

# Arm DesignStart – Introducción a Armv7-M

Resumen

Public - Cortex-M Profile Software Training

11/07/2020

Matias Vironi  
Versión 1

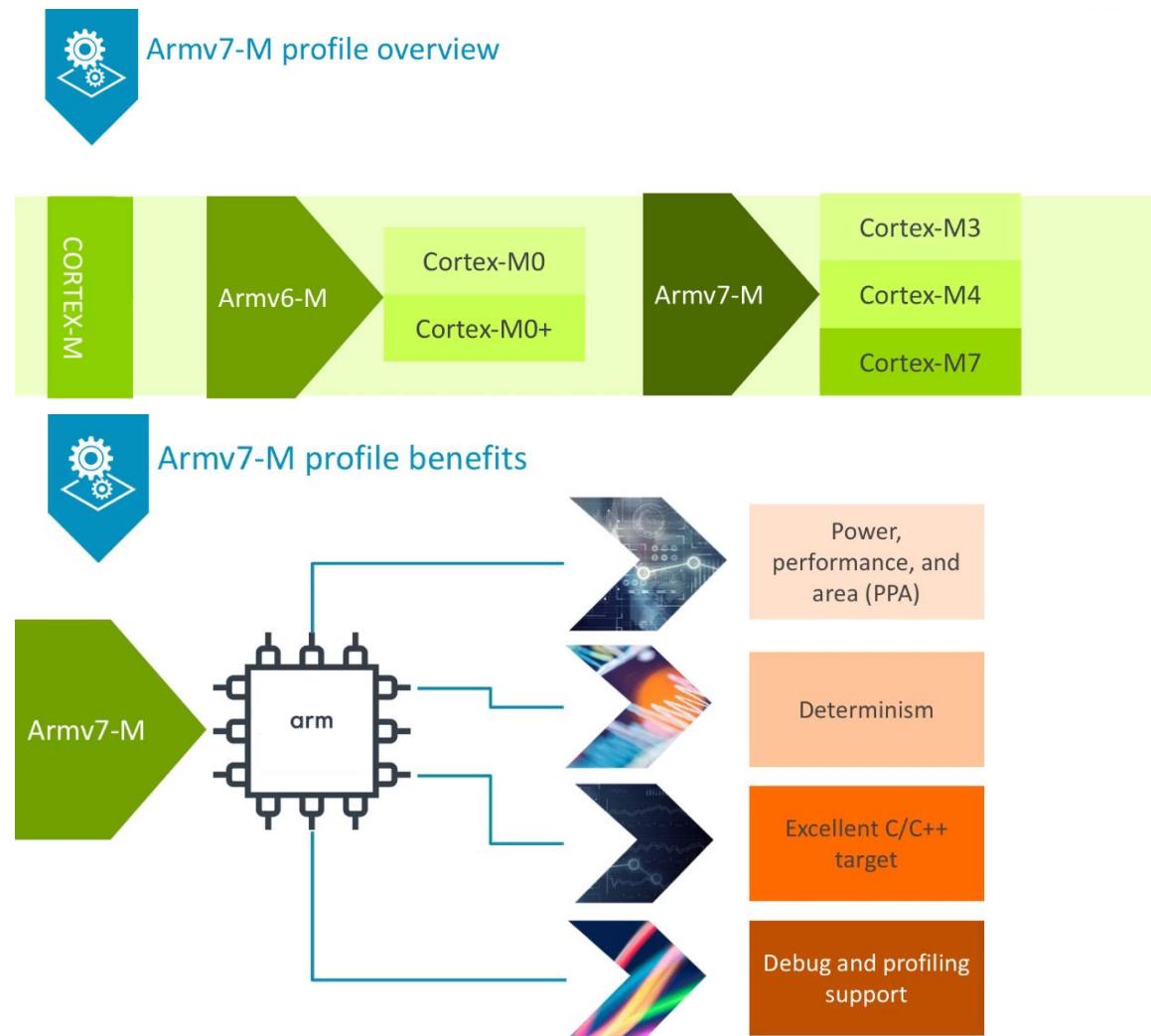
# Índice

<b>Visión general</b>	<b>3</b>
Descripción General	3
Procesador ARM Cortex-M3	3
<b>Modelo del programador</b>	<b>4</b>
Armv7-M - Descripción del modelo del programador	4
Tipos de datos	4
Conjunto de registro Armv7-M	5
Propiedades clave	5
Registros de estado del programa	7
Ejemplo de APSR	10
Ejemplo de EPSR	10
Modos, Privilegios y Pilas	10
Registros de Control	11
Tipos de excepción Armv7-M	12
Soporte del conjunto de instrucciones Armv7-M	13
Conjuntos de instrucciones escalables	14
Instrucciones de procesamiento de datos	14
Instrucciones de acceso a la memoria Armv7-M	14
Instrucciones de control de flujo Armv7-M	15
Uso del registro de Armv7-M	16
<b>Modelo de memoria</b>	<b>18</b>
Mapa de direcciones del sistema Armv7-M - parte 1	18
Mapa de direcciones del sistema Armv7-M - parte 2	18
Tipos y propiedades de memoria Armv7-M	19
Región del sistema Armv7-M	20
Armv7-M Private Peripheral Bus	20
Armv7-M System Control Space	21
Mapa de memoria de Cortex-M3	21
Resumen de MPU	22
<b>Modelo de excepción</b>	<b>24</b>
Resumen de excepciones	24
Mecanismos de interrupción micro-codificados	24
Interrupciones externas	25
Tablas de vectores Armv7-M	25
Comportamientos (Reset and Exception)	27
Entrada de excepción de apilamiento	28
Mecanismo de retorno de excepción	29
Ejemplo de entrada de excepción NMI	30
Ejemplo de retorno de excepción NMI	32
Resumen de prioridades de excepción(exception priorities)	33
Ejemplo de anidamiento (Nesting)	33
Ejemplo de encadenamiento de cola (Tail-chaining)	35
Ejemplo de Late-arriving	35
Excepciones durante la restauración del estado	36
Escribir interrupt handlers	37
Excepciones internas (SysTick, SVCall, PendSV)	37
Excepciones de fallas en Armv7-M	38
Estado de bloqueo (Lockup state)	40
Manejo de fallas (Fault handling)	41

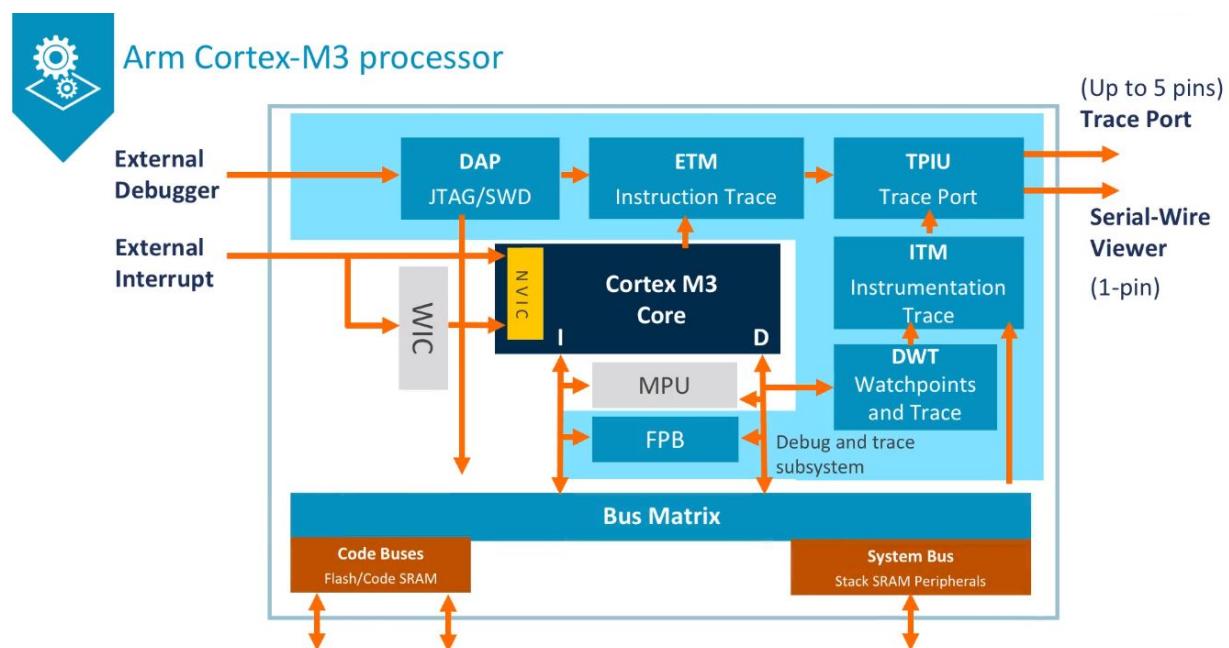
<b>Debug (Depurar)</b>	<b>42</b>
DesignStart - Descripción general de depuración	42
Estado de depuración de Armv7-M	42
Componentes de depuración Cortex-M	43
Breakpoints and watchpoints	44
Breakpoints de software y hardware	44
Armv7-M vector-catch (captura de vector)	45
Armv7-M semihosting	46
Características de perfil de Data Watchpoint and Trace (DWT)	46
The Instrumentation Trace Macrocell (ITM)	47
Embedded Trace Macrocell	47
Resumen (depuración Armv7-M)	48
<b>Desarrollo de software</b>	<b>49</b>
Desarrollo de software y herramientas de depuración	49
The Cortex Microcontroller Software Interface Standard (CMSIS)	49
Paquetes de software	51
Proceso de desarrollo embebido: secuencia de inicialización	52
Scatter-loading (Carga de dispersión)	52
Reorientación de la biblioteca C	55
Arm Cortex-M3 DesignStart Eval: Run a simple C test on RTL Simulation	55
Enlaces útiles - DesignStart Armv7-M	55

# Visión general

## Descripción General



## Procesador ARM Cortex-M3



# Modelo del programador

## Armv7-M - Descripción del modelo del programador

Este capítulo proporciona información sobre las características fundamentales que permiten a los desarrolladores programar un sistema Armv7-M, que incluye:

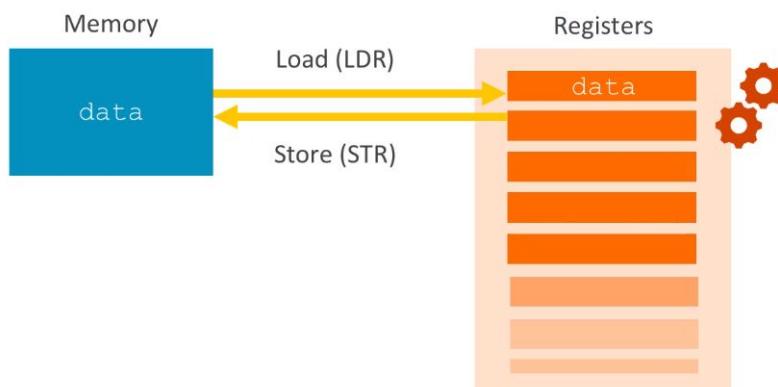
- Tipos de datos.
- Conjunto de registro central. (Core register set)
- Registros especiales. (Special purpose registers)
- Modos de procesador: Handler Mode and Thread Mode..
- Software Privilegiado vs No-privilegiado.
- Uso de pila (Stack).
- Manejo básico de excepciones.(Exception handling basics)
- Una introducción al conjunto de instrucciones Thumb.
- Cómo realizar una llamada de subrutina y el procedimiento de llamada estándar para la arquitectura de ARM. (the Procedure Call Standard for the Arm Architecture (AAPCS))

## Tipos de datos



### Load/store

Arm is a load/store architecture.



### Data types

Arm v7-M is a 32-bit load/store architecture.

Byte = 8 bits

Halfword = 16 bits

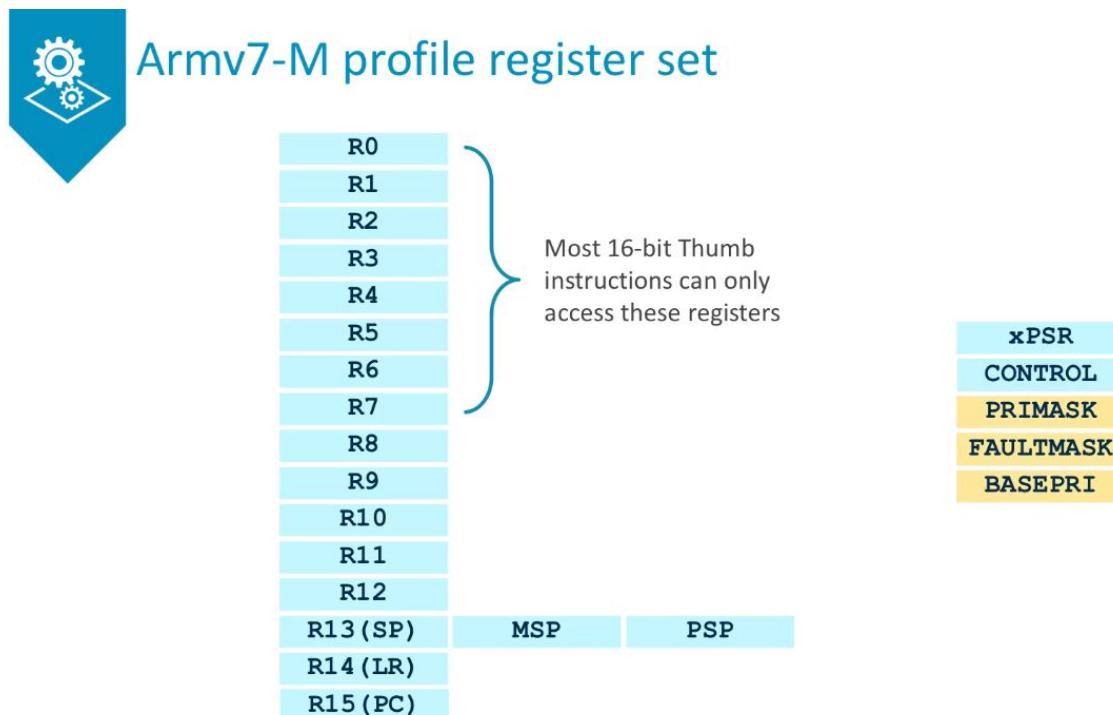
Word = 32 bits

Doubleword = 64 bits

These data types are used for all data operations in the Armv7-M architecture

## Conjunto de registro Armv7-M

- Hay 15 registros de propósito general y registros de propósito especial
- Los registros de **R0** a **R7** son conocidos como “Registros bajos” y son accesibles para todas las instrucciones **Thumb** de 16-bits
- Los registros de **R8** a **R12** son conocidos como “Registros altos” y están disponibles para todas las instrucciones **Thumb** de 32-bits y solo unas pocas instrucciones **Thumb** de 16-bits
- Un procesador ARMv7 implementa 2 pilas. La pila principal(**Main Stack**) a la que se accede mediante el puntero a pila principal (**MSP**) y la la pila de procesos(**Process Stack**) a la que se accede mediante el puntero a pila de procesos (**PSP**).
- Existen diferentes registros de estado del programa conocidos como **xPSR**
- Hay un registro de **CONTROL** para controlar algunas cosas. Ej: Que pila (**MSP** o **PSP**) utiliza el procesador cuando el software se refiere a **R13**.
- Finalmente hay algunos registros de máscara de propósito especial: **PRIMASK**, **FALMASK**, **BASEPRI** que se utilizan para cambiar la prioridad del contexto de ejecución actual



## Propiedades clave

El **program counter (PC)** se usa para buscar instrucciones en la tubería del procesador (processor pipeline). El software puede usar el **PC** para diferentes propósitos. Por ejemplo:

- a) La función **Branch** modifica el **PC** para cambiar el flujo del programa. En este ejemplo el procesador agrega el desplazamiento correcto al valor actual de **PC** para que comience a buscar instrucciones desde la misma ubicación.
- b) En este caso **PC** se usa para apuntar a algún dato.
- c) Aquí se muestra una dirección cargada en **R0**, seguido de una instrucción **BX R0** que bifurca a **R0**



## Key properties

Program counter  
(PC)

Link register (LR)

Stack pointer (SP)

```
a)      B foo
        :
        :

foo:
        // <code>

b)
        LDR r0, [pc,#8]
// pc points to data
        :
        :
        // <data>

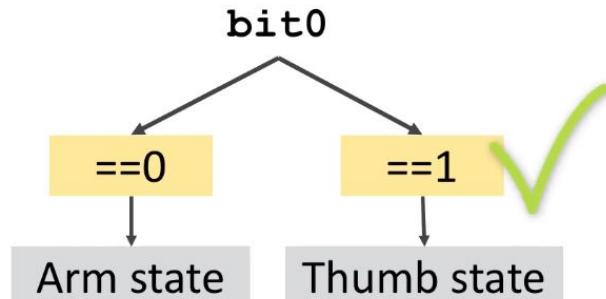
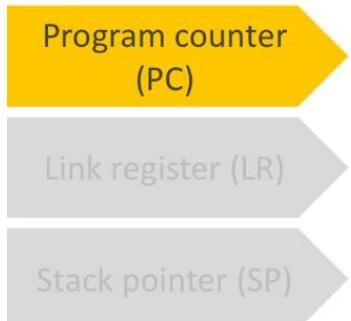
c)      LDR r0,=0x8001
        BX r0
```

En el ejemplo C, tenemos **LDR r0, =8001** en vez de **LDR r0, =8000**

La razón por la cual el bit inferior de la dirección se establece en 1 es para garantizar que el procesador permanezca en estado **Thumb**

En otras arquitecturas de ARM que admiten instrucciones de **ARM**. El bit inferior se usó para determinar en qué estado debe ingresar el procesador después de ejecutar la instrucción **BX**.

Esto se debe ejecutar con las instrucciones **BX** y **BLX**, de lo contrario se producirá un error de uso de estado no válido.



El **Link register (LR)** se usa para contener la dirección de retorno de una llamada de subrutina.

La **L** en la instrucción **BL** significa que el procesador guardará la dirección de retorno en el **Link Register**

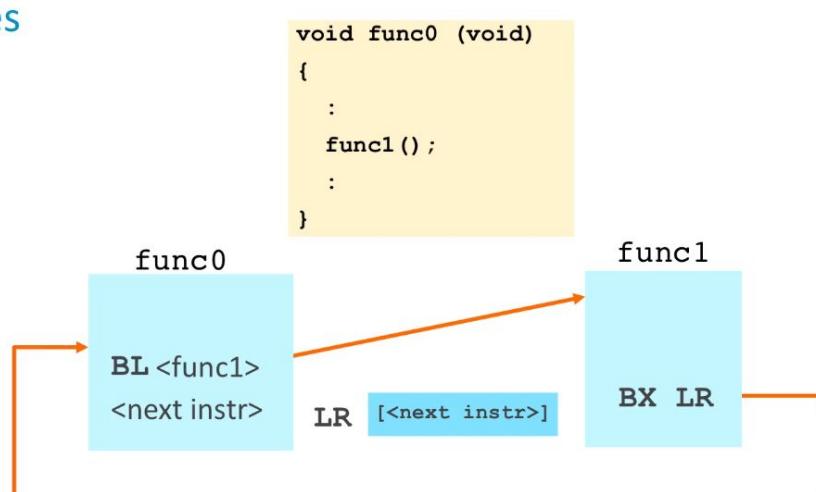


## Key properties

Program counter  
(PC)

Link register (LR)

Stack pointer (SP)



```
void func0 (void)
{
    :
    func1 ();
    :
}
```

LR [<next instr>]

func1

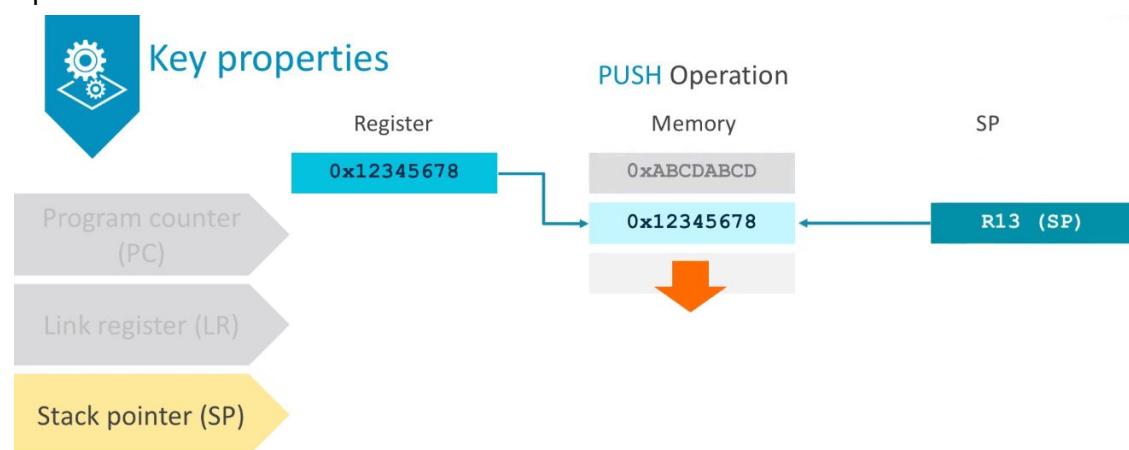
BX LR

En ARMv6-M y ARMv7-M el LR tiene otra funcionalidad relacionada con las excepciones.

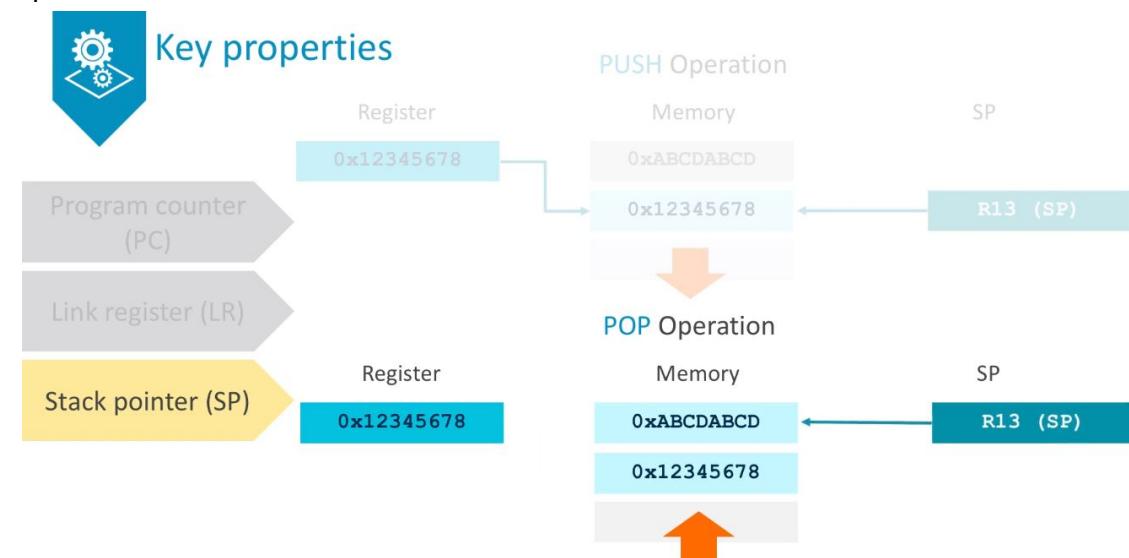
La **pila** se usa para guardar y restaurar el contenido del registro desde y hacia la memoria. La **pila** es necesaria para guardar los valores del registro de datos que de otro modo podrían sobreescribirse.

En ARMv7 las **pilas** son **Full Descending**, lo que significa que la pila crece hacia abajo en la memoria.

## Operación PUSH



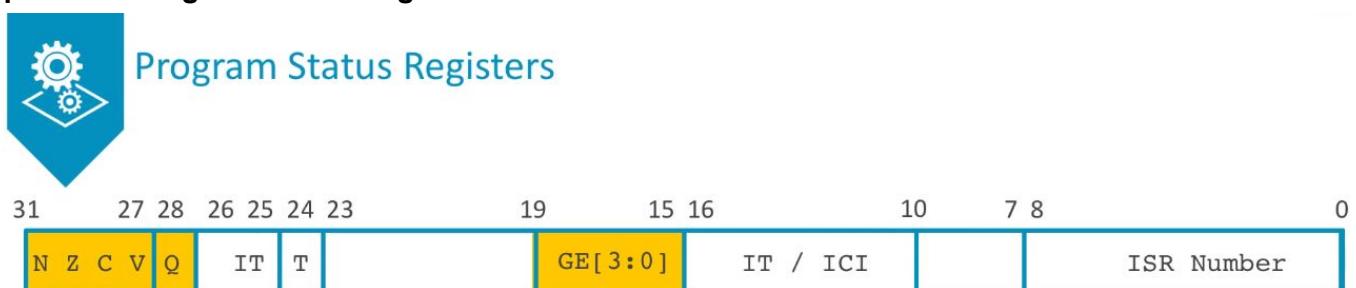
## Operación POP



Tienen animación las anteriores 2 imágenes para entender el **PUSH** y el **POP** (revisar el video).

## Registros de estado del programa

### Application Program Status Register

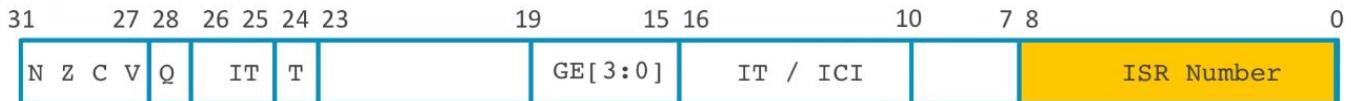


Application Program Status Register  
(APSR)

## Interrupt Program Status Register



### Program Status Registers

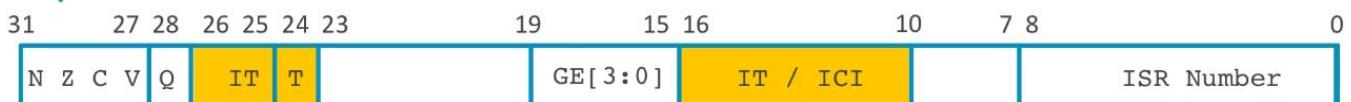


## Interrupt Program Status Register (IPSR)

## Execution Program Status Register



### Program Status Registers



## Execution Program Status Register (EPSR)

## Application Program Status Register

Contiene 5 banderas de condición y un campo **GE** que puede actualizarse mediante varias instrucciones.

**N:** Negativo, **Z:** Cero, **C:** Carry, **V:** Overflow. Se utilizan para controlar la ejecución de instrucción condicional.  
**Q:** Saturación (Indica que ha ocurrido una saturación). Este bit está seteado en 1 si una instrucción Signed Saturate (SSAT) o una Unsigned Saturate (USAT) resultó en saturación. También algunas instrucciones de multiplicación disponibles con las extensión **DSP** puede establecer el indicador **Q**.

La bandera **Q** una vez que se ha configurado, permanece establecida hasta que se borre explícitamente.

Los 4 bits **GE (Greater than or Equal)** están disponibles en implementaciones ARMv7-M con la extensión **DSP**. (Ej: Cortex M4 y M7).



### Program Status Registers



Application Program Status Register (APSR):

- Only ALU flags.
- GE bits only present in Armv7E-M (for example, Cortex-M4).

## Interrupt Program Status Register

El IPSR es actualizado por el procesador al entrar o regresar de una excepción.

Si **IPSR** se establece en 0: No hay extensiones activas y el procesador está en **modo Thread**. De lo contrario el procesador está en **modo Handler** y el **IPSR** indica que excepción está actualmente activa.

Si **IPSR** se establece en 2: Indica que la interrupción **Non-Maskable** es la excepción activa actual



Interrupt Program Status Register (IPSR):

- Interrupt/Exception Number

IPSR

0

Thread mode  
No active exceptions

IPSR

2

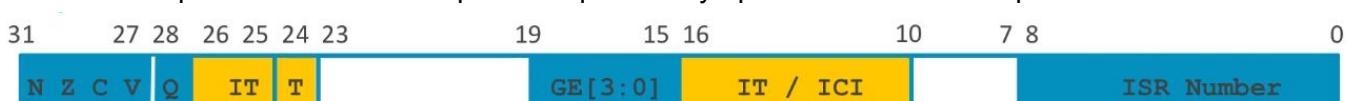
Handler mode  
Non-Maskable Interrupt

## Execution Program Status Register

Contiene los bits **IT** / **ICI** y el bit **T**

El bit **T** siempre debe establecerse en **1**, como los procesadores ARMv7-M solo admiten el estado **Thumb**. Cambiarlo a **0** usando software desencadenaría una falla.

Los bit **IT** / **ICI** son usados para la ejecución condicional de secuencias de instrucciones y para controlar qué instrucciones se pueden reanudar después de que se haya producido una interrupción.



Execution Program Status Register:

- T bit ( s/b =1, to show core is in T32 state)
- IT field – If/Then block information
- ICI field – Interruptible-Continuable Instruction Information

Finalmente el software puede referirse a todos los registros vistos como **xPSR** y el software puede referirse a un compuesto de 2 registros, de los cuales hay 3 instancias.

- 1) **IEPSR** que combina **IPSR** y **EPSR**.
- 2) **IAPSR** que combina **IPSR** y **APSR**.
- 3) **EAPSR** que combina **EPSR** y **APSR**.

El **MRS** y **MSR** son instrucciones que pueden ser usadas para acceder a estos registros

El **APSR** permite acceso a lectura y escritura. **IPSR** solo permite acceso de lectura. El **EPSR** se trata como RaZ / WI esto significa que siempre es Read-as-Zero, y todas las escrituras son ignoradas.



## Program Status Registers

### xPSR

- Composite of the 3 PSRs
- Stored on the stack on exception entry



### Application Program Status Register (APSR)

Read Write

IEPSR

### Interrupt Program Status Register (IPSR)

Read-only

IAPSR

### Execution Program Status Register (EPSR)

RaZ/WI

EAPSR

## Ejemplo de APSR



### Application Program Status Register (APSR) - Example

r0	LDR r0, =0x2 ; loop counter ; loop body
0	loop
N Z C V	SUBS r0, r0, #1 ; decrement loop counter BNE loop ; branch back to the beginning ; of the loop if the counter ; hasn't reached zero
0 1 1 0	

## Ejemplo de EPSR

[Ver el video](#)

## Modos, Privilegios y Pilas

En ARMv7-M hay 2 modos de operación: **Thread** y **Handler**

**Modo Thread:** Define si la ejecución del programa es una aplicación normal.

**Modo Handler:** Define si la ejecución del programa es un controlador de excepciones.

También hay 2 niveles de ejecución: **Privilegiada** y **No privilegiada**. Estos niveles proporcionan un mecanismo para controlar el acceso a recursos críticos.

El **modo Handler** siempre tiene privilegios mientras que el **modo Thread** tiene un nivel de privilegio programable.

Los procesadores ARMv7-M tienen dos pilas con los punteros a pila correspondientes. El **stack pointer** siempre está alineado con palabras.

El **Main Stack Pointer (MSP)** que es el Stack Pointer por defecto, es utilizado por los controladores de excepciones del sistema operativo y las aplicaciones que se ejecutan de manera privilegiada.

El **Process Stack Pointer (PSP)** es utilizado por aplicaciones que se ejecutan de manera no privilegiada



## Modes, privilege, and stacks

Thread	Handler
Normal application reset and processes	Exceptions
Privileged and Unprivileged	Privileged only
Main stack (MSP) Optional Process Stack (PSP)	Main stack (MSP) only

En este caso podemos ver todas las combinaciones posibles de los 2 **Stack pointer** y las niveles de privilegio para los dos modos de operación.

El **Current Stack Pointer** siempre es accedido usando el nombre del registro “**SP**”.

Para acceder específicamente al **MSP** o al **PSP** pueden usarse las instrucciones especiales de acceso al registro **MRS** y **MSR**.

Thread		Handler
<b>MSP + PRIV</b>	<b>PSP + PRIV</b>	<b>MSP + PRIV</b>
<b>MSP + UNPRIV</b>	<b>PSP + UNPRIV</b>	

Current Stack Pointer accessed using: SP

MRS and MSR instructions are used to access the MSP and PSP

## Registros de Control

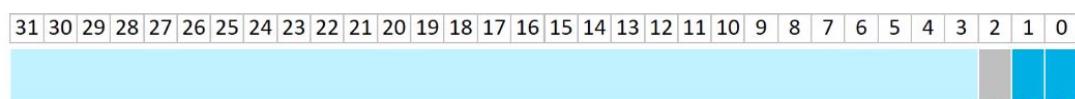
El Control de propósito especial es un registro de 2 o 3 bits. Se utiliza para definir el nivel de privilegio utilizando el campo **nPRIV** (Not Privileged).

Selecciona el **Modo Thread** del **Stack Pointer** usando el campo **SPSEL** (Stack Pointer Selection) y si la extensión de punto flotante está disponible, activa y desactiva la extensión de punto flotante en el contexto actual, utilizando el campo **Floating Point Context Active**. (No disponible en **Cortex-M3**).



## CONTROL Register

The special purpose CONTROL Register is a 2-bit or 3-bit register



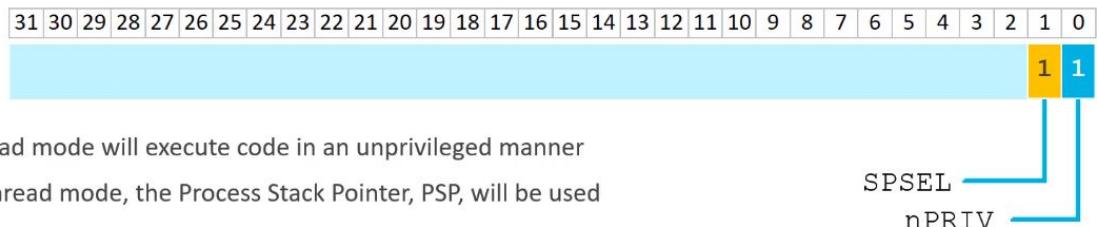
nPRIV – Defines the execution privilege in Thread mode

SPSEL – Defines the stack to be used in Thread mode

FPCA – Defines whether floating-point is active in the current context (not for Cortex-M3)

En el siguiente ejemplo podemos ver que el **Modo Thread** se ejecuta de manera **No privilegiada**. En **Modo Thread** se utilizará el Process Stack Pointer **PSP**.

The special purpose CONTROL Register is a 2-bit or 3-bit register

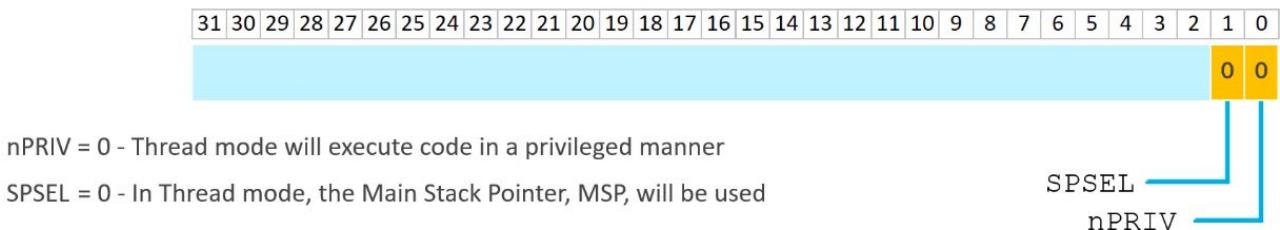


Fuera de **Reset**, el valor de registro es 0

Además de esto hay una serie de instrucciones que deben tenerse en consideración.

Primero, el único código que se ejecuta de manera privilegiada puede modificar el valor de este registro. Además, para garantizar que los cambios en el registro de control surtan efecto antes de ejecutar la siguiente instrucción se debe utilizar una instrucción de barrera de sincronización de instrucciones (Instruction Synchronization Barrier **ISB**)

The special purpose CONTROL Register is a 2-bit or 3-bit register



#### Access restrictions

No write access to CONTROL in Unprivileged mode.

Instruction Synchronization Barrier, ISB, required to ensure that changes to CONTROL register take effect.

## Tipos de excepción Armv7-M

### Manejo de excepciones (Exception handling)

Una excepción puede ser causada por la ejecución de una instrucción generadora de excepción o activado como respuesta a ciertos comportamientos del sistema tales como interrupciones, violaciones de protección de memoria y eventos de depuración.

Los procesadores **ARMv7-M** admiten los siguientes tipos de excepción:

**Reset**: Es un caso especial que se maneja en **Modo Thread**

**Non-maskable Interrupt (NMI)**: Esta excepción no se puede enmascarar o deshabilitar

**Faults**: Hay una serie de Fallas que se pueden configurar de forma independiente. Esto incluye **Permanently enabled Hard Fault**, que se usa comúnmente para errores irrecuperables del sistema y la **memory management fault** que pueden ser causadas por violaciones de protección de memoria.

Interrupciones **SVCALL** (Supervisor Call) y **PendSV** (Pending Supervisor Call): Se utilizan para manejar las llamadas al sistema generadas por el software.

La arquitectura también admite **496 Interrupciones Externas**

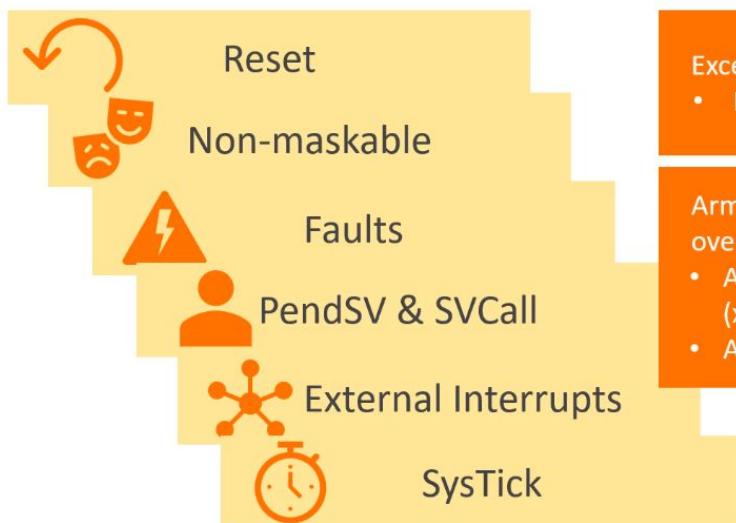
El temporizador obligatorio del sistema tiene la interrupción **SysTick**, que puede usarse para realizar ciertas tareas a intervalos configurables y regulares.

La arquitectura **ARMv7-M** puede guardar y restaurar automáticamente el contexto de ejecución actual tras la entrada y retorno de excepciones. Esto permite escribir el código del controlador de excepciones (exception handler) enteramente en **lenguaje C**.

Todas las excepciones se procesan en **Modo Handler**, por lo tanto el código de manejo de excepciones siempre se ejecuta de manera privilegiada.



## Exception handling



Exceptions processed in Handler mode (except Reset).

- Exceptions always run privileged.

Armv7-M exception handling reduces software overhead.

- Automatic save and restore of processor registers (xPSR, PC, LR, R12, R3-R0).
- Allows handler to be written entirely in 'C'.

## Soporte del conjunto de instrucciones Armv7-M

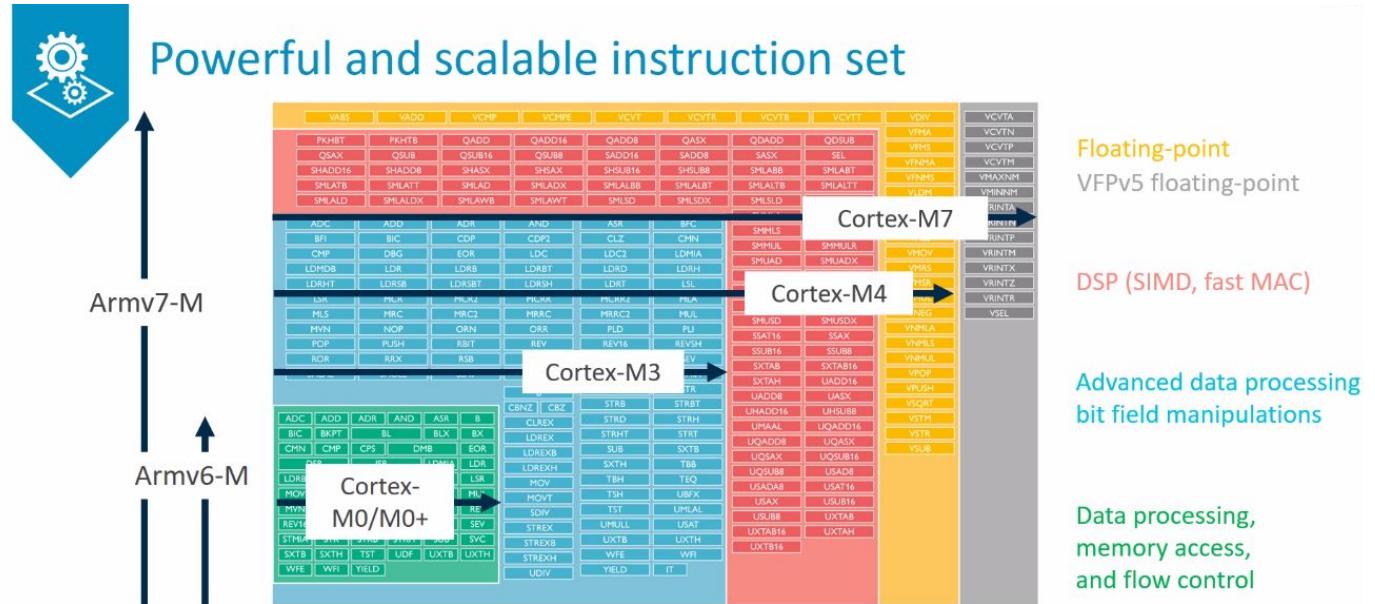
Los núcleos Armv7-M implementan el conjunto de instrucciones Thumb con la tecnología Thumb-2.

- Mezcla de instrucciones de 16 bits y 32 bits: implementa casi toda la funcionalidad del conjunto de instrucciones Arm.
- Superconjunto del conjunto completo de instrucciones Thumb de 16 bits.
- Conjunto de instrucciones de carga / almacenamiento; sin manipulación directa de los contenidos de la memoria.
- La longitud de la instrucción puede variar, dependiendo de la funcionalidad.

Hay dos extensiones de arquitectura opcionales disponibles:

- Armv7E-M** agrega instrucciones **DSP**.
  - Actualmente compatible con **Cortex-M4** y **Cortex-M7**.
- Instrucciones de punto flotante de simple precisión.
  - Actualmente compatible con **Cortex-M4** y **M7** con extensión de punto flotante.
- Instrucciones de punto flotante de doble precisión.
  - Actualmente compatible con **Cortex-M7** con extensión de punto flotante.

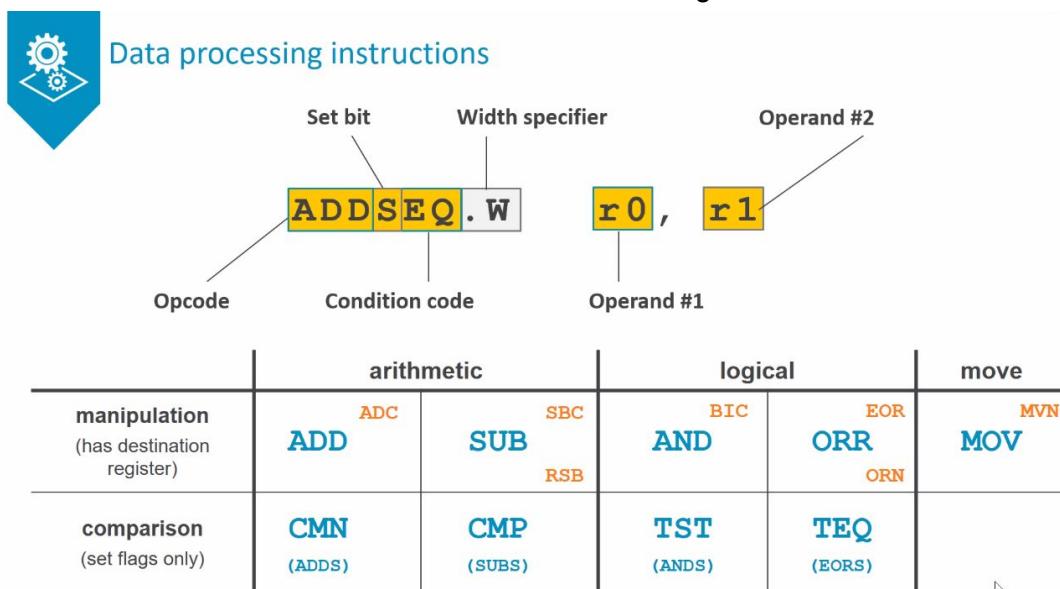
# Conjuntos de instrucciones escalables



# Instrucciones de procesamiento de datos

En la siguiente imagen tenemos un ejemplo de una instrucción de procesamiento de datos que realiza una operación de suma en los registros **R0** y **R1**.

Se puede especificar el ancho de la instrucción utilizando el **Width specifier**. Aquí .W indica que queremos generar la variante de **32 bits** de esta instrucción. En lugar de la variante de **16 bits**.



# Instrucciones de acceso a la memoria Armv7-M

Los registros de propósito general utilizados con las instrucciones de acceso a la memoria son de **32 bits**.

Para acceder a 1 palabra de memoria (**Word**) se puede utilizar 1 registro.

Para acceder a 1 palabra doble de memoria (**Doubleword**) se debe usar 2 registros.

Para acceder a datos de sub palabras como **Byte** y **Halfword** se utiliza 1 registro en combinación con extensión de signo (sign-extension) y relleno cero (zero-padding).

Para accesos de Halfwords tenemos las instrucciones **LDRH** y **STRH** que acceden a una media palabra con relleno cero. La instrucción **LDRSH** que carga una **Halfword** y realiza una extensión de signo.

Para accesos de Bytes tenemos las instrucciones **LDRB** y **STRB** que acceden a un byte con relleno cero. La instrucción **LDRSB** que carga un **Byte** y realiza una extensión de signo.

Tener en cuenta que para **Bytes** y **Halfwords** las instrucciones “store” se utilizan para almacenar valores con relleno de cero y signo extendido. Esto se debe a que las instrucciones de almacenamiento solo almacenan los bits necesarios, no leen ningún bit que esté fuera del rango de bits del tipo de datos que se está almacenando.



## Memory access instructions



## Instrucciones de control de flujo Armv7-M

Operaciones como Llamadas a funciones, Devoluciones y Bucles afectan el flujo de un programa, es decir requieren que un procesador pueda ejecutar instrucciones de manera no secuencial.

La arquitectura ARMv7-M incluye instrucciones para controlar el flujo de un programa de esta manera.

Estas instrucciones se conocen como las instrucciones de “Control de Flujo” o “Branch”. Y se pueden ver en la siguiente tabla:

**B:(Branch):** Bifurca a una dirección diferente sin afectar ningún registro.

**B<cond>:** Es como la instrucción Branch anterior, excepto que puede ser ejecutado condicionalmente.

**BL:** Actualiza el valor del Link Register cuando se toma el Branch. Permite llamadas a función y devoluciones.

**BX:** (Branch and Exchange): Se usa principalmente para retorno de funciones

**BLX:** (Branch with Link and Exchange)



## Flow control instructions

Branch instructions vary in size and range.

Instruction	Branch Range	
	16-bit	32-bit
<b>B</b>	+/- 2KB	+/- 16MB
<b>B&lt;cond&gt;</b>	-256 to +254	+/- 1MB
<b>BL</b>		+/- 16MB
<b>BX</b>	Any 32-bit address	
<b>BLX</b>	Any 32-bit address	
<b>CBZ</b>	+4 to 130 bytes	
<b>CBNZ</b>	+4 to 130 bytes	
<b>TBB</b>		+512 bytes
<b>TBH</b>		+128KB

# Uso del registro de Armv7-M

El **AAPCS** define cómo se deben utilizar los registros de propósito general **R0 a R15** para gestionar llamadas de funciones y retorno. Esto es importante para escribir código en Lenguaje Assembly que se puede llamar desde un lenguaje de alto nivel como C

## Arm Application Procedure Call Standard (AAPCS)

**AAPCS** divide los registros de propósito general en 4 grupos.

**R0 a R3** se pueden usar para pasar parámetros a una función y devolver el valor

Se puede pasar hasta 4 parámetros a la vez con parámetros adicionales que se pasan en la pila.

**R4 a R11** pueden usarse para cualquier operación dentro del cuerpo de una función, pero si se usan, sus valores deben ser preservados. La preservación se realiza almacenando sus valores en la pila al comienzo de la función y luego restaurando los valores de la pila antes de que regrese la función.

**R12** es el único registro que se puede usar como registro de memoria virtual. No hace falta preservar su valor.

**R13** se usa como el puntero de la pila, **R14** como registro de enlace y **R15** como contador de programa.

Los procesadores que implementan la arquitectura **ARMv7-M** pueden guardar automáticamente ciertos registros cuando ocurre una excepción. Los registros que se guardan son los que no están obligados a ser preservados por **AAPCS** que son los registros **R0 a R3, R12, SP, LR y PC**.



### Register usage

Passing parameters to functions  
Getting the return value

Register
R0
R1
R2
R3

Register variables  
Must be preserved

R4
R5
R6
R7
R8
R9
R10
R11

Scratch register  
(corruptible)

R12

Stack Pointer  
Link Register  
Program Counter

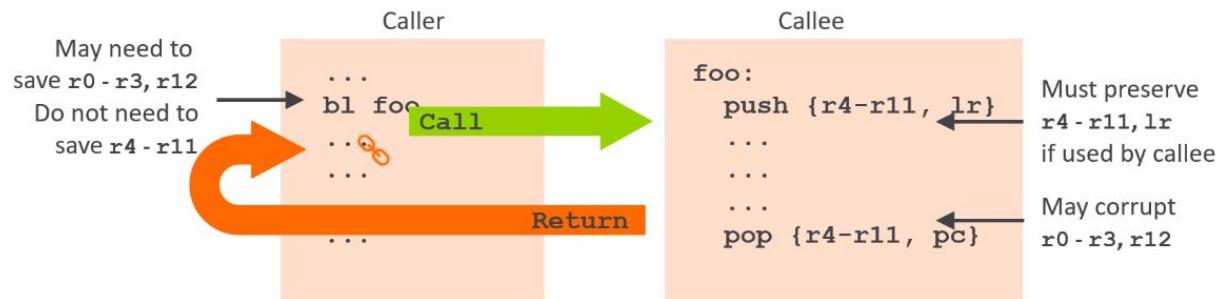
R13/SP
R14/LR
R15/PC

Tenga en cuenta que en este ejemplo no hay una instrucción explícita de **Branch and Exchange BX** utilizada para volver de la función de llamada. En cambio se usa la instrucción **POP**. Esta instrucción escribe en el Program Counter. Escribir en el PC implícitamente causa un bifurcación. Esto se hace para ahorrar espacio. Al comienzo de la función de llamada algunos registros deben guardarse en la pila utilizando una instrucción **PUSH**. Esta instrucción también se puede utilizar para guardar el **LR** como el último registro en la pila. En lugar de restaurar el **LR** de nuevo así mismo y luego usar una instrucción explícita “Branch” el **LR** se puede restaurar directamente hacia el **PC** usando la instrucción **POP**. Si se hace de esta forma causara un regreso del **Calle al Caller**.



## Register usage example

Parameters passed in **r0 - r3**



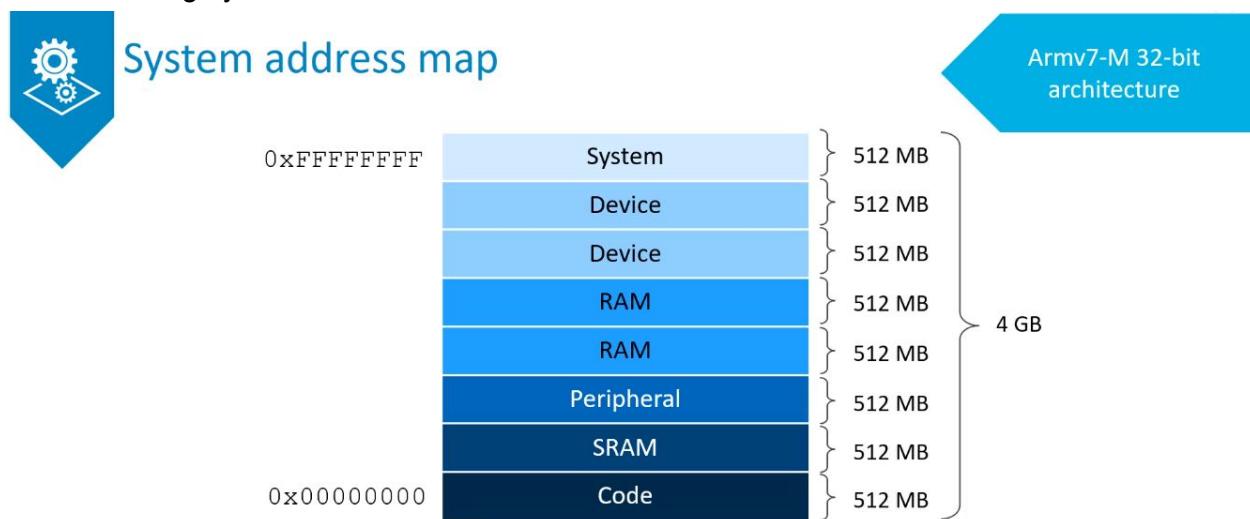
# Modelo de memoria

## Mapa de direcciones del sistema Armv7-M - parte 1

ARMv7-M es una arquitectura de 32 bits, y por lo tanto, el tamaño total del mapa de direcciones del sistema es de **4GB**.

Todas la direcciones de memoria utilizadas en sistemas de perfil M, Como **Cortex-M3** son direcciones físicas. Esto significa que el ARMv7-M es una arquitectura de memoria de mapeado plano (flat mapped memory architecture). El mapa predeterminado de memoria se divide en 8 segmentos primarios de 512 MB estas son: **Code, SRAM, Peripheral, 2 regiones RAM, 2 regiones Device y una región System.**

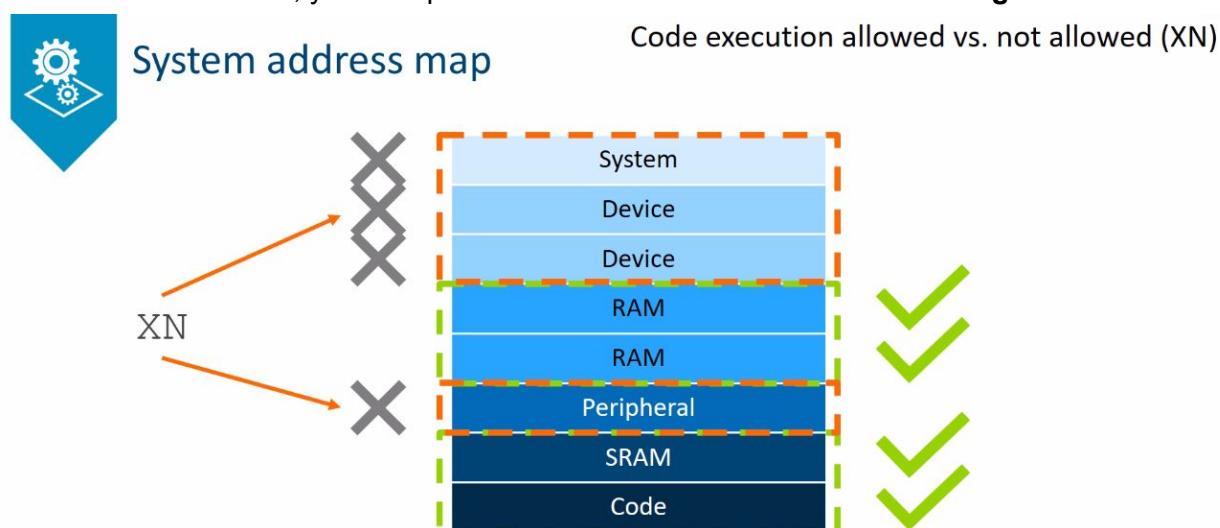
ARMv7-M es una arquitectura mapeada en memoria lo que significa que el procesador y los registros periféricos residen en direcciones específicas, y por lo tanto, se pueden acceder a través de operaciones estándar de Carga y Almacenamiento



## Mapa de direcciones del sistema Armv7-M - parte 2

Cada región tiene su propio conjunto de atributos. Un atributo importante informa al procesador si está permitido o no ejecutar código desde una dirección física particular. Por defecto la ejecución de código está permitida desde el **Code, SRAM, y 2 regiones RAM**.

Todas las demás regiones **Peripheral, las 2 regiones Device y la región System** son tratados por defecto como **eXecute Never (XN)**. Cualquier intento de ejecutar código desde una región marcada como XN desencadenara una falla, y más específicamente una extensión de **MemManage**.



# Tipos y propiedades de memoria Armv7-M

El modelo de memoria ARMv7-M admite 3 tipos diferentes de memoria. **Normal Memory**, **Device Memory** y **Strongly ordered Memory**. Cada tipo de memoria es adecuada para diferentes propósitos.

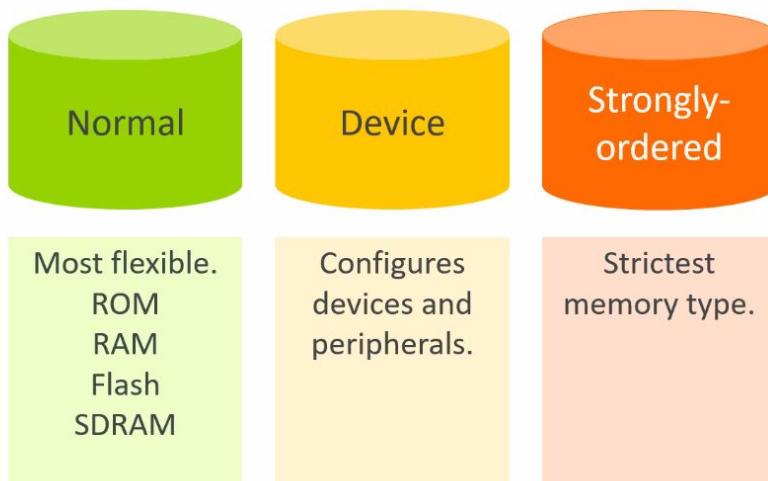
**Normal Memory** es el tipo de memoria más flexible adecuado para diferentes tipos de memoria (ROM, RAM)

**Device Memory** es adecuado para configurar dispositivos y periféricos, y es más estricto que el anterior. Por ejemplo: No es posible ejecutar código desde la memoria del dispositivo.

**Strongly ordered Memory** es la más estricta de los 3 tipos de memoria.



## Memory types and properties



### Reglas Importantes para tener en cuenta



- Unaligned accesses can be supported.
- Multi-cycle instructions can be interrupted.
- Data can be merged before accessing the memory system.
- Normal memory can be cached.

```
LDR    r1=0x20000010          0x20000010 0x200000012
STRH  r0, [r1,#0]
STRH  r0, [r1,#2]
```

**Device Memory** es adecuada para Periféricos y Dispositivos de E/S.



- Peripheral and I/O devices.
- Caches are not permitted.
- Unaligned accesses are unpredictable.
- Write buffers are supported.



- Similar to Device memory.
- Buffers are not supported.
- Private Peripheral Bus (PPB) is marked as strongly-ordered.

# Región del sistema Armv7-M

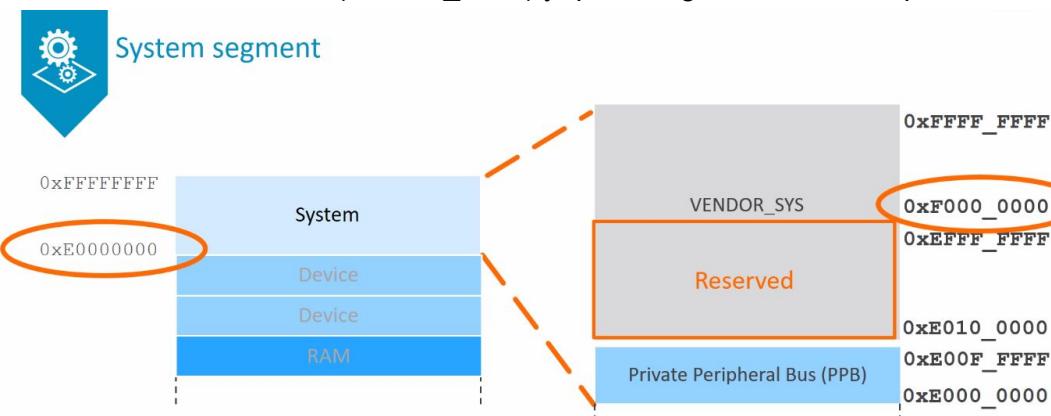
La región **System** del mapa de memoria, el cual comienza en la dirección de memoria 0xE0000000. Se divide en particiones principales.

Una región de **1 MB** conocida como **Private Peripheral Bus (PPB)** y una región de **511 MB** que comienza en un desplazamiento (offset) de **1 MB** desde la dirección base de la región System, que se conoce como región **Vendor system**.

El **PPB** proporciona al procesador un bus privado para controlar y configurar sus periféricos tales como

**Nested Vectored Interrupt Controller (NVIC), System Timer (SysTick), MPU y Debug Resources.**

La región **Vendor system** proporciona al proveedor de silicio la opción de agregar la implementación de recursos definidos dentro del espacio de direcciones físicas. ARM recomienda que los recursos del proveedor comiencen en la dirección (0xF000\_0000) y que la región de 255 MB que se muestra, esté reservada

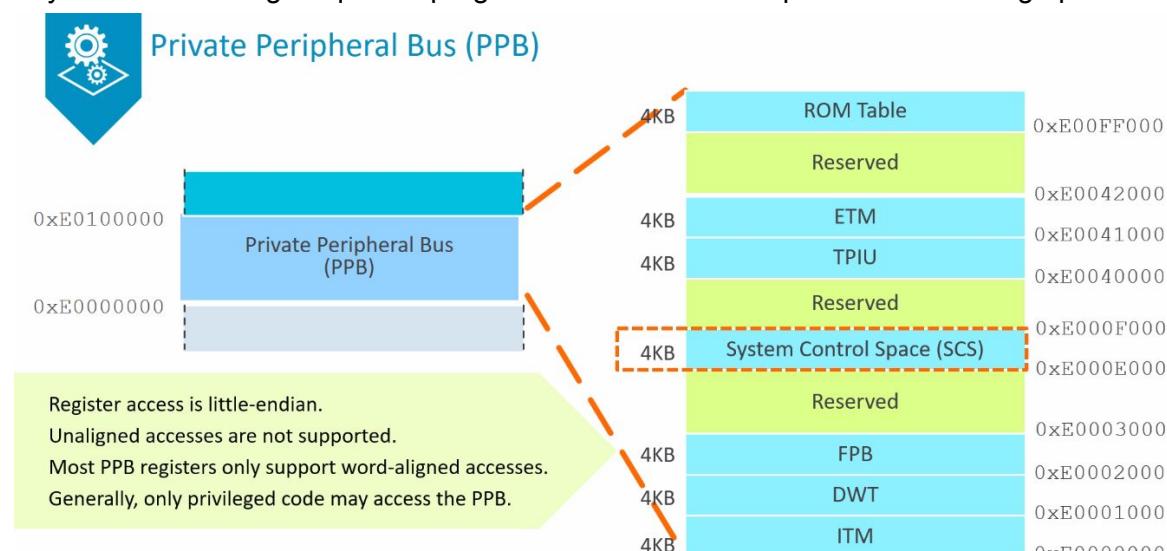


## Armv7-M Private Peripheral Bus

**PPB** contiene varios bloques de **4KB** de nivel superior diferentes. Muchos de los bloques están relacionados con la lógica de depuración dentro del procesador:

Instrumentation Trace Macrocell Unit (**ITM**), Data Watchpoint and Trace Unit (**DWT**), FlashPath and Breakpoint Unit (**FPB**), Trace Port Interface Unit (**TPIU**), Embedded Trace Macrocell (**ETM**), **CoreSight ROM Table** que proporciona herramientas de depuración con punteros a cada uno de los componentes de depuración. **System Control Space (SCS)** contiene información de configuración y muestra de información de estado para una amplia gama de otros recursos del procesador.

Hay una serie de reglas que los programadores deben cumplir al escribir código para el **PPB**



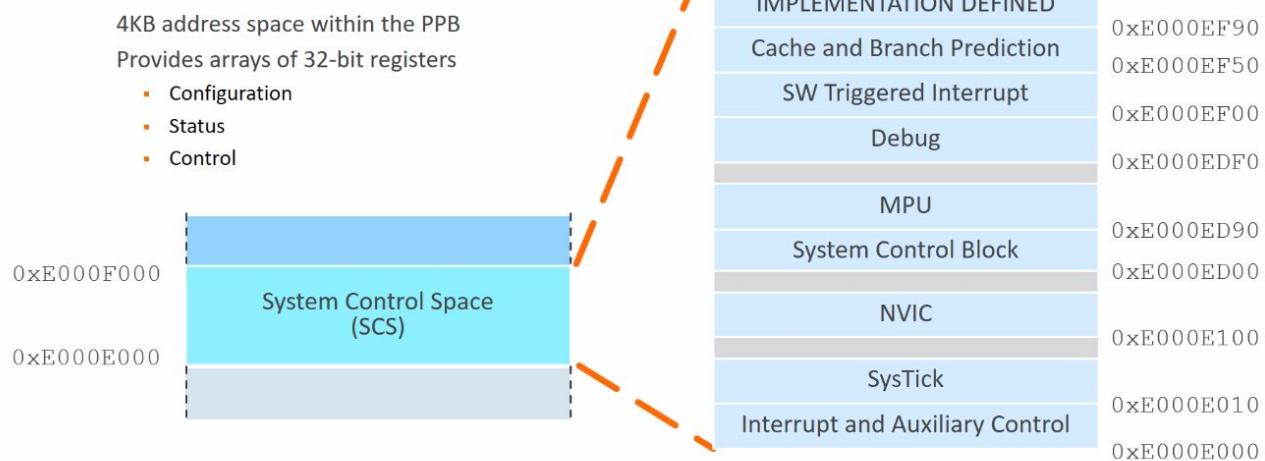
# Armv7-M System Control Space

El **SCS** es un espacio de direcciones de **4 KB** mapeado en memoria que proporciona principalmente registros de **32 bits** para configuración, informes de estado y de control. Los registros **SCS** se dividen en los siguientes grupos: La configuración del sistema, el estado y los registro de identificación en el Bloque de control del sistema (System Control Block SCB).

Se puede ampliar la información con el video



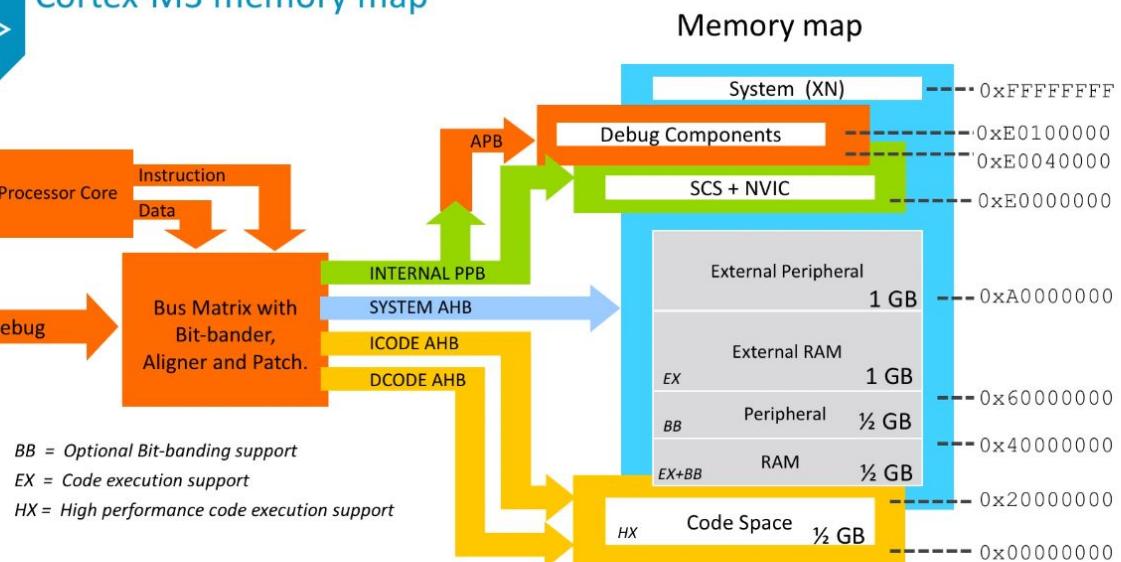
## System Control Space (SCS)



# Mapa de memoria de Cortex-M3



## Cortex-M3 memory map



El **Cortex-M3** contiene diferentes interfaces de bus para acceder a diferentes áreas de la memoria.

El Bus Matrix partitiona el acceso a la memoria usando los buses **AHB** y **PPB**. El bus **ICODE** y el bus **DCODE** siguen el protocolo de Bus Advanced High Performance (**AHB**). Estos buses son usados para acceder a los **512 MB** de memoria inferiores.

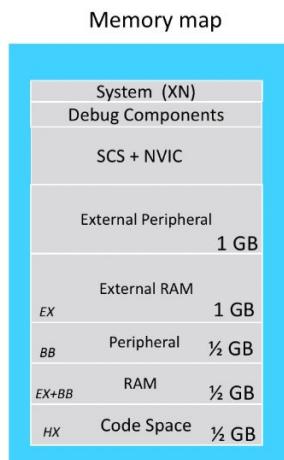
El **ICODE** está destinado a la obtención de instrucciones y el bus **DCODE** está destinado a accesos de datos.

El **SYSTEM AHB** se utiliza para accesos a la dirección 0x20000000 y superior.

Hay también el **PPB** el cual puede ser usado por el procesador y externamente. Por ejemplo: un agente de depuración externo

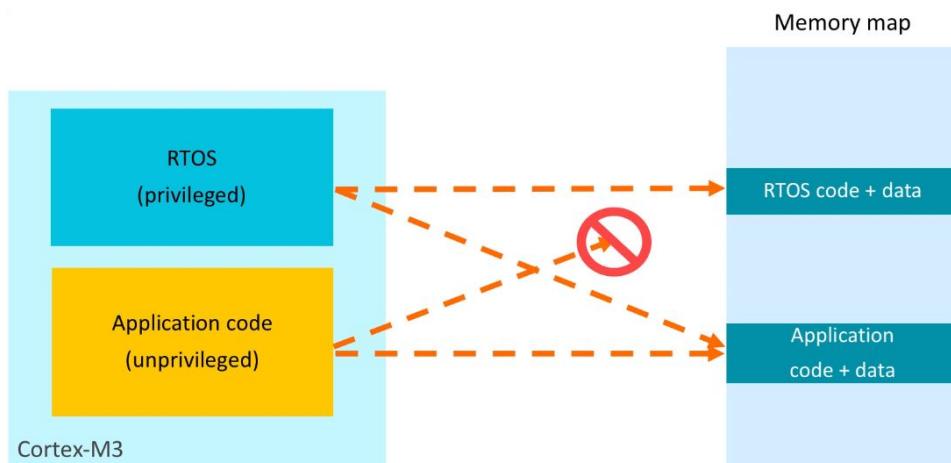
# Resumen de MPU

El mapa de direcciones del sistema predeterminado puede ser adecuado para algunas aplicaciones.



Sin embargo los sistemas más complejos pueden requerir flexibilidad adicional y protección adicional que puede ofrecer un mapa personalizado.

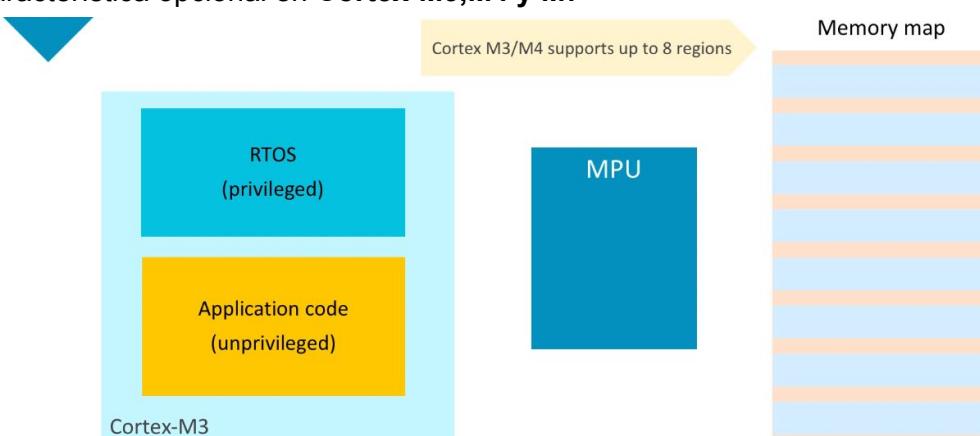
Por ejemplo Un sistema Operativo corriendo en ejecución privilegiada, necesitará acceso tanto al Sistema Operativo como a los códigos y datos de la Aplicación. Pero mientras que una aplicación debe tener acceso libre a su propio código asociado y regiones de datos, no se le debe dar acceso a códigos y datos asociados con el sistema operativo u otras aplicaciones.

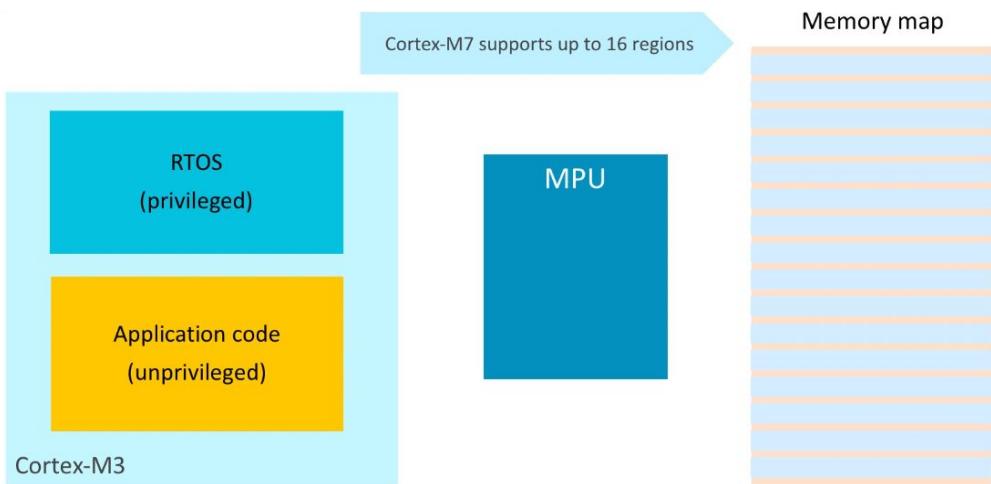


ARMv7-M soporta Protected Memory System Architecture (**PMSA**) donde está el espacio de direcciones protegido por una Unidad de Protección de Memoria (**MPU**).

La **MPU** se utiliza para definir un nuevo mapa de direcciones para reemplazar al mapa de direcciones predeterminado. Se puede usar la **MPU** para configurar regiones de memoria describiendo atributos que permiten al Core atrapar accesos ilegales antes de que lo intenten.

La **MPU** es una característica opcional en **Cortex-M3,M4 y M7**



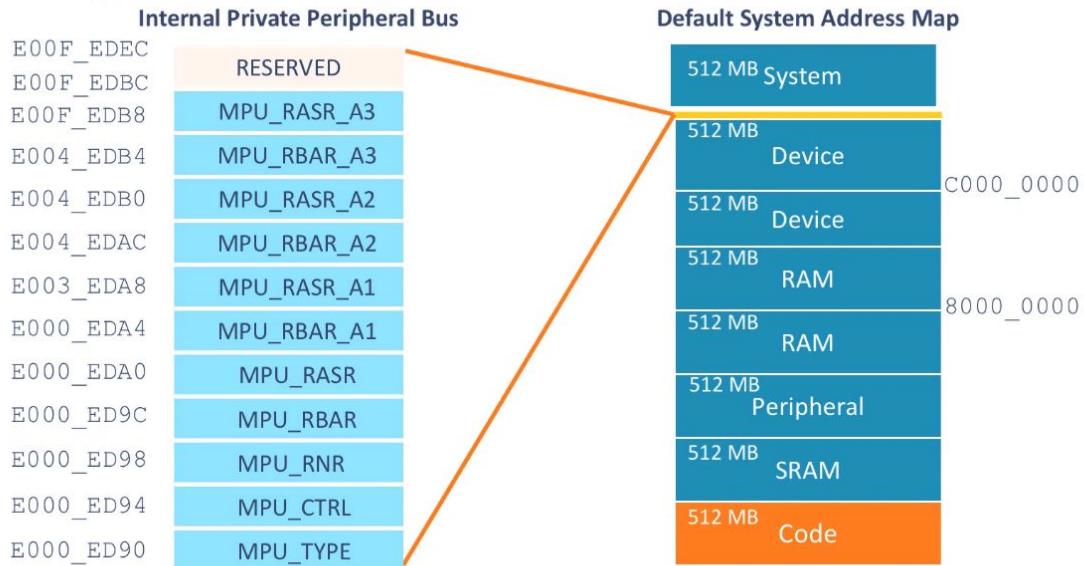


En la siguiente imagen podemos ver los registros del **MPU** en el **SCS**

Estos registros se utilizan para configurar la **MPU** y solo se puede acceder en modo privilegiado. Cualquier acceso sin privilegios genera una **BusFault Exception**



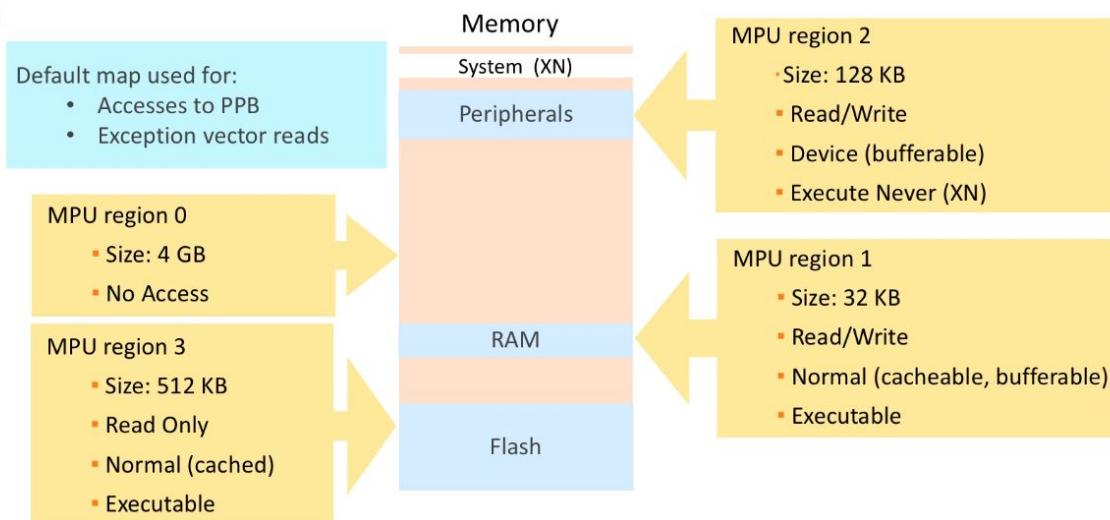
## MPU registers in the System Control Space



Ejemplo de sistema que contiene 3 memorias



## Memory Protection Unit (MPU) overview

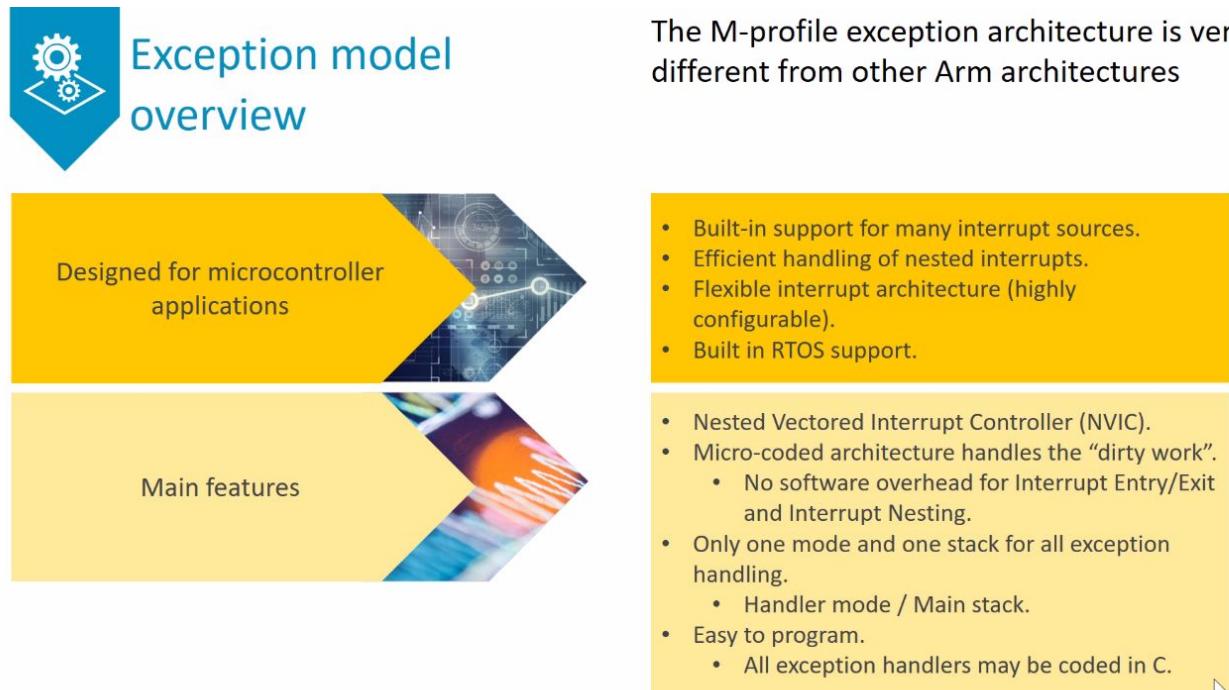


# Modelo de excepción

## Resumen de excepciones

Una característica clave de cualquier implementación de perfil M es la construcción en **Nested Vectored Interrupt Controller (NVIC)**.

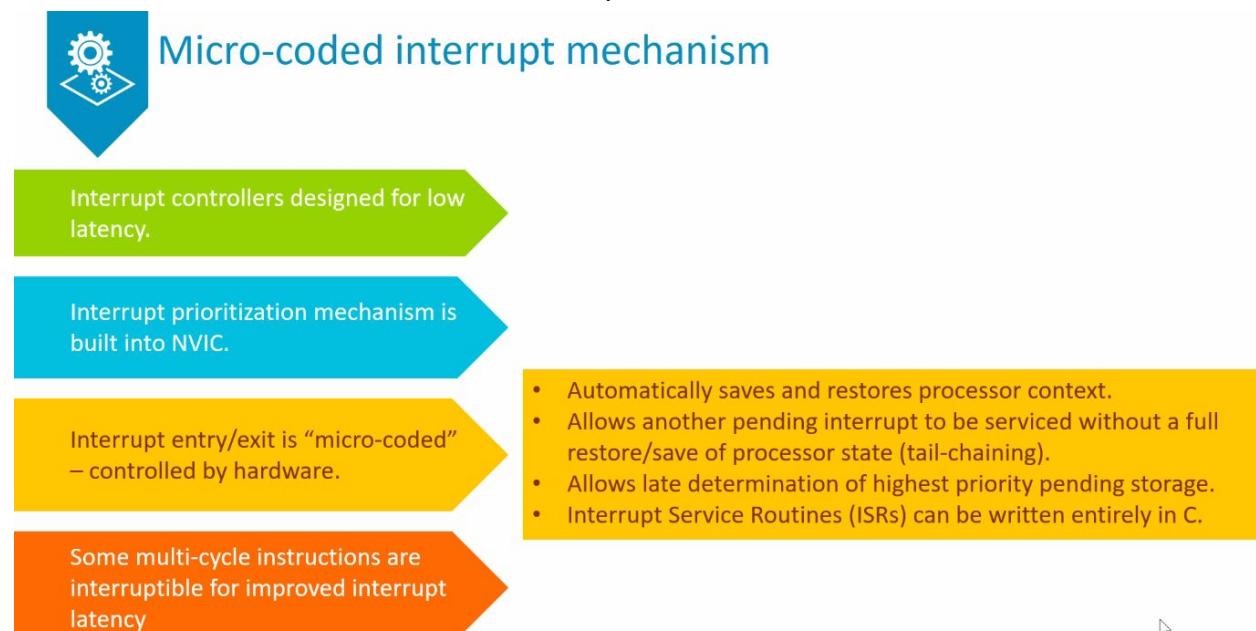
**NVIC** proporciona una gran cantidad de funciones integradas para manejar ciertas operaciones críticas que de otro modo tendría que ser realizado por software dentro de una rutina de servicio de interrupción. Lo que significa que el código del controlador de interrupciones se puede escribir completamente en C o C++.



## Mecanismos de interrupción micro-codificados

La arquitectura de interrupción está diseñada para baja latencia y hay un mecanismo integrado de priorización de interrupciones dentro del NVIC.

El hardware es capaz de manejar eficientemente las excepciones de entrada / salida guardando y restaurando automáticamente el contexto del procesador



## Interrupciones externas

El número de interrupciones externas que el **NVIC** debe poder manejar es definido por la implementación. La arquitectura ARMv7-M admite hasta **496** interrupciones externas y **1 Interrupción Non-Maskable** de alta prioridad, conocida como **NMI**.

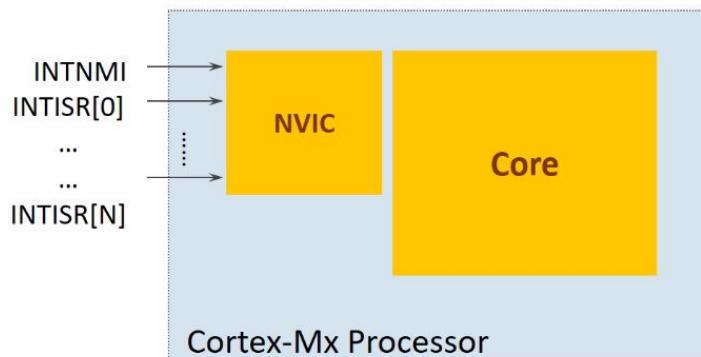
Cortex-M3 admite entre **1 y 240** interrupciones **externas** y **1 NMI**.



### External interrupts

External Interrupts handled by Nested Vectored Interrupt Controller (NVIC).

Tightly coupled with processor core.



## Tablas de vectores Armv7-M

Las tablas de vectores de los sistemas contienen instrucciones en lugar de datos.



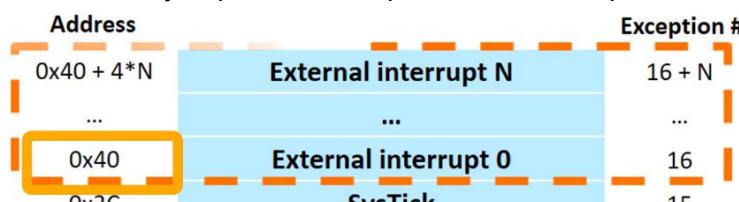
### Vector table for Armv7-M

Address	Exception #
0x40 + 4*N	External interrupt N
...	...
0x40	External interrupt 0
0x3C	SysTick
0x38	PendSV
0x34	Reserved
0x30	Debug Monitor
0x2C	SVCall
0x1C to 0x28	Reserved (x4)
0x18	UsageFault
0x14	BusFault
0x10	MemManage
0x0C	HardFault
0x08	NMI
0x04	Reset
0x00	SP_main
	N/A

En ARMv7-M las primeras 16 entradas de la tabla de vectores desde el desplazamiento **0x0** hasta la entrada **SysTick** en el desplazamiento **0x3C** están siempre presente.

0x3C	SysTick	15	0x3C	SysTick	15
0x38	PendSV	14	0x38	PendSV	14
0x34	Reserved	13	0x34	Reserved	13
0x30	Debug Monitor	12	0x30	Debug Monitor	12
0x2C	SVCall	11	0x2C	SVCall	11
0x1C to 0x28	Reserved (x4)	7-10	0x1C to 0x28	Reserved (x4)	7-10
0x18	UsageFault	6	0x18	UsageFault	6
0x14	BusFault	5	0x14	BusFault	5
0x10	MemManage	4	0x10	MemManage	4
0x0C	HardFault	3	0x0C	HardFault	3
0x08	NMI	2	0x08	NMI	2
0x04	Reset	1	0x04	Reset	1
0x00	SP_main	N/A	0x00	SP_main	N/A

Los vectores del desplazamiento **0x40** y superiores son para cada interrupción externa presente en el sistema



La primera entrada en la tabla de vectores es diferente a otros vectores ya que el valor ubicado aquí se almacena en **MSP** al reiniciar, mientras que todos los demás vectores, excepto los vectores reservados, afectan al **PC**

0x08	NMI	2
0x04	Reset	1
0x00	SP_main	N/A

El valor inicial del **PC** se toma del vector **Reset**.

El vector **NMI** se usa para una interrupción crítica como un temporizador de vigilancia, es la tercera entrada en el la tabla y a partir de ahí siguen los vectores para cada tipo de falla que podria ocurrir en el sistema.

0x18	UsageFault	6
0x14	BusFault	5
0x10	MemManage	4
0x0C	HardFault	3

Hay 5 vectores reservados para una futura expansión. De hecho, una de estas entradas es usado para una excepción **SecureFault** en ARMv8-M.

0x34	Reserved	13
0x30	Debug Monitor	12
0x2C	SVCall	11
0x1C to 0x28	Reserved (x4)	7-10

En este offset hay una llamada **SVCall** para manejar una excepción causada por una instrucción **SVC**

0x2C	Debug Monitor	12
0x28	SVCall	11
0x24 to 0x28	Reserved (x4)	7-10

El vector **PendSV** está relacionado con **SVCall** y está diseñado para ser utilizado por sistemas operativos embebidos o en tiempo real para tareas de baja prioridad como cambio de contexto.

0x38	PendSV	14
0x34 to 0x38	Reserved	N/A

El temporizador **SysTick** puede configurarse para generar una interrupción cuando su contador llega a cero, por lo que podría usarse para generar una interrupción regular, por ejemplo, para permitir un sistema de operación para cambiar Thread regularmente.



El vector final es el vector **DebugMonitor**. Este vector se puede usar para permitir que el procesador maneje eventos de depuración a través de un controlador de excepciones **DebugMonitor** en lugar de que el depurador detenga el procesador. **DebugMonitor** es una herramienta útil para depurar sistemas donde el procesador debe mantenerse funcionando.



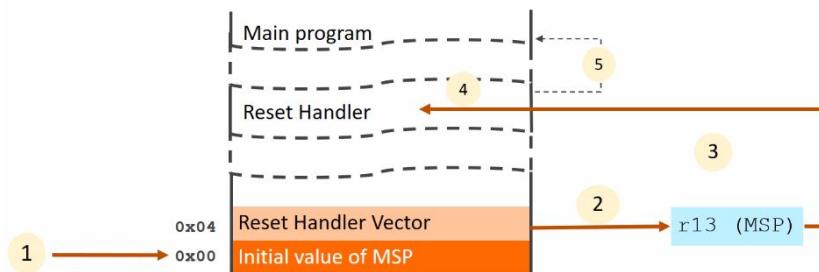
Cada excepción tiene un número, que se almacena en el **IPSR** para indicar que controlador de excepción se está ejecutando actualmente. El número también se almacena en el **Interrupt Control and State Register** una vez que se active una interrupción. Debido a la anidación, puede haber más de una interrupción activa.

Address		Exception #
0x40 + 4*N	External interrupt N	16 + N
...	...	...
0x40	External interrupt 0	16
0x3C	SysTick	15
0x38	PendSV	14
0x34	Reserved	13
0x30	Debug Monitor	12
0x2C	SVCall	11
0x1C to 0x28	Reserved (x4)	7-10
0x18	UsageFault	6
0x14	BusFault	5
0x10	MemManage	4
0x0C	HardFault	3
0x08	NMI	2
0x04	Reset	1
0x00	SP_main	N/A

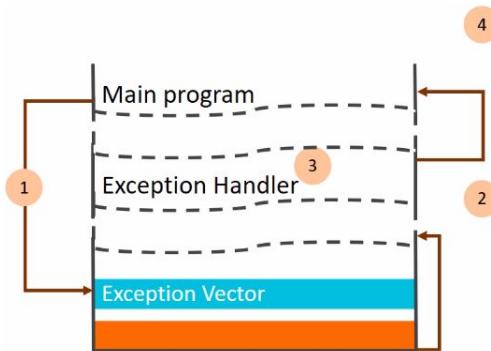
## Comportamientos (Reset and Exception)

Veamos que debería suceder fuera del **Reset** cuando la entrada de reinicio del procesador está impuesta. Como parte de la secuencia de reinicio en cualquier sistema de perfil M, el valor inicial de **MSP** se carga desde esta dirección (0x00) y la dirección **reset handler** es cargado en el **PC** desde esta dirección (0x04).

El **Reset Handler** se ejecuta en **Modo Thread privilegiado** y probablemente bifurcará a un **Main program** escrito en C o C++.

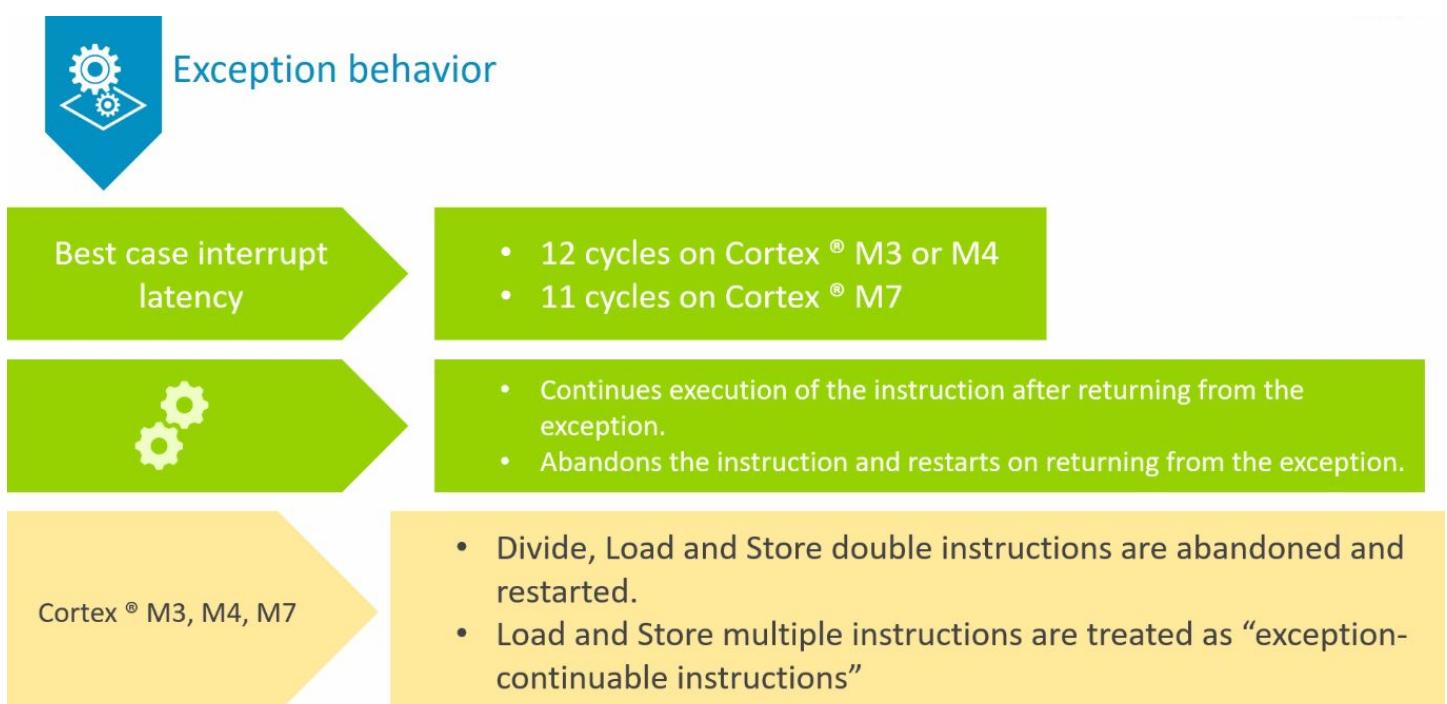


Si se produce una excepción y tiene una prioridad suficiente para ser atendida, el flujo de instrucciones actual se detiene, el estado del procesador se almacena en el pila actual, y el procesador accede a la tabla de vectores para leer la dirección del vector para la excepción asociada. El **Exception Handler** se ejecuta en **Modo Handler**. Una vez que la excepción ha sido atendida, su controlador puede volver al Thread interrumpido, asumiendo que no hay más excepciones pendientes con suficiente prioridad que podrían ser atendidas primeras. Otra cosa que sucede en la entrada de excepción es que el **LR** se modifica hacia el valor **EXC\_RETURN** especial, que indica como el procesador debe volver de forma segura al Thread previamente interrumpido.



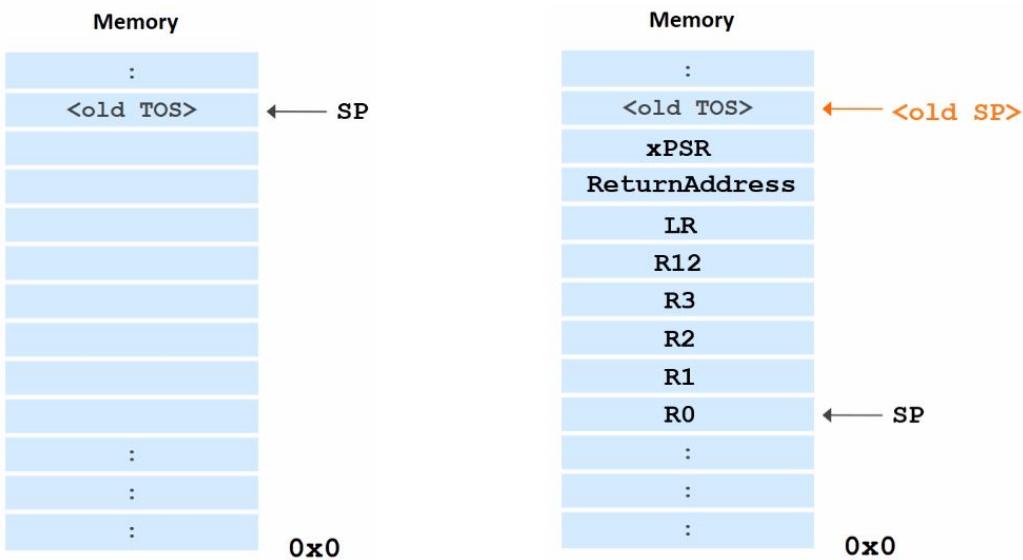
La latencia de interrupción del mejor caso (**Best case interrupt latency**) suponiendo estados de espera 0 en el sistema de memoria, es de **12 ciclos** en un **Cortex M3 o M4** y **11 ciclos** en un **Cortex-M7**. Sin embargo interrumpir las excepciones **late-arriving** y excepciones **tail-chaining** pueden mejorar la latencia interrupción. Al recibir una excepción el procesador termina de ejecutar la instrucción actual para la mayoría de las instrucciones, sin embargo, para minimizar la latencia de interrupción, el procesador puede tomar una excepción durante la ejecución de una instrucción de varios ciclos.

Al tomar una excepción durante una instrucción de varios ciclos, el procesador continúa la ejecución de la instrucción después de regresar de la instrucción o abandona la instrucción y reinicia la instrucción al regresar de la excepción.

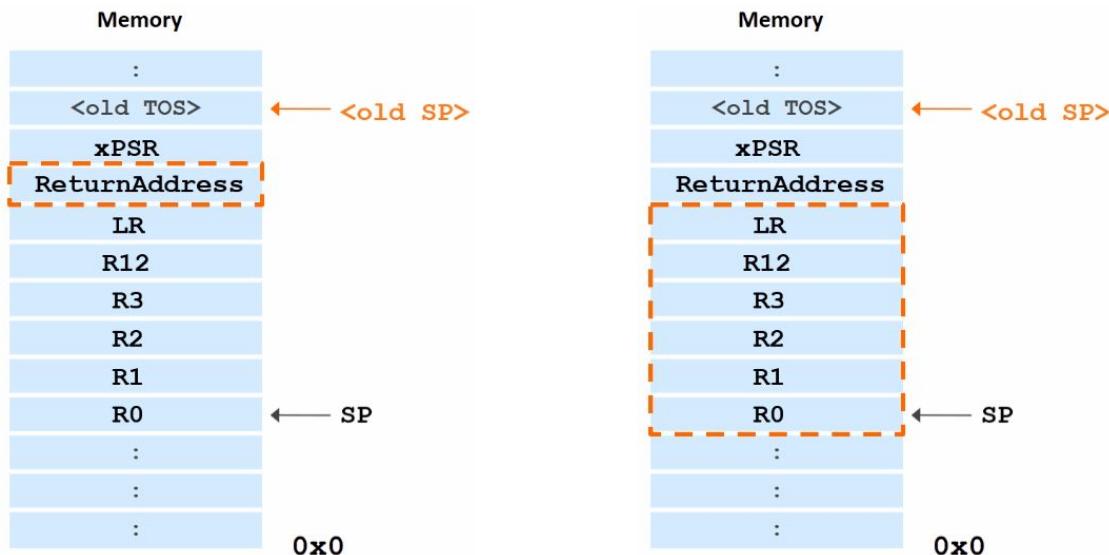


## Entrada de excepción de apilamiento

Veamos la información que el procesador introduce en la pila durante la entrada de excepción. La pila es **Full Descending** lo que significa que el hardware disminuye el **SP** al final del nuevo stack frame antes que almacene datos en la pila. Full significa que el **SP** apunta al último elemento conocido de la pila, que se muestra como **“old Top Of Stack”**. Cuando ocurre una excepción, el hardware guarda **8 Words** de **32 bits**, que comprenden **xPSR, Return Address, LR, R12, R3, R2, R1 y R0**.



El **xPSR** contiene el estado anterior del procesador, por lo tanto, es vital que esta información se conserve. También es vital guardar la dirección de donde el procesador debería retornar después de reparar la excepción. Para las mayorías de las excepciones, como las interrupciones y **SVCALL**, **ReturnAddress** es la dirección de la instrucción que se habría ejecutado a continuación si no hubiera ocurrido la excepción. Sin embargo, para algunas excepciones como **UsageFault**, **ReturnAddress** es la dirección de la instrucción que causó la excepción, para permitir que la instrucción que causó la falla se vuelva a ejecutar, después de que la falla haya sido reparada por el **handler code**.



Los registros restantes **R0-R3, R12** y **LR** deben guardarse de acuerdo con el **AAPCS**, para que **handler code** de excepciones no los corrompa. Al guardar estos registros en la pila automáticamente, el **handler code** de excepciones no necesita preocuparse por corromper el contenido de estos registros, reduciendo la sobrecarga del software y permitiendo que manipuladores de excepciones sean escritos como funciones normales de C. Las ubicaciones de pila para cada registro son fijas.

Finalmente este marco de pila (**stack frame**) de 8 registros es conocido como el “marco básico (**Basic Frame**)”

## Mecanismo de retorno de excepción

Una vez que los 8 registros se han apilado, el **LR** se modifica a un valor especial conocido como **EXC\_RETURN**. **EXC\_RETURN** no es un registro real. **EXC\_RETURN** es un valor especial definido arquitectónicamente por el mecanismo de retorno de excepción.

Se produce un retorno de excepción cuando una de las siguientes instrucciones carga un valor **EXC\_RETURN** en el **PC**. A continuación podemos verlo en la siguiente imagen



## Exception return mechanism

- POP/LDM that includes loading the program counter (PC).
- LDR with PC as a destination.
- BX with any register (for example, BX LR).

Cargar el PC con una dirección como indica **EXC\_RETURN**, por ejemplo la dirección 0xFFFFFE1, provocaría un error ya que todos los valores de **EXC\_RETURN** son direcciones ilegales para la ejecución de instrucciones.

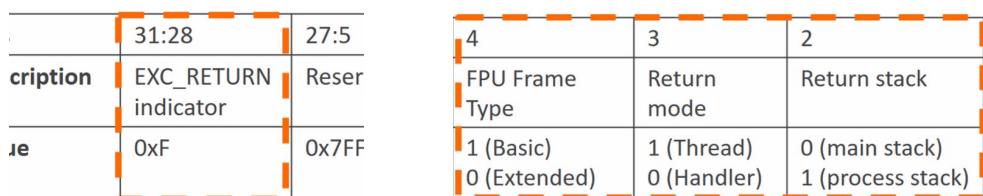


## Exception return mechanism

Bits	31:28	27:5	4	3	2	1	0
Description	EXC_RETURN indicator	Reserved	FPU Frame Type	Return mode	Return stack	Reserved	CPU state
Value	0xF	0x7FFFFF	1 (Basic) 0 (Extended)	1 (Thread) 0 (Handler)	0 (main stack) 1 (process stack)	0	1

EXC_RETURN	Condition
0xFFFFFE1	Return to Handler Mode and restore extended stack frame
0xFFFFFE9	Return to Thread mode using the main stack and restore extended stack frame
0xFFFFFED	Return to Thread mode using the process stack and restore extended stack frame
0xFFFFFFF1	Return to Handler Mode and restore basic stack frame
0xFFFFFFF9	Return to Thread mode using the main stack and restore basic stack frame
0xFFFFFFF9	Return to Thread mode using the process stack and restore basic stack frame

En cambio, cuando el procesador ve que los bits 28-31 de la dirección están configurados, se da cuenta de que necesita realizar un retorno de excepción, y en su lugar lee los bits 2, 3 y 4 de **EXC\_RETURN** para determinar qué hacer. El bit 2 indica que pila usar para restaurar el contenido anterior. El bit 3, indica al procesador el modo que se ingresa al regresar de la interrupción. El bit 4 se usa para determinar si el procesador debe restaurar un marco(frame) básico o extendido.



## Ejemplo de entrada de excepción NMI

Que ocurre en la entrada de excepción (exception entry) utilizando una excepción **NMI** como ejemplo.

Para empezar el procesador está en **Modo Thread** y está ejecutando una instrucción **ADD** cuando un **NMI** se impone. La instrucción **ADD** de ciclo único se completa antes de que el procesador tome las siguientes acciones principales.



## NMI exception entry example

Thread code is executed in Thread Mode → NMI interrupt is activated

Processor



8 registros se insertan en la pila actual. El registro de **CONTROL** indica que pila, **Main Stack o Process Stack**, se estaba utilizando en **Modo Thread**.

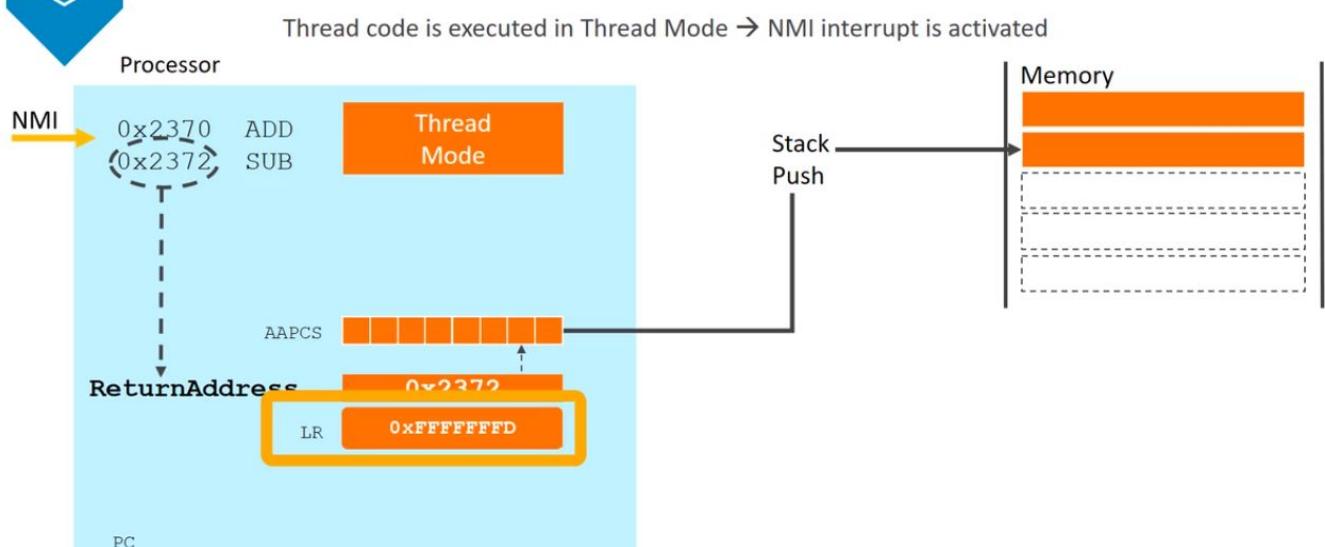
**ReturnAdress** se establece en la dirección de la instrucción **SUB**, siguiente.

Tener en cuenta que el valor **EXC\_RETURN** contiene el estado de la **CPU**, razón por la cual **ReturnAdress** no requiere el conjunto de bits inferior.

El **LR** se modifica a esta dirección por retorno de excepción, que indica que **Process Stack** estaba activada en el momento de la excepción.



## NMI exception entry example

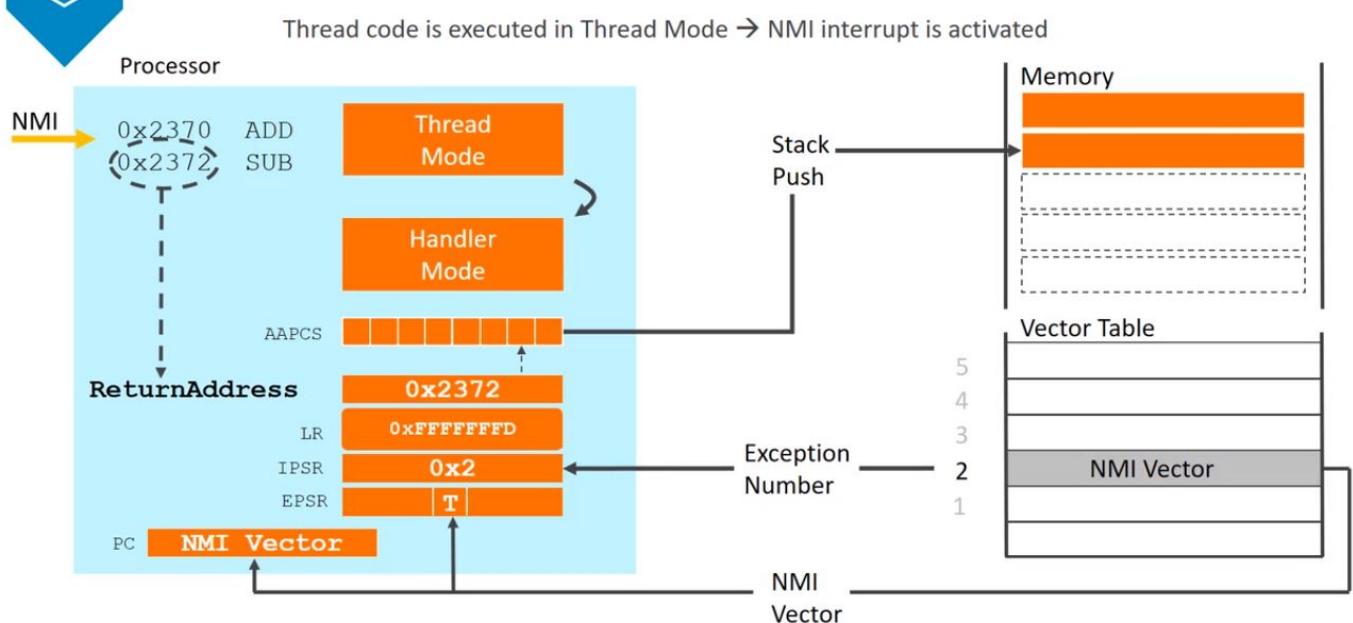


Durante (o después) el estado de ahorro la dirección del **NMI handler** se lee de la tabla de vectores. El procesador cambia a **Modo Handler**. El **IPSR** se actualiza con la número de excepción correspondiente al **NMI**. El **EPSR T-bit** se establece de acuerdo con el bit menos significativo del vector **NMI**. Si no se establece el bit **T** en 1, se producirá un error de excepcion de entrada. Se ejecuta la primera instrucción de la rutina del controlador NMI (**NMI handler**).

La **NMI handler** se ejecuta en **Modo Handler** utilizando la **Main Stack**.



## NMI exception entry example



## Ejemplo de retorno de excepción NMI

Es simple para el software volver a la tarea en primer plano desde el **NMI handler**.

El **LR**, **EXC\_RETURN** que contiene este valor (0xFFFFFFF7) indica que un retorno de excepción, retornara el procesador a **Modo Thread** utilizando **Main Process Stack**.

Si el valor **EXC\_RETURN** se mantiene en el **LR** al final de **NMI handler** simplemente puede ejecutar una operación **BX LR** para volver al contexto anterior.



### NMI exception return example

BX or POP instruction can be used for exception return

Processor

0x2370 ADD  
0x2372 SUB

Thread Mode

Handler Mode

LR 0xFFFFFFF7

BX  
POP

Return to Thread mode using the Main stack

Si el **LR** se guardó en la pila al comienzo de la rutina de servicio de interrupción el software simplemente puede usar una operación **POP** para cargar el valor **EXC\_RETURN** de la pila en el **PC**.

Al ejecutar una operación de retorno de excepción válida, el procesador hará **POP** el contexto anterior de la Main Stack en los 8 registros antes mencionado.

Configurar **PC** en **ReturnAddress** para recuperar la operación **SUB** y siguiendo las instrucciones en el **Processor's Pipeline**.

Y finalmente establecer el **IPSR** en este valor (0x0) que indica que el procesador está de nuevamente en **Modo Thread** y ya no maneja ninguna excepción.



### NMI exception return example

BX or POP instruction can be used for exception return

Processor

0x2370 ADD  
0x2372 SUB

Thread Mode

Handler Mode

LR 0xFFFFFFF7

AAPCS

BX  
POP

Return to Thread mode using the Main stack

Stack Pop

Memory

# Resumen de prioridades de excepción(exception priorities)

Cada excepción tiene una prioridad. Cuanto menor sea el número de prioridad, mayor será el nivel de prioridad de la excepción.

**Reset, NMI y HardFault** tienen prioridades fijas **-3, -2 y -1**.

Todos las demás excepciones tiene un nivel de prioridad programable. El nivel de prioridad para cada excepción que tiene una prioridad programable se encuentra en varios registros diferentes, **System Handler Priority Register 1 (SHPR1)** contiene 3 campos de tamaño byte para la configuración del nivel de prioridad para las fallas de **MemMangage, BusFault y UsageFault**.

**System Handler Priority Register 2 (SHPR2)** contiene un campo de tamaño byte para configurar el nivel de prioridad de **SVCall**.

**System Handler Priority Register 3 (SHPR3)** contiene 3 campos de tamaño byte para establecer los niveles de prioridad de **DebugMonitor, PendSV y SysTick**.

Finalmente hay varios registros de prioridad de interrupción (**NVIC\_IPRn**) dependiendo cuantas interrupciones externas se hayan implementado. Cada registro contiene 4 campos de prioridad, cada uno representando el nivel de prioridad para una interrupción dada. Si sólo hay 4 interrupciones externas en una implementación de ARMv7-M, solo hay un **Interrupt Priority Register** utilizado para programar el nivel de prioridad de cada interrupción.

Es arquitectónicamente posible para una implementación ARMv7-M tener **496 interrupciones externas**, lo que significa que habría **128 Interrupt Priority Register** para cada interrupción en el sistema



## Exception priorities overview

- The lower the priority number, the higher the priority level
- Priority level is stored in a byte-wide register, which is set to 0x0 at reset
- Exceptions with the same priority level follow a fixed priority order using the exception number
  - Lower exception number has higher priority level

Name	Exception Number	Exception Priority No.
Interrupts #0 - #495 (N interrupts)	16 to 16 + N	0-255 (programmable)
SysTick	15	0-255 (programmable)
PendSV	14	0-255 (programmable)
DebugMonitor	12	0-255 (programmable)
SVCall	11	0-255 (programmable)
UsageFault	6	0-255 (programmable)
BusFault	5	0-255 (programmable)
MemManage Fault	4	0-255 (programmable)
HardFault	3	-1
Non Maskable Interrupt (NMI)	2	-2
Reset	1	-3



## Ejemplo de anidamiento (Nesting)

En este ejemplo vemos cómo se manejan las interrupciones anidadas en una implementación ARMv7-M.

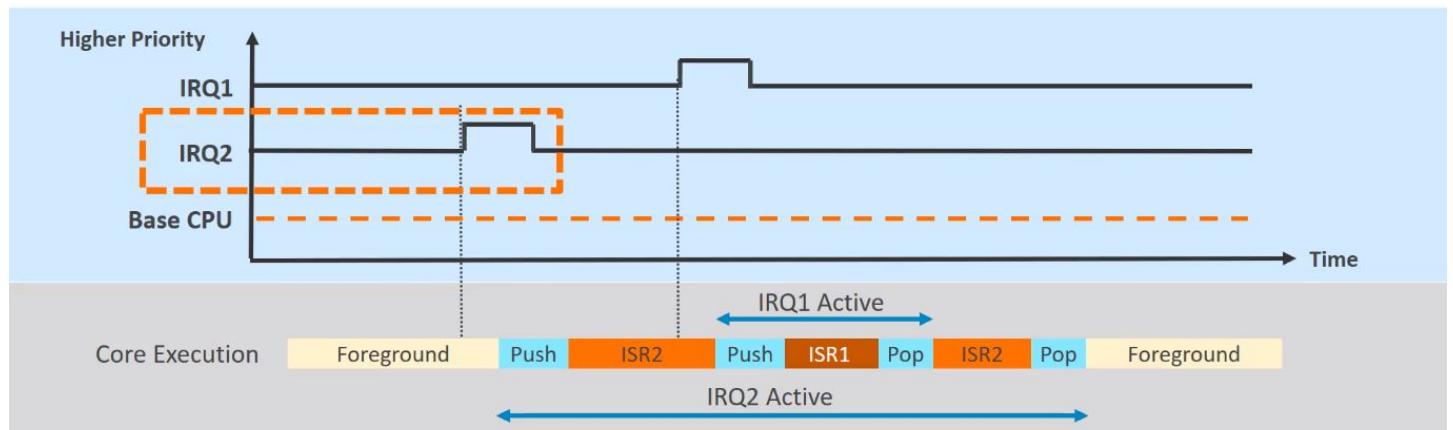
En este ejemplo hay 2 interrupciones externas **IRQ1** y **IRQ2**. Además, en este ejemplo, la tarea en primer plano (foreground task) se ejecuta en el “nivel base de prioridad de ejecución” lo que significa que el Core está en **Modo Thread**.

**IRQ2** que tiene una prioridad más baja que **IRQ1**, se impone primero, interrumpiendo la tarea en primer plano (foreground task).

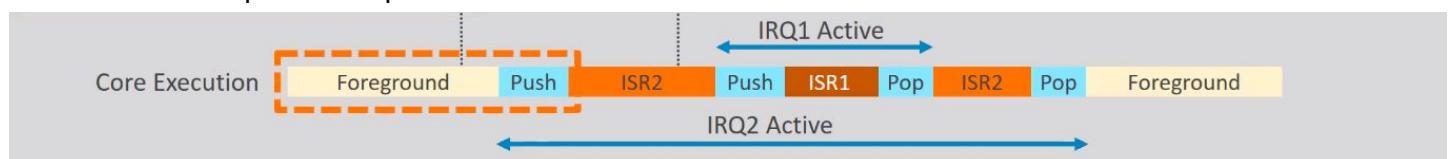
**Interrupt Request (IRQ)** : Pedido de interrupción



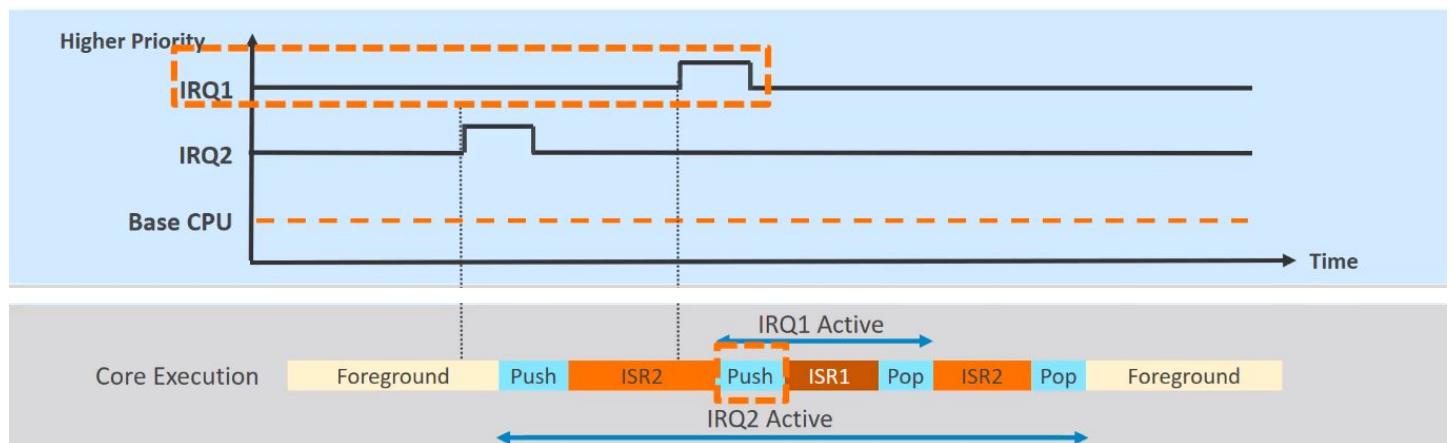
## Nesting example



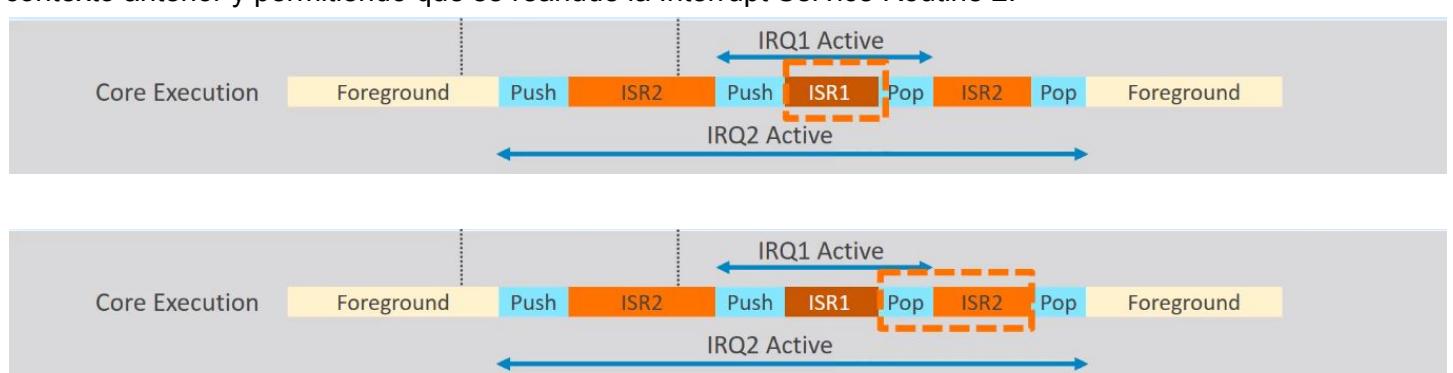
Esto da como resultado que el contexto de la tarea en primer plano (foreground task) sea automáticamente almacenada en la pila actual por el hardware



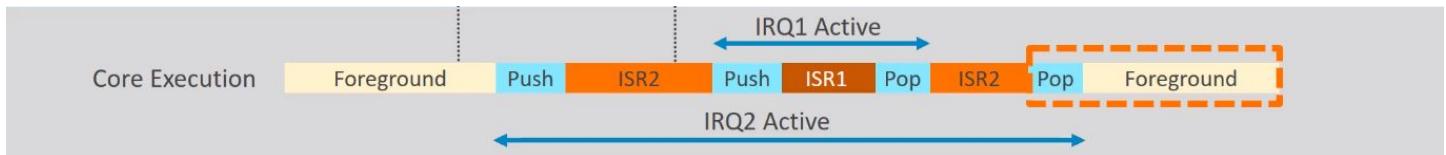
Mientras se ejecuta el Interrupt Service Routine para el **IRQ2**, **IRQ1** se impone, lo que da como resultado otra Stack Push: esta vez a la Main Stack ya que el procesador está en **Modo Handler**.



La Interrupt Service Routine para **IRQ1** se ejecuta hasta su finalización, resultando en la restauración del contexto anterior y permitiendo que se reanude la Interrupt Service Routine 2.



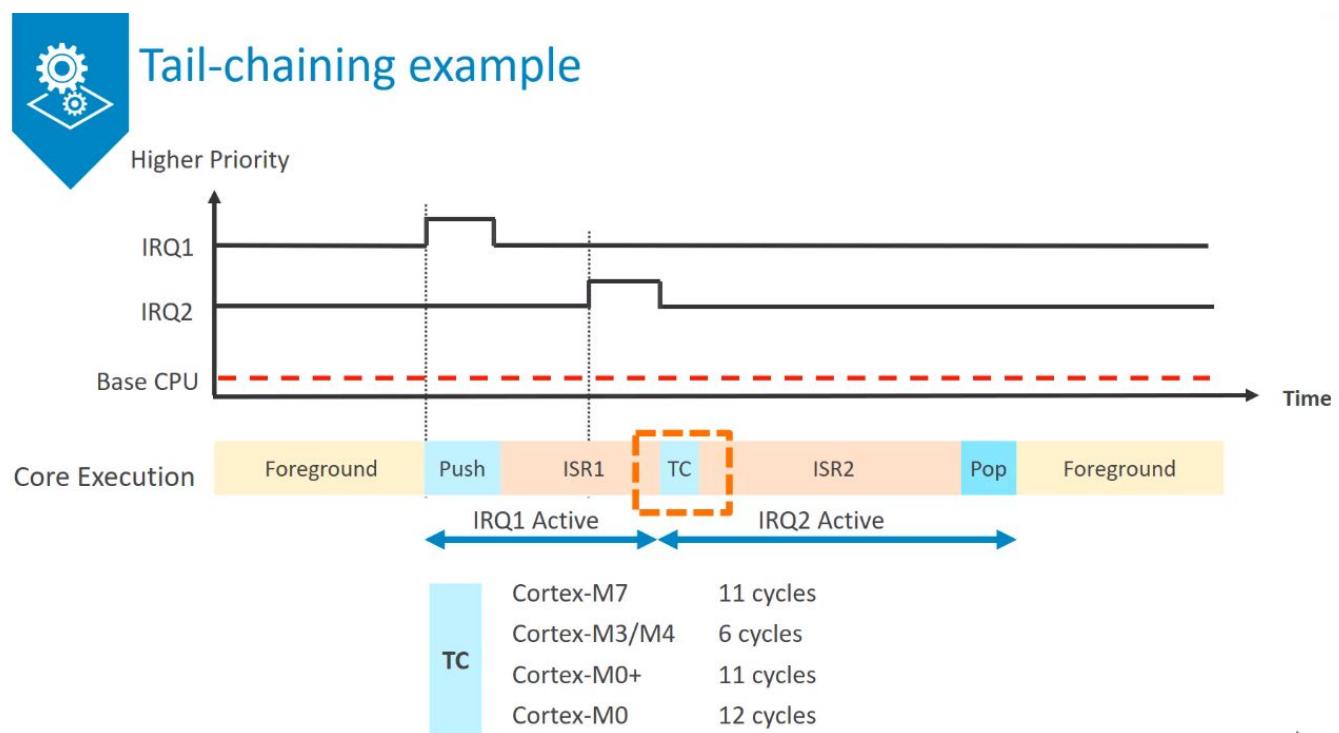
Cuando se completa la Interrupt Service Routine 2, hay un **POP** final de la pila siendo usado por la tarea en primer plano (foreground task), lo que permite que la tarea de primer plano (foreground task) reanude la ejecución.



Tener en cuenta que el **IRQ2** todavía está activo incluso cuando se está manejando el **IRQ1**. Esta información está presente en los registro de bits activos de interrupción.

## Ejemplo de encadenamiento de cola (Tail-chaining)

Esta vez **IRQ1** se impone primero. Mientras se revisa **IRQ1**, se impone **IRQ2** pero no tiene un nivel suficiente para interrumpir el **IRQ1**. El estado de primer plano (foreground state) fue guardado después de imponerse el **IRQ1**. Entonces en lugar de restaurar el contexto anterior cuando finaliza la **Interrupt Service Routine 1** y vuelve a guardar el contexto de la tarea en primer plano (foreground task) el procesador puede encadenar (tail change) a la **Interrupt Service Routine 2**. La optimización de encadenamiento de cola (tail-changing) mejora los tiempos de respuesta de interrupción. Ej: en **Cortex-M3** el mejor caso de latencia para tail-changing es de 6 ciclos, mientras que habría tomado 10 ciclos para restaurar y 12 ciclos para guardar el estado de contexto (context state). Lo que da un ahorro de 16 ciclos !!



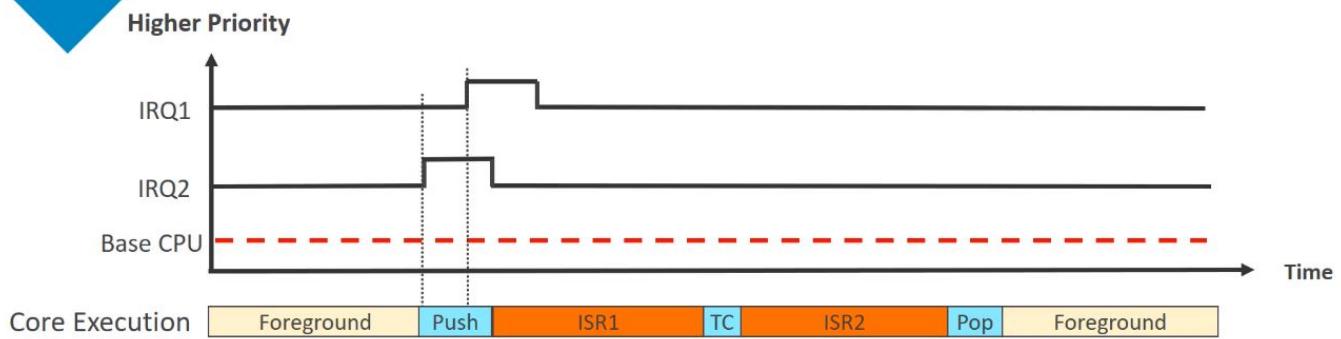
## Ejemplo de Late-arriving

Este ejemplo muestra la interrupción de menor prioridad imponiéndose primero pero mientras se guarda el contexto de la tarea en primer plano (foreground task), **IRQ1** afirma, que tiene un mayor nivel de prioridad, y debido a que la **Interrupt Service Routine** para **IRQ2** no ha comenzado, todavía es posible que la **Interrupt Service Routine 1** reciba servicio primero.

Además, una vez que **Interrupt Service Routine 1** se ha completado, es posible que el procesador encadene (tail-chain) a **Interrupt Service Routine 2** proporcionando beneficios adicionales en términos de latencia de interrupción.



## Late-arriving example



A pending higher priority exception is handled before an already pending lower priority exception even after exception entry sequence has started.

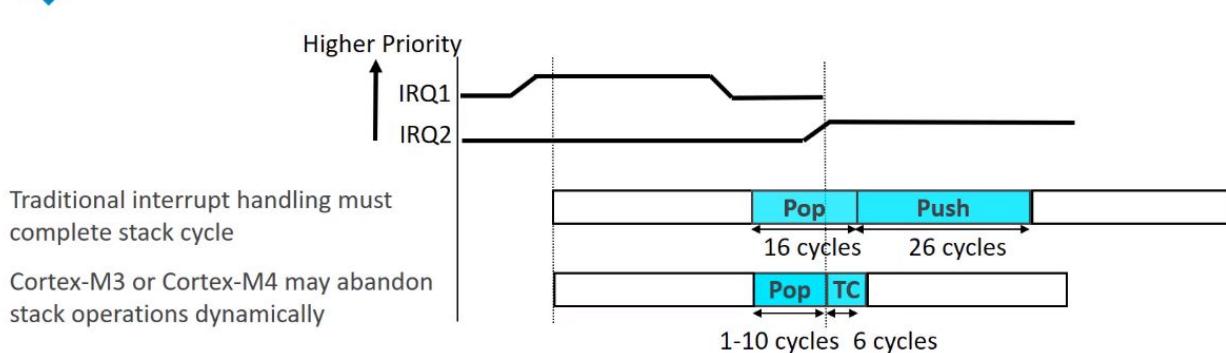
The lower priority exception is handled after the higher priority exception.

## Excepciones durante la restauración del estado

Cuando el procesador está restaurando el contexto anterior es posible que una excepción ocurra. En este ejemplo, **IRQ1** ha sido reparado y está en el proceso de restaurar el contexto anterior cuando **IRQ2** se impone. En esta situación un procesador como **Cortex-M3** puede abandonar las operaciones de apilamiento y la Tail-chain directamente a la excepción entrante, que en este ejemplo es la **Interrupt Service Routine** para **IRQ2**.



## Exceptions during state restore



### Armv7-A

Load Multiple uninterruptible, and hence the core must complete the POP and then full stack PUSH.

### Cortex®-M3 or M4

POP may be abandoned early if another interrupt arrives.  
If POP is interrupted it only takes 6 cycles to enter ISR2 (equivalent to tail-chaining).



## Escribir interrupt handlers

Los manipuladores de interrupción (Interrupt handlers) dirigidos a sistemas de perfil M pueden simplemente escribirse como las funciones estándar de C.

No requiere un código de entrada y salida de lenguaje ensamblador especial porque el hardware es capaz de guardar y restaurar el contexto.

La función **ISR (Interrupt Service Routines)** debe ser de tipo **void** y no puede aceptar argumentos y la palabra clave **\_\_irq** es opcional, para mayor claridad, como muestra este código:

```
__irq void SysTickHandler (void)
{
    num_sticks++;
}
```

Dicho esto, todavía es posible escribir controladores en lenguaje ensamblador.

**CMSIS-Core** proporciona controladores de excepciones ficticios predeterminados escritos en lenguaje ensamblador, que se repiten indefinidamente.

Es posible anular estas funciones con una función C con el mismo nombre, como se muestra aquí:

```
; CMSIS-Core Dummy Exception Handlers
; (infinite loops which can be modified)

SysTick_Handler      PROC
    EXPORT SysTick_Handler [WEAK]
    B .
ENDP
```

El compilador C o C ++ se encargará de guardar y restaurar cualquier registro que no esté almacenado en la pila automáticamente al ingresar una excepción. Alternativamente, también es posible que el programador modifique el controlador predeterminado, en cuyo caso deben asegurarse de guardar y restaurar manualmente cualquier registro que no esté almacenado en la pila al ingresar una excepción.

Cuando escribimos controladores en Assembly, debemos:

- Guardar y restaurar manualmente cualquier registro que no sea scratch, que se use:
  - **R4, R5, R6, R7, R8, R9, R10, R11, LR.**
- Mantenga la alineación de la pila de 8 bytes si realiza llamadas a funciones.
- Regrese de la interrupción usando **EXC\_RETURN** con las instrucciones adecuadas:
  - **PC LDR , ...**
  - **LDM / POP** que incluye cargar la **PC** .
  - **BX LR .**

## Excepciones internas (SysTick, SVCALL, PendSV)

### SysTick Timer

- Proporciona latidos del sistema para **RTOS** (o custom program executive).
- La interrupción periódica impulsa la programación de tareas del sistema.
- Mejor que una interrupción de temporizador estándar.
- Integrado en la **NVIC**.
- No varía entre la implementación del proveedor y las familias **Cortex-M**.

El temporizador **SysTick** es un temporizador de cuenta regresiva de **24 bits**, que se puede volver a cargar a un valor contenido en un Registro de recarga (**Reload Register**) cada vez que cuenta hasta cero, y opcionalmente generar una interrupción, que podría usarse como un latido del sistema para Un sistema operativo incorporado o en tiempo real. También tiene la ventaja de que es compatible con los sistemas **Armv6-M, Armv7-M y Armv8-M**, lo que facilita la transferencia de código de un sistema de perfil M a otro.

## Supervisor Call (SVCall)

- Similar a la instrucción "SVC" en otros núcleos Arm.
- Permite que el software no privilegiado realice llamadas al sistema.
- Solicitudes de servicios **RTOS** del modo no privilegiado.
- Proporciona protección a la funcionalidad importante del sistema.
- Ejemplos: abrir puerto serie, memoria de reasignación, ingresar al modo de bajo consumo, etc.

La excepción de **Supervisor Call** proporciona un mecanismo para que el software no privilegiado solicite una operación privilegiada, por ejemplo, el código de la aplicación podría usar una instrucción **SVC** para solicitar un servicio del sistema operativo subyacente, como abrir un puerto serie para que lo use. En arquitecturas anteriores, **SVC** se llamaba **SWI**, que significa **Software Interrupt**. La instrucción **SVC** permite que el software de la aplicación se desarrolle independientemente del software del sistema operativo. También es probable que el sistema operativo proporcione una API para el software de aplicación para los diferentes servicios, en lugar de exponer la operación **SVC** directamente.

## Pended System Call (PendSV)

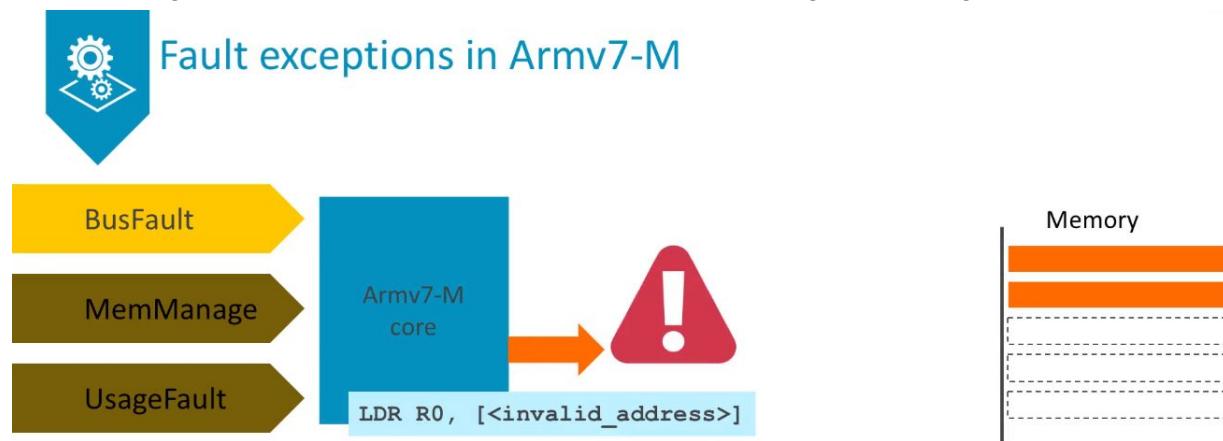
- Opera con **SVC** para facilitar el desarrollo de **RTOS**.
- Destinado a ser una interrupción para el uso de **RTOS**.

Finalmente, hay una excepción especial relacionada con el perfil M llamada **PendSV**, que puede ser habilitada por un software privilegiado a través del **Interrupt Control and State Register**, bit **PENDSVSET**. Una idea detrás de **PendSV** es proporcionar un sistema operativo con la capacidad de establecer una excepción de baja prioridad pendiente para proporcionar un mecanismo para ejecutar tareas de baja prioridad, como el cambio de contexto, solo una vez más las interrupciones críticas, y las tareas del sistema operativo han sido atendidas y completadas .

## Excepciones de fallas en Armv7-M

Hay 3 clases de fallas en ARMv7-M: **BusFault**, **MemManage** y **UsageFault**.

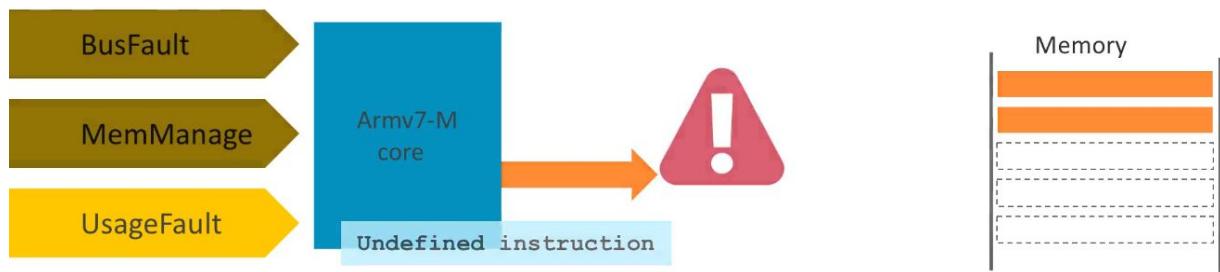
El **BusFault** puede ocurrir debido a un error de acceso a la memoria. Por ejemplo si el procesador intenta acceder a algunos datos desde una dirección que no está asignada a ninguna memoria.



**MemManage** Faults: ocurre debido a violaciones de permisos de la **MPU**. Por ejemplo si la **MPU** marca una dirección como de solo lectura y el software intenta escribir en esa dirección, se produce un error **MemManage**. La **MPU** detecta y atrapa los accesos antes de que se transfieran a la interfaz del bus.



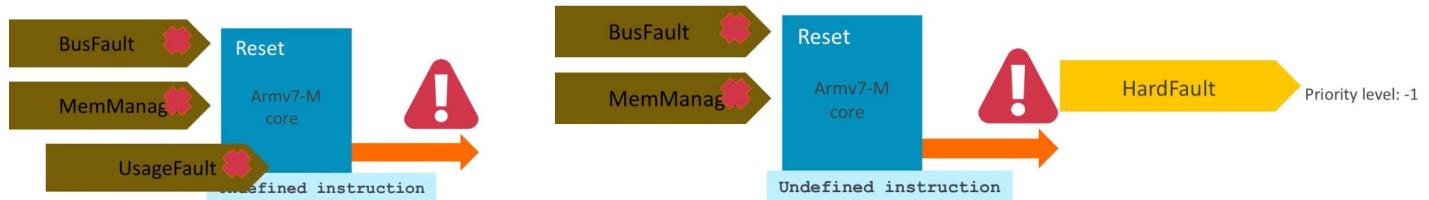
**UsageFault** puede ocurrir debido a una variedad de errores de estado de ejecución. Por ejemplo: intentar ejecutar una instrucción indefinida desencadenara un **UsageFault**



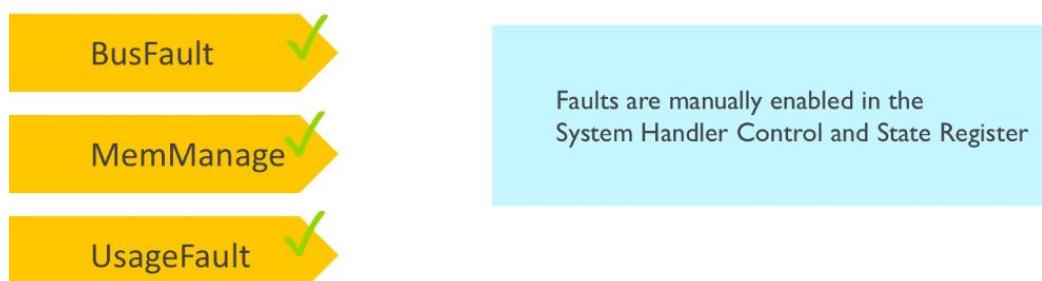
Si alguna de estas fallas ocurre cuando están deshabilitadas, el procesador los escala a **HardFault**, que siempre está habilitado con un nivel de prioridad de -1.



Por ejemplo: se decodifica una instrucción indefinida y el núcleo intenta señalar una **UsageFault**, pero la excepción **UsageFault** no está habilitada.

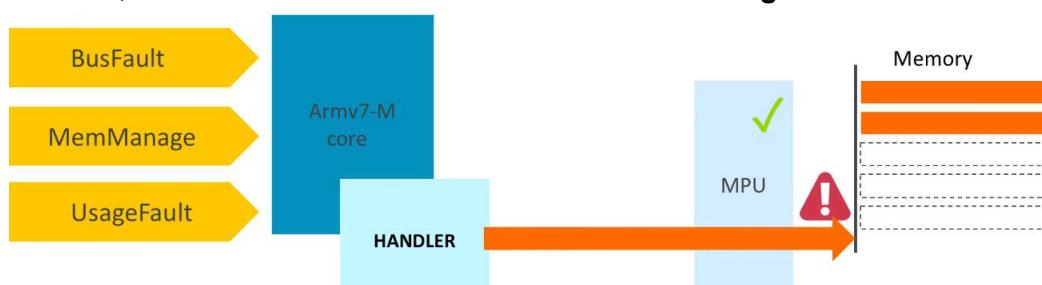


Si es necesario es posible habilitar las excepciones **BusFault**, **MemManaged** y **UsageFoult** a través del **System Handler Control** y **State Register**. Obviamente antes de hacer esto, es crucial que los manipuladores de excepción estén escritos para tratar este tipo de fallas.

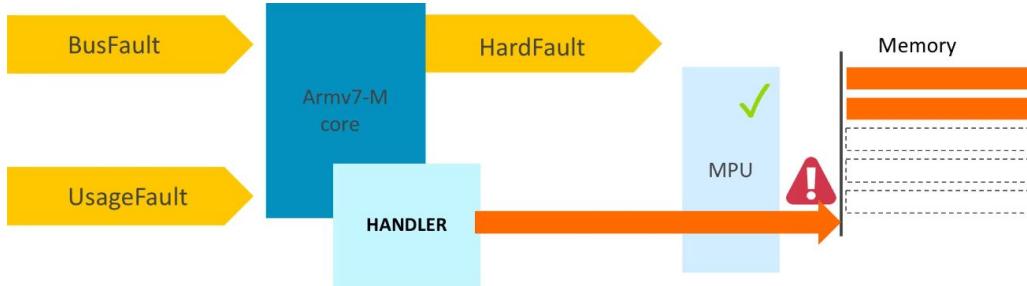


Escalamientos de fallas (Fault escalation) a una **HardFault** puede ocurrir si el controlador no tiene suficiente prioridad para ejecutarse o si el **Fault Handler** encuentra la misma falla.

Por ejemplo: Si la **MPU** está habilitada y un controlador de interrupciones realiza un acceso ilegal a la memoria, el core intentará señalar una falla de **MemManage**.



Pero si el nivel de prioridad de **MemManaged** es menor que la prioridad de la interrupción en ejecución, el error de **MemManaged** se escalará a una excepción **HardFault**



## Estado de bloqueo (Lockup state)

En casos de excepción irrecuperables, se ingresa al estado de bloqueo.

Puede experimentar situaciones desafortunadas en las que el procesador encuentra una excepción irrecuperable, por ejemplo, recibir un **HardFault** mientras se ejecuta en un nivel de prioridad de **-1** dentro de un controlador **HardFault**. En tales situaciones, el procesador entra en estado de bloqueo.

Estas fallas de la tabla de vectores incluyen:

- Fallo al leer el valor inicial de **MSP** o el vector **Reset**.
- Fallo en la lectura del vector **NMI**.
- Fallo en la lectura del vector **HardFault**.

En estado de bloqueo, el procesador obtiene repetidamente la misma instrucción de una dirección fija en el área **XN**.

Las implementaciones de Arm7-M como **Cortex-M3** obtienen una dirección de un área de memoria marcada como **eXecute Never (XN)**, por lo que el procesador no realiza ninguna ejecución de instrucción adicional. Esta búsqueda de instrucciones no es válida y no se ejecuta, y las implementaciones utilizan principalmente direcciones 0xFFFFFFFF o 0xFFFFFFF. Además, recomendamos que las implementaciones hagan valer una señal cuando se ingresa el estado de bloqueo. Esto permite que el sistema maneje el problema adecuadamente. Por ejemplo, el sistema podría conectar la señal a un perro guardián capaz de realizar un reinicio del sistema. Sin embargo, antes de reiniciar el sistema, todavía es posible depurar la falla deteniendo el procesador a través de una conexión de depuración.

El procesador puede bloquearse en 2 niveles de prioridad

El procesador puede bloquearse en el nivel de prioridad **-1** o **-2**, dependiendo de la causa del bloqueo. Por ejemplo, un error de lectura de vector en la entrada de **HardFault**, debido a que el procesador no puede leer el vector de **HardFault** produce un bloqueo en el nivel de prioridad **-1**, mientras que un error de lectura de vector en la entrada a un **NMI** da como resultado un bloqueo en el nivel de prioridad **-2**.

Cuando el procesador ingresa al estado de bloqueo en el nivel de prioridad **-1**, también es posible que la excepción **NMI** haga que el procesador salga del estado de bloqueo, en cuyo caso el controlador **NMI** podría intentar diagnosticar, informar e incluso rectificar la falla, si es posible.

El estado de bloqueo solo se puede salir en los siguientes casos:

- Excepción de **NMI** cuando la prioridad bloqueada es **-1**.
- Evento de depuración (extensiones de depuración).
- Reiniciar.

## Manejo de fallas (Fault handling)

Los sistemas simples generalmente solo usarían un controlador **HardFault**, mientras que los sistemas más complejos podrían habilitar otras fallas y proporcionar un controlador de fallas más específico.

Una ventaja de **Armv7-M** sobre **Armv6-M** es que hay algunos registros útiles para que los manipuladores de fallas lean para descubrir la causa de una falla. En primer lugar, hay algunos **Fault Address Registers** para fallas de **BusFault** y **MemManage**:

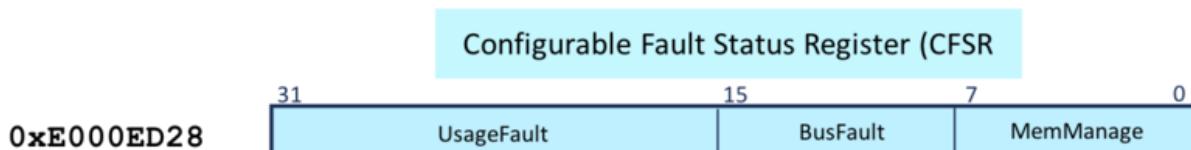
- **BusFault Address Register (BFAR)** muestra la dirección asociada con una falla de acceso a datos precisa.
- **MemManage Fault Address Register (MMFAR)** muestra la dirección de la ubicación de la memoria que causó la falla de la **MPU**.

Además, cada falla tiene un registro de estado, que proporciona más información sobre la falla.

**UsageFault Status Register** contiene el estado de algunas fallas de ejecución de instrucciones y de fallas de acceso a datos, por ejemplo, si **UsageFault** fue causado por un error **Divide by Zero**, un acceso no alineado o una instrucción indefinida. El **BusFault Status Register** contiene información de estado de los errores del bus resultantes de las captaciones previas de instrucciones y los accesos a los datos, por ejemplo, si un error de acceso a los datos fue preciso o impreciso, o si ocurrió una falla en el bus durante la entrada o devolución de la excepción.

La falla **MemManage** proporciona información de estado sobre fallas de **MPU**, por ejemplo, si la falla ocurrió debido a una violación de ejecución de instrucciones, como intentar ejecutar código desde un área de memoria marcada como **eXecute Never (XN)**.

Se puede acceder a cada uno de estos registros de estado de falla individualmente por software o colectivamente a través del **Configurable Fault Status Register(CFSR)**.



Finalmente, también hay un **HardFault Status Register** que indica si el **HardFault** se produjo debido a una escalada de prioridad o debido a un error al leer la tabla de vectores.

# Debug (Depurar)

## DesignStart - Descripción general de depuración

Armv7-M proporciona varias características de depuración para permitir a los desarrolladores depurar el código que se ejecuta en un sistema de perfil M. Para aquellos familiarizados con el portfolio de procesadores clásicos de Arm, hay muchas características de depuración tradicionales, por ejemplo, la **instrucción breakpoint**. Además, siempre ha habido tradicionalmente dos modos de depuración: uno que detiene el procesador y otro que no. Estos modos de depuración se conocen oficialmente como **halting debug** y **exception-based monitor debug**. **Halting debug** es donde el procesador es detenido por un agente de depuración externo como **DS-5 Debugger** o el depurador **uVision de Keil**. Por otro lado, **exception-based monitor debug** es donde todas las operaciones de depuración deben ser manejadas por un código personalizado a través de la excepción **DebugMonitor**. También es posible pasar por código una instrucción a la vez, con o sin interrupciones siendo capaz de interrumpir el flujo del programa. Otras características de depuración tradicionales incluyen la **vector catch** y la **Embedded Trace Macrocell**. También hay una serie de componentes de depuración **CoreSight** más nuevos, como la unidad **Flash Patch** y **Breakpoint**, **Data Watchpoint** y **Trace Unit** y la **Instrumentation Trace Macrocell**.

### Características tradicionales de depuración de ARM:

- Dos modos de depuración (**halting debug** y **monitor debug**).
- Dos modos de paso (con y sin interrupciones tomadas).
- Instrucción **BKPT**.
- Vector de captura(**Vector catch**).
- Embedded Trace Macrocell (**ETM**).

### Nuevas funciones de depuración

- Flash Patch and Breakpoint (**FPB**).
  - Instruction Breakpoints and Code Patching.
- Data Watchpoint and Trace (**DWT**).
  - Hardware Breakpoints, Event Counters and PC Sampling.
- Instrumentation Trace Macrocell (**ITM**).
  - Low bandwidth trace driven by application software or DWT.
  - Serial Wire Viewer.

## Estado de depuración de Armv7-M

ARM admite una variedad de mecanismos de depuración invasivos, como la capacidad de detener el procesador, Por ejemplo: En un **Breakpoint**. Además, aunque es menos intrusivo la excepción **DebugMonitor** se clasifica como un mecanismo de depuración invasivo.



## Debug state

### Arm Processor



If halting debug is enabled the processor enters Debug state.

When in Debug state the processor is stopped.

- No instructions are executed / No interrupts are serviced.

The processor is controlled through external debug interface.

La detención de la depuración se controla mediante una señal **DBGEN** que se refleja en este campo de **Debug Halting Control y Status Register**. El procesador ejecuta código en estado **Thumb**. Si se produce un evento de depuración (**halting debug**) cuando la detención de la depuración está habilitada y la excepción **DebugMonitor** está deshabilitada, el procesador se detendrá y entrará en estado de depuración.

Un evento de depuración puede ser interno o externo. Un evento de depuración interno es activado por una función de depuración dentro del procesador, por ejemplo el procesador, encuentra una instrucción **Breakpoint**. Los eventos de depuración también pueden ocurrir debido a una señal externa, por ejemplo, si hay 2 procesadores ARM en el mismo sistema, cada procesador puede ser capaz de señalar un evento de depuración al otro procesador. Si se produce un evento de depuración y tanto la detención de la depuración (**halting debug**) como la excepción **DebugMonitor** están deshabilitadas, se producirá una excepción **HardFault**.



## Debug state

### Arm Processor

External  
Debug  
Event

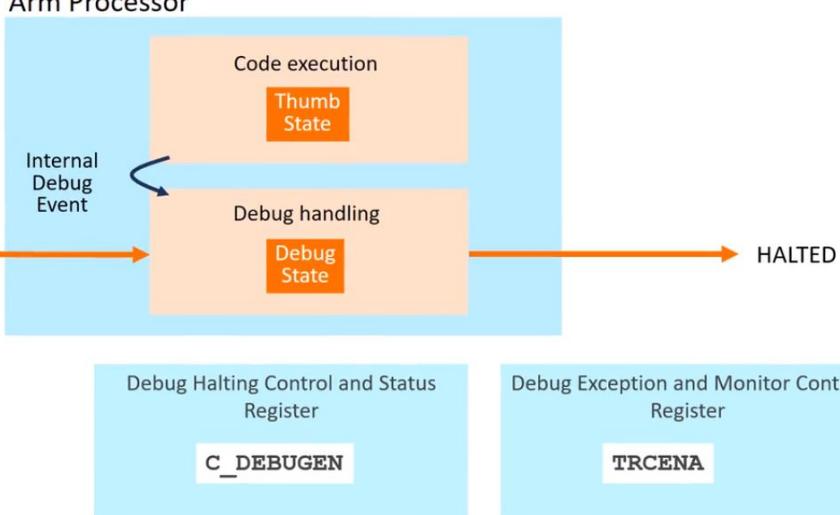
Internal  
Debug  
Event

If halting debug is enabled the processor enters Debug state.

When in Debug state the processor is stopped.

- No instructions are executed / No interrupts are serviced.

The processor is controlled through external debug interface.



Debug Halting Control and Status Register

`C_DEBUGEN`

Debug Exception and Monitor Control Register

`TRCENA`

ARMv7-M también admite técnicas de depuración no invasivas a través de funciones de rastreo y creación de perfiles en **DWT**, **ITM**, **ETM**. La depuración no invasiva se controla mediante una señal **NIDEN** que se refleja en este campo del **Monitor Control Register** y **Debug Exception**. Los mecanismos de depuración no invasivos no son intrusivos para la ejecución del programa y proveen una forma de observar el comportamiento del procesador sin alterar el estado del procesador.

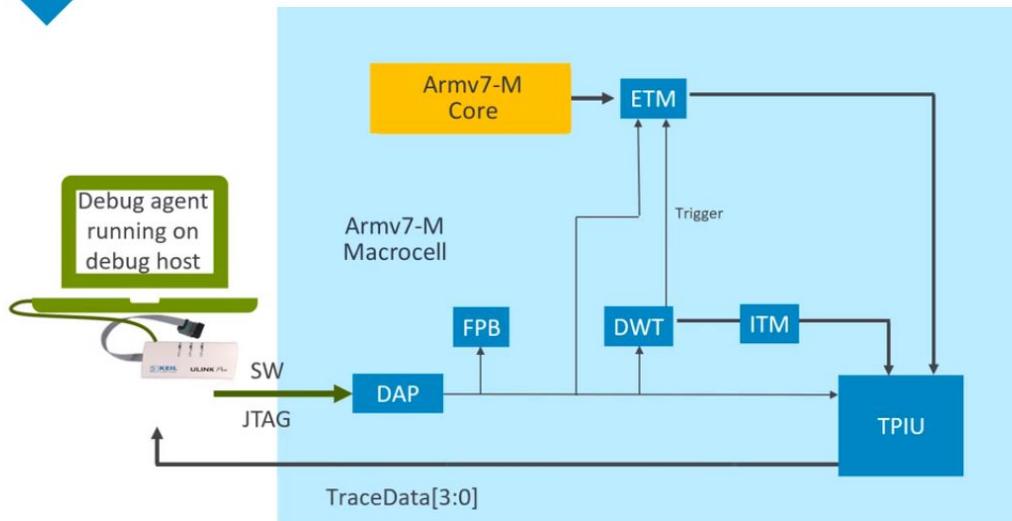
## Componentes de depuración Cortex-M

La arquitectura ARM describe varios componentes más profundos que pueden estar presentes en una implementación que el agente de depuración se conecta al sistema a través de un puerto de acceso de depuración utilizando un cable serie o el protocolo **JTAG**.

**TPIU:** Trace Port Interface Unit



## Cortex-v7M debug components



## Breakpoints and watchpoints

**Breakpoints** y **Watchpoint** son 2 características de depuración que pueden hacer que el procesador se detenga e ingrese al estado de depuración. Alternativamente, el procesador se puede configurar para generar una excepción **DebugMonitor**.



### Breakpoints and watchpoints



**Breakpoints** están relacionados con la ejecución del código. El desarrollador puede desear detener el procesador y depurar el sistema justo antes de que un código en particular esté a punto de ejecutarse como se puede ver a continuación.

Los **Breakpoints** pueden establecerse como **Disassembly Level** y como **Source Level** cuando la aplicación contiene información de depuración. Si el procesador está configurado para detenerse como resultado de un evento de depuración y encuentra un **Breakpoint**, se detendrá justo antes de que la instrucción ubicada en la dirección que se está comparando esté a punto de ejecutarse.

**Watchpoints**, por otro lado, están relacionados con los accesos a la memoria. Los **Watchpoints** permiten al desarrollador detener el procesador cuando se accede a una dirección de memoria en particular, por ejemplo, a través de una operación de carga o almacenamiento. En este ejemplo el desarrollador ha establecido un **Watchpoint** para activar cuando se lee 1 byte de esta dirección.

## Breakpoints de software y hardware

Hay 2 tipos de **Breakpoints**: **Software Breakpoints** y **Hardware Breakpoints**.

**Software Breakpoints**: Puede estar seteado por el agente de depuración (**debug agent**) en cualquier ubicación de la RAM. Cuando un desarrollador establece un **Breakpoint** en **RAM**, el **debug agent** simplemente puede reemplazar el código de operación (**opcode**) original con una instrucción **BKPT**. Si el

desarrollador alguna vez decide eliminar el **Breakpoint**, el agente de depuración simplemente restaurará el código de operación (opcode) original.

**Hardware Breakpoints** involucran comparadores en **FlashPatch** y **Breakpoint Unit**, que permite que al agente de depuración establecer **Breakpoints** en la **ROM**. El número de **Hardware Breakpoints** y **Watchpoints** disponible en un sistema es la implementación definida. La implementación de **ARM Cortex-M3 Design Start** proporciona 6 comparadores de **Breakpoints** y 4 comparadores de **Watchpoints**.



## Breakpoints and watchpoints

Software breakpoints and hardware breakpoints

Software breakpoints

- Set by debug agent on any location in RAM.
- Replaces opcode with a BKPT instruction.
- Opcode restored when BKPT removed.

Hardware breakpoints

- Involve comparators in the FlashPatch and Breakpoint Unit to allow the debug agent to set breakpoints in ROM.
- Number of breakpoints and watchpoints is implementation defined.
- The Arm Cortex-M3 DesignStart implementation provides:
  - Six Breakpoints.
  - Three Watchpoints.

## Armv7-M vector-catch (captura de vector)

Es común querer detener el procesador e ingresar al estado de depuración cuando se producen excepciones, una forma de lograr esto sería establecer **Breakpoints** en los manipuladores de interrupciones (**interrupt handler**) relacionados con excepciones que nos interesa depurar. El problema con esta solución es que los manipuladores de interrupciones (**interrupt handler**) generalmente se almacenan en **ROM** y solo han un número limitado de recursos de **Hardware Breakpoints** disponibles. En lugar de utilizar valiosos recursos de **Breakpoint**, es posible una característica llamada **Vector catch**.

**Vector catch** es una característica tradicional de depuración de ARM que se puede utilizar para atrapar ciertas excepciones. En ARMv7-M el **Debug Exception and Monitor Control Register (DEMCR)** contiene un número de bits de **Vector catch** que permiten a un depurador seleccionar qué excepción atrapar.

Hay campos en el **DEMCR** para atrapar **HardFaults**, **MemManaged**, **BusFault**, **UsageFaults** y **Reset**, y también errores en la entrada o retorno de excepciones.

Una limitación de ARMv7-M es que no es utilizar el mecanismo de **vector catch** para atrapar interrupciones externas. En cambio los desarrolladores tienen que establecer **Breakpoints** en los controladores de interrupción que están interesados en la depuración.



## Vector catch



Address	Vector
0x00	Initial Main SP
0x04	Reset
0x08	NMI
0x0C	HardFault
0x10	MemManage
0x14	BusFault
0x18	UsageFault
0x1C - 0x28	Reserved
0x2C	SVCall
0x30	Debug Monitor
0x34	Reserved
0x38	PendSV
0x3C	SysTick
0x40	IRQ0
....	More IRQs

## Armv7-M semihosting

**Semihosting** es un mecanismo para que los objetivos ARM de su dispositivo comuniquen la solicitud de entrada / salida del código de aplicación a una computadora host que ejecuta un depurador.

Este mecanismo podría usarse, por ejemplo, para permitir funciones en la biblioteca C, como **printf()** y **scanf()**, para usar la pantalla y el teclado del host en lugar de la pantalla y el teclado en el sistema de destino (target system). El mecanismo del **semihosting** no solo depende del soporte en el **debug tools**, pero también se basa en el soporte en las herramientas de compilación. Las herramientas de compilación que admiten el **semihosting** contienen un tipo especial de instrucciones para ciertas funciones de biblioteca como **printf()**. La instrucción especial **Breakpoint (BKPT)** 0xAB se utiliza como **semihosting call** para sistemas de perfil M. Cuando un depurador no admite **semihosting** y detecta uno de estas instrucciones, tratará la instrucción como una instrucción estándar de **Breakpoint** que hará que el procesador se detenga y entre en estado de depuración. Por lo tanto tales depuradores pueden proporcionar otro mecanismo similar E/S simuladas, como **Instrumentation Trace Macrocell** descrito en otro módulo.



## Semihosting

Application Code

Library Code

```
printf("Hello world\n");
```

Library Code



Communication with  
debugger running on host

## Características de perfil de Data Watchpoint and Trace (DWT)

La **Data Watchpoint and Trace Unit (DWT)** no solo es capaz de detener el procesador a través de **data watchpoints**, sino que también proporciona una variedad de características de seguimiento. Tiene cinco registros de perfiles para contar diferentes eventos, como cuántos ciclos ha estado durmiendo el procesador y cuántos ciclos adicionales se han perdido debido al apilamiento de excepciones y la prevención. El **DWT** también tiene un contador de ciclo de ejecución libre de 32 bits que incrementa cada **core cycle**, que es una herramienta realmente útil para comparar las aplicaciones que se ejecutan en sistemas de perfil M. El **DWT** está vinculado tanto con el **ITM**. Cuando un contador se desborda, es posible configurar el **DWT** para emitir un paquete de rastreo a través de la **Instrumentation Trace Macrocell**.

## DWT's event counters (8-bit):

- **Exception Overhead Counter:** número de ciclos perdidos para apilamiento y prevención.
- **Sleep Counter:** número de ciclos de sueño.
- **Fold Counter:** número de instrucciones plegadas.
  - Incrementos en cada ciclo donde se ejecuta más de una instrucción.
  - Cortex-M7 dual issue significa que este contador puede incrementarse con frecuencia.
- **Load / Store Counter:** número de ciclos generales para carga y almacenamiento.
- **CPI Counter:** número de ciclos de instrucción más allá del primero (contador general).
  - Tallies Interlock and Fetch instruction overhead.
  - Excluye Load / Store overhead (ya contados en el Load / Store Counter).

## The Instrumentation Trace Macrocell (ITM)

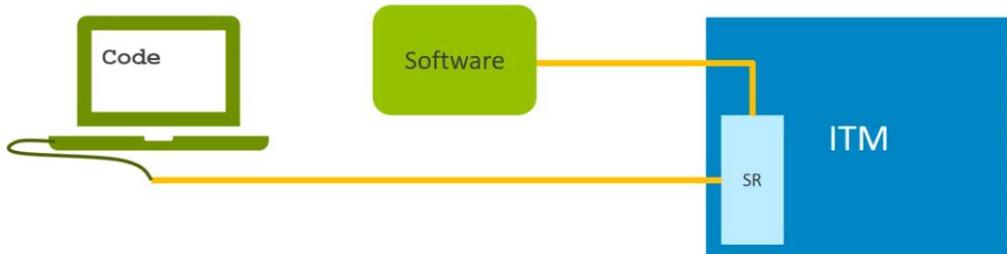
**Instrumentation Trace Macrocell (ITM)** contiene un número de stimulus register de 32 bits. los depuradores suelen proporcionar una interfaz para configurar si el software con o sin privilegios tiene permiso para acceder al registro de estímulo. El software puede escribir en los registros de estímulo, lo que resulta en paquetes de rastreo emitidos a través de un pin de salida de cable en serie del dispositivo.

Los depuradores pueden proporcionar un mecanismo llamado **Serial Wire Viewer** (o **Visor ITM**) para mostrar los datos a quien está depurando el dispositivo. Esencialmente, el **ITM** proporciona un mecanismo para “**printf debugging**”, donde se dirige el printf a la interfaz **ITM** en lugar del comportamiento predeterminado de la biblioteca C.

El **ITM** también es capaz de generar **local and Global timestamp information**. **Global timestamp information** se puede utilizar para sincronizar el seguimiento proveniente de múltiples procesadores en el mismo dispositivo.



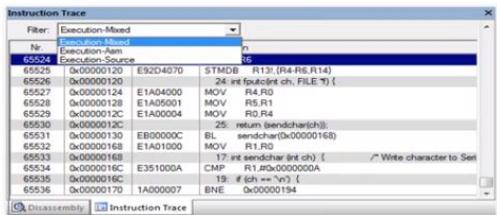
Instrumentation Trace Macrocell (ITM)



## Embedded Trace Macrocell

**Embedded Trace Macrocell (ETM)** admite instrucciones y rastreo de datos (**instruction trace and data trace**), aunque la mayoría de los procesadores de perfil M como **Cortex-M3** solo admiten el seguimiento de instrucciones a través del **ETM**.

Los procesadores **Cortex-M7** mas potentes y ricos admiten la opción de **data trace** también. El rastreo de **ETM** no es intrusivo para la ejecución del programa, por lo que es una excelente solución para el rastreo de aplicaciones en tiempo real. Con el **ETM** es posible rastrear todas las instrucciones que entran en la etapa de ejecución del pipeline y registrar con precisión qué instrucciones se ejecutaron. Si el código fuente original contiene información de depuración, algunos depuradores pueden mostrar una fuente intercalada y una vista de ensamblaje que muestra las instrucciones ejecutadas entrelazadas con las líneas individuales del código C relacionado. **ETM** es útil para que los depuradores la utilicen para generar información de cobertura de código para establecer qué parte del tiempo del programa se pasó ejecutando instrucciones específicas.



M3 M7  
✓ ✓ Instruction trace  
✓ Data trace



## Resumen (depuración Armv7-M)

Durante este módulo, describimos qué características de depuración de Armv7-M están disponibles para permitir a los desarrolladores depurar el código que se ejecuta en un sistema de perfil M. Aunque el hardware de depuración es extremadamente útil, tiene un costo en relación con el área de silicio, por lo que la depuración es opcional para permitir a los diseñadores de sistemas decidir qué características de depuración son necesarias para que sus desarrolladores trabajen eficientemente con su dispositivo. Por lo tanto, algunos sistemas de perfil M pueden incluir todas las características, algunos sistemas pueden incluir un número limitado de características y algunos sistemas pueden no incluir ninguna función de depuración. A veces, los diseñadores de sistemas integran funciones de depuración en una placa de desarrollo que contiene su unidad de microcontrolador, pero los desarrolladores de sistemas pueden terminar eliminando las funciones de depuración en sus productos finales, dependiendo de si desean depurar los dispositivos en el campo.

# Desarrollo de software

## Desarrollo de software y herramientas de depuración

Arm proporciona una gama de herramientas para desarrollar y depurar software integrado, como se muestra en la tabla a continuación.

### Desarrollo de software

Keil MDK incluye herramientas de compilación Arm, uVision IDE, Realtime Trace, Realtime Library. Se requiere una licencia para el despliegue legal. Se proporciona una licencia Keil de tres meses con DesignStart Eval. Vale la pena señalar que las versiones de evaluación de Keil Tools se pueden usar para la producción.

DS-5 incluye GCC (herramientas de compilación Arm con la versión Professional), depurador DS-5, modelos, entorno de desarrollo IDE basado en Eclipse Workbench, generador de perfiles opcional, etc.



Software Tools	Compilation Tools	Debug Adapters	Development Targets
• Keil MDK	<ul style="list-style-type: none"><li>• Arm Compiler 5 (armcc)</li><li>• Arm Compiler 6 (armclang)</li><li>• GNU Compiler (gcc)</li></ul>	<ul style="list-style-type: none"><li>• ULINKpro</li><li>• ULINKproD</li><li>• ULINK2</li><li>• ULINK-M</li></ul>	<ul style="list-style-type: none"><li>• MCU development boards</li><li>• μVision Simulator</li><li>• CMSIS-DAP enabled targets</li></ul>
• DS-5 Development Suite	<ul style="list-style-type: none"><li>• Arm Compiler 5 (armcc)</li><li>• Arm Compiler 6 (armclang)</li><li>• GNU Compiler (gcc)</li></ul>	<ul style="list-style-type: none"><li>• ULINKpro</li><li>• ULINKproD</li><li>• ULINK2</li><li>• DSTREAM-ST</li></ul>	<ul style="list-style-type: none"><li>• MCU development boards</li><li>• Fast Models</li><li>• Simulation Environment</li><li>• CMSIS-DAP enabled targets</li></ul>

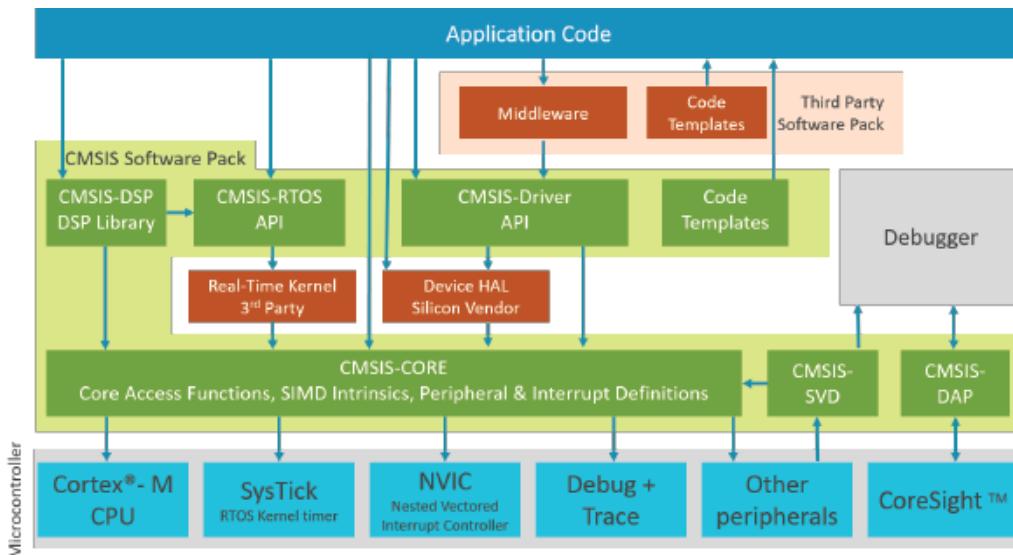
### Adaptadores de depuración

Puede conectar uVision Debugger / DS-5 Debugger a objetivos de hardware utilizando conectores JTAG o Trace. CMSIS-DAP sobre USB está disponible en CM3DS. Para obtener instrucciones para conectarse mediante CMSIS-DAP en Keil, consulte la guía de inicio rápido DesignStart Cortex-M3 FPGA. CMSIS-DAP también está disponible en DS-5.

## The Cortex Microcontroller Software Interface Standard (CMSIS)

El **Cortex Microcontroller Software Interface Standard (CMSIS)** está diseñado para permitir un soporte de dispositivo consistente e interfaces de software simples para el procesador y sus periféricos. **CMSIS** comenzó con **CMSIS-Core**, que es una capa de abstracción de hardware independiente del proveedor para procesadores **Cortex-M**. Desde entonces, **CMSIS** se ha expandido a áreas como la gestión de componentes de software y las interfaces de depuración de referencia.

**CMSIS** se define en estrecha cooperación con varios proveedores de software y silicio y proporciona un enfoque común para la interfaz con periféricos, sistemas operativos en tiempo real y componentes de middleware. Simplifica la reutilización del software, reduciendo la curva de aprendizaje para los nuevos desarrolladores de microcontroladores y reduciendo el tiempo de comercialización de los dispositivos.



En resumen:

**CMSIS-Core (Cortex-M)** es una API para el núcleo y los periféricos del procesador Cortex-M. Proporciona una interfaz estandarizada para Cortex-M0, Cortex-M0 +, Cortex-M3, Cortex-M4, Cortex-M7, Cortex-M23, Cortex-M33, SC000 y SC300. También se incluyen funciones intrínsecas SIMD para las instrucciones SIMD Cortex-M4, Cortex-M7 y Cortex-M33. [https://arm-software.github.io/CMSIS\\_5/Core/html/index.html](https://arm-software.github.io/CMSIS_5/Core/html/index.html)

**CMSIS-Core (Cortex-A)** es una API y un sistema básico de tiempo de ejecución para el núcleo y los periféricos del procesador Cortex-A5 / A7 / A9.

[https://arm-software.github.io/CMSIS\\_5/Core\\_A/html/index.html](https://arm-software.github.io/CMSIS_5/Core_A/html/index.html)

**CMSIS-Driver** define interfaces genéricas de controladores periféricos para middleware, lo que lo hace reutilizable en dispositivos compatibles. La API es independiente de RTOS y conecta periféricos de microcontroladores con middleware que implementa, por ejemplo, pilas de comunicación, sistemas de archivos o interfaces gráficas de usuario. [https://arm-software.github.io/CMSIS\\_5/Driver/html/index.html](https://arm-software.github.io/CMSIS_5/Driver/html/index.html)

**CMSIS-DSP** es una colección de biblioteca DSP con más de 60 funciones para varios tipos de datos: **punto fijo** (fraccional q7, q15, q31) y **punto flotante de precisión simple** (32 bits). La biblioteca está disponible para todos los núcleos Cortex-M. Las implementaciones que están optimizadas para el conjunto de instrucciones SIMD están disponibles para Cortex-M4, Cortex-M7 y Cortex-M33.

[https://arm-software.github.io/CMSIS\\_5/DSP/html/index.html](https://arm-software.github.io/CMSIS_5/DSP/html/index.html)

**CMSIS-RTOS v1** es una API común para sistemas operativos en tiempo real junto con una implementación de referencia basada en RTX. Proporciona una interfaz de programación estandarizada que es portátil para muchos RTOS y permite componentes de software que pueden funcionar en múltiples sistemas RTOS.

[https://arm-software.github.io/CMSIS\\_5/RTOS/html/index.html](https://arm-software.github.io/CMSIS_5/RTOS/html/index.html)

**CMSIS-RTOS v2** extiende CMSIS-RTOS v1 con soporte para arquitectura Armv8-M, creación dinámica de objetos, provisiones para sistemas multi-core e interfaz binaria compatible en compiladores compatibles con ABI. [https://arm-software.github.io/CMSIS\\_5/RTOS2/html/index.html](https://arm-software.github.io/CMSIS_5/RTOS2/html/index.html)

**CMSIS-Pack** describe con un archivo de descripción de paquete basado en XML (PDSC) las partes relevantes del usuario y del dispositivo de una colección de archivos (llamado paquete de software) que incluye archivos fuente, encabezado y biblioteca, documentación, algoritmos de programación Flash, plantillas de código fuente, y proyectos de ejemplo. Las herramientas de desarrollo y las infraestructuras web utilizan el archivo PDSC para extraer los parámetros del dispositivo, los componentes de software y las configuraciones de la placa de evaluación. [https://arm-software.github.io/CMSIS\\_5/Pack/html/index.html](https://arm-software.github.io/CMSIS_5/Pack/html/index.html)

**CMSIS-SVD** - System View Description para periféricos. Describe los periféricos de un dispositivo en un archivo **XML** y se puede utilizar para crear conciencia periférica en depuradores o archivos de encabezado con registro periférico y definiciones de interrupción.

[https://arm-software.github.io/CMSIS\\_5/SVD/html/index.html](https://arm-software.github.io/CMSIS_5/SVD/html/index.html)

**CMSIS-DAP** : Debug Access Port. Firmware estandarizado para una unidad de depuración que se conecta al puerto de acceso de depuración **CoreSight**. **CMSIS-DAP** se distribuye como un paquete separado y es muy adecuado para la integración en placas de evaluación. Este componente se proporciona como descarga por separado. [https://arm-software.github.io/CMSIS\\_5/DAP/html/index.html](https://arm-software.github.io/CMSIS_5/DAP/html/index.html)

**CMSIS-Zone** es una definición de recursos del sistema y particionamiento. Define métodos para describir los recursos del sistema y para dividir estos recursos en múltiples proyectos y áreas de ejecución.

[https://arm-software.github.io/CMSIS\\_5/Zone/html/index.html](https://arm-software.github.io/CMSIS_5/Zone/html/index.html)

## Paquetes de software

Los paquetes de software brindan soporte para dispositivos de microcontroladores y placas de desarrollo. Los paquetes de software también pueden contener componentes de software, como **drivers** y **middleware**, incluidos proyectos de ejemplo y plantillas de código.

Hay cinco tipos de respaldos de software disponibles. Cada uno se origina en una fuente diferente y está diseñado para diferentes tipos de uso:

Type	Device Family Pack	CMSIS Pack	Middleware Pack	Board Support Pack	In-house Software Pack
Source	Silicon Vendor, Tool Vendor	Arm	Silicon Vendor, Tool Vendor, 3rd Party	Board Vendor	Tool User
Use Case	Deploy support for new MCU families	Standard delivery of CMSIS components	Simplify integration of pre-build middleware	Support of evaluation boards with interfaces and example projects	Supply and update software components within a company

Puede encontrar una explicación sobre los paquetes de software y los enlaces a la documentación del paquete de software en esta publicación de blog .

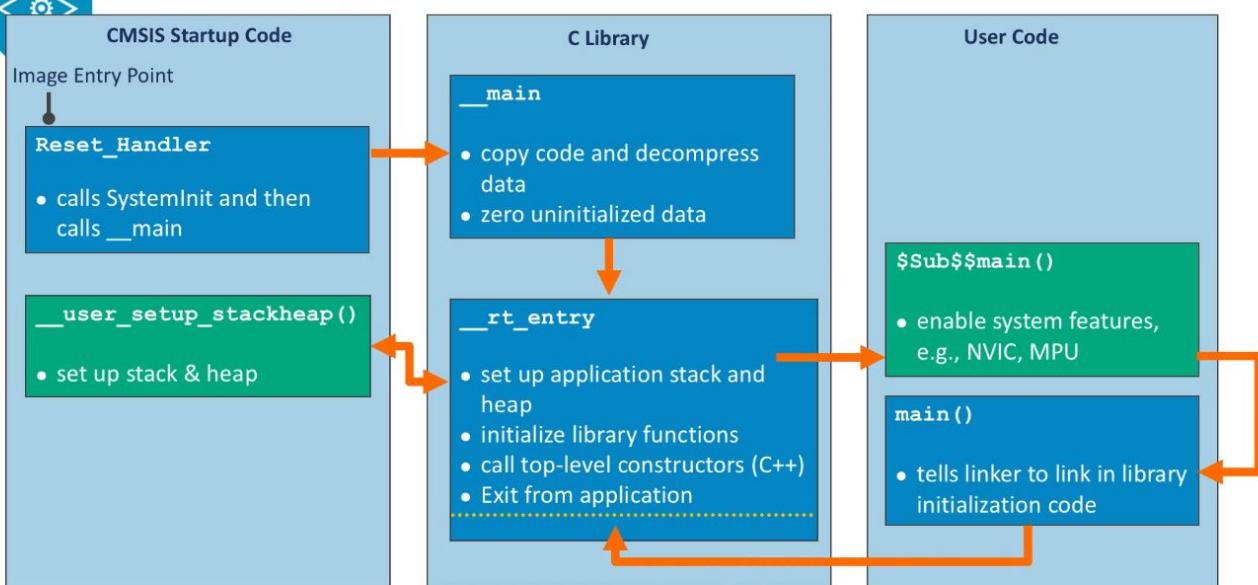
<https://community.arm.com/developer/tools-software/tools/b/tools-software-ides-blog/posts/mdk-arm-version-5-software-packs-explained>

# Proceso de desarrollo embebido: secuencia de inicialización

## Initialization sequence

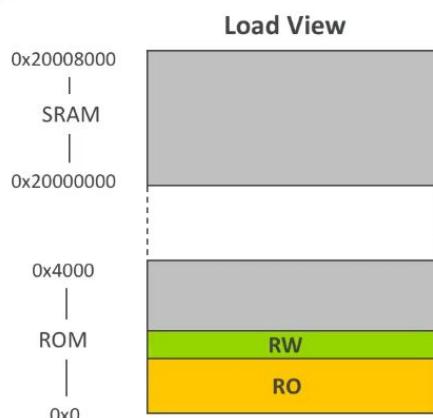


## Initialization sequence



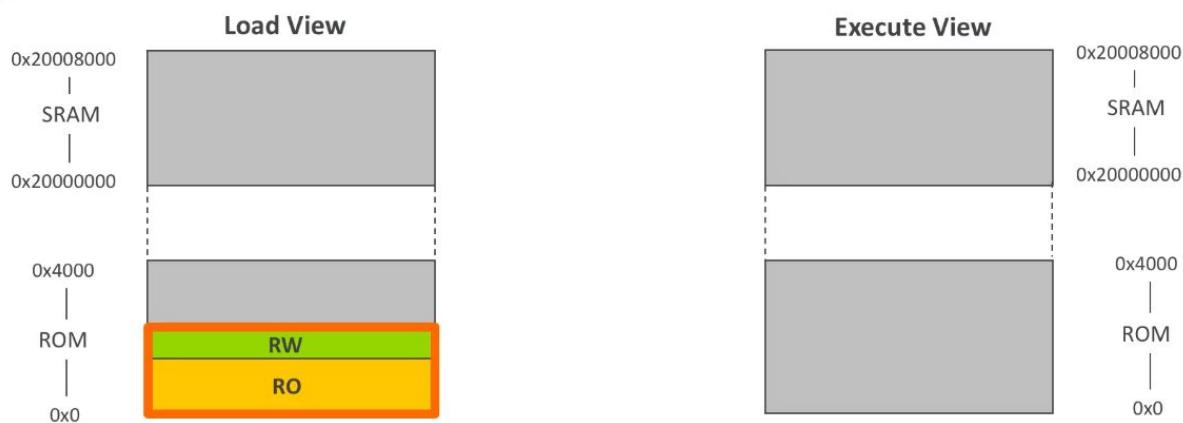
## Scatter-loading (Carga de dispersión)

## Scatter-loading - simple example

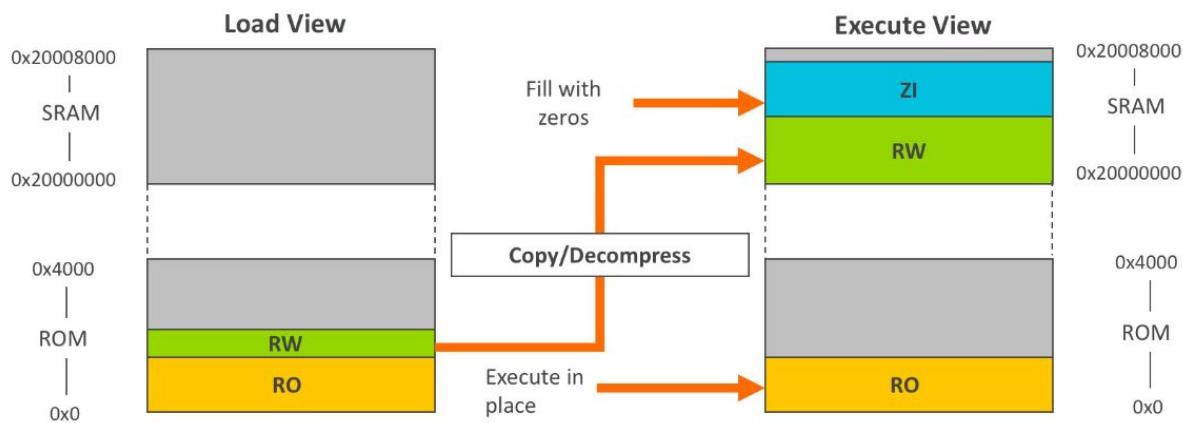




## Scatter-loading - simple example



## Scatter-loading - simple example



## Cortex-M3 DesignStart memory map

```

LR_IROM1 0x00000000 0x00400000 { ; load region size_region
    ER_IROM1 0x00000000 0x00400000 { ; load address = execution
        address
        *.o (RESET, +First)
        *(InRoot$$Sections)
        .ANY (+RO)
    }
    RW_IRAM1 0x20000000 0x00400000 { ; RW data
        .ANY (+RW +ZI)
    }
}

```

0xFFFF_FFFF	System (AHB and PPB expansion)
0xE000_0000	Device (AHB expansion)
0xC000_0000	Device (AHB expansion)
0xA001_0000	Radio interface (AHB expansion)
0xA000_0000	RAM (AHB expansion)
0x8000_0000	RAM (AHB expansion)
0x6000_0000	Peripheral (AHB expansion)
0x4001_0000	Peripheral (APB expansion)
0x4000_0000	SRAM (AHB expansion)
0x2000_0000	Code (AHB expansion)
0x0000_0000	

Scatter file from Blinky example in Arm V2M-MPS2 Board Support Pack for DesignStart Devices

```

ER_IROM1 0x00000000 0x00400000 { ; load address = execution
address
    *.o (RESET, +First)
    *(InRoot$$Sections)
    .ANY (+RO)
}
RW_IRAM1 0x20000000 0x00400000 { ; RW data
    .ANY (+RW +ZI)
}
}

```

Scatter file from Blinky example in Arm V2M-MPS2  
Board Support Pack for DesignStart Devices

```

*.o (RESET, +First)
*(InRoot$$Sections)
.ANY (+RO)
}
RW_IRAM1 0x20000000 0x00400000 { ; RW data
    .ANY (+RW +ZI)
}
}

```

Scatter file from Blinky example in Arm V2M-MPS2  
Board Support Pack for DesignStart Devices

```

*(InRoot$$Sections)
.ANY (+RO)
}
RW_IRAM1 0x20000000 0x00400000 { ; RW data
    .ANY (+RW +ZI)
}
}

```

Scatter file from Blinky example in Arm V2M-MPS2  
Board Support Pack for DesignStart Devices

```

.ANY (+RO)
}
RW_IRAM1 0x20000000 0x00400000 { ; RW data
    .ANY (+RW +ZI)
}
}

```

Scatter file from Blinky example in Arm V2M-MPS2  
Board Support Pack for DesignStart Devices

```

RW_IRAM1 0x20000000 0x00400000 { ; RW data
    .ANY (+RW +ZI)
}
}

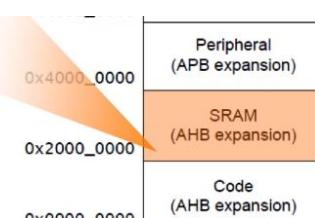
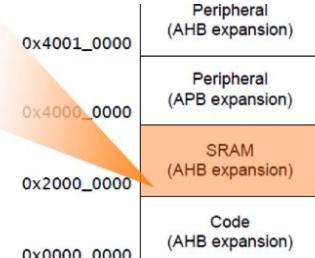
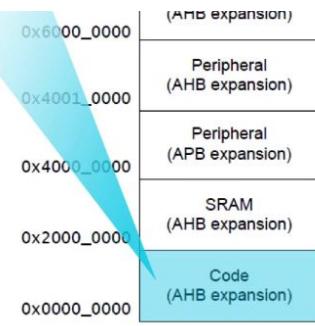
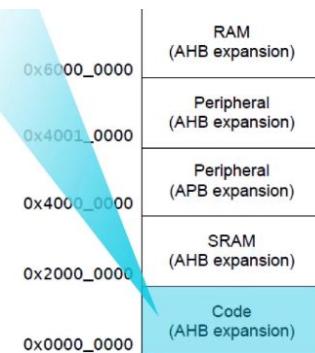
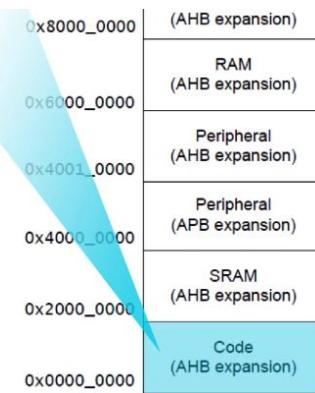
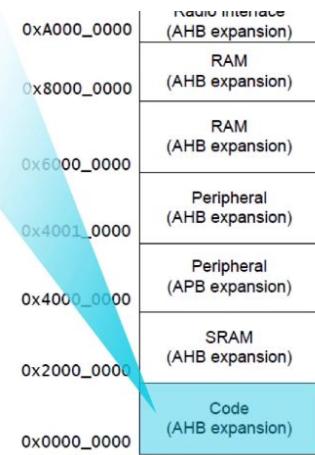
```

Scatter file from Blinky example in Arm V2M-MPS2  
Board Support Pack for DesignStart Devices

```

.ANY (+RW +ZI)
}
}
```

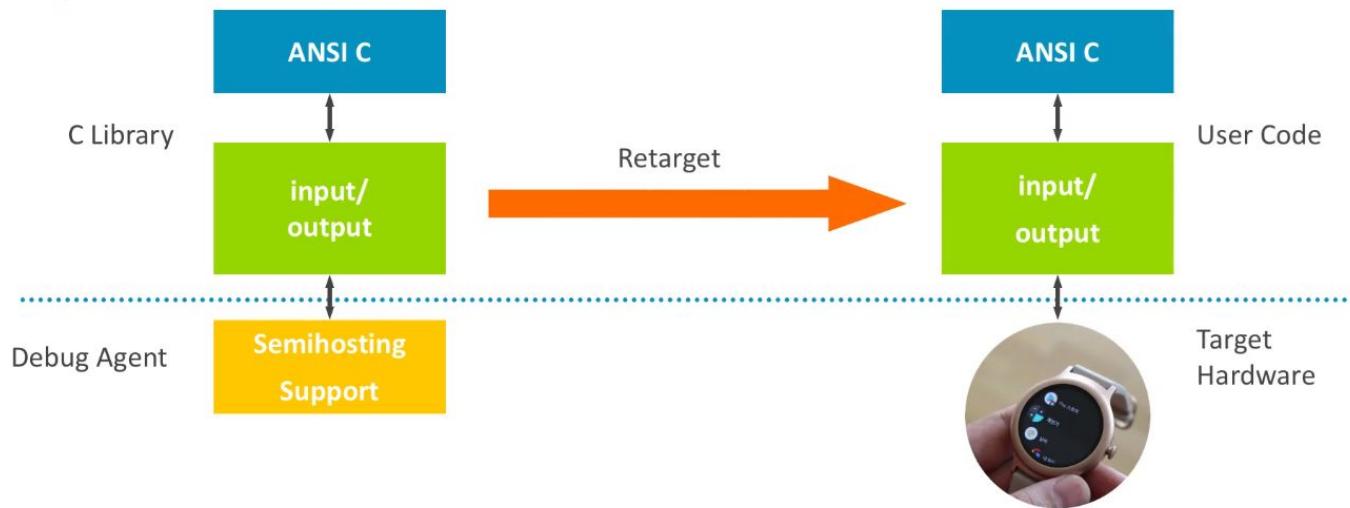
Scatter file from Blinky example in Arm V2M-MPS2  
Board Support Pack for DesignStart Devices



## Reorientación de la biblioteca C



### Retargeting the C library



Arm Cortex-M3 DesignStart Eval: Run a simple C test on RTL Simulation

Enlaces útiles - DesignStart Armv7-M

Enlaces

[Accediendo a DesignStart](#)

Recursos

[MPS2 + información y descargas](#)

[Uso de Cortex-M3 DesignStart y MPS2 + como destino mbed](#)

Paquetes de soporte de la placa CMSIS para MPS2:

- [http://www.keil.com/dd2/arm/ds\\_cm3/](http://www.keil.com/dd2/arm/ds_cm3/)
- [http://www.keil.com/dd2/arm/cmsdk\\_cm0/](http://www.keil.com/dd2/arm/cmsdk_cm0/)

[Comunidad: para blogs, debates y soporte a nivel de foro](#)

Documentación

[Documentación de perfil M](#)

[Documentación de Cortex-M3](#)

[Manual de referencia técnica de Cortex-M3](#)

[Manual de referencia de arquitectura Armv7-M](#)

[Documentación de Arm Compiler 6](#)

[Documentación de Arm Compiler 5](#)

[Documentación de Keil MDK](#)

[Documentación de DS-5 Development Studio](#)

## Libro

La guía definitiva para los procesadores Arm Cortex-M3 y Cortex-M4, por Joseph Yiu - ISBN 978-0-12-408082-9

### Certificado



Matias Vironi

Has passed

Arm DesignStart – Introduction to Armv7-M

Scoring:

80% (120 points)

on 7/7/2020

**arm**  
Training