en cada momento. Para el proceso de control del sistema operativo, este valor es cero (`PID = 0, z = 0`).

Otra estructura importante es el vector de PCBs `_gd_pcbs[]`, es decir, las estructuras que contienen información sobre el contexto del proceso alojado en cada zócalo. El zócalo cero siempre contiene el proceso de control del sistema operativo. Los otros 15 zócalos son para ejecutar los procesos del usuario, y pueden estar libres u ocupados; si un zócalo de proceso de usuario está libre, su `PID` será 0, si está ocupado, su `PID` será mayor que 0.

Por último, cabe destacar la estructura que contiene las 15 pilas para los procesos correspondientes a los programas de usuario, `_gd_stacks[]`. Cada pila dispone de 128 posiciones de tipo *word*, es decir, 512 bytes, lo cual supone la utilización de casi la mitad del espacio de la memoria DTCM.

Por su lado, el sistema operativo utiliza 3 pilas en la zona alta de la DTCM, correspondientes a los modos de ejecución **Sistema**, **Supervisor** e **IRQ** (ver apartado 3.9). La pila del modo **Sistema** será utilizada por el proceso de control del sistema operativo, mientras que los procesos de usuario utilizarán sus respectivas pilas ubicadas en el vector `_gd_stacks[]`.

3.8 Estructura de funciones y rutinas de GARLIC

En el fichero `garlic_system.h` también se encuentran declaradas las funciones y rutinas relativas a las tres facetas básicas del sistema: procesos (`progP`), memoria (`progM`) y gráficos (`progG`). Para distinguirlas claramente, se propone el uso de los prefijos “`_gp_`”, “`_gm_`” y “`_gg_`”, respectivamente:

```
//-----
//  Rutinas de gestión de procesos (garlic_itcm_proc.s)
//-----
typedef int  (* intFunc) (int);

/* _gp_WaitForVBlank:  sustituto de swiWaitForVBlank para Garlic */
extern void _gp_WaitForVBlank();

/* _gp_IntrMain:  manejador principal de interrupciones de Garlic */
extern void _gp_IntrMain();

/* _gp_rsiVBL:  manejador de interrupciones VBL (Vertical BLank)*/
extern void _gp_rsiVBL();
```

```

/* _gp_numProc: devuelve el número de procesos cargados, incluyendo
               el proceso en RUN y los procesos en READY */
extern int _gp_numProc();

/* _gp_crearProc: prepara un proceso para ser ejecutado, creando su
                 entorno de ejecución y colocándolo en la cola de READY;
                 Parámetros:
                     funcion    -> dirección de entrada al código del proc.
                     zocalo     -> identificador del zócalo (0 - 15)
                     nombre     -> nombre en clave del programa
                     arg        -> argumento del programa
                 Resultado: 0 si no hay problema, >0 si no se puede crear el
                 proceso
*/
extern int _gp_crearProc(intFunc funcion, int zocalo, char *nombre, int
arg);

//-----
// Funciones de gestión de memoria (garlic_mem.c)
//-----

/* _gm_initFS: inicializa el sistema de ficheros, devolviendo un valor
               booleano para indiciar si dicha inicialización ha tenido éxito;
*/
extern int _gm_initFS();

/* _gm_cargarPrograma: busca un fichero de nombre "(keyName).elf"
                       dentro del directorio "/Programas/" del sistema de ficheros
                       y carga los segmentos de programa a partir de una posición
                       de memoria libre, efectuando la reubicación de las
                       referencias a los símbolos del programa, según el
                       desplazamiento del código en la memoria destino;
                 Parámetros:
                     keyName -> vector de 4 caracteres con el nombre en clave
                 Resultado:
                     != 0 -> dirección de inicio del programa (intFunc)
                     == 0 -> no se ha podido cargar el programa
*/
extern intFunc _gm_cargarPrograma(char *keyName);

```

```
//-----
//  Funciones de gestión de gráficos (garlic_graf.c)
//-----

/* _gg_iniGraf: inicializa el procesador gráfico A para GARLIC 1.0 */
extern void _gg_iniGrafA();

/* _gg_generarMarco: dibuja el marco de la ventana que se indica por
    parámetro */
extern void _gg_generarMarco(int v);

/* _gg_escribir: escribe una cadena de caracteres en la ventana
    indicada;
    Parámetros:
        formato -> cadena de formato, terminada con '\0';
                    admite '\n' (salto de línea), '\t'
                    (tabulador, 4 espacios) y códigos entre
                    32 y 159 (los 32 últimos son caracteres
                    gráficos), además de códigos de formato
                    %c, %d, %x y %s (max. 2 códigos por
                    cadena)
        val1  -> valor a sustituir en primer código de
                    formato, si existe
        val2  -> valor a sustituir en segundo código de
                    formato, si existe
                - los valores pueden ser un código ASCII (%c),
                  un valor atural de 32 bits (%d, %x) o un
                  puntero a string (%s)
        ventana -> número de ventana (de 0 a 3)
*/
extern void _gg_escribir(char *formato, unsigned int val1,
                        unsigned int val2, int ventana);
```

Algunas de estas funciones y rutinas ya se encuentran implementadas, como `_gp_WaitForVBlank()` o `_gm_initFS()`, pero el resto se encuentran solo declaradas, puesto que constituyen el trabajo a realizar.







Las rutinas relativas a **progP** serán todas en lenguaje ensamblador, mientras que las funciones relativas a **progM** y **progG** se escribirán en lenguaje C, a excepción de algunas rutinas de soporte en lenguaje ensamblador que se comentarán más en detalle en los apartados dedicados a las tareas de cada programador. Las rutinas de **progT** serán prácticamente todas en lenguaje ensamblador, aunque también se definirán algunas funciones en C.

3.9 Distribución en memoria de los componentes del sistema

Para terminar la sección de especificaciones generales, a continuación se muestran los mapas de memoria donde se ubicarán los componentes del sistema operativo GARLIC.

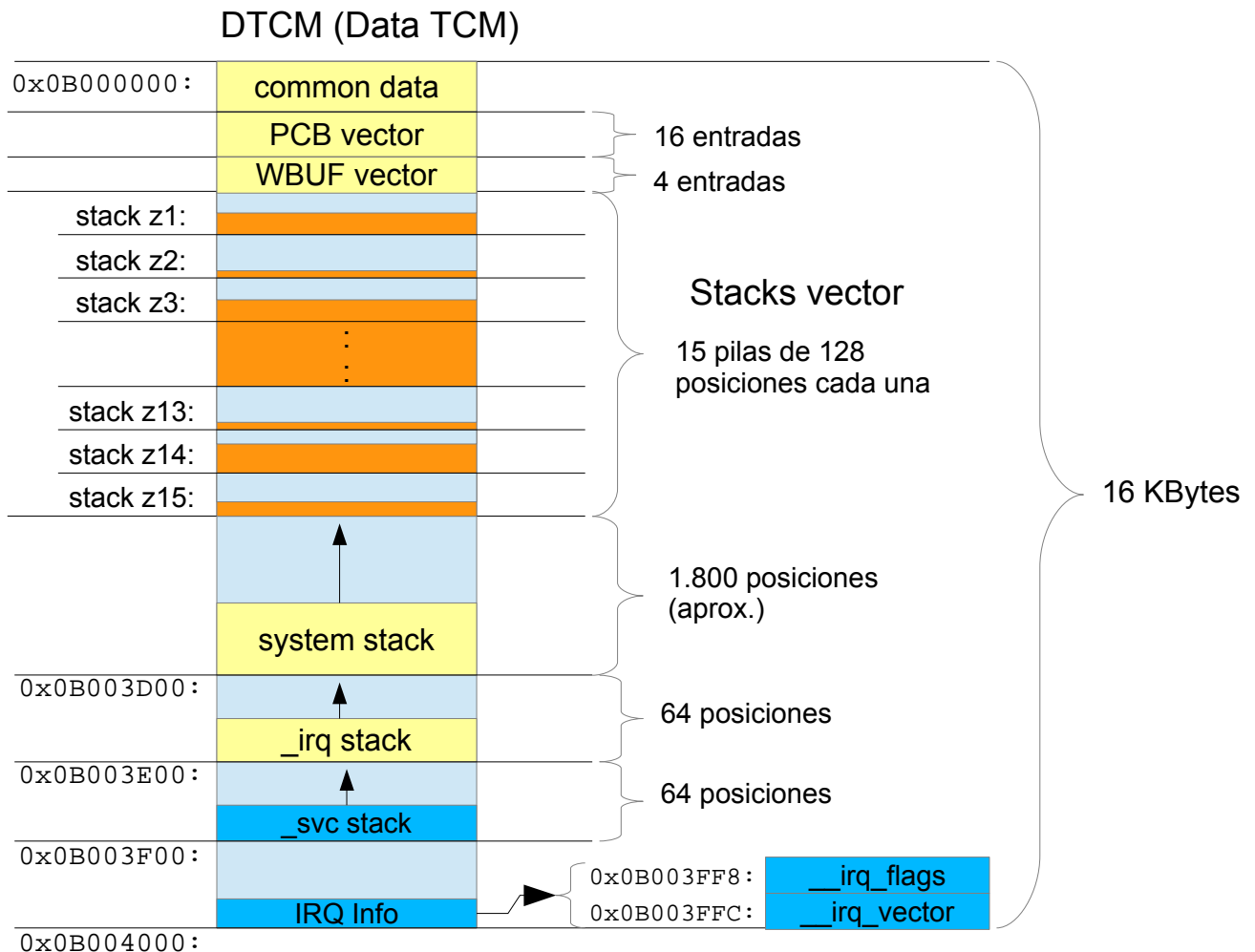
Los nuevos mapas se pueden comparar con los mostrados en el apartado 2.4, pero el código de colores se ha variado ligeramente para distinguir el sistema operativo, que sustituye al *"user program"*, y los programas para GARLIC (procesos de usuario), que introducen un nuevo color (naranja):

Código de colores:

	→	Memoria libre
	→	BIOS
	→	Librerías (C, libnds, etc.)
	→	Gestión de IRQs
	→	Sistema GARLIC
	→	Programas para GARLIC

3.9.1 Data Tightly Coupled Memory

En esta zona se añadirán las variables principales de trabajo de GARLIC:

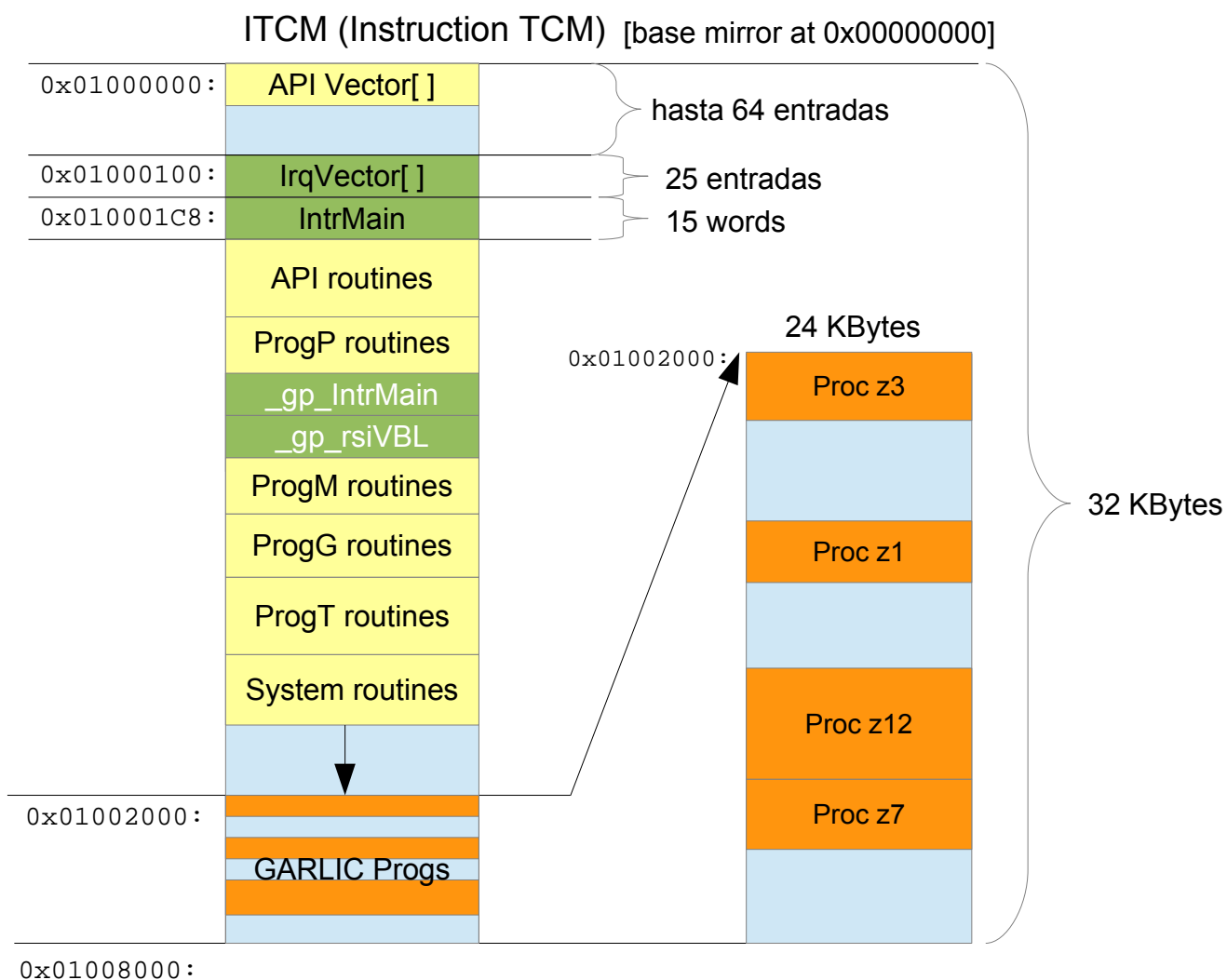


En la nueva configuración de la memoria DTCM, se han utilizado un poco más de la mitad de la memoria (8 KBytes) en datos del sistema y pilas de los procesos de GARLIC (z1-z15).

La pila "user stack" del mapa del apartado 2.4.1 pasa a denominarse "system stack" porque es la pila para el proceso de control del sistema operativo (z0). Aunque inicialmente dispone de muchas posiciones, este espacio se irá reduciendo a medida que se vayan añadiendo nuevas variables de sistema, sobretodo en la fase 2, pero se supone que, al menos, dispondrá del mismo espacio que el resto de las pilas, es decir, 128 posiciones (words).

3.9.2 *Instruction Tightly Coupled Memory*

En esta zona se añadirán el vector de direcciones y el código de las rutinas del API, así como el código de las RSIs y de las rutinas en ensamblador de todos los programadores y del sistema operativo. Además, se debe alojar el código y los datos de todos los procesos de usuario:

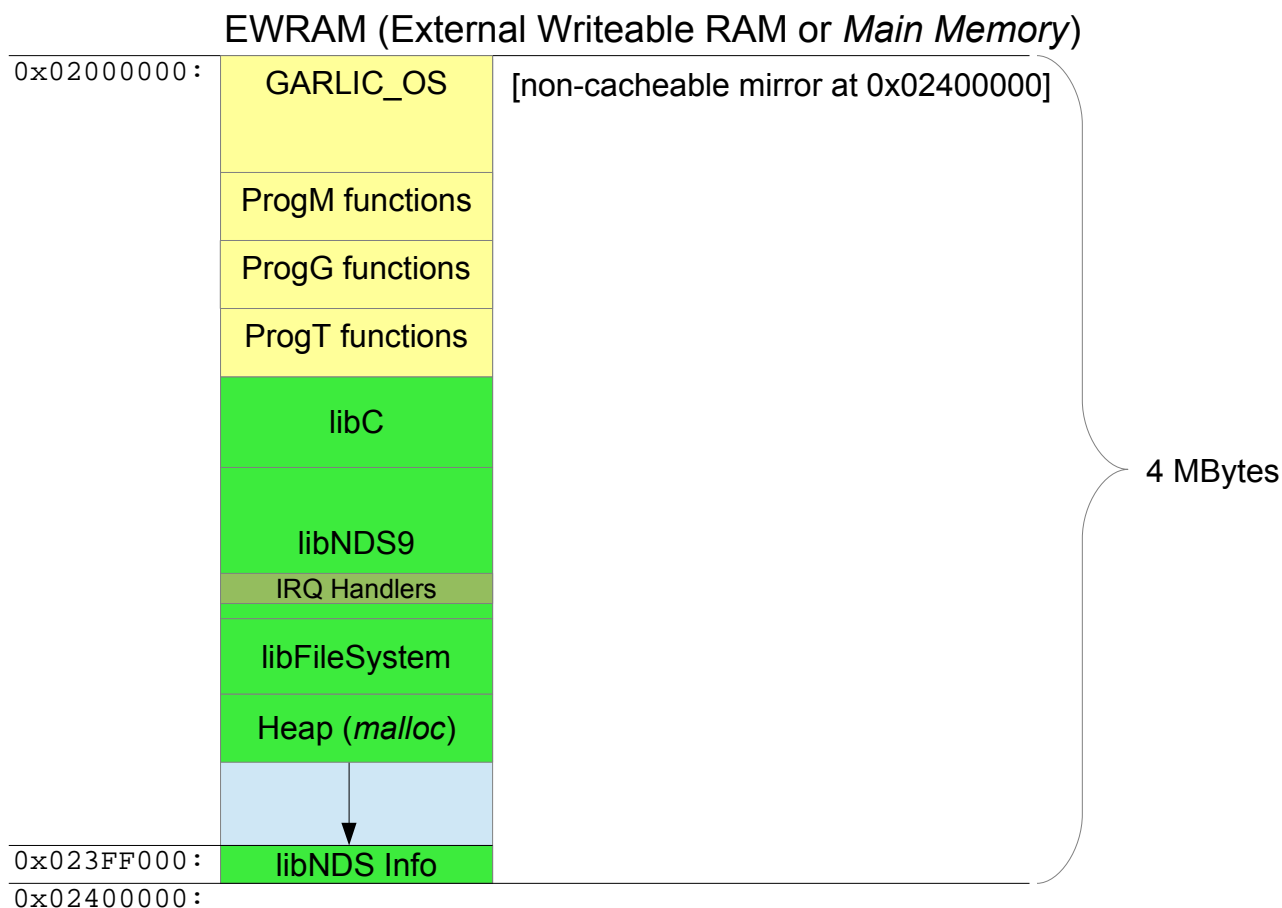


Se puede observar que se conserva el vector de direcciones de las RSIs definido por **libnds** `IrqVector[]` y la rutina principal de gestión de interrupciones `IntrMain()`, aunque esta última ya no se utilizará puesto que se instalará la rutina del propio sistema operativo `_gp_IntrMain()`.

Atención: se supone que todo el código en ensamblador del sistema operativo **no superará los 8 Kbytes**, ya que hay que reservar 24 Kbytes para los procesos de usuario. De otro modo, habrá que mover parte del código del sistema operativo a la memoria principal.

3.9.3 Main Memory

En la memoria principal se ubicará el código del sistema operativo que no se encuentra en la zona ITCM, es decir, el código escrito en lenguaje C (código del proceso de control del sistema operativo más el código en C de los programadores progM, progG y progT):



En definitiva, se pretende colocar el máximo de código posible dentro de las zonas de memoria DTCM e ITCM de la NDS, con el fin de optimizar al máximo el rendimiento del *hardware*. Sin embargo, algunas partes del sistema se programarán en C para aprovechar las funcionalidades que ofrecen las librerías **libC** y **libFileSystem** para gestionar el acceso al sistema de ficheros, así como algunas funciones de la librería **libNDS9** que facilitan la gestión del *hardware*, como por ejemplo la configuración y control de los procesadores gráficos.

4 Tareas de gestión del procesador (progP)

4.1 Rutina principal de gestión de interrupciones de GARLIC

El sistema GARLIC dispone de su propia rutina principal de gestión de interrupciones IRQ, cuyo código es el siguiente:

```
.global _gp_IntrMain
_gp_IntrMain:
    mov    r12, #0x4000000
    add    r12, r12, #0x208
    ldr    r2, [r12, #0x08]    @; R2 = REG_IE (interr. permitidas)
    ldr    r1, [r12, #0x0C]    @; R1 = REG_IF (interr. activadas)
    and    r1, r1, r2          @; filtrar int. activadas con permitidas
    ldr    r2, =irqTable
.Lintr_find:                  @; buscar manejadores de interrupciones específicos
    ldr    r0, [r2, #4]        @; R0 = máscara de int. del manejador
    cmp    r0, #0              @; si máscara = cero, fin de vector
    beq    .Lintr_setflags
    ands   r0, r0, r1          @; determinar si manejador atiende a la
    beq    .Lintr_cont1        @; interrupción activada
    ldr    r3, [r2]            @; R3 = dirección de salto del manejador
    cmp    r3, #0
    beq    .Lintr_ret          @; abandonar si dirección = 0
    mov    r2, lr              @; guardar dirección de retorno
    blx    r3                  @; invocar el manejador
    mov    lr, r2              @; recuperar dirección de retorno
    b      .Lintr_ret          @; salir del bucle de búsqueda
.Lintr_cont1:
    add    r2, r2, #8
    b      .Lintr_find
.Lintr_ret:
    mov    r1, r0              @; indica qué interrupción se ha servido
.Lintr_setflags:
    str    r1, [r12, #0x0C]
    ldr    r0, =__irq_flags
```

```

ldr    r3, [r0]
orr    r3, r3, r1
str    r3, [r0]
mov    pc, lr @; retornar al gestor de la excepción IRQ de la BIOS

```

En esencia, la rutina busca el manejador de interrupciones específico para una de las IRQs que se encuentre activada.

Para instalar esta rutina y reemplazar la de la librería **libnds**, desde el programa principal **main.c** se invocará la siguiente llamada:

```
irqInitHandler(_gp_IntrMain);
```

Esto provoca que la dirección de la función `_gp_IntrMain()` se copie en la posición de memoria `_irq_vector`, de modo que el gestor de excepciones IRQ de la BIOS invocará a la rutina de principal de gestión de interrupciones de GARLIC cada vez que se provoque cualquier interrupción (ver apartados 2.2 y 2.3).

Otro efecto que provoca la instalación del nuevo gestor de interrupciones es que todas las RSI que estaban instaladas anteriormente quedarán desactivadas (pero no borradas). Este comportamiento de la función `irqInitHandler()` es lógico porque podría ocurrir que el nuevo gestor no fuese compatible con las antiguas RSIs.

En efecto, esto es cierto para las RSIs de control de la comunicación *fifo*, `fifoInternalRecvInterrupt()` / `fifoInternalSendInterrupt()`, ya que dichas RSIs requieren que la rutina principal de gestión de interrupciones haya reactivado el *flag I* del registro de estado del procesador CPSR (I=0), con el fin de que se permitan las **interrupciones anidadas** o reentrantes (ver apartado 2.5).

En principio, el proceso de multiplexación de procesos GARLIC **no** puede permitir interrupciones anidadas, puesto que el manejo de los contextos de programa requeriría que **cada proceso dispusiera de varias pilas (sistema, supervisor, IRQ)**, lo cual incrementaría considerablemente la complejidad del proyecto.

Por este motivo, se ha decidido que las RSIs de comunicación *fifo* queden desactivadas, a costa de **perder la comunicación con el procesador ARM7**. Esto significa que el sistema GARLIC no podrá utilizar la pantalla táctil, por ejemplo, ni los botones **X** e **Y**, a excepción de que se defina un protocolo específico de comunicación con el ARM7 (habrá que reprogramar dicho procesador).

4.2 Rutina de Servicio de la Interrupción IRQ_VBL

GARLIC utilizará la interrupción por retroceso vertical de la pantalla (*Vertical Blank*) para realizar el intercambio de procesos. Esto significa que se tendrá que programar e instalar la RSI para dicha interrupción. Esta RSI se denominará `_gp_rsiVBL()`, y se instalará del siguiente modo:

```
irqSet(IRQ_VBLANK, _gp_rsiVBL);  
irqEnable(IRQ_VBLANK);
```

El cuerpo de la RSI se tendrá que programar en lenguaje ensamblador dentro del fichero `garlic_itcm_proc.s`, en el siguiente contexto:

```
.global _gp_rsiVBL  
@; Manejador de interrupciones VBL (Vertical BLank) de Garlic:  
@; se encarga de actualizar los tics, intercambiar procesos, etc.  
_gp_rsiVBL:  
    push {r4-r7, lr}  
  
    pop {r4-r7, pc}  
  
    @; Rutina para salvar el estado del proceso interrumpido en la  
    @; entrada correspondiente del vector _gd_pcb  
_gp_salvarProc:  
    push {r8-r11, lr}  
  
    pop {r8-r11, pc}  
  
    @; Rutina para restaurar el estado del siguiente proceso en la  
    @; cola de READY  
_gp_restaurarProc:  
    push {r8-r11, lr}  
  
    pop {r8-r11, pc}
```

Básicamente, el trabajo de la RSI para IRQ_VBL consiste en los siguientes puntos:

- incrementar el contador de tics general `_gd_tickCount`,
- detectar si existe algún proceso pendiente en la cola de *Ready*: en caso negativo, la RSI finalizará sin cambio de contexto,
- si el proceso actual a desbancar es el del sistema operativo, pasar a salvar el contexto del proceso (penúltimo punto de esta lista),
- si el proceso actual a desbancar no es el del sistema operativo pero su PID es cero, significará que se trata de un proceso de programa que ha terminado su ejecución; en este caso **no** hay que salvar el contexto del proceso actual (saltar al último punto de la lista),
- salvar el contexto del proceso actual,
- restaurar el proceso del siguiente proceso de la cola de *Ready*.

4.3 Rutinas de salvar y restaurar contexto

Las rutinas `_gp_salvarProc()` y `_gp_restaurarProc()` tienen por objetivo salvar y restaurar el contexto de los procesos que se intercambian entre el estado de *Run* y la cola de *Ready*.

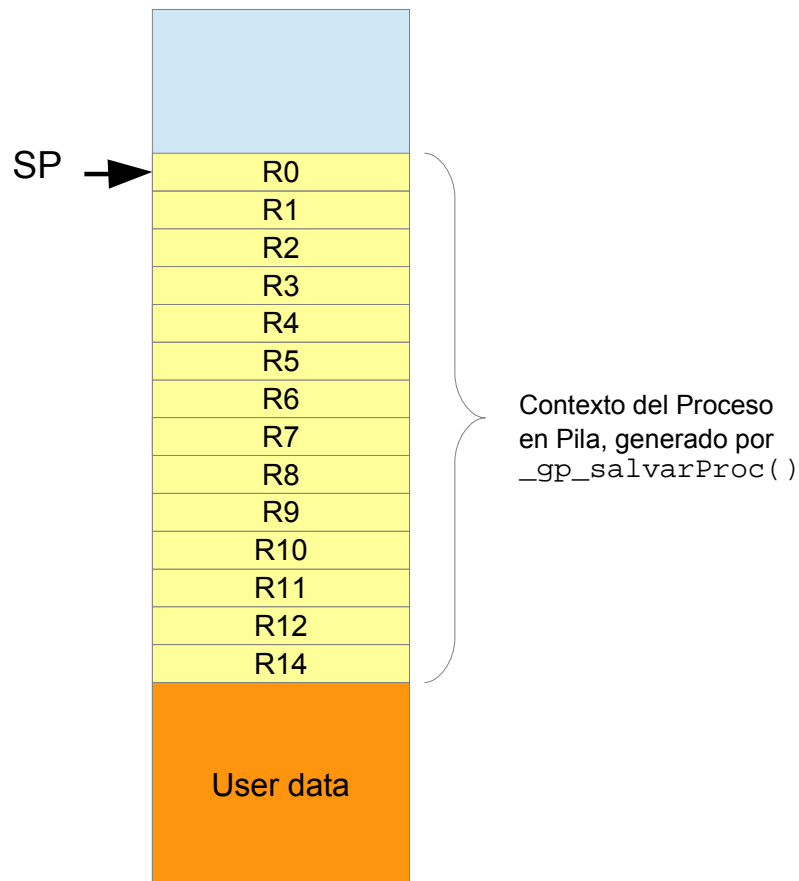
El proceso que está en *Run* es el proceso que se está ejecutando actualmente, cuyo PID y zócalo se encuentran almacenados en la variable `_gd_pidz`.

Los procesos que están preparados para ejecutarse tienen su número de zócalo en la cola de *Ready*.

Cuando se produce un cambio de contexto (*swap*), el **primer proceso** de la cola de *Ready* pasa a *Run*, y el proceso que estaba en *Run* pasa a la **última posición** de la cola de *Ready* (*Round Robin* sin prioridades).

El contexto de un proceso es el valor de todos los registros `R0-R15` y de la palabra de estado `CPSR`. Los registros `R0-R12` y `R14` se guardarán en la pila del proceso, mientras que los registros `R13`, `R15` y `CPSR` se guardarán en la entrada del vector `_gd_pcb[]` correspondiente al zócalo del proceso, en los campos `SP`, `PC` y `Status`, respectivamente.

El valor del `SP` guardado en la estructura `garlicPCB` del proceso tendrá que ser el *top* de la pila después de guardar en ella todos los registros de datos `R0-R12` más el `R14`, de modo que la pila de un proceso que está en *Ready* tiene la siguiente organización:

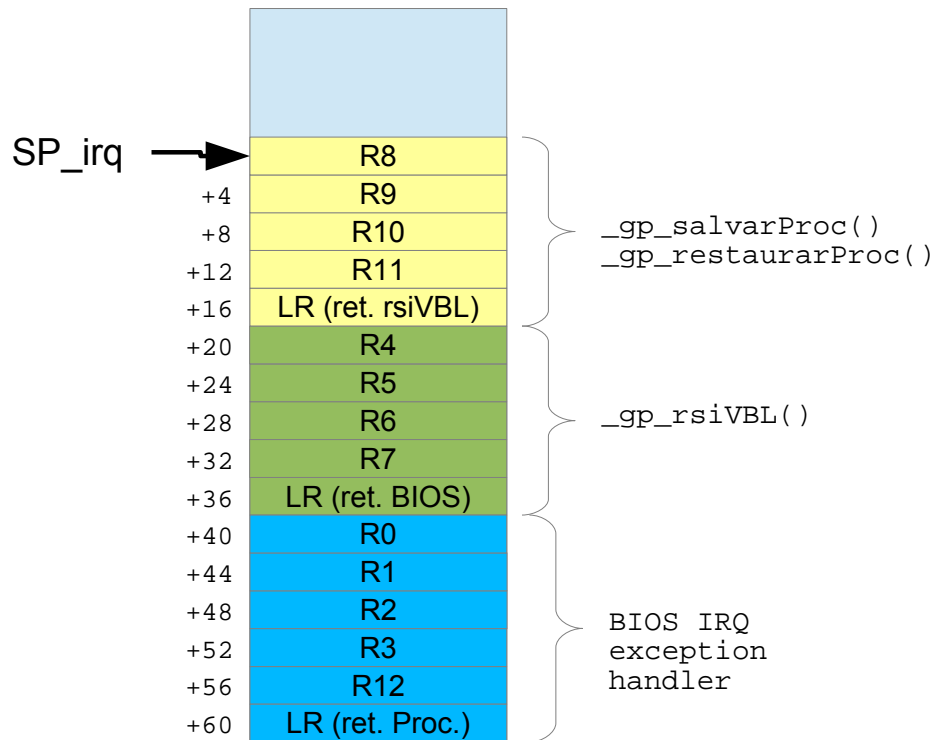


El color naranja indica el contenido de la pila que ha generado el propio proceso al utilizar las instrucciones `push` para sus tareas. El color amarillo indica la parte de la pila generada por la rutina de salvar contexto. Aunque se trata del contenido de los registros del proceso, es una información que almacena la rutina `_gp_salvarProc()`, motivo por el cual se ha utilizado el color amarillo.

La cuestión ahora es cómo la rutina de salvar el contexto del proceso puede acceder al contenido de los registros del proceso, teniendo en cuenta que, cuando se atiende a la interrupción, se ejecutan una serie de rutinas que modifican dichos registros (gestión de excepciones de la BIOS, rutina principal de gestión de interrupciones, RSI de la `IRQ_VBL`).

Además, hay que recordar que el contenido de los registros `R13` y `R14` habrá cambiado, puesto que el procesador se encontrará en modo de ejecución **IRQ** cuando ejecute la rutina de salvar contexto y que el valor del registro de estado `CPSR` estará almacenado en el registro `SPSR_irq` (apartado 2.1).

Para solucionar el problema del acceso a los registros `R0-R12`, hay que utilizar el contenido de la pila del modo **IRQ** que, dentro de las rutinas `_gp_salvarProc()` y `_gp_restaurarProc()`, tendrá la siguiente estructura:



Esta estructura viene determinada por los `push` y `pop` que se realizan en los distintos niveles de gestión de una interrupción `IRQ_VBL`. Por lo tanto, **no se permite modificar** la lista de los registros que se salvan y se restauran en las rutinas `_gp_rsiVBL()`, `_gp_salvarProc()` y `_gp_restaurarProc()`.

Para poder copiar el contenido de los registros `R0-R12` desde la pila de interrupciones hasta la pila del proceso a desbancar, será necesario copiar el contenido del `SP_irq` en otro registro (`R8`, `R9`, `R10` o `R11`), **cambiar el modo de ejecución** (modificando el `CPSR` con la instrucción `msr`) y apilar los valores de los registros en la pila de usuario.

También se tendrá que apilar el valor del `R14` (`LR`) del modo de ejecución en el que se encontraba el proceso interrumpido (típicamente en modo **sistema**). El valor del `R13` (`SP`) del modo **sistema** se podrá guardar en la estructura `garlicPCB` **después** de realizar todas estas copias.

Además, el valor del `CPSR` del proceso a desbancar estará guardado en el registro `SPSR_irq`, por lo que habrá que copiarlo a la estructura `garlicPCB` desde el modo **IRQ**.

Por último, el valor del `R15` (`PC`) del proceso interrumpido es el valor más bajo de la pila de interrupciones, es decir, `[SP_irq + 60] LR (ret. Proc.)`, que también habrá que copiarlo en la estructura `garlicPCB`.

A modo de lista (no ordenada), las tareas de `_gp_salvarProc()` son las siguientes:

- incrementar el contador de procesos pendientes `_gd_nReady`,
- guardar el número de zócalo del proceso a desbancar en la última posición de la cola de *Ready*,
- guardar el valor del `R15` del proceso a desbancar en el campo `PC` del elemento `_gd_pcbs[z]`, donde `z` es el número de zócalo del proceso a desbancar,
- guardar el `CPSR` del proceso a desbancar en el campo `Status` del elemento `_gd_pcbs[z]`,
- guardar el `SP_irq` en un registro de trabajo libre (`R8-R11`),
- cambiar al modo de ejecución del proceso interrumpido,
- apilar el valor de los registros `R0-R12 + R14` del proceso a desbancar en su propia pila,
- guardar el valor del registro `R13` del proceso a desbancar en el campo `SP` del elemento `_gd_pcbs[z]`,
- volver al modo de ejecución **IRQ** y retornar de `_gp_salvarProc()`.

En el caso de la rutina `_gp_restaurarProc()`, el proceso es el inverso, es decir:

- decrementar el contador de procesos pendientes `_gd_nReady`,
- recuperar el número de zócalo del proceso a restaurar de la primera posición de la cola de *Ready*, y desplazar el vector `_gd_qReady[]` para que la cola empiece por el zócalo del siguiente proceso a restaurar,
- construir el valor combinado `PIDz` para guardarlo en la variable global `_gd_pidz`, a partir del `PID` i número de zócalo del proceso a restaurar,
- recuperar el valor del `R15` anterior del proceso a restaurar y copiarlo en la posición correspondiente de pila del proceso,

- recuperar el `CPSR` del proceso a restaurar y copiarlo sobre el registro `SPSR_irq`,
- guardar el puntero de la pila del modo **IRQ** en un registro de trabajo libre (`R8-R11`),
- cambiar al modo de ejecución del proceso a restaurar,
- recuperar el valor del registro `R13` del proceso a restaurar,
- desapilar el valor de los registros `R0-R12 + R14` de la pila del proceso a restaurar, y copiarlos en la pila del modo **IRQ**,
- volver al modo de ejecución **IRQ** y retornar de `_gp_restaurarProc()`.

4.4 Rutina de crear proceso

La rutina `_gp_crearProc()` presenta la siguiente disposición inicial:

```
.global _gp_crearProc
@; prepara un proceso para ser ejecutado, creando su entorno de
@; ejecución y colocándolo en la cola de READY
@;Parámetros
@; R0: intFunc funcion,
@; R1: int zocalo,
@; R2: char *nombre
@; R3: int arg
@;Resultado
@; R0: 0 si no hay problema, >0 si no se puede crear el proceso
_gp_crearProc:
    push {lr}

    pop {pc}
```

Las tareas que debe realizar son las siguientes (no tienen que seguir este orden):

- rechazar la llamada si zócalo = 0 (reservado para sistema operativo), o si el zócalo ya está ocupado por otro proceso, lo cual se verifica consultando el campo `PID` del elemento `_gd_pcb[s]`, teniendo en cuenta que un zócalo está libre si su PID es 0,
- obtener un PID para el nuevo proceso, incrementando la variable global `_gd_pidCount`, y guardarlo en el campo `PID` del `_gd_pcb[s]`,
- guardar la dirección de la rutina inicial del proceso (primer parámetro) en el campo `PC` del elemento `_gd_pcb[s]`, sumándole 4 (una instrucción) para compensar el decremento que sufrirá la primera vez que se restaure el proceso, debido al código de retorno de la *BIOS IRQ exception handler* (ver apartado 2.2),
- guardar los cuatro primeros caracteres del nombre en clave del programa (tercer parámetro) en el campo `keyName` del elemento `_gd_pcb[s]`,
- calcular la dirección base de la pila del proceso,
- guardar en la pila del proceso el valor inicial de los registros `R0-R12 + R14`, que será cero para todos excepto para `R0`, que tendrá que contener el valor del argumento (cuarto parámetro), y excepto `R14`, que deberá contener la dirección de retorno del proceso, concretamente, la dirección de la rutina `_gp_terminarProc()`, que se encargará de realizar las tareas de finalización del proceso,
- guardar el valor actual del registro `R13` del proceso a crear en el campo `SP` del elemento `_gd_pcb[s]`,
- guardar el valor inicial del registro `CPSR` en el campo `Status` del elemento `_gd_pcb[s]`, para que la ejecución del proceso sea en modo **sistema**, se permite la generación de interrupciones (*flag I* = 0), tipo de juego de instrucciones ARM (*flag T* = 0) y el resto de *flags* a cero,
- inicializar otros campos del elemento `_gd_pcb[s]`, como el contador de tics de trabajo `workTicks`,
- guardar el número de zócalo en la última posición de la cola de *Ready* e incrementar en número de procesos pendientes en la variable `_gd_nReady`.

Si se realizan correctamente todos estos pasos, la RSI de intercambio de procesos empezará a ejecutar el proceso creado cuando le toque el turno. Cuando el proceso termine, se ejecutará la rutina `_gp_terminarProc()`, lo cual provocará que el proceso se elimine del circuito de ejecución.

Además, hay que tener en cuenta que **no** es necesario invocar a la rutina de creación de procesos para el proceso del sistema operativo; simplemente hay que inicializar la variable `_gd_pidz` a cero y algún campo más del elemento `_gd_pcb[0]`, como el nombre en clave del programa ("GARL").

El primer cambio de contexto se producirá cuando carguemos el primer proceso de programa, y ya se encargará la rutina `_gp_salvarProc()` de salvar convenientemente el estado del proceso de control del sistema operativo.

4.5 Otras rutinas de gestión de procesos

Existen otras rutinas de la rama `progP` que ya están implementadas en la versión inicial del proyecto `GARLIC_OS`. Sin embargo, es necesario entender su funcionamiento para una comprensión general de toda la gestión de procesos.

La rutina `_gp_numProc()` devuelve el número total de procesos en el sistema, contando el proceso que está en *Run* más los procesos que están en la cola de *Ready*.

La rutina `_gp_terminarProc()` indica al sistema que un proceso ha terminado. El procedimiento concreto es el siguiente:

- pone a cero la parte del PID de la variable `_gd_pidz` sin cambiar la parte del número de zócalo, lo cual permite a la rutina de cambio de contexto detectar que el proceso ha terminado su ejecución y, por lo tanto, no salvará su contexto ni lo guardará en cola de *Ready*.
- Poner a cero el campo `PID` del elemento `_gd_pcb[z]`, para que el zócalo `z` se pueda volver a utilizar por otro proceso con la rutina de creación de procesos,
- llamar a la rutina `_gp_WaitForVBlank()` hasta forzar el cambio de contexto.

La rutina `_gp_WaitForVBlank()` sustituye a la función `swiWaitForVBlank()` de **libnds**, que internamente llama a una rutina de la BIOS para esperar el siguiente retroceso vertical.

Es importante entender que los procesos de Garlic **no** pueden llamar a una rutina SWI, puesto que esto supone el cambio de modo de ejecución (de **Sistema** a **Supervisor**), lo que conllevaría un cambio de pila. Por lo tanto, si no queremos manejar dos pilas para cada proceso de usuario, estos programas no deben utilizar funciones de la BIOS.

La excepción a esta regla es el proceso de sistema, puesto que **sí** tiene una pila en modo **supervisor**. Por este motivo, cuando se produce un cambio de contexto del proceso de sistema, puede que esté en modo de ejecución del **supervisor**, lo cual se deberá tener en cuenta en el código de salvar y restaurar procesos.

4.6 Programa principal para progP

Para poder realizar las tareas específicas de esta rama sin tener que depender del trabajo realizado por los compañeros de las otras ramas, se dispone el siguiente programa principal **main.c**:

```
/*-----
    "main.c" : fase 1 / programador P
-----*/

#include <nds.h>
#include <stdio.h>

#include <garlic_system.h>

#include <GARLIC_API.h> // inclusión de la API para simular un proceso
int hola(int);          // función que simula la ejecución del proceso
...

//-----
void inicializarSistema() {
//-----

    consoleDemoInit();
    ...
    irqInitHandler(_gp_IntrMain); // instalar rutina principal inter.
    irqSet(IRQ_VBLANK, _gp_rsiVBL); // instalar RSI de VBlank
    irqEnable(IRQ_VBLANK);          // activar inter. VBlank
    REG_IME = IME_ENABLE;           // activar inter. en general

    _gd_pcbs[0].keyName = 0x4C524147; // "GARL"
}
```

```
//-----
int main(int argc, char **argv) {
//-----

    inicializarSistema();

    printf("*****");
    printf("*");
    printf("* Sistema Operativo GARLIC 1.0 *");
    printf("*");
    printf("*****");
    printf("*** Inicio fase 1_P\n");

    _gp_crearProc(hola, 7, "HOLA", 1);
    _gp_crearProc(hola, 14, "HOLA", 2);

    while(_gp_numProc() > 1) {
        _gp_WaitForVBlank();
        printf("*** Test\t%d\t%d\n", _gd_tickCount, _gp_numProc());
    }
        // esperar a que termine el proceso pendiente

    printf("*** Final fase 1_P\n");

    while(1) {
        _gp_WaitForVBlank();
    }
        // parar el procesador en un bucle infinito
    return 0;
}
```

En este extracto se muestra la inclusión de cabeceras (`#include`), una función de inicialización y la función principal `main()`.

En el apartado de cabeceras se ha incluido el fichero `GARLIC_API.h` con el fin de poder llamar a las funciones del API desde la función de prueba que describiremos más adelante, aunque **esto no estará permitido en la versión final**, puesto que el sistema operativo no tiene que invocarse a sí mismo a través de las funciones del API, sino que puede llamar a sus funciones internas directamente (más eficiente).

En la función de inicialización hay que observar como se llama a la función `consoleDemoInit()` de la librería **libnds**, lo cual permitirá emitir mensajes por pantalla con la función `printf()`, o sea, sin depender del trabajo de la rama `progG`.

Además, en la función de inicialización se instala la rutina principal de gestión de interrupciones de GARLIC, así como la RSI para las interrupciones

de retroceso vertical, que se tendrá que programar convenientemente para realizar el multiplexado de los procesos.

En la función principal `main()` se inicializa el sistema y se crean dos procesos de prueba con la función `hola()`, con dos argumentos diferentes (1 y 2), después se imprimen unos mensajes de test mientras los procesos no terminan su ejecución y, finalmente, se entra en un bucle infinito.

Hay que destacar que los procesos de prueba **no** se cargarán en memoria mediante el procedimiento que se tendrá que programar en la rama `progM`, para evitar dependencias entre las dos ramas. Por este motivo, la función `hola()` se ha definido como una función más del fichero `main.c`:

```
/* Proceso de prueba */
//-----
int hola(int arg) {
//-----

    unsigned int i, j, iter;

    if (arg < 0) arg = 0;           // limitar valor máximo y
    else if (arg > 3) arg = 3;      // mínimo del argumento

                                   // escribir mensaje inicial
    GARLIC_printf("-- Programa HOLA - PID (%d) --\n", GARLIC_pid());

    j = 1;                         // j = cálculo de 10 elevado a arg
    for (i = 0; i < arg; i++)
        j *= 10;
        // cálculo aleatorio del número de iteraciones 'iter'
    GARLIC_divmod(GARLIC_random(), j, &i, &iter);
    iter++;                        // asegurar que hay al menos una iteración

    for (i = 0; i < iter; i++)      // escribir mensajes
        GARLIC_printf("(%d)\t%d: Hello world!\n", GARLIC_pid(), i);

    return 0;
}
```

Se puede observar que el código de la función de prueba es exactamente el mismo del programa `hola.c` del proyecto `GARLIC Progs/HOLA`, mostrado en el apartado 3.6 de este manual, salvo la cabecera de la función.

Sin embargo, esta función de prueba realiza llamadas a la función del API `GARLIC_printf()`, lo que provoca la invocación de la rutina interna `_ga_printf()`, que se ha modificado convenientemente en el fichero `garlic_itcm_api.s` del proyecto `GARLIC_OS` para la rama `progP`:

```

.global _ga_printf
@;Parámetros
@; R0: char * format,
@; R1: unsigned int val1 (opcional),
@; R2: unsigned int val2 (opcional)
_ga_printf:
    push {r4, lr}
    ldr r4, =_gd_pidz      @; R4 = dirección _gd_pidz
    ldr r3, [r4]
    and r3, #0x3          @; R3 = ventana de salida (zócalo actual MOD 4)
    bl _gp_waitForVBlank
    push {r12}
    bl printf              @; llamada de prueba
    pop {r12}
    pop {r4, pc}

```

Comparando este código modificado con su contenido original (ver apartado 3.5), se observa que se ha substituido la llamada `bl _gg_escribir` por una llamada `bl _gp_waitForVBlank` y otra `bl printf`, junto con un par de instrucciones `push/pop` para salvar el registro `r12` (las llamadas a funciones de C lo pueden modificar). Es decir, se sustituye la llamada final a la función `_gg_escribir()` por una llamada de prueba a la función `printf()` estándar, con lo cual se conseguirá visualizar los mensajes que se transmitan a través de la `GARLIC_printf()` en la ventana inferior de la NDS, sincronizados con la interrupción de `VBlank`.

Hay que tener en cuenta que este cambio es **temporal**, solo para verificar el funcionamiento de las rutinas de la rama `progP` sin disponer del código final de la rama `progG`. Evidentemente, este cambio no funcionará sobre el entorno de ventanas de Garlic, el cual es completamente desconocido para la función `printf()` estándar.

Para que toda la parte del programador `progP` pueda compilar y *linkar*, en el fichero `Makefile` de esta rama se ha indicado que se enlace el proyecto con el fichero `GARLIC_API.o`, lo cual **no estará permitido en la versión final del proyecto**:

```

%.elf:
    @echo linking $(notdir $@)
    $(LD) $(LDFLAGS) $(OFILES) $(LIBPATHS) $(LIBS) $(SFILES)
        $(GARLICAPI)/GARLIC_API.o -o $@

```


El resultado final que se observará en pantalla de la NDS será similar al siguiente:

```
*****
* Sistema Operativo GARLIC 1.0 *
*****
*** Inicio fase 1_P
*** Test 4:3
-- Programa HOLA - PID (1) --
-- Programa HOLA - PID (2) --
*** Test 7:3
(1) 0: Hello world!
(2) 0: Hello world!
*** Test 10:3
(1) 1: Hello world!
(2) 1: Hello world!
*** Test 13:3
(1) 2: Hello world!
(2) 2: Hello world!
*** Test 16:2
(2) 3: Hello world!
*** Test 18:2
(2) 4: Hello world!
.
.
(2) 21: Hello world!
*** Test 52:2
(2) 22: Hello world!
*** Test 54:2
(2) 23: Hello world!
*** Test 56:2
(2) 24: Hello world!
*** Test 58:2
(2) 25: Hello world!
*** Test 60:2
(2) 26: Hello world!
*** Test 62:2
(2) 27: Hello world!
*** Test 64:2
(2) 28: Hello world!
*** Test 66:2
(2) 29: Hello world!
*** Test 68:2
(2) 30: Hello world!
*** Test 70:2
(2) 31: Hello world!
*** Test 72:1
*** Final fase 1_P
```

En las pantallas anteriores se puede observar que el código para la rama **progP** funciona porque los mensajes del proceso de control del sistema operativo (función `main()`) y los procesos de usuario se entrelazan (función `hola()`), lo cual indica que su ejecución es concurrente gracias a la multiplexación.

En el caso de los procesos de usuario, los mensajes empiezan por el número de PID del proceso (1 o 2), seguido del número de veces que se ha emitido el mensaje "Hello World!". Como el proceso 1 habitualmente genera menos mensajes que el proceso 2 debido a la diferencia en sus argumentos, llega un momento en que el proceso 1 acaba pero el 2 no.

Por otro lado, los mensajes del sistema operativo empiezan por "*** Test" e indican el contador de tics (retrocesos verticales) y el número de procesos totales que se están ejecutando en cada momento. Cuando los dos procesos de usuario terminan, el número de procesos en ejecución será 1; entonces el control del sistema operativo emitirá un mensaje de finalización y entrará en un bucle infinito.

Hay que tener en cuenta que el número de zócalo que asignamos a los procesos de prueba **no** puede ser muy bajo, ya que al utilizar la función `printf()` se pueden requerir entre 300 y 500 posiciones de pila, por lo tanto, la ejecución de un proceso de usuario puede **invadir la pila** de los dos o tres zócalos anteriores. Por este motivo, se han utilizado los zócalos 7 y 14.

Además, cada proceso escribe sus mensajes en la ventana correspondiente a su número de zócalo módulo 4. Por lo tanto, cuando el sistema de ventanas esté en funcionamiento (tarea de `progG`), el proceso de control del sistema operativo utilizará la ventana 0, mientras que los procesos de usuario utilizarán las ventanas 3 y 2 (desde los zócalos 7 y 14).

5 Tareas de gestión de la memoria (progM)

5.1 Formato de fichero ejecutable ELF

Entre las tareas de esta rama hay que programar la lectura e interpretación de ficheros `.elf`, que tienen que estar guardados dentro del directorio `GARLIC_OS/nitrofiles/Programas/`. Entre la estructura de ficheros de ejemplo para esta rama se proporciona el proyecto `GARLIC_Progs/HOLA`, que generará el fichero ejecutable de ejemplo `hola.elf`, a partir del fichero fuente `hola.c` y del fichero objeto `GARLIC_API/garlic_api.o`.

Por lo tanto, es necesario entender la estructura interna de los ficheros ejecutables según las especificaciones **ELF** (*Executable and Linkable Format*), cuya descripción en detalle se puede consultar en el fichero `Generic_ELF.pdf` disponible en el espacio *Moodle* de la asignatura.

Sin embargo, en esta sección del manual se explicarán las estructuras básicas imprescindibles para poder realizar la práctica.

Uno de los conceptos básicos iniciales es que el formato ELF está diseñado para codificar ficheros objeto, programas ejecutables y librerías dinámicas, aunque en esta práctica solo se utilizará la versión para programas ejecutables.

La siguiente idea importante es que un fichero ejecutable ELF está formado por **secciones** y también por **segmentos**:

Linking View	Execution View
ELF Header	ELF Header
Program Header Table <i>optional</i>	Program Header Table
Section 1	Segment 1
...	
Section <i>n</i>	Segment 2
...	
...	...
Section Header Table	Section Header Table <i>optional</i>

Las **secciones** son los fragmentos de código o datos que se generan al compilar/ensamblar el programa. Por ejemplo, las secciones más habituales son `".text"`, `".rodata"`, `".data"` y `".bss"`, aunque también existen

secciones específicas de la plataforma sobre la que se compila, por ejemplo `".dtcm"` y `".itcm"`. Existen además otras secciones para registrar información del compilador o ensamblador, por ejemplo para poder reubicar las referencias a posiciones de memoria absoluta (`".rel.text"`).

Los **segmentos** son los fragmentos de código o datos que se tienen que cargar en memoria a la hora de ejecutar el programa. Un segmento puede contener una o varias secciones.

La visión de secciones o segmentos depende del contexto en el cual se trate el fichero ELF: las secciones se manipulan por el compilador/ensamblador y enlazador (*linker*), mientras que los segmentos están orientados a facilitar el trabajo al cargador de programas en memoria (**loader**), cuya implementación es la **tarea básica** del programador **progM**, aunque también deberá interpretar el contenido de algunas secciones.

5.2 Tipos de datos ELF

Antes de analizar las estructuras ELF, es necesario definir el propósito (y el tamaño) de los tipos de datos definidos específicamente para este formato:

Tipo	Bytes	Propósito
Elf32_Addr	4	dirección de memoria
Elf32_Half	2	medio entero (sin signo)
Elf32_Off	4	desplazamiento dentro del fichero (sin signo)
Elf32_Sword	4	entero con signo
Elf32_Word	4	entero (sin signo)
unsigned char	1	entero pequeño (sin signo)

A nivel de codificación, lo más importante es el número de bytes que ocupa cada tipo de datos.

A nivel de significado, hay que destacar que las direcciones de memoria se pueden referir a un espacio de memoria física o virtual, aunque en nuestro caso coinciden (memoria virtual = memoria física).

Por otro lado, el desplazamiento dentro del fichero (*offset*) hace referencia al número de bytes que hay que "saltar" desde el inicio del fichero para llegar a una determinada posición de la información, dentro del fichero ELF.

5.3 Cabecera de un fichero ELF

La cabecera ELF permite definir el tamaño y la posición de las estructuras más elementales del formato, es decir, la tabla de secciones y la tabla de segmentos. La estructura de la cabecera ELF es la siguiente:

```
#define EI_NIDENT 16
typedef struct {
    unsigned char    e_ident[EI_NIDENT];
    Elf32_Half       e_type;
    Elf32_Half       e_machine;
    Elf32_Word       e_version;
    Elf32_Addr       e_entry;
    Elf32_Off        e_phoff;
    Elf32_Off        e_shoff;
    Elf32_Word       e_flags;
    Elf32_Half       e_ehsize;
    Elf32_Half       e_phentsize;
    Elf32_Half       e_phnum;
    Elf32_Half       e_shentsize;
    Elf32_Half       e_shnum;
    Elf32_Half       e_shstrndx;
} Elf32_Ehdr;
```

Los campos más relevantes para esta práctica son los siguientes:

- `e_entry`: punto de entrada del programa (dirección de memoria de la primera instrucción de la rutina `_start()`)
- `e_phoff`: desplazamiento de la tabla de segmentos (*program header*),
- `e_shoff`: desplazamiento de la tabla de secciones (*section header*),
- `e_phentsize`: tamaño de cada entrada de la tabla de segmentos,
- `e_phnum`: número de entradas de la tabla de segmentos,
- `e_shentsize`: tamaño de cada entrada de la tabla de secciones,
- `e_shnum`: número de entradas de la tabla de secciones.

Para poder observar el contenido de la cabecera ELF del fichero **HOLA.elf** podemos ejecutar la utilidad **arm-none-eabi-readelf**, que se encuentra en el directorio **devkitPro/devkitARM/bin**. Si tenemos este directorio añadido a la variable **PATH** del sistema, podremos invocarlo desde el directorio que contiene el fichero **HOLA.elf**, ya sea desde **GARLIC_Progs/HOLA** o bien desde **GARLIC_OS/nitrofiles/Programas** (para que el fichero esté en este segundo directorio, habrá que copiarlo manualmente desde el primer directorio, pasando el nombre de minúsculas a mayúsculas):

```
$ arm-none-eabi-readelf -h HOLA.elf
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                               ELF32
  Data:                                   2's complement, little endian
  Version:                             1 (current)
  OS/ABI:                               UNIX - System V
  ABI Version:                         0
  Type:                                 EXEC (Executable file)
  Machine:                             ARM
  Version:                             0x1
  Entry point address:                  0x8000
  Start of program headers:             52 (bytes into file)
  Start of section headers:             2224 (bytes into file)
  Flags:                                0x5000202, Version5 EABI,
  Size of this header:                  52 (bytes)
  Size of program headers:              32 (bytes)
  Number of program headers:            1
  Size of section headers:              40 (bytes)
  Number of section headers:            9
  Section header string table index: 6
```

En este listado vemos que la dirección de entrada es la 0x8000, que la tabla de segmentos tiene una sola entrada de 32 bytes que se encuentra a partir del byte 52 del fichero (*offset*), mientras que en la tabla de secciones hay nueve entradas de 40 bytes cada una, que se encuentran a partir del byte 2.224 del fichero (*offset*).

5.4 Tabla de segmentos

La tarea básica de la rama **progM** consistirá en leer todas las entradas de la tabla de segmentos y cargar los segmentos a partir de una dirección de memoria libre.

La estructura de cada entrada de la tabla de segmentos es la siguiente:

```
typedef struct {
    Elf32_Word      p_type;
    Elf32_Off       p_offset;
    Elf32_Addr      p_vaddr;
    Elf32_Addr      p_paddr;
    Elf32_Word      p_filesz;
    Elf32_Word      p_memsz;
    Elf32_Word      p_flags;
    Elf32_Word      p_align;
} Elf32_Phdr;
```

Los campos más relevantes para esta práctica son los siguientes:

- **p_type**: tipo del segmento; solo se cargarán segmentos de tipo 1 (**PT_LOAD**),
- **p_offset**: desplazamiento en el fichero del primer byte del segmento,
- **p_paddr**: dirección física donde se tendría que cargar el segmento; en nuestro caso solo servirá de referencia, puesto que el sistema operativo cargará el programa en sus propias direcciones de memoria (reubicación),
- **p_filesz**: tamaño del segmento dentro del fichero,
- **p_memsz**: tamaño del segmento dentro de la memoria; podría ser diferente al campo anterior, por ejemplo para segmentos con zonas de datos no inicializadas (**.bss**), que no ocupan espacio en el fichero pero sí en memoria,
- **p_flags**: este campo indica si el contenido del segmento es de lectura (**R**), escritura (**W**), ejecutable (**E**) o cualquier combinación de las tres modalidades, aunque en esta primera fase de la práctica no se utilizará esta información.

Para poder ver los segmentos del programa **HOLA.elf**, podemos invocar a *arm-none-eabi-readelf* con la opción **-l** (guión ele minúscula):

```
$ arm-none-eabi-readelf -l HOLA.elf

Elf file type is EXEC (Executable file)
Entry point 0x8000
There are 1 program headers, starting at offset 52

Program Headers:
   Type   Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
LOAD 0x000400 0x00008000 0x00008000 0x0019f 0x0019f R E 0x400

Section to Segment mapping:
Segment Sections...
00          .text .rodata
```

En este listado vemos que el único segmento existente en este fichero es de tipo **LOAD**, su contenido empieza en el byte 0x000400 del fichero (*offset*), que se tendría que copiar en la posición física 0x00008000 de memoria, que ocupa 415 bytes (0x19f) tanto en fichero como en memoria, y que su contenido es de solo lectura y ejecutable.

Además, la utilidad nos indica también qué secciones contiene cada segmento, que en este caso se corresponden a la sección de código de programa **".text"** más una sección de datos de solo lectura **".rodata"** (*read only data*), que almacena las constantes del programa, en nuestro caso, los *strings* de formato de texto para llamar a **GARLIC_printf()**.

5.5 El problema de la reubicación de direcciones

Para entender el problema de la reubicación de direcciones, veamos primero cómo el código fuente del programa **hola.c** (en verde) se convierte en lenguaje ensamblador **hola.s** (en rojo), y como quedan las posiciones de memoria (en azul) si cargamos el segmento del programa **HOLA.elf** a partir de la dirección 0x01002000 (lenguaje máquina en negro):

```

        .text
        .align 2
        .global _start
        .arm
        .type _start, %function
int _start(void)  _start:
{
    push {lr}                                0x1002000: str lr, [sp, #-4]!
    unsigned int i, iter;

        sub sp, sp, #28                      0x1002004: sub sp, sp, #28
        str r0, [sp, #4]                     0x1002008: str r0, [sp, #4]
        ldr r3, [sp, #4]                     0x100200C: ldr r3, [sp, #4]

    if (arg < 0) arg = 0;

        cmp r3, #0                           0x1002010: cmp r3, #0
        bge .L2                             0x1002014: bge 0x1002024
        mov r3, #0                           0x1002018: mov r3, #0
        str r3, [sp, #4]                     0x100201C: str r3, [sp, #4]
        b .L3                               0x1002020: b 0x1002038

        .L2:
    else if (arg > 3) arg = 3;

        ldr r3, [sp, #4]                     0x1002024: ldr r3, [sp, #4]
        cmp r3, #3                           0x1002028: cmp r3, #3
        ble .L3                             0x100202C: ble 0x1002038
        mov r3, #3                           0x1002030: mov r3, #3
        str r3, [sp, #4]                     0x1002034: str r3, [sp, #4]

        .L3:
    GARLIC_printf("-- Programa HOLA -- PID (%d) --\n", GARLIC_pid());

        bl GARLIC_pid                       0x1002038: bl 0x1002114
        mov r3, r0                           0x100203C: mov r3, r0
        mov r1, r3                           0x1002040: mov r1, r3
        ldr r0, .L9                           0x1002044: ldr r0, [0x100210C]
        bl GARLIC_printf                     0x1002048: bl 0x1002150

    j = 1;

        mov r3, #1                           0x100204C: mov r3, #1
        str r3, [sp, #20]                     0x1002050: str r3, [sp, #20]

    for (i = 0; i < arg; i++)

        mov r3, #0                           0x1002054: mov r3, #0

```

```

        str r3, [sp, #16]      0x1002058: str r3, [sp, #32]
        b .L4                 0x100205C: b 0x1002084

.L5:
j *= 10;        ldr r2, [sp, #20]      0x1002060: ldr r2, [sp, #20]
                mov r3, r2            0x1002064: mov r3, r2
                lsl r3, r3, #2        0x1002068: mov r3, r3, lsl #2
                add r3, r3, r2        0x100206C: add r3, r3, r2
                lsl r3, r3, #1        0x1002070: mov r3, r3, lsl #1
                str r3, [sp, #20]      0x1002074: str r3, [sp, #20]
                ldr r3, [sp, #16]      0x1002078: ldr r3, [sp, #16]
                add r3, r3, #1        0x100207C: add r3, r3, #1
                str r3, [sp, #16]      0x1002080: str r3, [sp, #16]

.L4:
        ldr r2, [sp, #16]      0x1002084: ldr r2, [sp, #16]
        ldr r3, [sp, #4]       0x1002088: ldr r3, [sp, #4]
        cmp r2, r3            0x100208C: cmp r2, r3
        bcc .L5               0x1002090: bcc 0x1002060

GARLIC_divmod(GARLIC_random(), j, &i, &iter);
        bl  GARLIC_random      0x1002094: bl 0x1002128
        mov r3, r0            0x1002098: mov r3, r0
        mov r0, r3            0x100209C: mov r0, r3
        add r3, sp, #12        0x10020A0: add r3, sp, #12
        add r2, sp, #16        0x10020A4: add r2, sp, #16
        ldr r1, [sp, #20]      0x10020A8: ldr r1, [sp, #20]
        bl  GARLIC_divmod      0x10020AC: bl 0x100213C

iter++;
        ldr r3, [sp, #12]      0x10020B0: ldr r3, [sp, #12]
        add r3, r3, #1         0x10020B4: add r3, r3, #1
        str r3, [sp, #12]      0x10020B8: str r3, [sp, #12]

for (i = 0; i < iter; i++)
{
    mov r3, #0                0x10020BC: mov r3, #0
    str r3, [sp, #16]          0x10020C0: str r3, [sp, #16]
    b .L6                     0x10020C4: b 0x10020EC

.L7:
        bl  GARLIC_pid         0x10020C8: bl 0x1002114
        mov r1, r0            0x10020CC: mov r1, r0
        ldr r3, [sp, #16]      0x10020D0: ldr r3, [sp, #16]
        mov r2, r3            0x10020D4: mov r2, r3

GARLIC_printf("(%d)\t%d: Hello world!\n", GARLIC_pid(), i);
        ldr r0, .L9+4          0x10020D8: ldr r0, [0x1002110]
        bl  GARLIC_printf      0x10020DC: bl 0x1002150
        ldr r3, [sp, #16]      0x10020E0: ldr r3, [sp, #16]
        add r3, r3, #1         0x10020E4: add r3, r3, #1
        str r3, [sp, #16]      0x10020E8: str r3, [sp, #16]

.L6:
        ldr r2, [sp, #16]      0x10020EC: ldr r2, [sp, #16]
        ldr r3, [sp, #12]      0x10020F0: ldr r3, [sp, #12]

```

```

    cmp r2, r3                0x10020F4: cmp r2, r3
    bcc .L7                   0x10020F8: bcc 0x10020C8
}
return 0;

    mov r3, #0                0x10020FC: mov r3, #0x0
    mov r0, r3                0x1002100: mov r0, r3
    add sp, sp, #28           0x1002104: add sp, sp, #28
    pop {pc}                  0x1002108: ldr pc, [sp], #4
}

.L10:
    .align 2
.L9:
    .word .LC0                0x100210C: 0x00008164
    .word .LC1                0x1002110: 0x00008188

.ident "GCC: (devkitARM release 46) 6.3.0"

```

En el listado anterior se han resaltado en negrita los contenidos conflictivos: las instrucciones `ldr r0, .L9` y `ldr r0, .L9+4` sirven para cargar en `R0` las direcciones de memoria donde empiezan los *strings* de formato de texto del programa, que luego se pasarán como primer parámetro de la función `GARLIC_printf()`.

En las posiciones de memoria relativas a la etiqueta `.L9` encontramos dos *words*, que en la versión en ensamblador hacen referencia a las etiquetas `.LC0` y `.LC1`. Estas etiquetas se definen de la siguiente forma:

```

.section .rodata
.align 2
.LC0:
.ascii "-- Programa HOLA - PID (%d) --\012\000"
.align 2
.LC1:
.ascii " (%d)\011%d: Hello world!\012\000"

```

Es decir, en la sección `".rodata"` se guardan los códigos ASCII de los *strings* de formato de texto del programa. Sin embargo, cuando esta información la carguemos en memoria a partir de la posición `0x01002000`, la dirección inicial de estos mensajes ya **no** se corresponderá con las direcciones guardadas en `0x100210C` y `0x1002110`. Es decir, los valores **0x00008164 y 0x00008188 son incorrectos**.

Esto es debido a que el programa se compiló para ubicarse a partir de la dirección de memoria `0x00008000`, pero nuestro sistema operativo lo cargará a partir de la dirección `0x01002000` (por ejemplo). Para solucionar este problema hay que aplicar un algoritmo de **reubicación**, tal como explicaremos a continuación.

5.6 Tabla de secciones

Para poder identificar las direcciones de memoria que necesitan una reubicación, hay que descifrar una sección del fichero .ELF específica que contiene unas estructuras especiales denominadas **relocs** o **reubicadores**.

Antes de describir estas estructuras, primero será necesario entender como interpretar la información de la tabla de secciones, donde cada entrada presenta la siguiente estructura:

```
typedef struct {
    Elf32_Word      sh_name;
    Elf32_Word      sh_type;
    Elf32_Word      sh_flags;
    Elf32_Addr      sh_addr;
    Elf32_Off       sh_offset;
    Elf32_Word      sh_size;
    Elf32_Word      sh_link;
    Elf32_Word      sh_info;
    Elf32_Word      sh_addralign;
    Elf32_Word      sh_entsize;
} Elf32_Shdr;
```

Los campos más relevantes para esta práctica son los siguientes:

- **sh_type**: tipo de la sección; las secciones de reubicadores son del tipo 9 ([SHT_REL](#)),
- **sh_offset**: desplazamiento en el fichero del primer byte de la sección,
- **sh_size**: tamaño de la sección dentro del fichero,
- **sh_link**: para una sección de tipo [SHT_REL](#), este campo indica el índice de la sección que contiene la tabla de símbolos asociada a los reubicadores,
- **sh_info**: para una sección de tipo [SHT_REL](#), este campo indica el índice de la sección sobre la cual se deberán aplicar los reubicadores,
- **sh_entsize**: para una sección de tipo [SHT_REL](#), este campo indica el tamaño en bytes de cada reubicador (típicamente, 8 bytes).

Para poder ver las secciones del programa **HOLA.elf**, podemos invocar a *arm-none-eabi-readelf* con la opción **-S** (guión ese mayúscula):

```
$ arm-none-eabi-readelf -S HOLA.elf
There are 9 section headers, starting at offset 0x8b0:

Section Headers:
[Nr]  Name           Type            Addr           Off           Size         ES Flg Lk  Inf
[ 0]                NULL           00000000       000000 000000    00      0 0
[ 1]  .text           PROGBITS       00008000       000400 00164    00    AX 0 0
[ 2]  .rel.text       REL            00000000       000828 00040    08    I 7 1
[ 3]  .rodata         PROGBITS       00008164       000564 0003b    00    A 0 0
[ 4]  .comment        PROGBITS       00000000       00059f 00022    01   MS 0 0
[ 5]  .ARM.attri     ARM_ATTRI      00000000       0005c1 00025    00      0 0
[ 6]  .shstrtab       STRTAB         00000000       000868 00046    00      0 0
[ 7]  .symtab         SYMTAB         00000000       0005e8 00190    10      8 11
[ 8]  .strtab         STRTAB         00000000       000778 000af    00      0 0

Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), T (TLS), E (exclude), ...
```

En el listado anterior se ha resaltado (en rojo) la única sección de tipo **SHT_REL** que existe en **HOLA.elf**, cuyo desplazamiento dentro del fichero es 0x828, que la sección de símbolos asociada es la número 7, y que la sección donde hay que aplicar las reubicaciones es la número 1, es decir, **".text"**.

Además, se observa también que el tamaño total de la sección es 0x40, o sea, 64 bytes, que divididos por el tamaño de cada reubicador da un total de 8 reubicadores, que veremos en el siguiente apartado.

5.7 Estructura de los reubicadores

En el formato ELF existen dos tipos de reubicadores, pero para esta práctica solo utilizaremos el tipo más básico, que presenta la siguiente estructura:

```
typedef struct {  
    Elf32_Addr      r_offset;  
    Elf32_Word      r_info;  
} Elf32_Rel;
```

El significado de estos dos campos es el siguiente:

- `r_offset`: para un fichero ejecutable, este campo indica la dirección de memoria virtual (o física) sobre la que hay que aplicar la reubicación,
- `r_info`: este campo se divide en dos partes, los 8 bits bajos contienen el tipo de reubicación que hay que aplicar, y el resto de los bits altos indican el índice del símbolo sobre el que se aplicará la reubicación.

Por lo tanto, en teoría necesitaríamos acceder también a la sección de símbolos referenciada en el campo `sh_link` de la descripción de la sección de reubicadores para poder obtener la información del símbolo sobre el que se aplica la reubicación. Sin embargo, para resolver las reubicaciones en esta práctica **no** será necesario profundizar más en el contenido de las secciones de símbolos.

Para observar los reubicadores de `HOLA.elf`, podemos invocar a `arm-none-eabi-readelf` con la opción `-r`:

```
$ arm-none-eabi-readelf -r HOLA.elf
```

```
Relocation section '.rel.text' at offset 0x828 contains 8 entries:
```

Offset	Info	Type	Sym.Value	Sym. Name
00008038	00000b1c	R_ARM_CALL	00008114	GARLIC_pid
00008048	00000f1c	R_ARM_CALL	00008150	GARLIC_printf
00008094	00000d1c	R_ARM_CALL	00008128	GARLIC_random
000080ac	0000181c	R_ARM_CALL	0000813c	GARLIC_divmod
000080c8	00000b1c	R_ARM_CALL	00008114	GARLIC_pid
000080dc	00000f1c	R_ARM_CALL	00008150	GARLIC_printf
0000810c	00000202	R_ARM_ABS32	00008164	.rodata
00008110	00000202	R_ARM_ABS32	00008164	.rodata

En el listado anterior se han resaltado los reubicadores que afectan a las direcciones de los strings de formato de texto. Concretamente, se trata de los que tienen el tipo `R_ARM_ABS32`, cuyo código numérico es el 2 (en 8 bits bajos).

En la columna "Offset" (primera por la izquierda) observamos las direcciones sobre las que hay que aplicar la reubicación, es decir, 0x0000810c y 0x00008110. Si recordamos las direcciones de las posiciones que contienen los valores erróneos del programa son 0x0100210c y la 0x01002110, o sea, se trata de las mismas posiciones pero desplazadas en memoria desde 0x00008000 a 0x01002000.

Además, en la columna "Sym.Value" (valor del símbolo) aparece la dirección inicial de la sección ".rodata", puesto que en el código ensamblador se hace referencia a las etiquetas `.LC0` y `.LC1`.

Por último, observar que **no** es necesario prestar atención a los otros reubicadores de tipo `R_ARM_CALL`, puesto que se trata de llamadas con una **dirección de salto relativa**, lo cual significa que no importa a partir de qué posición de memoria se copie el contenido del segmento: la diferencia entre la dirección de la instrucción de salto (`b` o `bl`) hasta la dirección destino se mantendrá intacta, y el salto se efectuará correctamente, tal como se muestra en las direcciones de salto del código máquina del apartado 5.5.

5.8 Algoritmo de carga y reubicación

Para conseguir realizar la carga de los segmentos del programa a partir de una determinada dirección de memoria destino, hay que programar la siguiente función en lenguaje C, dentro del fichero **garlic_mem.c**:

```
/* _gm_cargarPrograma: busca un fichero de nombre "(keyName).elf"
   dentro del directorio "/Programas/" del sistema de ficheros
   y carga los segmentos de programa a partir de una posición
   de memoria libre, efectuando la reubicación de las
   referencias a los símbolos del programa, según el
   desplazamiento del código en la memoria destino;

   Parámetros:
       keyName -> vector de 4 caracteres con el nombre en clave

   Resultado:
       != 0 -> dirección de inicio del programa (intFunc)
       == 0 -> no se ha podido cargar el programa
*/
intFunc _gm_cargarPrograma(char *keyName)
```

El algoritmo de la función puede seguir los siguientes pasos:

- buscar el fichero "**(keyname).elf**" en el directorio **Programas** del sistema de ficheros **Nitro**, contenido dentro del sistema operativo, donde "(keyname)" será el nombre en clave del programa (**por convenio, solo cuatro caracteres en mayúsculas**),
- si encuentra el fichero, cargarlo íntegramente dentro de un buffer de memoria dinámica, que permitirá acceder a su contenido en los siguientes pasos del algoritmo de forma más eficiente,
- acceder a la cabecera ELF para obtener la posición (*offset*) y el tamaño de la tabla de segmentos,
- acceder a la tabla de segmentos; para cada segmento de tipo **PT_LOAD**:
 - obtener la dirección de memoria inicial del segmento a cargar (campo **p_paddr**), así como el desplazamiento dentro del fichero donde empieza el segmento (campo **p_offset**),
 - cargar el contenido del segmento a partir de una dirección de memoria destino apropiada (ver más adelante),

- efectuar la reubicación de todas las posiciones sensibles a la dirección del código en memoria, invocando a la rutina `_gm_reubicar()` (ver más adelante).
- si todo el proceso ha funcionado correctamente, devolver la dirección de inicio del programa para el segmento que contenga el punto de entrada `e_entry`, convenientemente reubicada,
- en caso de que se haya detectado algún inconveniente, devolver 0.

Este algoritmo se puede simplificar para el caso del programa `HOLA.elf`, porque **solo tiene un único segmento a cargar**. Sin embargo, en la segunda fase de la práctica se utilizarán otros programas cuya estructura será un poco más compleja (uno o dos segmentos a cargar).

En la parte del algoritmo que se refiere a “*una dirección de memoria destino apropiada*”, se debe entender que podemos fijar cualquier dirección de memoria de la NDS a partir de la cual exista suficiente cantidad de posiciones libres para albergar el contenido de cada segmento a cargar.

Tal como se indica en el apartado 3.9.2 de este manual, se pide que la memoria para albergar los segmentos de los procesos se obtenga a partir de la posición `0x01002000`, es decir, en memoria ITCM. El fichero `garlic_mem.c` dispone de una definición simbólica para hacer referencia a dicha posición inicial:

```
#define INI_MEM_PROC    0x01002000    // dirección de inicio de memoria
                                // de los procesos de usuario
```

Además de este símbolo, se debe crear una variable global que contenga la primera posición de memoria libre dentro de la zona de procesos de usuario, con el fin de permitir **cargar los segmentos de varios procesos a la vez**, uno a continuación del otro, suponiendo que nunca se llegará hasta el límite de los 24 Kbytes.

Para realizar la copia de los bytes de los segmentos no se podrá utilizar la función `dmaCopy()` de la librería **libnds**, puesto que el controlador de DMA no puede acceder a la zona de memoria ITCM. Por lo tanto, se proporciona una rutina específica para realizar copias de memoria mediante CPU, de nombre `_gs_copiaMem()`, dentro del fichero `garlic_itcm_sys.s`:

```
/* _gs_copiaMem: copiar un bloque de numBytes bytes, desde una posición
   de memoria inicial (*source) a partir de otra posición de memoria
   destino (*dest), teniendo en cuenta que ambas posiciones de
   memoria deben estar alineadas a word */
extern void _gs_copiaMem(const void *source, void *dest, unsigned int
numBytes);
```

Para realizar la reubicación de las posiciones sensibles a la dirección de memoria destino, se debe programar la siguiente **rutina en lenguaje ensamblador**, dentro del fichero **garlic_itcm_mem.s**:

```
.global _gm_reubicar

@; rutina para interpretar los 'relocs' de un fichero ELF y
@; ajustar las direcciones de memoria correspondientes a las
@; referencias de tipo R_ARM_ABS32, restando la dirección de
@; inicio de segmento y sumando la dirección de destino en la
@; memoria;
@;Parámetros:
@; R0: dirección inicial del buffer de fichero (char *fileBuf)
@; R1: dirección de inicio de segmento (unsigned int pAddr)
@; R2: dirección de destino en la memoria (unsigned int *dest)
@;Resultado:
@; cambio de las direcciones de memoria que se tienen que ajustar
_gm_reubicar:
    push {lr}

    pop {pc}
```

El algoritmo para esta rutina debe realizar los siguientes pasos:

- acceder a las secciones de reubicadores, dentro del buffer del fichero ELF,
- para cada reubicador de tipo `R_ARM_ABS32`:
 - obtener la **dirección de memoria destino** de la posición que hay que reubicar, aplicando la reubicación a la **dirección de memoria origen** registrada en el reubicador,
 - obtener el **contenido** de dicha dirección destino, que corresponderá a la dirección origen de un dato, aplicarle la reubicación y actualizar la memoria según el resultado.

Como en esta primera fase vamos a suponer que solo habrá un único segmento de código para cada programa a cargar, **todas** las reubicaciones se realizarán siempre en base a la misma dirección origen y dirección destino de dicho segmento, según se indique en los parámetros de la rutina.

En la segunda fase, sin embargo, los programas podrán tener dos segmentos, uno de código y otro de datos, lo cual doblará el número de direcciones a gestionar e incrementará la complejidad del algoritmo para aplicar la reubicación en base a unas direcciones u otras.

Además, en esta primera fase se proporciona otro programa de ejemplo para GARLIC_OS, contenido en el directorio `GARLIC_Progs/PRNT`, que consiste en una serie de tests específicos para la rutina `GARLIC_printf()`.

Hay que tener en cuenta que el fichero `PRNT.elf` generado contiene un único segmento de código, com corresponde a esta primera fase de la práctica, pero presenta dos secciones de reubicadores:

```
$ arm-none-eabi-readelf -S PRNT.elf
There are 10 section headers, starting at offset 0x1024:

Section Headers:
[Nr]  Name           Type          Addr          Off   Size      ES Flg Lk  Inf
[ 0]                NULL          00000000      000000 00000      00      0 0  0
[ 1] .text            PROGBITS      00008000      000400 00268      00  AX  0  0
[ 2] .rel.text        REL           00000000      000e60 000f8      08   I  8  1
[ 3] .rodata          PROGBITS      00008268      000668 0053f      00   A  0  0
[ 4] .rel.rodata      REL           00000000      000f58 00080      08   I  8  3
[ 5] .comment          PROGBITS      00000000      000ba7 00022      01  MS  0  0
[ 6] .ARM.attri        ARM_ATTRI     00000000      000bc9 00025      00      0 0  0
[ 7] .shstrtab          STRTAB        00000000      000fd8 0004a      00      0 0  0
[ 8] .symtab            SYMTAB        00000000      000bf0 001b0      10      9 11  0
[ 9] .strtab            STRTAB        00000000      000da0 000be      00      0 0  0

Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), T (TLS), E (exclude), ...
```

Esto significa que será necesario recorrer todas las secciones del fichero ELF para detectar múltiples secciones de reubicadores. Ahora bien, de momento para la fase 1 de la práctica podemos suponer que todas las secciones de reubicadores harán referencia al único segmento (de código) del programa.

5.9 Programa principal para progM

Para poder realizar las tareas específicas de esta rama sin tener que depender del trabajo realizado por los compañeros de las otras ramas, se dispone el siguiente programa principal **main.c**:

```
/*-----
    "main.c" : fase 1 / programador M
-----*/

#include <nds.h>
#include <stdio.h>
#include <garlic_system.h>
...

//-----
void inicializarSistema() {
//-----
    consoleDemoInit();
    ...
    if (!_gm_initFS()) {
        printf("ERROR: ;no se puede inicializar el sistema de
            ficheros!");
        exit(0);
    }
}

//-----
int main(int argc, char **argv) {
//-----
    intFunc start;
    inicializarSistema();

    printf("*****");
    printf("*");
    printf("* Sistema Operativo GARLIC 1.0 *");
    printf("*");
    printf("*****");
    printf("*** Inicio fase 1_M\n");

    printf("*** Carga de programa HOLA.elf\n");
    start = _gm_cargarPrograma("HOLA");
    if (start)
    {
        printf("*** Direccion de arranque :\n\t\t%p\n", start);
        printf("*** Pulse \'START\' ::\n\n");
    }
}
```

```

        while(1) {
            swiWaitForVBlank();
            scanKeys();
            if (keysDown() & KEY_START) break;
        }
        start(1);          // llamada al proceso HOLA con argumento 1
    } else
        printf("*** Programa \"HOLA\" NO cargado\n");

    printf("*** Carga de programa PRNT.elf\n");
    start = _gm_cargarPrograma("PRNT");
    if (start)
    {
        printf("*** Direccion de arranque :\n\t\t%p\n", start);
        printf("*** Pusle \'START\' ::\n\n");
        while(1) {
            swiWaitForVBlank();
            scanKeys();
            if (keysDown() & KEY_START) break;
        }
        start(0);          // llamada al proceso PRNT con argumento 0
    } else
        printf("*** Programa \"PRNT\" NO cargado\n");

    printf("*** Final fase 1_M\n");

    while(1) {
        swiWaitForVBlank();
    }
    // parar el procesador en un bucle infinito
    return 0;
}

```

En la función `inicializarSistema()` se llama a la función `_gm_initFS()`, cuyo código se tendrá que programar en el fichero `garlic_mem.c`, para inicializar el sistema de ficheros **Nitro**.

Además, en la función `main()` se llama a la función `_gm_cargarPrograma()`, que deberá realizar la carga dinámica y reubicación de los ficheros ejecutables en formato ELF.

Nota: para saber cómo utilizar la librería **Nitro** de acceso a ficheros, se recomienda consultar el ejemplo disponible en el fichero `devkitPro/examples/nds/filesystem/nitrofs/nitrodir/source/directory.c`

Si la función `_gm_cargarPrograma()` puede cargar y reubicar el programa en memoria, devolverá la dirección de inicio en memoria del programa, sino devolverá 0. La función `main()` invocará dicha dirección de inicio para ejecutar el programa cargado en memoria. Esto se consigue con la línea `start(x)`, donde x es el argumento del programa (número entero entre 0 y

3), aunque hay que entender que la función `start()` **no está definida en ningún fichero del proyecto GARLIC_OS**, sino que corresponde al código del segmento cargado desde fichero ELF y reubicado convenientemente en memoria.

Este mecanismo tan “extraño” de transferir la ejecución a un código externo al código del proyecto funciona gracias a que el lenguaje C permite manejar direcciones de funciones como un tipo de dato más del lenguaje. En nuestro caso, se ha definido el tipo `intFunc` en el fichero `garlic_system.h`:

```
typedef int (* intFunc)(int);
```

Gracias a este tipo de definiciones, el lenguaje C puede utilizar el resultado de la función `_gm_cargarPrograma()` como la dirección de una función que tiene un argumento de tipo `int` y devuelve un resultado de tipo `int`, y que se puede invocar como cualquier otra función compilada y enlazada dentro del código del proyecto.

El resultado que se observará en pantalla para la ejecución del programa `HOLA.elf` será similar al siguiente:

```
*****
*
* Sistema Operativo GARLIC 1.0 *
*
*****
*** Inicio fase 1_M
*** Carga de programa HOLA.elf
> segmento de programa copiado:
  despl. en fichero : 1024
  tam. en fichero : 415
  dir. en memoria : 0x8000
*** Direccion de arranque :
  0x1002000
*** Pulse tecla 'START' ::

-- Programa HOLA - PID (0) --
(0) 0: Hello world!
(0) 1: Hello world!
(0) 2: Hello world!
(0) 3: Hello world!
(0) 4: Hello world!
(0) 5: Hello world!
(0) 6: Hello world!
(0) 7: Hello world!
```

...

La información que se muestra después del carácter '>' se ha generado desde `_gm_cargarPrograma()`, con el único propósito de poder verificar los datos para la carga del segmento. En la versión definitiva de la práctica, su visualización se tendrá que suprimir.

La parte que demuestra que la carga y reubicación se ha hecho correctamente son las líneas que ha impreso el programa **HOLA**, es decir, la lista de mensajes “Hello world!” y la línea de cabecera.

Después de volver a pulsar la tecla START, el resultado que se observará en pantalla para la ejecución del programa `PRNT.elf` será similar al siguiente:

```

*** Carga de programa PRNT.elf
> segmento de programa copiado:
  despl. en fichero : 1024
  tam. en fichero : 1959
  dir. en memoria : 0x8000
*** Direccion de arranque :
  0x10021a0
*** Puse tecla 'START' ::

Prueba frases:
Por fin lleg . Salimos en seguid
a para Carmona.
El chofer alzaba una ceja, pisab
a el acelerador y dec a, volvien
dose a medias hacia nosotras:
-Podridita que est la carret
era.

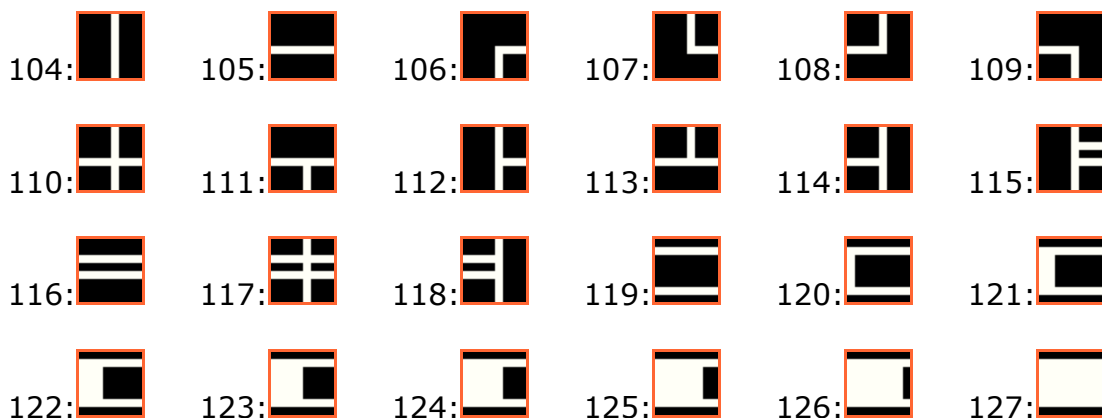
Pruebas mixtas::

%a% prueba string%%char: @0
%b% aleatorio decimal: 17346%
    hexadecimal: 0x43c2%
%c% codigos de formato reconoc
idos: %c %d %% %s
%d% codigos de formato no reco
nocidos: %i %f %e %g %p
*** Final fase 1_M

```

En esta versión de prueba no se realiza ningún tipo de multiplexación de procesos, ya que se ha definido de forma independiente al código del programador `progP`. Además, la función `_ga_printf()` también está modificada del mismo modo que se ha explicado en el apartado 4.6, de modo que las llamadas a la función `GARLIC_printf()` que realizan los programas de usuario se convierten en llamadas a la función `printf()` estándar del lenguaje C. Una vez mas, cabe recordar que este "bypass" solo se debe realizar mientras no se disponga del código fuente de la función `_gg_escribir()`, que tendrá que escribir el programador `progG`.

Hay que observar también que, a diferencia del proyecto para el programador `progP`, en esta ocasión **no hace falta enlazar el fichero `GARLIC_API.o`** con el proyecto del sistema operativo, ya que el código de los programas cargados ya incorporan dicho fichero objeto y, por tanto, invocan la API del sistema operativo a través del vector de funciones.



6.2 Inicialización del entorno gráfico

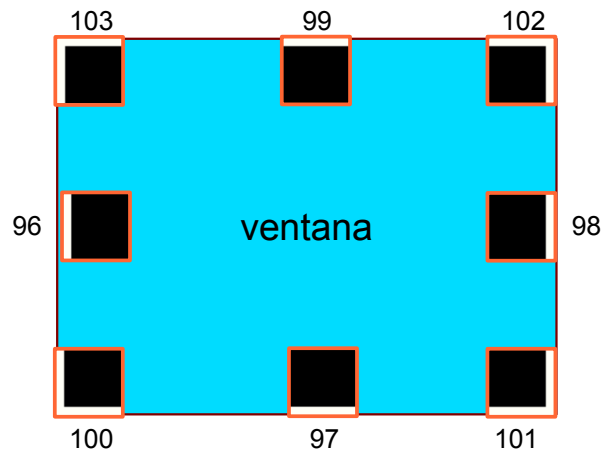
Dentro del fichero `garlic_graf.c`, el programador `progG` tiene que escribir la función `_gg_iniGrafA()`, en la que hay que realizar las siguientes tareas:

- inicializar el procesador gráfico principal (**A**) en modo 5, con salida en la pantalla superior de la NDS,
- reservar el banco de memoria de vídeo A,
- inicializar los fondos gráficos 2 y 3 en modo *Extended Rotation*, con un tamaño total de 512x512 píxeles,
- fijar el fondo 3 como más prioritario que el fondo 2,
- descomprimir el contenido de la fuente de letras sobre una zona adecuada de la memoria de vídeo,
- copiar la paleta de colores de la fuente de letras sobre la zona de memoria correspondiente,
- generar los marcos de las ventanas de texto en el fondo 3,
- escalar los fondos 2 y 3 para que se ajusten exactamente a las dimensiones de una pantalla de la NDS (reducción del 50%).

Para realizar todas estas tareas se pueden usar las funciones de **libnds**: `videoSetMode()`, `vramSetBankA()`, `bgInit()`, `bgSetPriority()`, `bgSetScale()`, `bgUpdate()`, `decompress()`, o cualquier otra función o definición para la configuración y gestión de los gráficos.

6.3 Dibujo de los marcos de las ventanas

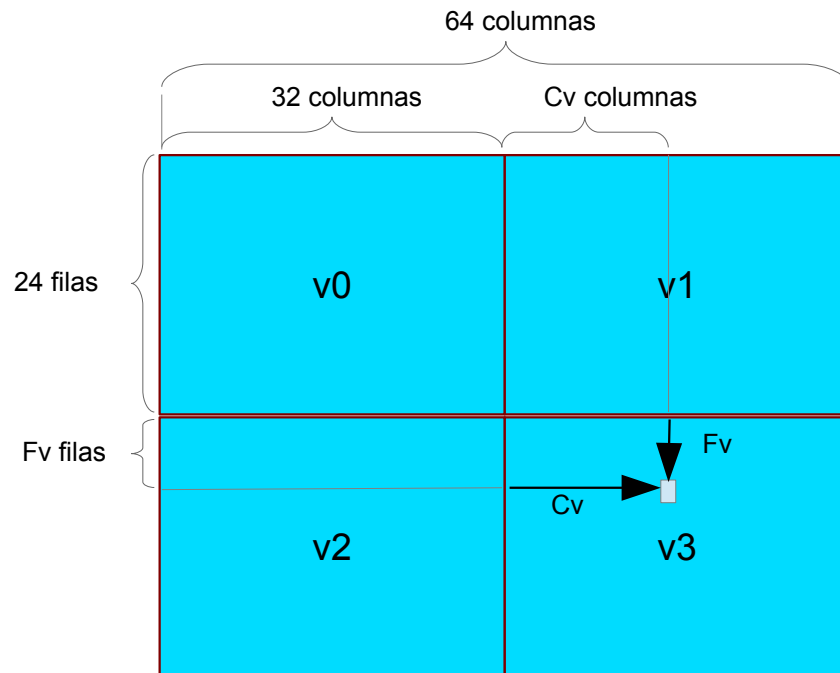
Otra función a realizar dentro de `garlic_graf.c` es `_gg_generarMarco()`, la cual tiene que dibujar los marcos de las ventanas utilizando los caracteres gráficos del 96 al 103:



Los códigos de estos caracteres gráficos son los índices de las baldosas de la fuente de texto que se cargan en la inicialización de los gráficos. Por lo tanto, para realizar el dibujo de los marcos hay que escribir dichos códigos en las posiciones adecuadas del mapa de caracteres del fondo 3.

El procesador gráfico de la NDS interpretará el color negro de los caracteres como el color transparente, de modo que los marcos de las ventanas en el fondo 3 se dibujarán sobre el texto de las ventanas, que se almacenará en el mapa de caracteres para el fondo 2.

La función `_gg_generarMarco()` recibe un único parámetro (`int v`), que indica sobre qué ventana hay que dibujar el marco. En la fase 1, este parámetro será un número del 0 al 3, que corresponderá a las ventanas que se indican en el siguiente gráfico:



Por lo tanto, hay que programar el acceso a los caracteres del mapa del fondo 3 o 2 según las coordenadas del carácter de la ventana (F_v , C_v) y el número de ventana a la que se tiene que acceder, para calcular el desplazamiento global de la posición correspondiente del mapa de caracteres.

Además, en la segunda fase de la práctica se trabajará con 16 ventanas, lo cual significa que **la programación del acceso a los mapas de caracteres tiene que ser lo más paramétrica posible**, para que cuando se cambie de fase solo sea necesario cambiar unas pocas constantes y el código pueda funcionar igualmente con las nuevas dimensiones.

Con el fin de realizar esta programación paramétrica, dentro del fichero **garlic_graf.c** se introducirá el siguiente conjunto de definiciones:

```
/* definiciones para realizar cálculos relativos a la posición de los
caracteres dentro de las ventanas gráficas, que pueden ser 4 o 16 */
#define NVENT      4           // número de ventanas totales
#define PPART      2           // número de ventanas horizontales o verticales
                                // (particiones de pantalla)
#define VCOLS      32          // columnas y filas de cualquier ventana
#define VFILS      24
#define PCOLS      VCOLS * PPART // número de columnas totales
#define PFILS      VFILS * PPART // número de filas totales
```

En las definiciones anteriores, cuando se pase a la fase 2 solo se tendrán que cambiar `NVENT` de 4 a 16 y `PPART` de 2 a 4. El número de filas y columnas de la ventana quedará igual, mientras que el número total de filas y columnas de la pantalla se actualizará según el cambio en `PPART`.

6.4 La escritura de texto

La función de API de la fase 1 para escribir texto es la `GARLIC_printf()`:

```
/* GARLIC_printf: escribe string en la ventana del proceso actual */
extern void GARLIC_printf(char * format, ...);
```

A través del fichero `GARLIC_API.o` y del vector de direcciones `APIVector`, esta función se enlaza con la rutina `_ga_printf()` definida en el fichero `garlic_itcm_api.s`:

```
.global _ga_printf
@;Parámetros
@; R0: char * format,
@; R1: unsigned int val1 (opcional),
@; R2: unsigned int val2 (opcional)
_ga_printf:
    push {r4, lr}
    ldr r4, =_gd_pidz      @; R4 = dirección _gd_pidz
    ldr r3, [r4]
    and r3, #0x3          @; R3 = ventana de salida (zócalo actual MOD 4)
    bl _gg_escribir
    pop {r4, pc}
```

Esta rutina recibe un *string* de formato por referencia por el registro `R0`, además de dos valores opcionales por los registros `R1`, `R2`, añade el número de ventana correspondiente al proceso actual en el registro `R3`, y redirecciona la llamada a otra función de nombre `_gg_escribir()`, incluida en el fichero `garlic_graf.c`, cuya descripción es la siguiente:

```
/* _gg_escribir: escribe una cadena de caracteres en la ventana
   indicada;
   Parámetros:
       formato -> cadena de formato, terminada con '\0';
                  admite '\n' (salto de línea), '\t'
                  (tabulador, 4 espacios) y códigos entre
```

```

32 y 159 (los 32 últimos son caracteres
gráficos), además de códigos de formato
%c, %d, %x y %s (max. 2 códigos por
cadena)
    vall1 -> valor a sustituir en primer código de
              formato, si existe
    val2 -> valor a sustituir en segundo código de
              formato, si existe
    - los valores pueden ser un código ASCII (%c),
      un valor atural de 32 bits (%d, %x) o un
      puntero a string (%s)
    ventana -> número de ventana (de 0 a 3)
*/
extern void _gg_escribir(char *formato, unsigned int vall1,
                        unsigned int val2, int ventana);

```

Por lo tanto, esta función deberá realizar los siguientes pasos (no definidos en orden estricto):

- convertir el *string* de formato y los valores pasados por parámetro en un mensaje de texto definitivo, sustituyendo los códigos de formato %c, %d, %x, %s y %% en los caracteres ASCII correspondientes a los valores tratados según el tipo de formato especificado,
- leer el campo `pControl` de la entrada `_gd_wbfs[ventana]`, y obtener la fila actual y el número de caracteres almacenados en el campo `pChars[]` (vector de 32 caracteres),
- analizar los caracteres del mensaje de texto definitivo, uno a uno, y añadir los códigos ASCII que correspondan al final del buffer de línea de la ventana, o sea, en el campo `pChars[]`:
 - si se trata de un tabulador ('\t'), añadir espacios en blanco hasta la próxima columna (posición del buffer) con índice múltiplo de 4,
 - si se trata de un carácter literal, añadir su código ASCII tal cual,
 - si se trata de un salto de línea ('\n') o se ha llenado el buffer de línea de la ventana, esperar el siguiente período de retroceso vertical, invocando la rutina `swiWaitForVBlank()`, para asegurar que el controlador de gráficos no está accediendo a la memoria de vídeo, y transferir los caracteres del buffer sobre las posiciones de memoria de vídeo correspondientes a la línea actual de escritura en ventana, utilizando la rutina `_gg_escribirLinea()` (ver más adelante),
- incrementar la línea actual de escritura; en el caso que el número de línea anterior fuera 23 (última fila), será necesario realizar un desplazamiento hacia arriba (*scroll*) con la rutina `_gg_desplazar()`

(ver más adelante) antes de transferir el contenido del buffer (punto anterior), para dejar sitio a la nueva línea,

- seguir con este proceso hasta el final del mensaje de texto definitivo, actualizando el campo `pControl` según el estado final de la transferencia (última posición de inserción).

La función `_gg_escribir()` tiene que diseñarse en base a una función auxiliar de nombre `_gg_procesarFormato()`, con una cabecera similar a la siguiente:

```
/* _gg_procesarFormato: copia los caracteres del string de formato
    sobre el string resultante, pero identifica las
    marcas de formato precedidas por '%' e inserta
    la representación ASCII de los valores
    indicados por parámetro.

    Parámetros:
        formato    ->    string con marcas de formato;
        val1, val2 ->    valores a transcribir, sean número de
                        código ASCII (%c), un número natural (%d,
                        %x) o un puntero a string (%s);
        resultado  ->    mensaje resultante.

    Observación:
        Se supone que el string resultante tiene reservado espacio
        de memoria suficiente para albergar todo el mensaje,
        incluyendo los caracteres literales del formato y la
        transcripción en código ASCII de los valores.
*/
void _gg_procesarFormato(char *formato, unsigned int val1,
                        unsigned int val2, char *resultado)
```

Es decir, se trata de una función (se puede escribir en lenguaje C) que convertirá el *string* de formato según las marcas de formato '%' y los valores de los parámetros, guardando el texto definitivo correspondiente sobre el espacio de memoria apuntado por la dirección que se pasa como tercer parámetro.

Por lo tanto, desde la función `_gg_escribir()` se tiene que reservar el espacio suficiente para el *string* con el texto definitivo, que se pasará por referencia a la función `_gg_procesarFormato()`. Para evitar tener que reservar una cantidad excesiva de memoria, vamos a suponer que el texto definitivo nunca superará las tres líneas de ventana, aunque se podría escribir un programa de usuario que superase esta restricción, por lo tanto, habría que prever esta situación e ignorar el texto que sobrepase el límite.

Para realizar la transformación de valores numéricos a sus correspondientes caracteres ASCII, se deben invocar las rutina de sistema implementadas

dentro del fichero `garlic_itcm_sys.s`, de nombre `_gs_num2str_dec()` y `_gs_num2str_hex()`, según se requiera formato decimal o hexadecimal.

Además de la función `_gg_procesarFormato()`, se tendrán que programar las dos rutinas auxiliares (en lenguaje ensamblador) que faltan para completar el algoritmo de escritura, dentro del fichero `garlic_itcm_graf.s`:

```
.global _gg_desplazar
@; Rutina para desplazar una posición hacia arriba todas las
@; filas de la ventana (v), y borrar el contenido de última fila
@;Parámetros:
@;   R0: ventana a desplazar (int v)
_gg_desplazar:

.global _gg_escribirLinea
@; Rutina para escribir toda una línea de caracteres almacenada
@; en el buffer de la ventana especificada;
@;Parámetros:
@;   R0: ventana a actualizar (int v)
@;   R1: fila actual (int f)
@;   R2: número de caracteres a escribir (int n)
_gg_escribirLinea:
```

Las dos rutinas auxiliares `_gg_escribirLinea()` y `_gg_desplazar()` también tienen que estar programadas de forma paramétrica respecto al número de ventanas, para que se puedan adaptar con facilidad a una nueva configuración (16 ventanas de la fase 2). Por lo tanto, dentro del fichero `garlic_itcm_graf.s` se proporcionan, en lenguaje ensamblador, los símbolos equivalentes a las definiciones propuestas para describir la geometría de las ventanas en las funciones de C:

```
NVENT = 4                @; número de ventanas totales
PPART = 2                @; número de ventanas horizontales o verticales
                        @; (particiones de pantalla)
L2_PPART = 1             @; log base 2 de PPART

VCOLS = 32               @; columnas y filas de cualquier ventana
VFILS = 24
PCOLS = VCOLS * PPART    @; número de columnas totales (en pantalla)
PFILS = VFILS * PPART    @; número de filas totales (en pantalla)

WBUFS_LEN = 36           @; longitud de cada buffer de ventana (32+4)
```

A parte de los símbolos ya explicados, se ha introducido el símbolo `L2_PPART` para poder realizar operaciones lógicas con el número de ventana (AND, OR, desplazamiento, etc.). Además, se introduce el símbolo `WBUFS_LEN` para poder calcular el desplazamiento de las entradas del vector `_gd_wbfs[]`.

Por último, en la rutina `_gg_escribirLinea()` hay que tener en cuenta que será necesario restar 32 al código ASCII de los caracteres imprimibles del mensaje, con el fin de obtener el índice de la baldosa que representa el carácter (gráfico de la fuente de letras), índice que se deberá copiar en la posición correspondiente del mapa de caracteres del fondo 2, siguiendo las mismas reglas explicadas en el apartado de dibujo de los marcos de las ventanas.

6.5 Programa principal para progG

Para poder realizar las tareas específicas de esta rama sin tener que depender del trabajo realizado por los compañeros de las otras ramas, se dispone el siguiente programa principal `main.c`:

```
/*-----
    "main.c" : fase 1 / programador G
    -----*/
#include <nds.h>
#include <garlic_system.h>

#include <GARLIC_API.h>
int hola(int);           // función que simula la ejecución del proceso
extern int prnt(int);    // otra función (externa) de test
                        // correspondiente a un proceso de usuario
...

//-----
void inicializarSistema() {
//-----
    int v;

    _gg_iniGrafA();       // inicializar procesador gráfico A
    for (v = 0; v < 4; v++) // para todas las ventanas
        _gd_wbfs[v].pControl = 0; // inicializar buffers
    ...
}
```



```
//-----
int main(int argc, char **argv) {
//-----

    inicializarSistema();

    _gg_escribir("*****", 0, 0, 0);
    _gg_escribir("*", 0, 0, 0);
    _gg_escribir("* Sistema Operativo GARLIC 1.0 *", 0, 0, 0);
    _gg_escribir("*", 0, 0, 0);
    _gg_escribir("*****", 0, 0, 0);
    _gg_escribir("*** Inicio fase 1_G\n", 0, 0, 0);

    _gd_pidz = 6;    // simular zócalo 6
    hola(0);

    _gd_pidz = 7;    // simular zócalo 7
    hola(2);

    _gd_pidz = 5;    // simular zócalo 5
    prnt(1);

    _gg_escribir("*** Final fase 1_G\n", 0, 0, 0);

    while(1) {
        swiWaitForVBlank();
    }
    // parar el procesador en un bucle infinito
    return 0;
}
```

La función `inicializarSistema()` llama a la función `_gg_iniGrafA()` para que se inicialice el entorno gráfico de la NDS, además de inicializar los buffers de las ventanas.

La función `main()` realiza llamadas a la función `_gg_escribir()` en vez de utilizar la función `printf()` de la librería C estándar, utilizando la ventana 0 para emitir los mensajes y pasando 0 como valores opcionales, ya que no hay marcas de formato en dichos mensajes.

Para simular la ejecución del proceso **HOLA** sin tener que cargar el fichero **HOLA.elf**, se dispone también de la función `hola()` programada dentro del fichero **main.c**, tal como se propone en la configuración de test para el programador **progP** (ver apartado 4.6).

La función `main()` llamará dos veces a la función `hola()` después de fijar el valor de la variable `_gd_pidz` a 6 y a 7, lo cual sirve para que las llamadas a la función `GARLIC_printf()` desde `hola()` utilicen la ventana 2 y 3 (número de zócalo módulo 4) para emitir sus mensajes.

Además, también se utiliza una llamada externa a la función `prnt()`, que se corresponde con la implementación del programa de usuario homónimo como función del sistema operativo. Al contrario que la función `hola()`, se ha decidido no incluirla dentro del fichero `main.c` debido a su considerable longitud. Aunque esté declarada en otro fichero `prnt.c`, se trata de una función que se incluirá dentro del código ejecutable del sistema operativo, puesto que todavía no podemos confiar que el programador `progM` haya terminado su función de carga de ficheros `.ELF`. Evidentemente, el fichero `prnt.c` **no** deberá formar parte del código fuente de la versión definitiva del sistema operativo GARLIC.

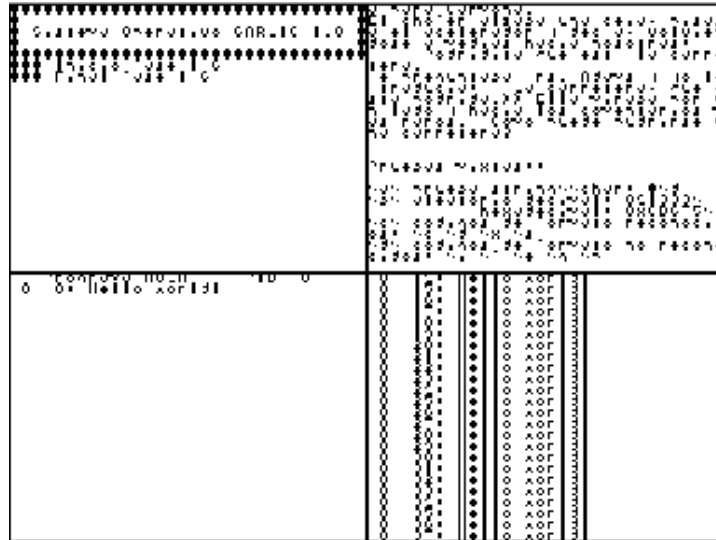
El resultado a obtener será similar al siguiente:

[illegible]

El programa funciona porque los mensajes emitidos por la función `main()` se han transferido a la ventana 0, los mensajes que emitidos por la función `hola()` se han transferido a las ventanas 2 y 3, así como los mensajes emitidos por la función `prnt()` se han transferido a la ventana 1.

También se comprueba que funciona el desplazamiento, cuando se emiten más de 24 líneas desde el inicio del proceso.

Sin embargo, la visualización anterior se ha obtenido con la opción **View Maps** del menú **Tools** del emulador **DeSmuME**. En realidad, la visualización de la ventana resultante en la pantalla superior de la NDS es similar a la siguiente:



Esto es debido a la reducción del zoom al 50%, con lo que resulta imposible leer los mensajes. Sin embargo, el propósito de la visualización reducida es simplemente observar la escritura de texto en ventanas diferentes, en este caso, de la 0 a la 3.

Igual que en la rama **progP**, para que toda la parte del programador **progG** pueda compilar y *linkar*, en el fichero **Makefile** de estas ramas se ha indicado que se enlace el proyecto con el fichero **GARLIC_API.o**, lo cual **no estará permitido en la versión final del proyecto**:

```
%elf:
    @echo linking $(notdir $@)
    $(LD) $(LDFLAGS) $(OFILES) $(LIBPATHS) $(LIBS) $(SFILES)
        $(GARLICAPI)/GARLIC_API.o -o $@
```

7 Tareas de gestión de teclado (progT)

Para la programación de la gestión de teclado no se proporciona ningún fichero de partida, de modo que hay que programar “**desde cero**”. De todos modos, se puede utilizar el código proporcionado para los otros roles como referencia.

A continuación se describen todos los pasos a realizar, indicados por ficheros a modificar o crear:

0. Crear la rama **progT** (ver apartado 1.5).
1. **GARLIC_API.h**: crear una nueva definición para la función de leer teclado:

```
extern int GARLIC_getstring(char * string, int max_char);
```

la cual recibe por parámetro la dirección de un vector de caracteres donde guardar el *string* introducido por teclado, así como el número máximo de caracteres que puede contener el vector (excluido el centinela), y devuelve como resultado el número de caracteres leídos finalmente (excluido el centinela).

2. **GARLIC_API.s**: crear la rutina `GARLIC_getstring()`, utilizando la siguiente entrada libre del vector de direcciones; recompilar el proyecto **GARLIC_API**.
3. **garlic_vectors.s**: crear una nueva entrada en el **APIVector**, que apunte a una nueva rutina de nombre `_ga_getstring()`.
4. **garlic_itcm_api.s**: crear la rutina `_ga_getstring()`, de modo similar a la rutina `_ga_printf()`, es decir, añadiendo un parámetro más para pasar el número de zócalo del proceso invocador y llamando a una rutina del **progT** para leer un *string* y pasárselo al proceso invocador (siguiente punto).
5. **garlic_itcm_tecl.s**: crear este nuevo fichero, que contendrá las rutinas en lenguaje ensamblador relativas a la gestión de teclado. Dentro de este fichero, crear la rutina `_gt_getstring()`, que recibirá los siguientes parámetros:

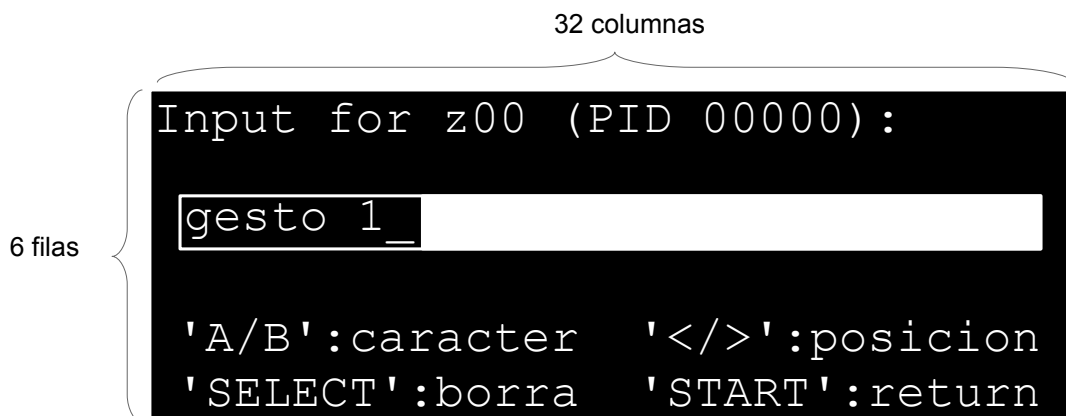
```
R0: string -> dirección base del vector de caracteres (bytes)
R1: max_char -> número máximo de caracteres del vector
R2: zocalo -> número de zócalo del proceso invocador
```

el algoritmo de la rutina `_gt_getstring()` tiene que ser similar al siguiente:

- si la interfaz de teclado está desactivada (oculta), mostrarla (ver punto 8) y activar la RSI de teclado (ver punto 9).
 - añadir el número de zócalo sobre un vector global `_gd_kbwait[]`, que se comportará como una cola en la cual estarán registrados los procesos que esperan la entrada de un *string* por teclado,
 - esperar a que el bit de una variable global `_gd_kbsignal`, correspondiente al número de zócalo indicado, se ponga a 1,
 - poner el bit anterior a cero, copiar el *string* leído sobre el vector que se ha pasado por parámetro, filtrando el número total de caracteres y añadiendo el centinela, y devolviendo el número total de caracteres leídos (excluido el centinela).
6. **garlic_dtcn.s**: añadir las variables globales descritas en el punto anterior, más las variables adicionales que se requieran (p.ej. longitud de la cola de procesos esperando al teclado).
 7. **garlic_tecl.c**: crear un nuevo fichero para contener las funciones necesarias para la gestión de teclado que sea más conveniente escribir en lenguaje C. Por ejemplo, crear una función para inicializar toda la información necesaria para la gestión del teclado, como las variables globales, los gráficos, instalar la RSI de teclado, etc.

```
void _gt_initKB();
```

8. **garlic_tecl.c**: añadir dos funciones para mostrar y ocultar la interfaz de teclado sobre la pantalla inferior de la NDS, utilizando las baldosas gráficas que se muestran en el apartado 6.1, creando un patrón similar al siguiente:



Para crear este patrón se tiene que inicializar el procesador gráfico secundario con un fondo para texto (baldosas), paleta, etc.; todas estas tareas son similares a las descritas en los apartados 6.2 y 6.3.

En la cabecera de la ventana hay que generar los caracteres para indicar el zócalo y el PID del proceso que pide la información (hay que sustituir los ceros).

En el pie de la ventana se informa al usuario de las teclas para manejar la interfaz:

- '**A/B**': botones '**A**' y '**B**' de la NDS para decrementar e incrementar el código ASCII del carácter donde se encuentra situado el cursor,
- '</>': flechas izquierda y derecha para mover el cursor entre todos los caracteres introducidos, más una posición final para introducir un nuevo carácter,
- '**SELECT**': permitirá borrar el carácter donde se encuentra situado el cursor, enganchando el resto del *string* a continuación,
- '**START**': dará por terminado el *string*.

Las rutinas para mostrar y ocultar el teclado pueden tener nombres como:

```
void _gt_showKB();
```

```
void _gt_hideKB();
```

9. **garlic_itcm_tecl.s**: crear la RSI del teclado, que permitirá el manejo de la interfaz de teclado mostrada en el apartado anterior, procesando las teclas tal como se ha explicado y, cuando se pulse la tecla '**START**', poner a 1 el bit correspondiente al zócalo del proceso para el cual va destinado el *string* en la variable `_gd_kbsignal`, además de ocultar la interfaz o reiniciarla para el siguiente proceso que esté esperando entrada de información.

```
void _gt_rsiKB();
```

10. **garlic_dtcmm.s**: obviamente, será necesario declarar más variables auxiliares para toda la gestión de las teclas (posición cursor, código ASCII actual, contenido actual del *string*, etc.).

11. **garlic_system.h**: crear las definiciones externas en C para cualquier variable, rutina o función de la gestión de teclado que tenga que ser accesible desde distintos ficheros del sistema operativo.
12. **main.c**: crear un programa principal, que inicialmente puede ser similar al suministrado para el **progP** (ver apartado 4.6), ya que se tiene que poder probar la llamada a la función `GARLIC_getstring()` sin cargar el programa en memoria (tareas del **progM**) y sin requerir la gestión de ventanas de texto (tareas del **progG**); por lo tanto, hay que cambiar el contenido de la función `hola()` para simular la lectura por teclado desde un programa para GARLIC.

Por último, cabe remarcar que la gestión de teclado tiene que ser **compatible con la concurrencia de procesos**, de modo que más de un proceso pueda estar "bloqueado" esperando entrada de texto, aunque la interfaz de entrada de texto los atenderá de manera secuencial (según orden de llegada a la cola).

Esta capacidad, sin embargo, solo la podremos probar si las tareas de gestión de procesos (**progP**) **se terminan con antelación a la fecha límite de entrega del proyecto**, de modo que dé tiempo para ajustar las tareas de gestión del teclado (**progT**) que deben funcionar de forma concurrente entre diversos procesos del sistema. En cualquier caso, no será posible realizar dichas pruebas si la rama **progT** no se fusiona con la rama **progP**, de modo que será obligatorio la presentación de las dos ramas para que el programador de teclado tenga opción a los 2 puntos adicionales de su rol, lo cual explica la limitación de puntuación en el caso de que **progT** realice una entrega individual (ver apartado 1.4).

8 Tareas de integración del código (master)

Las tareas de fusión e integración de código de las distintas ramas se repartirá entre los propios programadores.

Al final hay que conseguir una única versión del sistema operativo, fusionando sobre la rama **master** todo el código implementado. El último *commit* de fusión sobre la rama **master** se considerará como la versión definitiva del sistema **GARLIC_OS 1.0**.

Para realizar las fusiones se debe utilizar el comando `git merge --no-ff id_rama`, habitualmente desde la rama **master**. Se recomienda empezar las fusiones con la rama **progG**. Esta primera fusión es trivial, puesto que el contenido de la rama **master** se igualará al contenido de la rama **progG** sin ningún conflicto:

```
(progG)$ git checkout master
(master)$ git merge --no-ff progG
```

A continuación, se recomienda realizar la fusión de la rama **master** con la rama **progP**, o con la fusión de la rama **progP** y **progT**. En el segundo caso, la fusión del código de gestión de procesos y de gestión de teclado se puede realizar de forma independiente a la rama **master**, por ejemplo, sobre la rama **progT**.

Inevitablemente, las nuevas fusiones generarán conflictos debido a las versiones divergentes entre diversos ficheros. **Hay que arreglar dichos conflictos manualmente**, editando los ficheros correspondientes y efectuando un *commit* de fusión.

Por último, se debe realizar la fusión de la rama **master** con la rama **progM**, para que el sistema pueda cargar los procesos desde el fichero **HOLA.elf**, el fichero **PRNT.elf** o cualquier otro fichero ejecutable creado por los miembros del grupo (recordemos que cada alumno debe presentar un programa distinto para GARLIC).

Hay que tener en cuenta que, en la compilación del proyecto **GARLIC** final, **no se deberá enlazar el fichero **GARLIC_API.o****, ya que este fichero está destinado a los programas de usuario, no al sistema operativo. Para verificar que esto no ocurre, hay que **cerciorarse** de que las líneas del fichero **Makefile** que se encargan de enlazar el proyecto son las siguientes:

```
%.elf:
    @echo linking $(notdir $@)
    $(LD) $(LDFLAGS) $(OFILES) $(LIBPATHS) $(LIBS) $(SFILES) -o $@
```


Por otro lado, hay que consolidar todo el código para que las rutinas y funciones de los distintos programadores se llamen entre sí cuando sea pertinente. El ejemplo más representativo de esta directriz es la sustitución de las llamadas a la función `swiWaitForVBlank()`, dentro del código de `_gg_escribir()` (o de cualquier otra rutina o función), por llamadas a `_gp_waitForVBlank()`, ya que es la rutina propia de GARLIC para sincronizarse con el retroceso vertical.

Al final, el aspecto de la evolución de las versiones del proyecto puede ser similar al siguiente:




Nota: la representación gráfica real en el **gitk** variará significativamente según el orden de los *commits* efectuados.

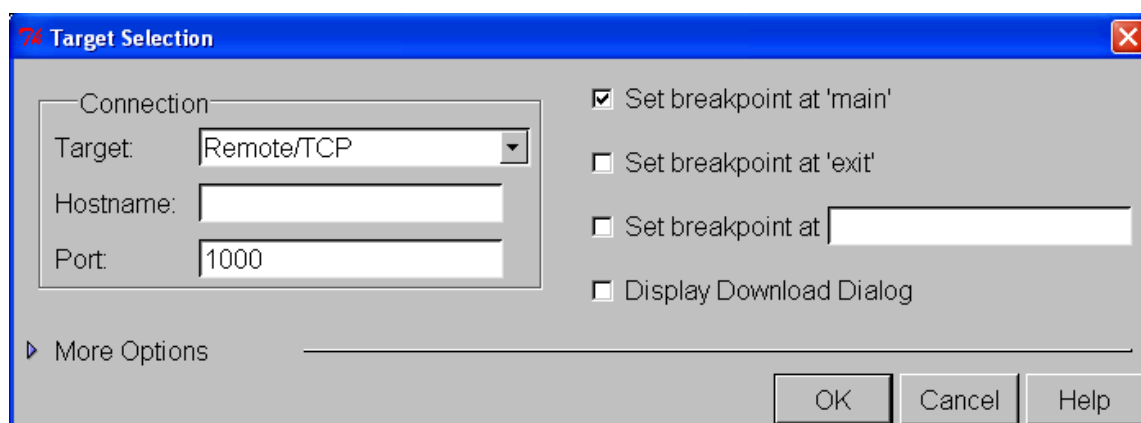
9 Consejos para la depuración del código

En todas las ramas se tendrán que efectuar procesos de depuración exhaustiva del código fuente en C o en ensamblador.

El entorno de depuración se activa automáticamente invocando la opción `debug` del comando `make`, lo cual se puede realizar añadiendo un nuevo comando (si todavía no se ha definido) al menú **Tools** del entorno de desarrollo **Programmer's Notepad**.

La opción `debug` del fichero **Makefile** invoca al emulador **DeSmuME** en modo depuración (**DeSmuME_dev.exe**), conectándolo con el depurador **Insight** a través del puerto TCP número 1000.

Cuando se pulsa el botón de **Run** , hay que indicar al depurador que se conecte al destino (**Target**) mediante la opción "**Remote/TCP**", además de activar el *breakpoint* de inicio de programa ('**main**')




A partir de este momento, el programa que constituye el sistema GARLIC ya estará cargado en el **Insight**. Cuando realizamos la ejecución paso a paso, el emulador y el depurador se enviarán mensajes a través del puerto TCP indicado para ir efectuando la ejecución de instrucciones y modificación del entorno de la plataforma NDS.

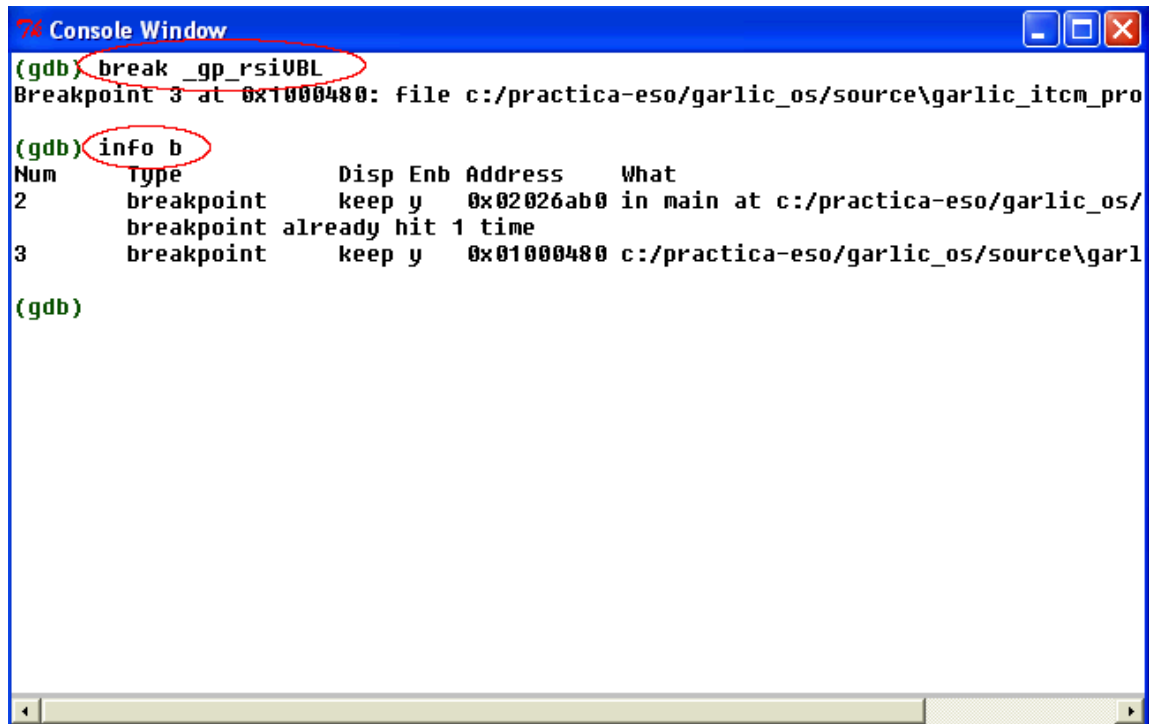
Sin embargo, puede resultar muy tedioso realizar la ejecución instrucción a instrucción hasta llegar al punto del programa que queremos comprobar.

Por otro lado, **con la ejecución paso a paso no se podrán depurar las RSIs**, porque el emulador no generará las interrupciones a menos que se le permita "correr".

Para subsanar estos dos problemas no queda más remedio que usar puntos de ruptura o *breakpoints*. El depurador **Insight** no permite activar

correctamente los *breakpoints* desde su interfaz gráfica. Por este motivo, habrá que fijarlos mediante la **consola** del **Insight**, que se abre mediante el botón .

Una vez dentro de la consola, se podrán añadir *breakpoints* con el comando `break` más el nombre de una función que queramos depurar, por ejemplo, la `_gp_rsiVBL()`. Para consultar los *breakpoints* instalados, se puede invocar el comando `info b`:




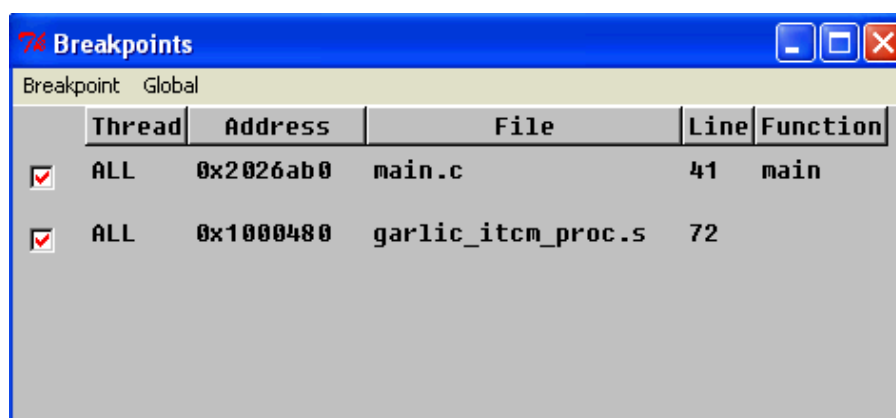
```

74 Console Window
(gdb) break _gp_rsiVBL
Breakpoint 3 at 0x1000480: file c:/practica-eso/garlic_os/source\garlic_itcm_pro

(gdb) info b
Num      Type             Disp Enb Address      What
2        breakpoint        keep y   0x02026ab0 in main at c:/practica-eso/garlic_os/
        breakpoint already hit 1 time
3        breakpoint        keep y   0x01000480 c:/practica-eso/garlic_os/source\garl


(gdb)
  
```

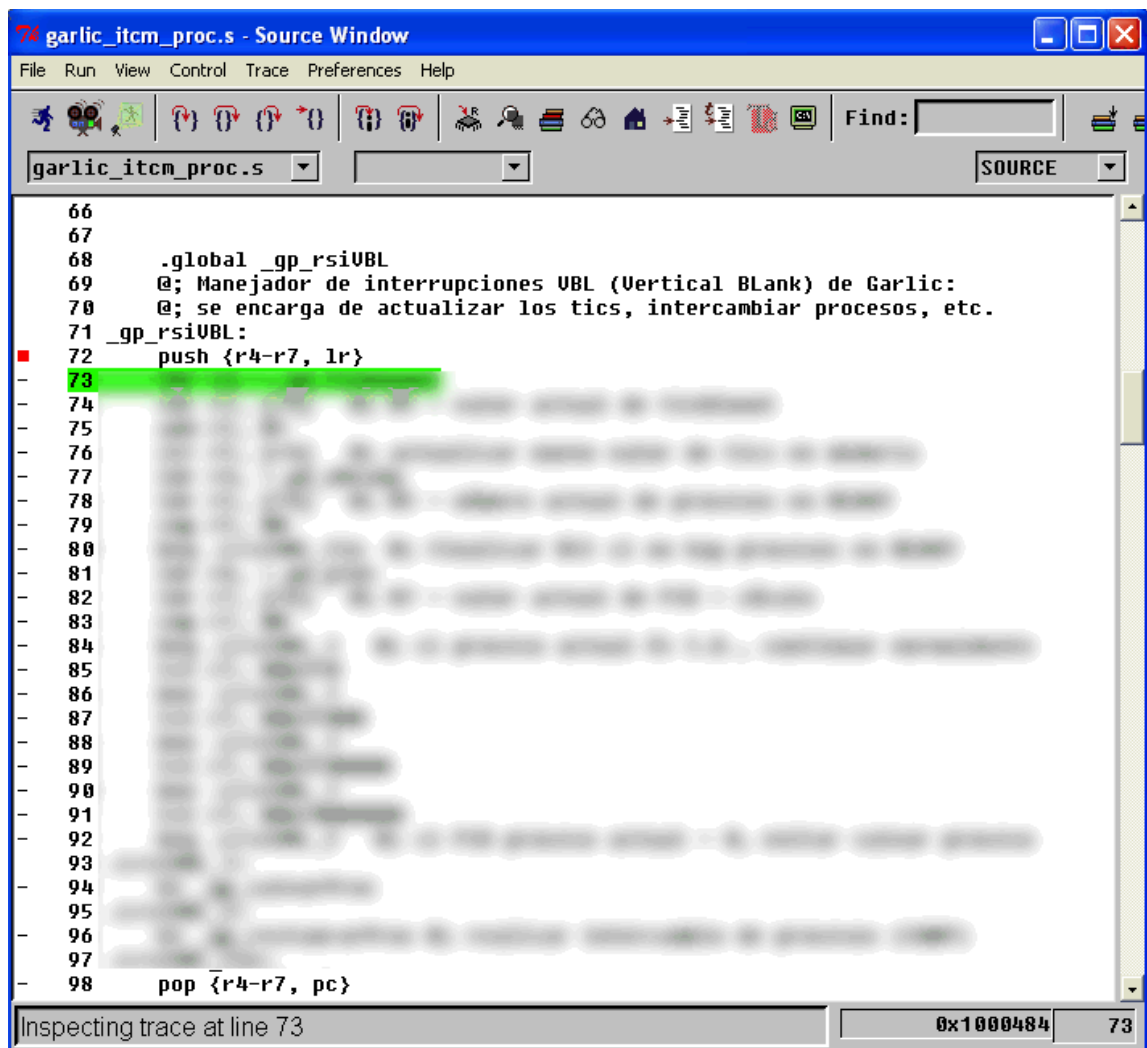
Para consultar los *breakpoints* también resulta útil la ventana de control de *breakpoints* que se abre cuando pulsamos el botón :




Breakpoint Global					
	Thread	Address	File	Line	Function
<input checked="" type="checkbox"/>	ALL	0x2026ab0	main.c	41	main
<input checked="" type="checkbox"/>	ALL	0x1000480	garlic_itcm_proc.s	72	

Desde esta ventana se puede consultar, activar, desactivar y borrar cualquier *breakpoint*. Sin embargo, la única forma **fiable** de crearlos es a través del comando `break`.

Una vez fijado el *breakpoint* en `_gp_rsiVBL()`, podemos indicar al emulador que siga la ejecución con el botón de continuar . La primera IRQ de retroceso vertical detendrá la ejecución en el *breakpoint*:



Otro defecto el depurador **Insight** es que a veces no para exactamente en el punto de ruptura. En la imagen anterior se observa que el *breakpoint* está correctamente fijado en la primera instrucción de la RSI, pero la ejecución se ha parado en la siguiente instrucción. Esto es debido a las diferentes variantes del valor del registro `PC` debidas al proceso de segmentación de la ejecución de las instrucciones del programa que es está ejecutando, pero no interferirá excesivamente en el proceso de depuración.

Otro problema que pueden surgir es que el depurador se detenga antes de llegar a la instrucción que tiene el punto de ruptura. En este caso se recomienda avanzar paso a paso con el botón de siguiente instrucción de lenguaje máquina .

A pesar de todos los inconveniente del depurador gráfico **Insight**, en general resulta más conveniente que utilizar la versión de texto **gdb** que funciona por debajo, en el sentido que la interfaz gráfica de **Insight** ofrece mucha más información simultánea y es obvio que también resulta más fácil de manipular.

Para más información sobre la configuración del sistema de desarrollo y depuración, referirse al manual de prácticas de la asignatura de *Computadores*.