Tema 4:

Programación a bajo nivel en C

Programación de sistemas y dispositivos

José Manuel Mendías Cuadros
Dpto. Arquitectura de Computadores y Automática
Universidad Complutense de Madrid

Contenidos

- ✓ Introducción
- ✓ Organización y estructura de un programa.
- ✓ Tipos de datos y operadores.
- ✓ Manipulación de bits.
- ✓ Punteros.
- ✓ Clases de almacenamiento.
- ✓ Inclusión de código máquina.
- ✓ Referencias cruzadas ensamblador-C.
- ✓ Compilación de funciones.
- ✓ Funciones in-line.
- ✓ Compilación de rutinas de excepción.
- ✓ Enlazado y ubicación de código.

C, ¿por qué?

- Las aplicaciones que corren en un sistemas empotrado se diferencian de las aplicaciones convencionales en que:
  - Deben respetar ligaduras de índole física: tiempos de respuesta, consumo, etc.
  - Hacen uso de un conjunto de recursos limitado.
  - Deben interactuar directamente y en tiempo real con el entorno.
  - Para ellas el hardware no es transparente.
  - No tienen por qué depender de terceros: bibliotecas, sistema operativo, etc.
  - Deben ser especialmente robustas.
- El lenguaje C es uno de los preferidos para programación de sistemas:
  - El código objeto es pequeño y eficiente (el lenguaje añade poca sobrecarga).
  - Permite optimizar el uso de memoria y el tiempo de ejecución.
  - Permite un control directo del hardware y los dispositivos.
  - Frente a ensamblador, reduce el tiempo de desarrollo y mantenimiento.
- No obstante frente a otros lenguajes:
  - Ofrece poca protección en tiempo de compilación y ninguna en tiempo de ejecución.
  - Es bastante críptico.

C para programadores C++

guía rápida (i)

- El lenguaje C no soporta:
  - Construcciones orientadas a POO (`class` / `public` / `private` / `friend` ...)
  - Funciones operador (`operator`).
  - Sobrecarga de funciones.
  - Argumentos implícitos.
  - Aserciones (`assert`)
  - Espacios de nombres (`namespace` / `using`)
  - Declarar en las funciones parámetros formales por referencia (`&`)
    - En su lugar, las variables de salida se declaran de tipo puntero
  - Las primitivas de memoria dinámica (`new` / `delete`)
    - En su lugar, usar las funciones `malloc` / `free` de la biblioteca `stdlib`
  - El tipo `bool`
    - En su lugar, 0 es falso, el resto es cierto
  - El tipo `string`
    - En su lugar, usar cadenas de `char` y como centinela de fin de cadena el carácter `'\0'`
  - Flujos de caracteres, ni operadores de extracción/inserción (`cin >>` / `cout <<`)
    - En su lugar se usan funciones `printf`, `scanf` de la biblioteca `stdio`

J.M. Mendías
2022

C para programadores C++

guía rápida (ii)

- Además, en C a diferencia de C++
  - Las variables locales (incluidos iteradores) deben declararse al inicio de la función
  - Las variables globales deben declararse al inicio del código
  - Los arrays se pasan por referencia y el parámetro formal debe ser de tipo puntero

- Por último, téngase en cuenta que en un sistema empotrado puede que no haya SO ni un conjunto de dispositivos convencionales:
  - No hay donde volver tras la finalización de una aplicación
    - Se programa para que no tenga fin.
  - Las bibliotecas de entrada/salida por consola convencionales son inútiles.
    - Es necesario adaptarlas al dispositivo de E/S disponible (por ejemplo, UART)
  - El acceso a ficheros puede no tener sentido
    - En caso de tenerlo, será necesario adaptar las bibliotecas al soporte físico disponible.
  - En general, el uso de bibliotecas convencionales se debe hacer con precaución
    - Están programadas genéricamente y pueden ser ineficientes para un sistema concreto.
  - Ídem para la elección del compilador cruzado y selección de opciones de compilación.

Organización de un programa

- Un programa C es una colección de ficheros fuente en C
  - Ficheros de implementación (`*.c`): contienen definiciones de variables y funciones tanto globales como locales al módulo.
    - La entrada al programa se hace a través de la función `main`
  - Ficheros de cabecera (`*.h`): contienen declaraciones de tipos, variables y/o prototipos de funciones globales.
    - Cada fichero de módulo debe tener un fichero de cabecera asociado
  - Los ficheros de cabecera deben ser incluidos en aquellos ficheros de implementación que necesitan las declaraciones a través de la directiva del preprocesador `#include`

- Previo a su compilación, todo módulo C es preprocesado según lo indicado en las directivas que incluye el código, lo que supone:
  - Combinar ficheros fuente
  - Expandir macros
  - Incluir selectivamente secciones de código fuente
  - Eliminar comentarios

J.M. Mendías
2022

Directivas del preprocesador

- **#define**
  - Permite la definición de macros y, en particular, de constantes simbólicas.
  - Especialmente utilizado para disponer de nemónicos de los recursos hardware.

```
#define SEGS (*(volatile unsigned char *)0x2140000)
```

- **#include**
  - Permite incluir en un fichero el contenido de otro (típicamente de cabecera)

```
#include <fichero.h>
```
el fichero se encuentra en alguno de los directorios indicados por la configuración del proyecto

```
#include "fichero.h"
```
el fichero se encuentra en el directorio del fichero que lo incluye

- **#ifdef / #ifndef y #endif**
  - Permiten incluir una sección de código según esté o no definida una cierta macro.
  - Especialmente utilizado para evitar referencias circulares y redeclaraciones.

```
#ifndef __SEGS_H__
#define __SEGS_H__
...
#endif
```
la sección de código comprendida entre **#ifndef** y **#endif** se procesa solo si no ha sido previamente definida la macro **__SEGS_H__**

Estructura de un programa

```
#ifndef __SEGS_H__
#define __SEGS_H__

#define SEGS_OFF 0xff
#define SEGS (*(volatile unsigned char *)0x2140000)

void segs_init( void );
void segs_putchar( char ch );
...

#endif
```

declaración de macros y nemotécnicos

declaración de prototipos de funciones públicas

Fichero cabecera

```
#include <segs.h>

static char segs_state = SEGS_OFF;

void segs_init( void )
{
    segs_state = SEGS_OFF;
    SEGS = SEGS_OFF;
}
```

declaración de variables globales privadas al módulo

definición de funciones

Fichero implementación

Estructura de un programa

```
#include <common_types.h>
#include <system.h>
#include <segs.h>

uint32 bar;

int32 foo( ... )
{
    ...
}

void main( void )
{
    char i;

    segs_init();

    while( 1 ){
        for( i=0; i<=0xF; i++)
            segs_putchar( i );
        ...
    };
}
```

inclusión de declaraciones de macros y prototipos

declaración de variables globales

definición de funciones

punto de entrada: función principal

declaración de variables locales a la función

fase de inicialización

fase de operación,
es un bucle infinito ya que no hay SSOO al que volver.

Fichero principal

Tipos de datos

taxonomía

- Atómicos:
  - Enteros: [**signed**/**unsigned**] **char**, [**signed**/**unsigned**] [**long**/**short**] **int**, **enum**
  - Reales: **float**, **double** , **long double**
  - Sin tipo: **void**
  - Punteros (*tipo* *): el dato es la dirección de almacenamiento de otro dato o función.
- Compuestos:
  - Array: colección de datos un mismo tipo.
  - Estructura (**struct**): colección de datos de distinto tipo.
  - Campo de bits (**struct**): colección de enteros de tamaños no estándar.
  - Unión (**union**): colección de datos de distinto tipo que comparten ubicación en memoria.
- El programador puede definir nuevos nombres de tipo con **typedef.**
- Los tipos pueden ser calificados:
  - Constante (**const**): el dato no puede modificar su valor.
  - Volátil (**volatile**): el dato puede ser modificado por un agente ajeno al programa
    - por ejemplo, registro de un dispositivo mapeado en memoria.
    - indica al compilador que en toda referencia al dato debe recurrir a la dirección de memoria que ocupa (en lugar almacenarlo con antelación en un registro y acceder a éste)
    - Importante: en sistemas con cache no implica que físicamente se acceda a memoria, luego cuando se necesite habrá que indicarlo al controlador de cache (marcado como no cacheable)

Tipos de datos

tamaño en memoria (i)

- El tamaño de los tipos atómicos depende de la implementación
  - puede conocerse con **`sizeof( tipo )`**
  - se recomienda definir y usar tipos propios en lugar de los predefinidos.

S3C44B0X - GCC

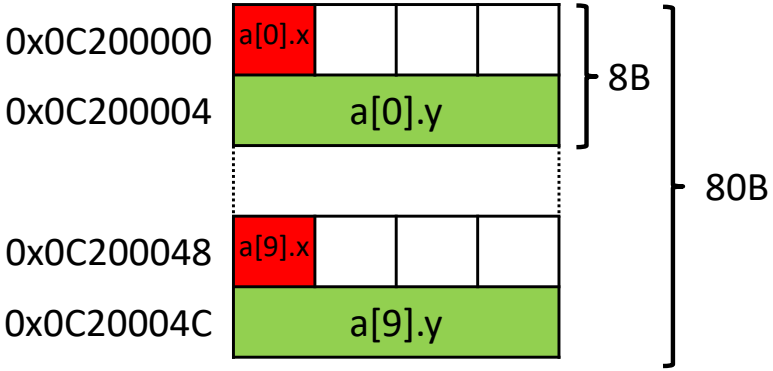
tipo predefinido	tamaño	common_types.h
`---`	8b	`boolean`
`[signed/unsigned] char`	8b	`int8/uint8`
`[signed/unsigned] short int`	16b	`int16/uint16`
`[signed/unsigned] int`	32b	`int32/uint32`
`[signed/unsigned] long int`	32b	`---`
`[signed/unsigned] long long int`	64b	`int64/uint64`
`float`	32b	`---`
`double`	64b	`---`
`long double`	64b	`---`
`puntero`	32b	`---`

- El tamaño de los tipos compuestos depende del tamaño de los atómicos y de su alineamiento.

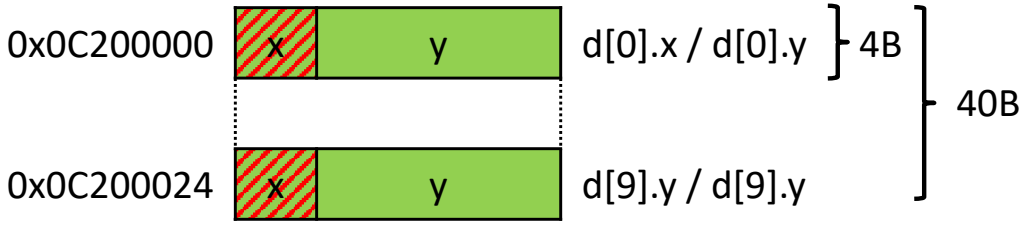
Tipos de datos

tamaño en memoria (ii)

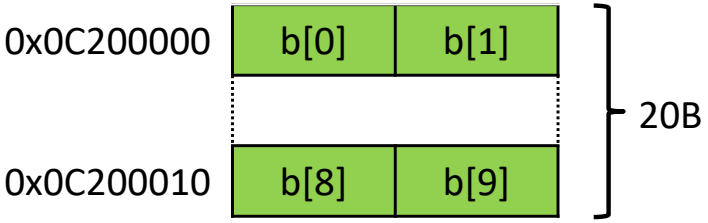
```
struct foo {
  int8 x; int32 y
} a[10];
```



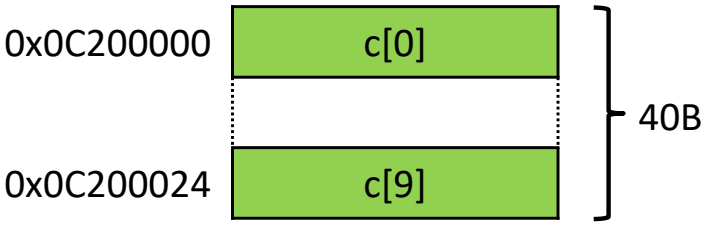

```
union bar {
  int8 x; int32 y
} d[10];
```




```
int16 b[10];
```




```
int32 c[10];
```

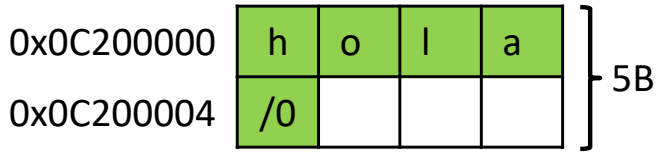



```
struct pllcon
{
  uint32 mdiv: 7;
  uint32 pdiv: 6;
  uint32 sdiv: 2;
} e;
```

¡¡ NO USAR !!

la ubicación exacta de los campos dentro de la palabra depende de la implementación

```
char f[] = "hola";
```



Operadores

- Como todo lenguaje, C dispone de una colección de operadores básicos de tipo aritmético, lógico y relacional.
  - Que se completan con un conjunto estándar de funciones matemáticas (math.h)

operador		operación
`++x`	`x++`	pre/post incremento
`--x`	`x--`	pre/post decremento
`-x`		negación aritmética
`x + y`	`x += y`	suma
`x - y`	`x -= y`	resta
`x * y`	`x *= y`	multiplicación
`x / y`	`x /= y`	división
`x % y`	`x %= y`	módulo

operador	operación
`!x`	negación lógica
`x && y`	y-lógica
`x || y`	o-lógica
`x > y`	mayor
`x >= y`	mayor o igual
`x < y`	menor
`x <= y`	menor o igual
`x == y`	igual
`x != y`	distinto

no confundir con operadores bit a bit `x&y` `x|y`

no confundir con asignación `x = y`

Consideraciones sobre aritmética

- Intentar usar el tipo de dato más pequeño capaz de cumplir su función.
  - Al mezclar distintos tipos de datos en una expresión, el compilador promociona los de menor tamaño al mayor de ellos, para controlarlo usar *casting*: **`(tipo)expresion`**
- Intentar usar al máximo suma/resta entera
  - Las variantes con/sin signo no afectan a la velocidad.
  - Las variantes de tamaño no afectan a la velocidad si no superan el tamaño de palabra.
- Evitar multiplicación/división entera y el uso de punto flotante a menos que el procesador disponga de hardware específico para ello.
  - El core ARM7TDI del S3C44B0X tiene un multiplicador entero de 32x8 bits, no dispone de divisor entero y no da soporte a punto flotante.
  - Las operaciones se realizan por funciones software que pueden no estar optimizadas.
- Reducir al mínimo el uso de funciones de bibliotecas matemáticas.

tipo	+	*	/
`int8 / uint8 / int16 / uint16 / int32 / uint32`	1	1	~100
`int64 / uint64`	2	5	~500
`float`	~50	~30	~100

num. instrucciones (S3C44B0X-GCC)

Manipulación de bits

- El lenguaje C dispone de operadores específicos para manipular bits.
  - Se usan para acceder a bits concretos de los registros de control de periféricos.
  - Típicamente el argumento derecho es una constante que se usa de máscara.

operador		operación	utilidad
`x & y`	`x &= y`	y-lógica bit a bit	poner algunos bits a 0
`x | y`	`x |= y`	o-lógica bit a bit	poner algunos bits a 1
`x ^ y`	`x ^= y`	o-exclusiva-lógica bit a bit	complementar algunos bits
`~ x`		negación bit a bit	complementar todos los bits
`x << n`	`x <<= n`	desplazamiento a izquierdas	poner un valor en cierta posición
`x >> n`	`x >>= n`	desplazamiento a derechas	

- Los literales numéricos pueden expresarse en:
  - Octal: comienzan por **`0`**.
  - Hexadecimal: comienzan por **`0x`**.
  - Decimal: no comienzan por **`0`**.

```
foo = 0x2a;
foo = 052;
foo = 42;
```

son equivalentes

Manipulación de bits

```
uint8 a;
...
a = a | 0x4;
a |= 0x4;
a |= (1 << 2);
a |= (1 << 5) | (1 << 2);
...
a = a & ~0x8;
a &= ~0x8;
a &= ~(1 << 3);
a &= ~((1 << 4) | (1 << 3));
...
a ^= (1 << 5);
a ^= (1 << 5) | (1 << 0);
...
if( a & (1 << 4) ){
   ...
}
```

son equivalentes:
ponen a 1 el bit 2 (máscara = 0x4 = 00000100)

pone a 1 los bits 5 y 2 (máscara = 00100100)

son equivalentes:
ponen a 0 el bit 3 (máscara = ~ 0x8 = 11110111)

pone a 0 los bits 4 y 3 (máscara = 11100111)

invierte el bit 5 (máscara = 00100000)

invierte los bits 5 y 0 (máscara = 00100001)

el cuerpo del if se ejecuta si el bit 4 vale 1 (máscara = 00100000)

```
#define PLLCON (*(volatile uint32 *)(0x01d80000))
..
PLLCON = 0x38021;
PLLCON = (56 << 12) | (2 << 4) | (1 << 0);
PLLCON = (56 << 12) + (2 << 4) + 1;
```

PLLCON = 0x38021 (0011.1000.XX00.0010.XX01)

PLLCON[19:12] = 56	MDIV
PLLCON[9:4] = 2	PDIV
PLLCON[1:0] = 1	SDIV

Manipulación de bits

- Por comodidad pueden definirse macros de manipulación de bits
  - pero por eficiencia jamás como funciones.

```
#define BITSET( var, bitnum ) ((var) |= (1 << (bitnum)))
#define BITCLR( var, bitnum ) ((var) &= ~(1 << (bitnum)))
#define BITSW( var, bitnum )  ((var) ^= (1 << (bitnum)))
#define BITTST( var, bitnum ) ((var) & (1 << (bitnum)))
...
unsigned char a;
...
a |= (1 << 2);
BITSET( a, 2 );
...
a &= ~(1 << 3);
BITCLR( a, 3 );
...
if( a & (1 << 4) ){
   ...
}
if( BITTST(a, 4) ){
   ...
};
```

son equivalentes:
ponen a 1 el bit 2 (máscara = 00000100)

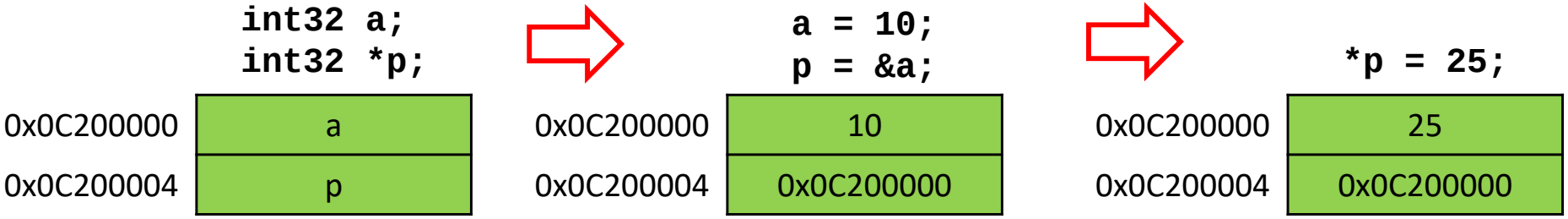
son equivalentes:
ponen a 0 el bit 3 (máscara = 11110111)

son equivalentes:
el cuerpo del if se ejecuta si el bit 4 vale 1 (máscara = 00100000)

J.M. Mendías
2022

Punteros

- Todo dato atómico está en memoria a partir de una dirección inicial.
  - La dirección del dato (referencia) puede obtenerse utilizando el operador **&**.
  - La dirección del dato puede almacenarse en una variable de tipo puntero.
  - Un puntero puede desreferenciarse (obtener el contenido de la dirección a la que apunta) utilizando el operador *****.

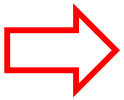


- Todo array está en memoria contigua a partir de una dirección inicial.
  - El identificador de un array es un puntero a su dirección de inicio.
  - Una indexación de dicho identificador (con corchetes) desreferencia a dicho puntero
- Toda función está en memoria contigua a partir de una dirección inicial.
  - El identificador de la función es un puntero a su dirección de inicio.
  - Una llamada a función (con paréntesis) desreferencia a dicho puntero.

J.M. Mendías
2022

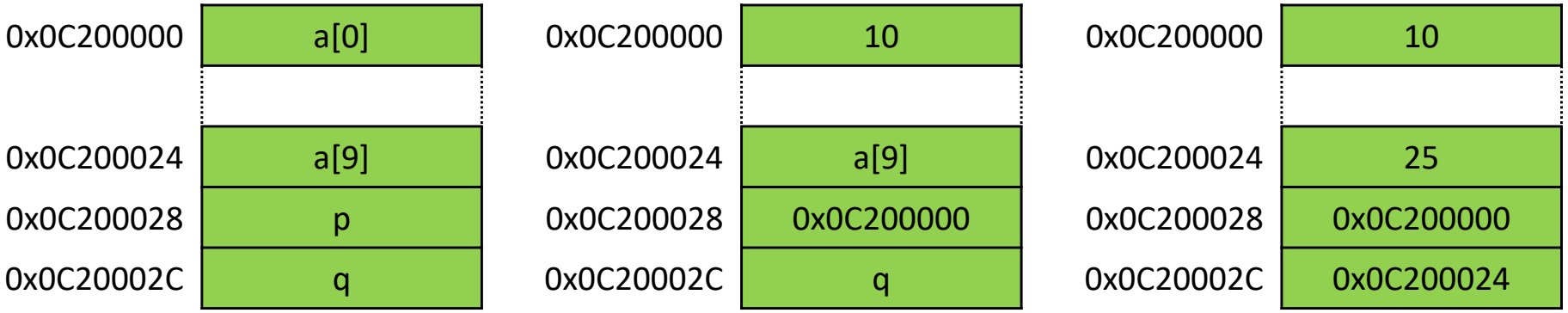
Punteros

```
int32 a[10];
int32 *p, *q;
```



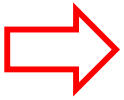
```
p = a;
*p = 10;
```

```
q = p + 9;
*q = 25;
```

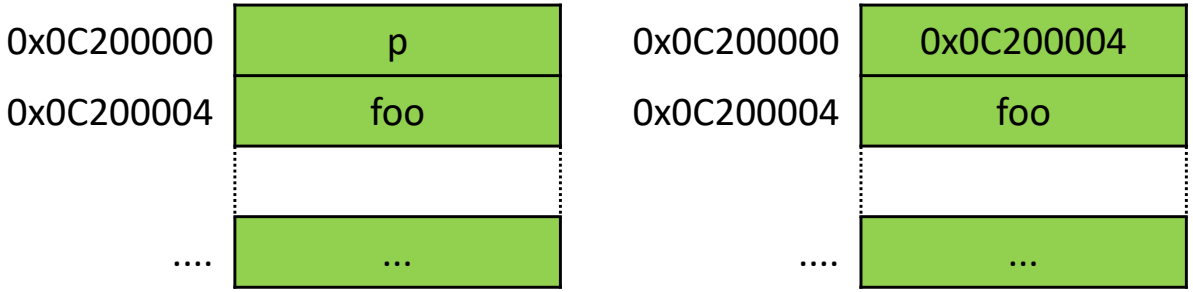



```
a     ≡ &(a[0])
*a    ≡ a[0]
a[9]  ≡ *(a+9)
a+9   ≡ 0x0C200000
          +9*sizeof(int32)
```

```
void (*p)( void );
void foo( void )
{
...
};
```



```
p = foo;
(*p)();
```




```
foo() ≡ (*p)()
```

Punteros

- Los punteros pueden ser dobles, triples...
  - El identificador de un array bidimensional es un doble puntero

```
int32 a;
int32 *p;
int32 **dp;
```

0x0C200000	a
0x0C200004	p
0x0C200008	dp

⇨

```
a = 10;
p = &a;
dp = &p;
```

0x0C200000	10
0x0C200004	0x0C200000
0x0C200008	0x0C200004

⇨

```
**dp = 25;
```

0x0C200000	25
0x0C200004	0x0C200000
0x0C200008	0x0C200004

- En C, el paso de parámetros por referencia se hace explícitamente
  - La función invocante pasa a la función invocada punteros a las variables que quiere que ésta última modifique.
  - Los arrays, estructuras y funciones siempre se pasan por referencia.

```
void foo( int32 formal_in, int32 *formal_out )      ..... prototipo función
{
    *formal_out = formal_in;                        ..... acceso a parámetros
}
...
foo( actual_in, &actual_out );                      ..... llamada a función
```

Punteros

- Los punteros son imprescindibles en estructuras de datos enlazadas
  - Para su recorrido se usa el operador **->**

```
struct nodo
{
  int32 valor;
  struct nodo *next;
} a, b *p;
```

```
p = &a;
p->valor = 10;
p->next = &b;
```

```
p = p->next;
p->valor = 20
p->next = NULL;
```

0x0C200000	a.valor
0x0C200004	a.next
0x0C200008	b.valor
0x0C20000C	b.next
0x0C200010	p

0x0C200000	10
0x0C200004	0x0C200008
0x0C200008	b.valor
0x0C20000C	b.next
0x0C200010	0x0C200000

0x0C200000	10
0x0C200004	0x0C200008
0x0C200008	20
0x0C20000C	0x00000000
0x0C200010	0x0C200008

p->valor ≡ (*p).valor

J.M. Mendías
2022

Punteros

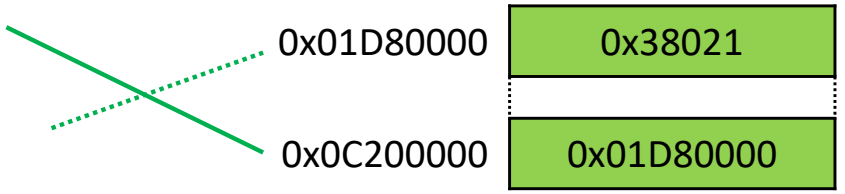
- Todos los punteros (con independencia a su tipo) son compatibles
  - Son direcciones de memoria y su tipo sirve para desreferenciarlos correctamente.
  - Los tipos de los datos a los que apuntan no tienen por qué ser compatibles.
  - El tipo genérico de un puntero es (void *) y requiere un *cast* para desreferenciarse.
  - NULL = (void *) (0x0)
- Las calificaciones de tipos (**`const`** o **`volatile`**) pueden aplicarse al puntero o al tipo apuntado.

```
int32 *p;                          puntero a un entero
const int32 *p_const;              puntero a una constante entera
int32 *const const_p;              puntero constante a un entero
const int32 *const const_p_const;  puntero constante a una constante entera
...
p  = ...;
*p = ...;
p_const  = ...;
*p_const = ...;                    ERROR EN COMPILACIÓN
const_p  = ...;                    ERROR EN COMPILACIÓN
*const_p = ...;
const_p_const = ...;               ERROR EN COMPILACIÓN
*const_p_const = ...;              ERROR EN COMPILACIÓN
```

Punteros

- Los registros de controladores mapeados en memoria
  - Son variables volátiles (cambian sin intervención del software) con direcciones absolutas contantes (fijadas por el hardware)
- Las direcciones absolutas de memoria son punteros constantes.

```
uint32 *const PLLCON_dir = 0x01d80000;
...
(volatile uint32) *PLLCON_dir = 0x38021;
```




```
#define PPLCON_dir ((volatile uint32 *)0x01d80000)
...
*PLLCON_dir = 0x38021;
bar = *PLLCON_dir;
```


```
#define PLLCON (*(volatile uin32 *)0x01d80000)
...
PLLCON = 0x38021;
bar = PLLCON;
```

Clases de almacenamiento

- Aplicados a variables indican su ubicación y visibilidad
- Automático (**`auto`**)
  - Se ubican en tiempo de ejecución en la pila del sistema.
    - Se crean tras la llamada a la función y se destruyen al retorno.
  - Solo son visibles dentro de la función.
  - Es el tipo por defecto de las variables locales y parámetros formales de una función.
- En registro (**`register`**)
  - Son declaraciones automáticas pero ubicadas (si el compilador lo ve posible) en un registro del procesador.
- Estático (**`static`**)
  - Se ubican en tiempo de enlazado en una posición fija de memoria.
    - Se crean al arrancar el programa y se destruyen al finalizar
    - Conservan su valor aún cuando sean locales a una función.
  - Declaradas dentro de una función, solo son visibles dentro de la misma.
  - Declaradas fuera, solo son visibles dentro del módulo (fichero) en que se definen.
- Externo (**`extern`**)
  - Son declaraciones estáticas pero visibles globalmente.
  - Es el tipo por defecto de las variables globales.

J.M. Mendías
2022

Clases de almacenamiento

```
int32 a;
static int32 b;

int32 foo( int32 c )
{
  static int32 d;
  int32 e;
  ...
}
```

a: dirección fija (conserva valor) – visible desde otros ficheros

b: dirección fija (conserva valor) – solo visible en este fichero

c: en pila – solo visible en esta función

d: dirección fija (conserva valor) – solo visible en esta función

e: en pila – solo visible en esta función

- Aplicados a funciones indican su visibilidad
- Externa (**`extern`**)
  - Visibles globalmente.
  - Es el tipo por defecto de toda función.
- Estática (**`static`**)
  - Solo son visibles dentro del módulo (fichero) en que se definen.

Memoria dinámica

- El tiempo de acceso a datos automáticos o estáticos es fijo y similar
  - Los datos automáticos son preferibles ya que la memoria que ocupan se reutiliza cuando la función finaliza.
  - Si el número de niveles de anidación es muy alto (por ejemplo, las funciones son recursivas) existe el riesgo de agotar la pila.
- En sistemas empotrados la memoria dinámica (ubicada en el heap) se usa con moderación y siempre que es posible se evita:
  - Los algoritmos software de asignación/liberación son impredeciblemente lentos y sufren problemas de fragmentación.
  - Si el número de datos dinámicos vivos o la fragmentación es muy alta existe el riesgo de agotar el heap.

```
#include <stdlib.h>
int32 *a;
...
a = (int32 *) malloc( 1000 * sizeof( int32 ) );   ····· crea un array dinámico de 1000 elementos
a[50] = 10;   ····· se accede como los arrays estáticos
free( a );   ····· libera el espacio que ocupa el array
```

Inclusión de código máquina

- Los compiladores permiten incluir instrucciones en ensamblador dentro del código fuente C:
  - Permite ejecutar instrucciones específicas del procesador no disponibles desde C.
  - Puede tomar como argumentos expresiones en C.
  - Para que el código sea reusable se definen como macros en ficheros aparte.

- En GCC se realiza usando la directiva **`asm`**

`asm` [`volatile`] ("*instrucción*" : *args salida* : *args entrada* [: *regs modificados*]);

  - **`volatile`**: indica al compilador que no optimice el código
  - *instrucción*: puede referenciar a argumentos por orden de aparición en la sentencia %0, %1...
  - *args salida*: toman la forma "=*categoría de operando máquina*" (*expresión C*)
    - La categoría de operando pueden ser: memoria (m), registro (r), constante inmediata (i)
  - *args entrada*: toman la forma "*categoría de operando máquina*" (*expresión C*)
  - regs modificados: en el caso de que en la instrucción se haga referencia explicita a registros o posiciones de memoria, informa de esto al compilador.
    - Toman la forma: "r1", "r2"... "memory"

Inclusión de código máquina

- Por ejemplo, en C no existe el operador de rotación de bits
  - puede implementarse como una función
  - pero es más eficiente usar la instrucción del repertorio del ARM7TDMI

```
int32 foo = 0x1;
...
asm( "mov %0, %1, ror #1": "=r" (foo): "r" (foo) );
```

```
ldr  r3, [ ... ]
mov r3, r3, ror #1
str  r3, [ ... ]
```

```
#define ROTATE( var ) asm( "mov %0, %1, ror #1": "=r" (var): "r" (var) )
...
int32 foo = 0x1;
...
ROTATE( foo );
```

Referencias cruzadas

desde ensamblador a C

- Para llamar desde ensamblador a una función definida en C:
  - Basta con saltar normalmente a la función utilizando su identificador
    - Por defecto, todas las funciones C son de clase **`extern`** por lo que son visibles globalmente
  - Si la función tiene argumentos deberán pasarse siguiendo el estándar definido por la arquitectura
    - Usando los registros R0..R3 y la pila
  - No obstante, el nombre de la función C suele declararse en el código ensamblador.
- Para acceder desde ensamblador a una variable global declarada en C:
  - Basta con usar normalmente su identificador como argumento
    - Por defecto, todas las variables globales C son de clase **`extern`** por lo que son visibles globalmente.
  - No obstante, el nombre de la variable C suele declararse en el código ensamblador.

```
.extern main
.extern foo
...
ldr r1, =foo
bl  main
...
```

```
int foo;
...
void main( void )
{
...
}
```

J.M. Mendías
2022

Referencias cruzadas

desde C a ensamblador

- Para llamar desde C a una función definida en ensamblador:
  - El código ensamblador debe hacer visible globalmente el nombre de la función
    - Utilizando la directiva `.global`
  - El código C debe declarar el prototipo de la función ensamblador como externo.
    - Para que puedan pasarse argumentos según el estándar definido por la arquitectura
  - La función puede llamarse desde C normalmente
- Para acceder desde C a una variable global declarada en ensamblador:
  - El código ensamblador debe hacer visible globalmente la etiqueta
  - El código C debe declarar la variable como externa.
  - La variable puede accederse normalmente.

```
extern int32 _bar;
extern void _foo( void );
...
_bar = 0x0;
_foo()
...
```

```
.bss
.global _bar
_bar: .word
...
.text
.global _foo
_foo:
cuerpo de la función
```

Funciones

- Una llamada a una función C supone:

1.	Paso de parámetros actuales	*función* ***invocante***
2.	Almacenamiento de la dirección de retorno	
3.	Salto a la dirección de comienzo de la función	
4.	Almacenamiento de los registros modificados por la función	*prólogo*
5.	Reserva de espacio para variables locales a la función	
6.	Inicialización de las variables locales	
7.	Procesamiento y cálculo del valor de retorno	*función* ***invocada***
8.	Actualización de parámetros de salida	
9.	Restauración de los registros modificados por la función	*epílogo*
10.	Salto a la dirección de retorno (la siguiente a la que hizo la llamada)	

- La manera en que se traduce a ensamblador depende de:
  - Arquitectura del procesador
    - Repertorio de instrucciones
    - Estándar de llamada a funciones definido por la arquitectura
  - Compilador

Funciones

estándar de llamada ARM (i)

- Según el estándar de llamada definido para ARM, típicamente:
  - Los primeros 4 parámetros se pasan por los registros r0-r3 y los restantes por pila.
  - El salto se realiza relativo al PC usando la instrucción BL
    - Instrucción que además de saltar, almacena la dirección de retorno en el registro LR (r14)
  - Los registros modificados por la función se almacenan en la pila, en particular:
    - FP (r11), para no perder el puntero al marco de activación de la función invocante.
    - SP (r13), para no perder el puntero a la cima de pila antes de la llamada.
    - LR (r14), para no perder la dirección de retorno si la función invocada llama a otra función.
    - No es necesario almacenar: r0-r3 (parámetros de entrada), ni IP (r12: auxiliar para construcción marco), ni PC (la dirección de retorno está en LR).
  - Las variables locales (automáticas) se ubican en la pila.
    - Incluyendo los parámetros actuales de las funciones a las que se llame
  - El valor de retorno se almacena en r0.
- Toda la información local de una llamada a función ubicada en la pila se denomina marco de activación
  - El marco de activación se direcciona relativo al FP (puntero de marco)
    - en posiciones decrecientes se acceden los argumentos
    - en posiciones crecientes el resto de elementos

Funciones

estándar de llamada ARM (ii)

```
int32 foo( ... )
{
  int32 bar = 0xff;
  ...
}
```

```
...
baz = foo( ... );
...
```

```
foo:
mov     ip, sp
stmfd   sp!, {... fp, ip, lr, pc}
sub     fp, ip, #4
sub     sp, sp, #...
mov     r3, #255
str     r3, [fp, ...]
...
ldr     r0, ...
sub     sp, fp, #...
ldmfd   sp, { ... fp, sp, lr}
bx      lr
```

PC y LR se apilan o no según opciones del compilador

almacenamiento de registros

actualización del FP

reserva de espacio para variables locales

inicialización de variables locales

almacenamiento del valor de retorno

libera el espacio de variables locales

restauración de registros

salto a la dirección de retorno

```
...
ldr     r3, ...
str     r3, [sp, ...]
ldr     r0, ...
ldr     r1, ...
ldr     r2, ...
ldr     r3, ...
bl      foo
str     r0, ...
...
```

paso de parámetros actuales

almacenamiento de dirección de retorno y salto a la dirección de inicio de la función

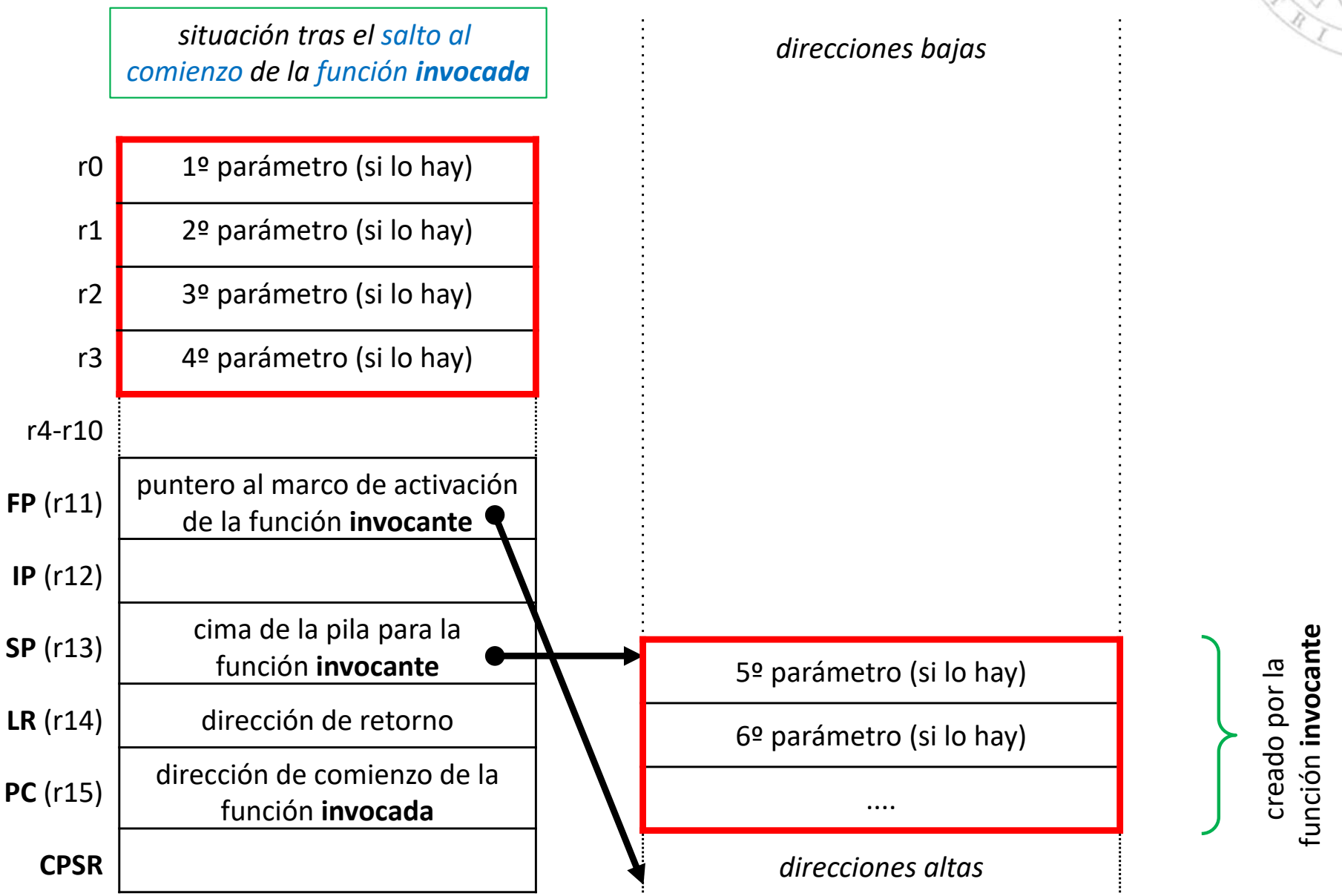
recuperación del valor de retorno

Recuérdese:

```
stmfd ≡ push ≡ stmdb
ldmfd ≡ pop ≡ ldmia ≡ ldm
```

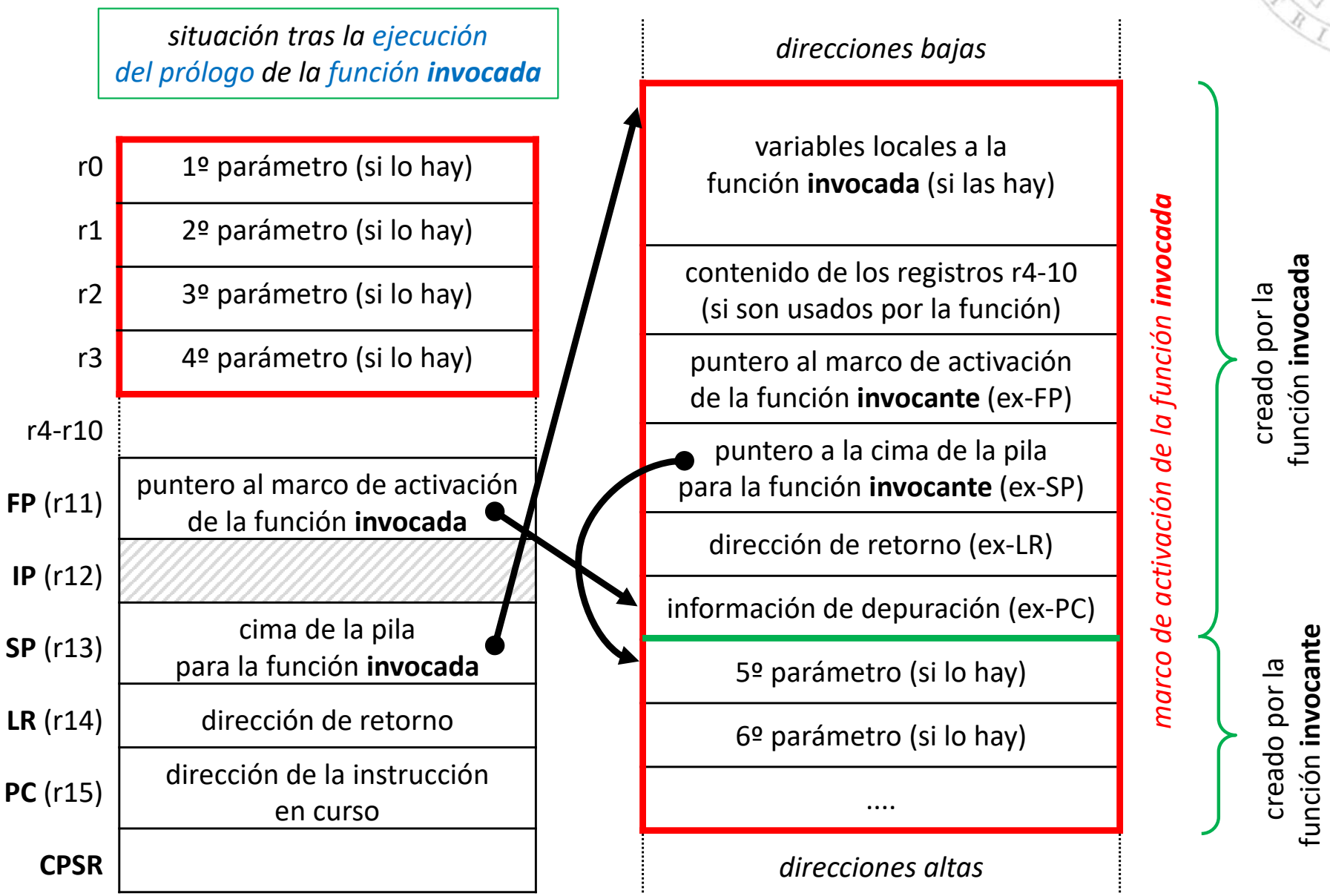
Funciones

estándar de llamada ARM (iii)



Funciones

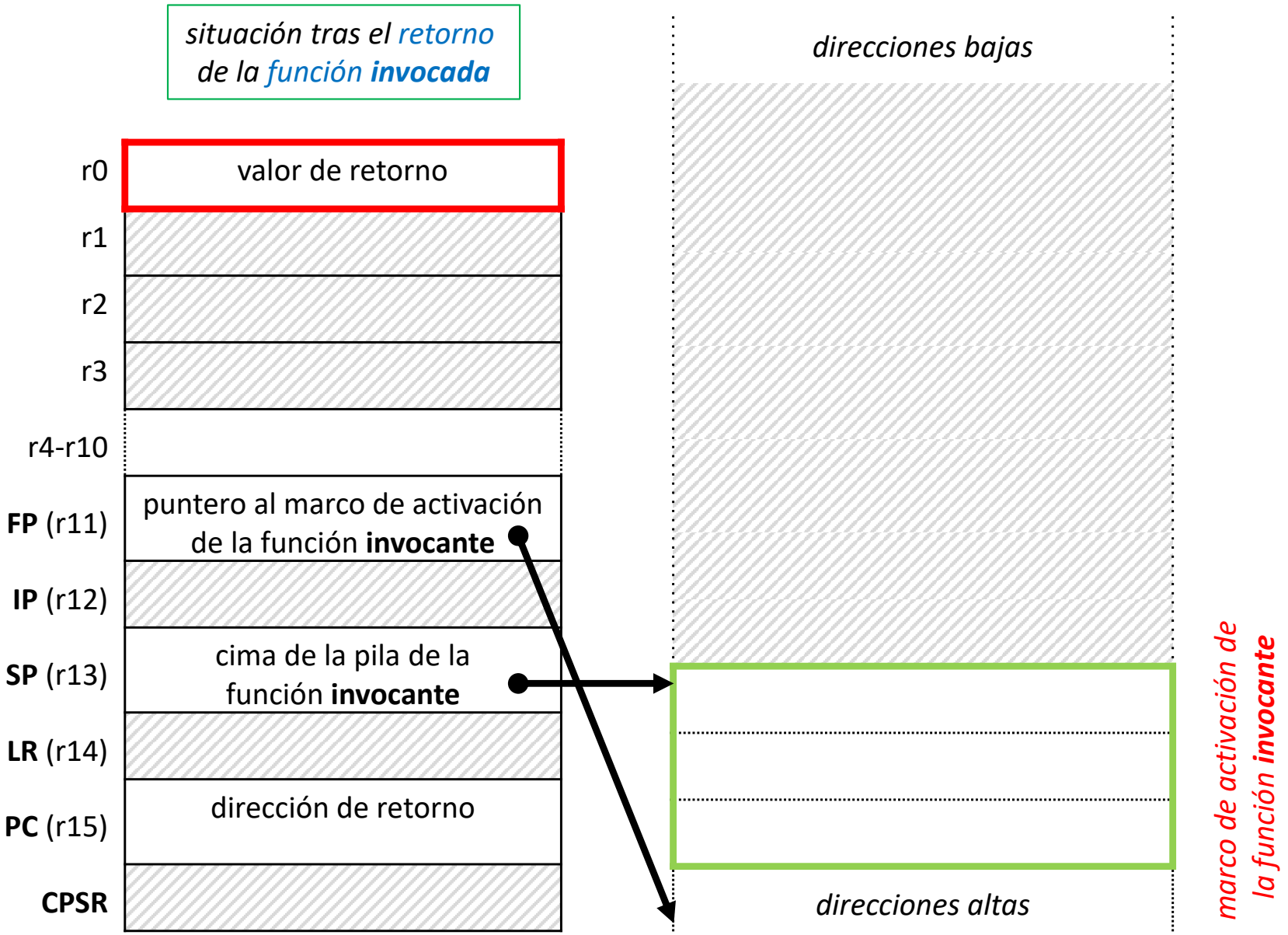
estándar de llamada ARM (iv)



J.M. Mendías
2022

Funciones

estándar de llamada ARM (iv)



J.M. Mendías
2022

Funciones inline

- Al declarar una función como **`inline`**, el programador indica al compilador que integre el código de la función invocada dentro del código de la función invocante.
  - La ejecución es más rápida ya que elimina la sobrecarga de del paso de parámetros.
  - Además si algún parámetro actual es constante el compilador optimiza incluso llegando al caso de que ningún código tenga que ser integrado.
  - Es tan rápido como una macro de preprocesador.

- No obstante depende del grado de optimización del compilador el que se integre o no.
  - Para obligar a GCC siempre la integre, debe usarse el atributo **`always_inline`**

```
inline void foo( char ) __attribute__ ((always_inline));
```

J.M. Mendías
2022

Rutinas de servicio de excepción

- Una rutina de servicio de excepción (excepto SWI) es una función que:
  - es llamada asíncronamente por un evento hardware
    - su ubicación en memoria debe ser conocida por el hardware
    - tras su retorno debe dejar intacto el estado del procesador
  - no admite parámetros y no retorna valor
    - si el programa principal necesita comunicarse con la RTI deberá usar variables globales

- Las rutinas de servicio de SWI se diferencian de las anteriores:
  - Son llamadas síncronamente por el programador
    - para acceder a un servicio del sistema
    - mediante la instrucción **`SWI <num>`**
  - Admiten al menos un parámetro de entrada
    - número de excepción: valor inmediato codificado en la propia instrucción
    - el resto de parámetros se pasan como en una función convencional
  - No obstante el mecanismo de servicio es análogo al resto de excepciones.

J.M. Mendías
2022

Rutinas de servicio de excepción

- El programador al declarar la función debe informar al compilador:
  - Que una función es una RTI, en GCC, usando el atributo **`interrupt`**
  - El modo en que se servirá la función, en GCC, usando el parámetro **`IRQ`**, **`FIQ`**, **`SWI`**, **`ABORT`** o **`UNDEF`**.
    - Para que se generen un prólogo y un epílogo adecuados.

```
void foo( void ) __attribute__ ((interrupt("IRQ")));
```

- Adicionalmente en el cuerpo del programa debe:
  - Configurar el controlador de interrupciones.

```
INTMOD = ...;
INTCON = ...;
...
```

  - Enlazar la función en la correspondiente tabla de vectores de excepción.

```
.extern foo
vectors:
...
b foo
...
```
volcado en ROM

```
#define pISR_IRQ (*(uint32 *)(0x0c7fff18))
...
pISR_IRQ = foo;
...
```
volcado en RAM

  - Habilitar globalmente las interrupciones e individualmente las que correspondan.

```
INTMSK &= ~( (1 << 26) | ... );
```

Rutinas de servicio de excepción

- Una llamada a una rutina servicio de excepción en el ARM7TDMI supone:

1.	Copiado de la dirección de retorno en LR_*<modo>*	*hardware*
2.	Copiado del estado del procesador (CPSR) en SPSR_*<modo>*	
3.	Cambio al modo de operación que corresponda (CPSR.M=*<modo>*)	
4.	Cambio al estado ARM (CPSR.T=0)	
5.	Si *<modo>* = (RST o FIQ) deshabilita FIQ (CPSR.F = 1)	
6.	Deshabilita IRQ (CPSR.I = 1)	
7.	Carga el PC con la dirección del vector correspondiente	
8.	Almacenamiento en pila de los registros modificados por la función	*prólogo*
9.	Reserva de espacio en pila para variables locales a la función	
10.	Inicialización de las variables locales	
11.	Procesamiento	*rutina de servicio*
12.	Si <modo> = (FIQ o IRQ) borrado del flag de interrupción pendiente (I_ISPC/F_ISPC)	
13.	Restauración de los registros modificados por la función	*epílogo*
14.	Restauración del CPSR con el valor de SPSR_*<modo>*	
15.	Restauración del PC con el valor de LR_*<modo>*	

Rutinas de servicio de excepción

ejemplo IRQ (i)

```
void foo( void ) __attributte__ ((interrupt ("IRQ")))
{
  uint32 bar = 0xff;
  ...
}
```

PC y LR se apilan o no según opciones del compilador.

```
foo:
str     ip, [sp, -#4]!
mov     ip, sp
stmfd   sp!, {... fp, ip, lr, pc}
sub     fp, ip, #4
sub     sp, sp, #...
mov     r3, #255
str     r3, [fp, ...]
...
sub     sp, fp, #...
ldmfd   sp, {... fp, sp, lr}
ldr     ip, [sp], #4
subs    pc, lr, #4
```

almacenamiento de registros

actualización del FP

reserva de espacio para variables locales

inicialización de variables locales

libera el espacio de variables locales

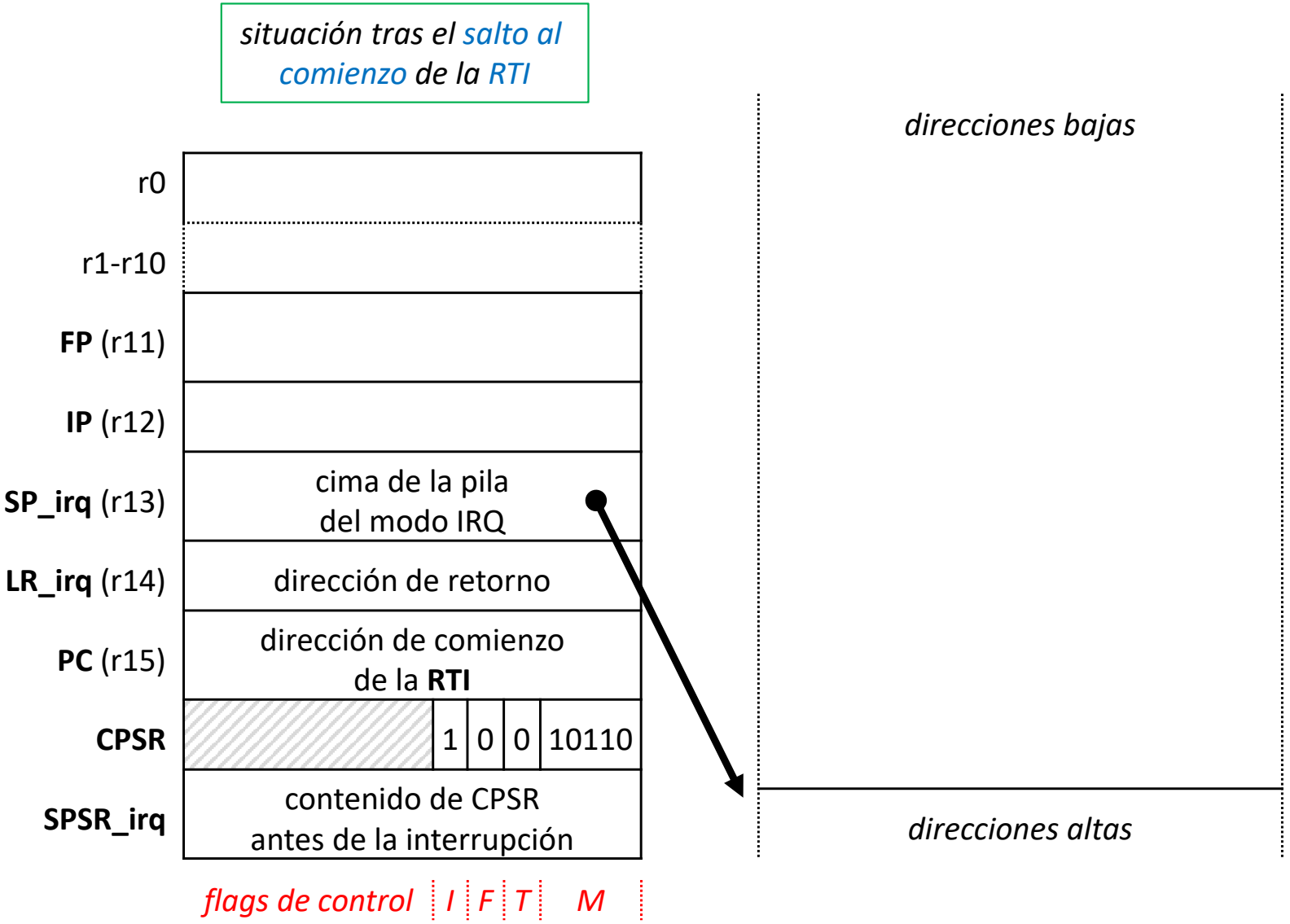
restauración de registros

restauración de CPSR y salto a la dirección de retorno

idéntico a una función convencional

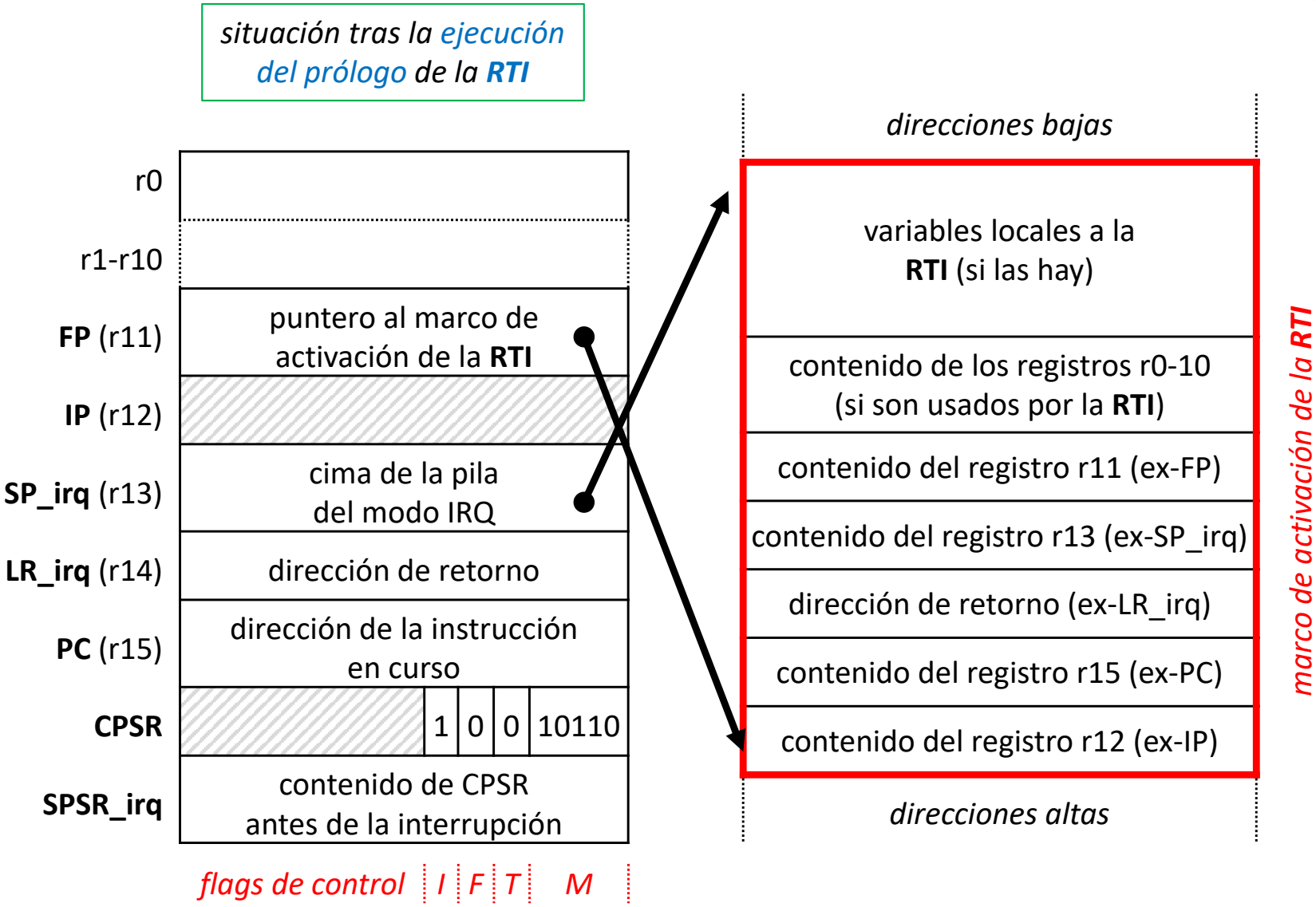
Rutinas de servicio de excepción

ejemplo IRQ (ii)



Rutinas de servicio de excepción

ejemplo IRQ (iii)



J.M. Mendías
2022

Rutinas de servicio de excepción

ejemplo SWI (i)

- GCC no da soporte a llamadas a SWI desde C, luego debe hacerse en ensamblador ejecutando la instrucción la instrucción **SWI <num>**

```
#define SWI( num ) asm volatile ( "swi %0" : : "i" (num) : )
```

- La rutina de servicio de SWI debe conocer el número de excepción para hacer un tratamiento específico para cada tipo:
  - Está ubicado en los bits [23:0] de la propia instrucción que provoca la excepción.
- Para recuperar dicho número, debe tenerse en cuenta que:
  - La dirección de retorno se almacena en el LR_svc
  - La dirección de la instrucción que provocó la excepción es la anterior a la de retorno, es decir LR_svc - 4

almacena en r0 la instrucción SWI que provocó la excepción

```
#define GET_SWI_NUMBER( num_p ) asm volatile (                \
  "ldr r0, [lr, #-4]              \n"                         \
  "bic r0, r0, #0xff000000        \n"                         \
  "str r0, %0                       " : "=m" (*num_p) : : "r0" );
```

borra los 8 bits más significativos para quedarse únicamente con el número de excepción

Rutinas de servicio de excepción

ejemplo SWI (ii)

```
void foo( void ) __attribute__ ((interrupt("SWI")));
...
void foo( void )
{
  uint32 num;

  GET_SWI_NUMBER( &num );
  switch( num ){
    case 0: /* Código de la SWI número 0 */
      ...
      break;
    case 1 : /* Código de la SWI número 1 */
      ...
      break;
      ...
    default: /* SWI desconocida - error */
      ...
  }
}
...
void main( void )
{
  ...
  SWI( 7 );
  ...
}
```

obtención del número de excepción

tratamiento diferenciado según el número de excepción

no llamar a rutinas de la biblioteca estándar de C ya que pueden realizar un número arbitrario de llamadas anidadas a funciones que pueden agotar la pila de supervisor

llamada a la SWI número 7

Rutinas de servicio de excepción

ejemplo SWI (iii)

- Los argumentos adicionales se pasan por registros antes de la llamada.
  - Por ejemplo, una macro para hacer una llamada a SWI con 1 argumento sería:

```
#define SWI( num, arg0 ) asm volatile (                \
  "ldr r0, %0  \n"                                     \
  "swi %1          " : : "m" (arg0), "i" (num) : "r0" );
```

  - Replicando la primera línea de la macro pueden pasar más argumentos:

```
#define SWI( num, arg0, arg1 ) asm volatile (                          \
  "ldr r0, %0  \n"                                                     \
  "ldr r1, %1  \n"                                                     \
  "swi %2          " : : "m" (arg0), "m" (arg1), "i" (num) : "r0 ", "r1" );
```

- Lo primero que deberá hacer la RTE es recuperar los argumentos:
  - Por ejemplo, la macro en ensamblador para recuperar el número de excepción y 1 argumento sería (análogamente para más argumentos):

```
#define GET_SWI_NUMBER( num_p, arg0_p) asm volatile (                  \
  "str r0, %0                  \n"                                     \
  "ldr r0, [lr, #-4]           \n"                                     \
  "bic r0, r0, #0xff000000     \n"                                     \
  "str r0, %1                    " : "=m" (*arg0_p), "=m" (*num_p) : : "r0“ );
```

J.M. Mendías
2022

Rutinas de servicio de excepción

ejemplo SWI (iv)

```
void foo( void ) __attribute__ ((interrupt("SWI")));
...
void foo( void )
{
  uint32 num, arg0;

  GET_SWI_NUMBER( &num, &arg0 );
  switch( num ){
    case 0: /* Código de la SWI número 0 */
      ...
      break;
    case 1 : /* Código de la SWI número 1 */
      ...
      break;
      ...
    default: /* SWI desconocida - error */
      ...
  }
}
...
void main( void )
{
  ...
  SWI( 7, arg0 );
  ...
}
```

Rutinas de servicio de excepción

otros casos

- La rutina de servicio de FIQ:
  - Necesariamente debe hacer pooling cuando debe servir a múltiples periféricos, ya que el controlador de interrupciones del S3X44BOX solo vectoriza las IRQ.
  - No debe permitir anidamientos.
  - En ese caso, no necesita salvar R8-R14 ya que el modo los tiene propios.
  - Al estar su vector al final de la tabla de vectores de excepción, la RTI puede ubicarse directamente en esa dirección para evitar la instrucción de salto (mayor rapidez).
- La rutina de servicio de Undefined instruction:
  - Realiza la emulación por SW de instrucciones de coprocesamiento, recuperando la instrucción que provocó la excepción y ejecutando la rutina SW que corresponda.
- Las rutinas de servicio de Prefetch/Data abort:
  - Tratan los fallos de memoria virtual en sistemas con MMU, recuperando la dirección que provocó la excepción y restaurándola en memoria física
  - Retorna a la instrucción que provocó la excepción (no a la siguiente).
- La rutina de servicio de Reset:
  - Es el punto de entrada al bootloader y se programa directamente en ensamblador
  - No retorna.

Sobre el enlazador

- Un enlazador, siguiendo las directivas del programador, crea un código ejecutable partiendo de uno o varios códigos objeto:
  - Combinando secciones del código de entrada con requisitos de localización comunes (suelen tener el mismo nombre) en una única sección de código de salida.
  - Reubicando cada sección en una región de memoria física.
    - Toda sección de entrada tiene como dirección de comienzo la 0 y todas sus etiquetas locales tienen direcciones relativas a dicho comienzo.
    - Cuando se crea la sección de salida se fija la dirección de comienzo definitiva y las etiquetas se reubican.
  - Resolviendo símbolos, es decir, reemplazando etiquetas globales por sus direcciones.

- Al menos, todo código objeto tiene definidas las siguientes secciones:
  - .text código del programa
  - .rodata constantes globales (incluyendo cadenas constantes de caracteres)
  - .data variables globales inicializadas
  - .bss variables globales no inicializadas (aunque C lo hace a 0)

Téngase en cuenta que las variables locales, parámetros, etc. de un programa en C se ubican en la pila del sistema.

Sobre el enlazador

script de enlazado (i)

- Un script de enlazado permite que el programador pueda controlar el proceso de enlazado.
  - Comando MEMORY: define el mapa físico de memoria.
  - Comando SECTIONS: define cómo las secciones de entrada se ubican en memoria.
  - Comando ENTRY: indica la dirección de primera instrucción ejecutable
  - En todos ellos puede utilizar expresiones con sintaxis C.

- Comando MEMORY
  - El linker por defecto asume un mapa de memoria contiguo e ilimitado.
  - Este comando permite definir el nombre, tipología, dirección de comienzo y tamaño de cada uno de los bloques que forman el mapa físico de memoria.

```
MEMORY
{
   ROM  (rx)  : org = 0x00000000, len = 2M
   RAM  (rwx) : org = 0x0C000000, len = 8M
   SP   (rwx) : org = 0x10000000, len = 8K
}
```

nombre tipo dirección tamaño

mapa de memoria de la placa Embest S3CEV40

J.M. Mendías
2022

Sobre el enlazador

script de enlazado (ii)

- Comando SECTIONS
  - Permite definir el nombre, ubicación y contenido de las secciones de salida del ejecutable.

```
SECTIONS
{
  .text    0x0C000000 : { *  (.text)  }
  .data    0x0C100000 : { *  (.data)  }
  .rodata  0x0C200000 : { *  (.rodata)}
  .bss     0x0C300000 : { *  (.bss)   }
}
```

secciones de salida

direcciones

archivos objeto

secciones de entrada

las secciones de entrada de un mismo tipo de todos los ficheros objeto se combinan en una sección de salida que se ubica en la dirección indicada

el orden de ubicación de las distintas secciones de entrada se puede definir indicando el nombre de los ficheros objeto

```
SECTIONS
{
  .text   : { * (.text)   } > RAM
  .data   : { * (.data)   } > RAM
  .rodata : { * (.rodata) } > RAM
  .bss    : { * (.bss)    } > RAM
}
```

las distintas secciones de entrada se van ubicando consecutivamente en el bloque correspondiente de memoria física

J.M. Mendías
2022

Sobre el enlazador

script de enlazado (iii)

- Comando SECTIONS
  - Pueden definirse y usarse variables (referenciables desde código fuente)
    - location counter (.) = variable que contiene la dirección actual de la sección de salida

```
SECTIONS
{
  . = 0x0C100000;
  .text   : { * (.text)   }
  .data   : { * (.data)   }
  .rodata : { * (.rodata) }
  .bss    : { * (.bss)    }
}
```

asigna al *location counter* la dirección 0x0C100000

las distintas secciones de entrada se van ubicando consecutivamente a partir de la dirección indicada

```
SECTIONS
{
  . = 0x0C100000;
  _inicio_programa = .;
  .text   : { *(.text) }
  _tam_codigo = SIZEOF (.text);
  .data   : { *(.data) }
  .rodata : { *(.rodata) }
  .bss    : { *(.bss) }
}
```

esta variable contiene la dirección donde comienza el programa

esta variable contiene el tamaño del código

J.M. Mendías
2022

Sobre el enlazador

script de enlazado (v)

- Comando SECTIONS
  - Permite indicar una dirección de carga distinta a la ubicación final en ejecución

```
SECTIONS
{
  . = 0x00000000;
  .text   : { * (.text) }
  .rodata : { * (.rodata) }
  _fin = .;
  . = 0x0C000000;
  .data   : AT (_fin) { * (.data) }
  .bss    : { * (.bss) }
}
```

las variables globales se ubicarán en tiempo de ejecución a partir de la dirección 0x0C000000, pero sus valores iniciales estarán cargados tras el segmento .rodata

- Comando ENTRY
  - Este comando permite indicar el punto de entrada a la aplicación

```
ENTRY( main )
```

identifica la función main como punto de entrada

J.M. Mendías
2022

Ubicación de código

- Por defecto, el compilador ubica:
  - el código del programa en la sección .text del código objeto
  - los datos globales, según corresponda, en las secciones .rodata, .data o .bss

- Sin embargo, a veces es deseable que ciertas funciones o datos se ubiquen en secciones especiales
  - para que a su vez el enlazador los ubique correctamente dentro el mapa de memoria
- Para ello se usa el atributo **`section`**
  - que toma parámetro el nombre de la sección.
  - no puede usarse con datos automáticos ya que estos siempre se ubican en pila.

```
char bar[10000] __attribute__ ((section (".buffer"))) = { 0 };
...
void foo( void ) __attribute__ ((section (".system")));
```

J.M. Mendías 2022

Referencias cruzadas

desde C al script de enlazado

- Desde C es posible hacer referencia a las variables definidas en el script de enlazado.

```
SECTIONS
{
  . = 0x0C100000;
  _inicio_programa = .;
  .text   : { *(.text) }
  _tam_codigo = SIZEOF (.text);
  ...
}
```

```
extern const uint8 _inicio_programa[];
extern const uint8 _tam_codigo[];
...
void foo( void )
{
  uint32 inicio, tamaño;

  inicio = (uint32) _inicio_programa;
  tamaño = (uint32) _tam_codigo;
  ...
}
```

declararlas como constantes externas (de tipo array de bytes)

usarlas haciendo el casting que corresponda

Acerca de *Creative Commons*

- Licencia CC (Creative Commons)
  - Ofrece algunos derechos a terceras personas bajo ciertas condiciones. Este documento tiene establecidas las siguientes:

Reconocimiento (*Attribution*):
En cualquier explotación de la obra autorizada por la licencia hará falta reconocer la autoría.

No comercial (*Non commercial*):
La explotación de la obra queda limitada a usos no comerciales.

Compartir igual (*Share alike*):
La explotación autorizada incluye la creación de obras derivadas siempre que mantengan la misma licencia al ser divulgadas.

Más información: https://creativecommons.org/licenses/by-nc-sa/4.0/