Electrónica Digital II

Santiago Rúa Pérez, PhD.

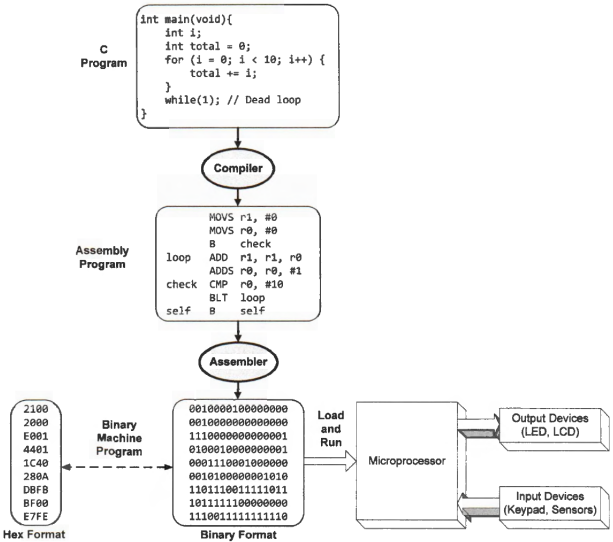
18 de septiembre de 2022

INTRODUCCIÓN A LOS MICROCONTROLADORES

Objetivos

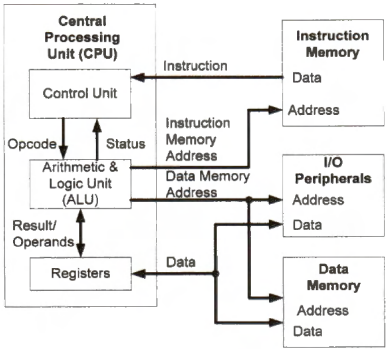
- Entender la estructura de funcionamiento de un microcontrolador.
- Entender como se traduce un programa de C a ejecución en el microcontrolador.
- Entender la diferencia en Hardvard y Von Neumann.

Lenguaje C a lenguaje de máquina

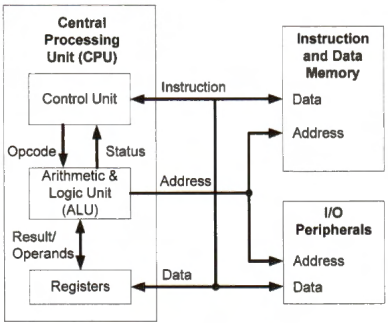


Hardware vs Von Neumann

Hardvard

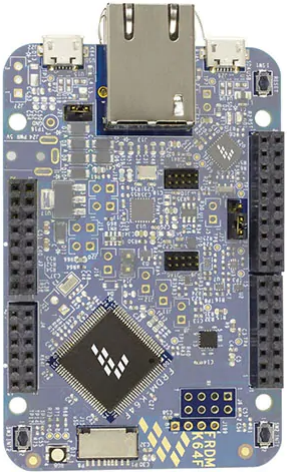


Von Neumann



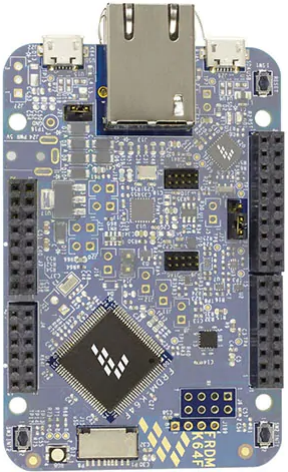
Tarjeta a trabajar

- Procesador Cortex M4 de 32-bit
- Freescale Kinetis MK64FN1M0VLL12
  - 120 Mhz max clock, 1 MB flash, 256 kB RAM
  - Bajo consumo.
  - Cantidad grande de periféricos.
- Costo 47USD (Digikey)
- Perifericos externos: led RGB, acelerometro y magnetómetro (FXOS8700CQ), dos pulsadores, SD, Ethernet
- Compatible con los pines de arduino.
- mbed.org habilitado

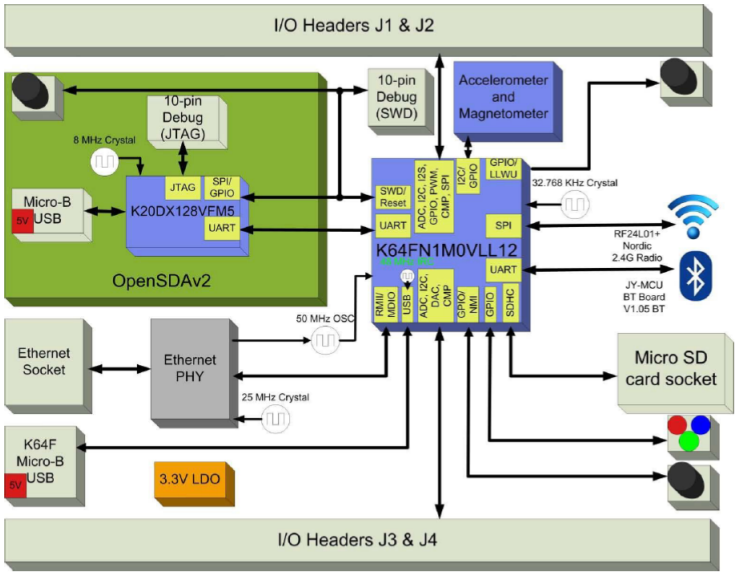


Tarjeta a trabajar

- Procesador Cortex M4 de 32-bit
- Freescale Kinetis MK64FN1M0VLL12
  - 120 Mhz max clock, 1 MB flash, 256 kB RAM
  - Bajo consumo.
  - Cantidad grande de periféricos.
- Costo 47USD (Digikey)
- Perifericos externos: led RGB, acelerometro y magnetómetro (FXOS8700CQ), dos pulsadores, SD, Ethernet
- Compatible con los pines de arduino.
- mbed.org habilitado



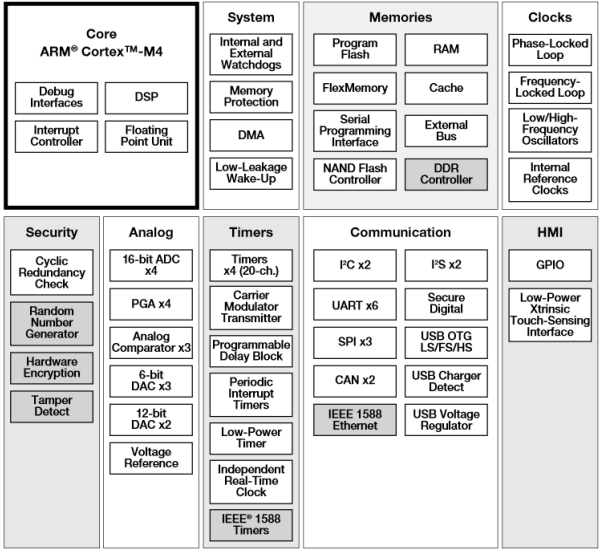
Tarjeta a trabajar



Documentos Importantes

- Kinetis K64: 120MHz Cortex-M4F up to 1MB Flash 100-144pin . Necesario para revisar todos los registros y capacidades del procesador.
- FRDMK64FUG, FRDM-K64F Freedom Module User's Guide. Necesario para revisar las capacidades y conexiones de la tarjeta FRDMK64F.
- FRDMK64 Schematic. Necesario para revisar las conexiones físicas del sistemas.
- Get Started. Guia de inicio para probar la tarjeta de desarrollo. Hacerlo y validarlo inmediatamente.

Kinetis K60


Core
ARM® Cortex™-M4
Debug Interfaces
DSP
Interrupt Controller
Floating Point Unit
System
Internal and External Watchdogs
Memory Protection
DMA
Low-Leakage Wake-Up
Memories
Program Flash
RAM
FlexMemory
Cache
Serial Programming Interface
External Bus
NAND Flash Controller
DDR Controller
Clocks
Phase-Locked Loop
Frequency-Locked Loop
Low/High-Frequency Oscillators
Internal Reference Clocks
Security
Cyclic Redundancy Check
Random Number Generator
Hardware Encryption
Tamper Detect
Analog
16-bit ADC x4
PGA x4
Analog Comparator x3
6-bit DAC x3
12-bit DAC x2
Voltage Reference
Timers
Timers x4 (20-ch.)
Carrier Modulator Transmitter
Programmable Delay Block
Periodic Interrupt Timers
Low-Power Timer
Independent Real-Time Clock
IEEE® 1588 Timers
Communication
I²C x2
I²S x2
UART x6
Secure Digital
SPI x3
USB OTG LS/FS/HS
CAN x2
USB Charger Detect
IEEE 1588 Ethernet
USB Voltage Regulator
HMI
GPIO
Low-Power Xtrinsic Touch-Sensing Interface

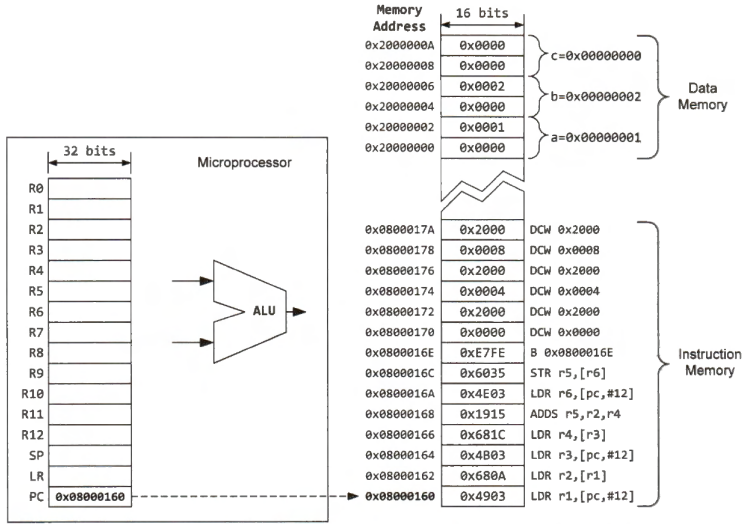
Comparing ARM

Arm Core	Cortex M0[2]	Cortex M0+[3]	Cortex M1[4]	Cortex M3[5]	Cortex M4[6]	Cortex M7[7]	Cortex M23[8]	Cortex M33[12]	Cortex M35P	Cortex M55
ARM architecture	ARMv6-M[9]	ARMv6-M[9]	ARMv6-M[9]	ARMv7-M[10]	ARMv7E-M[10]	ARMv7E-M[10]	ARMv8-M Baseline[15]	ARMv8-M Mainline[15]	ARMv8-M Mainline[15]	Armv8.1-M
Computer architecture	Von Neumann	Von Neumann	Von Neumann	Harvard	Harvard	Harvard	Von Neumann	Harvard	Harvard	Harvard
Instruction pipeline	3 stages	2 stages	3 stages	3 stages	3 stages	6 stages	2 stages	3 stages	3 stages	4 to 5 stages
Thumb-1 instructions	Most	Most	Most	Entire	Entire	Entire	Most	Entire	Entire	Entire
Thumb-2 instructions	Some	Some	Some	Entire	Entire	Entire	Some	Entire	Entire	Entire
Multiply instructions 32x32 = 32-bit result	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Multiply instructions 32x32 = 64-bit result	No	No	No	Yes	Yes	Yes	No	Yes	Yes	Yes
Divide instructions 32/32 = 32-bit quotient	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Saturated instructions	No	No	No	Some	Yes	Yes	No	Yes	Yes	Yes
DSP instructions	No	No	No	No	Yes	Yes	No	Optional	Optional	Optional
Single-Precision (SP) Floating-point instructions	No	No	No	No	Optional	Optional	No	Optional	Optional	Optional
Double-Precision (DP) Floating-point instructions	No	No	No	No	No	Optional	No	No	No	Optional

Ejecutando programa en arquitectura Cortex M

C Program	Assembly Program		Machine Program	
			Binary	Hex
		AREA myData, DATA		
		ALIGN		
int a = 1;	a	DCD 1	0000000000000000	0x0000
			0000000000000001	0x0001
int b = 2;	b	DCD 2	0000000000000000	0x0000
			0000000000000010	0x0002
int c = 0;	c	DCD 0	0000000000000000	0x0000
			0000000000000000	0x0000
		AREA myCode, CODE		
		EXPORT __main		
		ALIGN		
		ENTRY		
int main(){	__main	PROC		
c = a + b;		LDR r1, =a	0100100100000011	0x4903
while(1);		LDR r2, [r1]	0110100000001010	0x680A
}		LDR r3, =b	0100100110000011	0x4B03
		LDR r4, [r3]	0110100000011100	0x681C
		ADDS r5, r2, r4	0001100100011001	0x1915
		LDR r6, =c	0100111000000011	0x4E03
		STR r5, [r6]	0110000000111001	0x6035
	stop	B stop	1110011111111110	0xE7FE
		ENDP	0000000000000000	0x0000
		END	0010000000000000	0x2000
			0000000000000100	0x0004
			0010000000000000	0x2000
			0000000000001000	0x0008
			0010000000000000	0x2000

Arquitectura Cortex M-4



Arquitectura Cortex M-4

Dirección – Instrucción de ensamblador – Resultado.

- 0x08000160 — LDR r1, [pc, #12] — r1 = 0x2000000.
  Carga la direccion de memoria de la variable global a en el registro r1.
- 0x08000162 — LDR r2, [r1] — r2 = 0x00000001.
  Carga el valor de la variable global a al registro r2.
- 0x08000164 — LDR r3, [pc, #12] — r3 = 0x20000004.
  Carga la direccion de memoria de la variable global b en el registro r3
- 0x08000166 — LDR r4, [r3] — r4 = 0x00000002.
  Carga el valor de la variable global b al registro r4.
- 0x08000168 — ADDS r5, r2, r4 — r5 = 0x00000003.
  Suma el valor de a y b

Arquitectura Cortex M-4

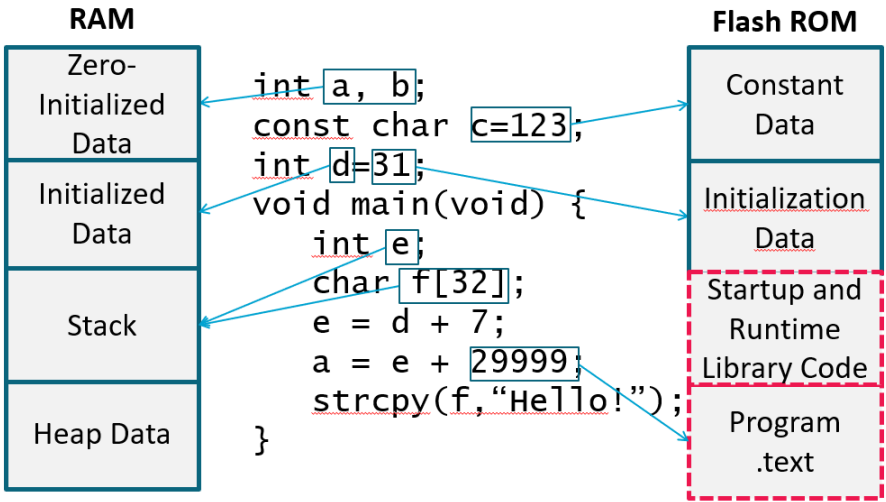
Dirección – Instrucción de ensamblador – Resultado.

- 0x0800016A — LDR r6, [pc, #12] — r6 = 0x20000008.
  Carga la direccion de memoria de la variable global c en el registro r6.
- 0x0800016C — STR r5, [r6] — mem[0x20000008] = 0x00000003.
  Guarda la suma en la memoria de datos
- 0x0800016E — B 0x0800016E — PC = 0x0800016E.
  A punta a la direccion de memoria de sis mismo

Requerimientos de memoria

- Que se requiere almacenar en la memoria?
  - Código
  - Datos estáticos de solo lectura
  - Datos estáticos de escritura
  - Heap y el Stack
- Puede cambiar la información almacenada?
  - No? Ponerla en memoria no volatil, solo lectura, usar const. Ej: instrucciones, contantes, valores iniciales.
  - Si? Ponerla en memoria de escritura y lectura. Ej: variables, resultados intermedios, direcciones de retorno.
- Cuanto tiempo que existir el dato?
  - Reutilizar memoria desde que se pueda.

Requerimientos de memoria



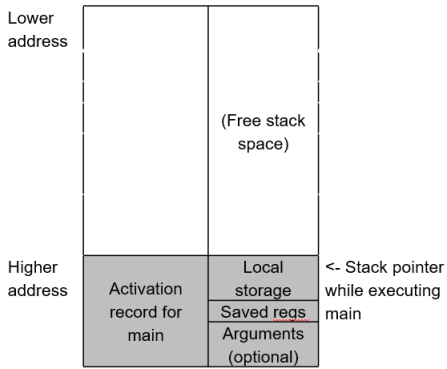
Calificadores para las variables

- Modificar la declaración de las variables posibilita hacer entender al compilador el comportamiento que queremos.
- Contantes: nunca escritas o modificadas por el programa, almacenandolas en ROM y no RAM.
- Volatil: pueden cambiar por fuera del scope del programa, por ejemplo una interrupcion, entre otros. El compilador no debe realizar optimizaciones sobre este tipo de variables.
- Estaticas: declaradas o usadas dentro de las funciones, para retener valores de llamado a las mismas. Solo tienen alcance dentro de la función.

Pila de llamado a funciones

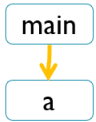
```
int main(void) {
    auto vars
    a();
}

void a(void) {
    auto vars
    b();
}

void b(void) {
    auto vars
    c();
}

void c(void) {
    auto vars
    …
}
```

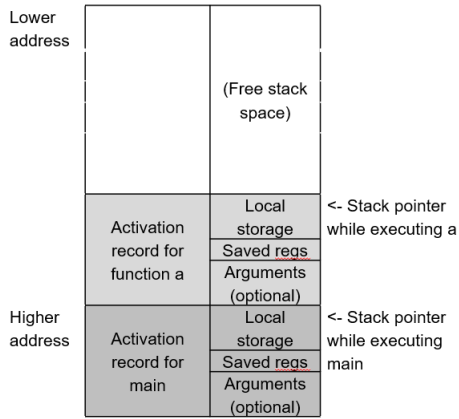
main



Pila de llamado a funciones

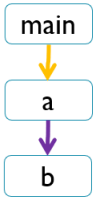
```
int main(void) {
    auto vars
    a();
}

void a(void) {
    auto vars
    b();
}

void b(void) {
    auto vars
    c();
}

void c(void) {
    auto vars
    …
}
```

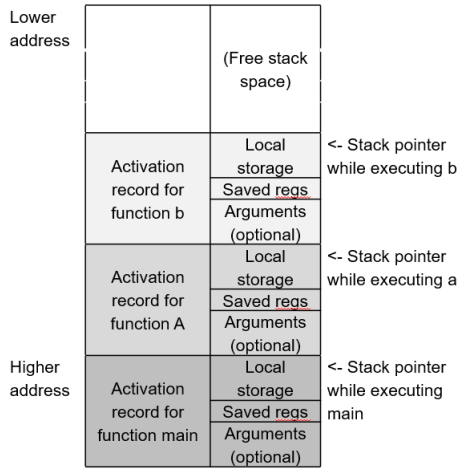




Pila de llamado a funciones

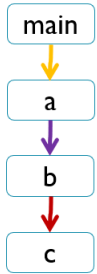
```
int main(void) {
    auto vars
    a();
}

void a(void) {
    auto vars
    b();
}

void b(void) {
    auto vars
    c();
}

void c(void) {
    auto vars
    …
}
```

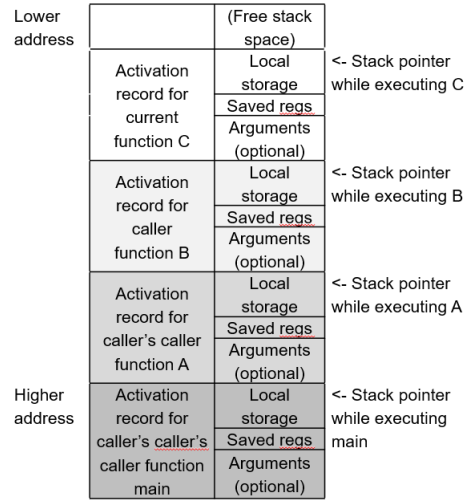




Pila de llamado a funciones

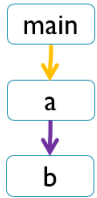
```
int main(void) {
    auto vars
    a();
}

void a(void) {
    auto vars
    b();
}

void b(void) {
    auto vars
    c();
}

void c(void) {
    auto vars
    …
}
```

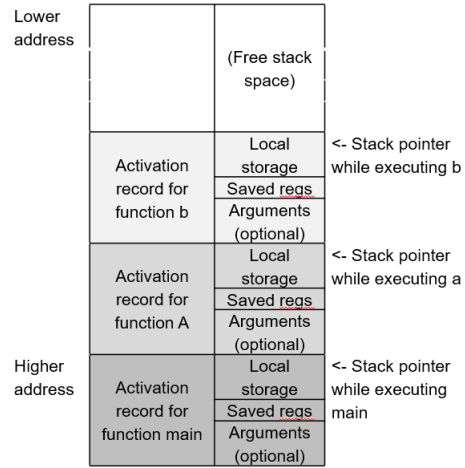




Pila de llamado a funciones

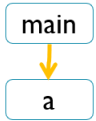
```
int main(void) {
    auto vars
    a();
}

void a(void) {
    auto vars
    b();
}

void b(void) {
    auto vars
    c();
}

void c(void) {
    auto vars
    …
}
```

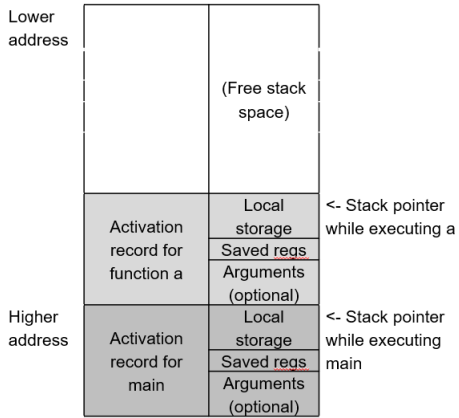




Pila de llamado a funciones

```
int main(void) {
    auto vars
    a();
}

void a(void) {
    auto vars
    b();
}

void b(void) {
    auto vars
    c();
}

void c(void) {
    auto vars
    …
}
```

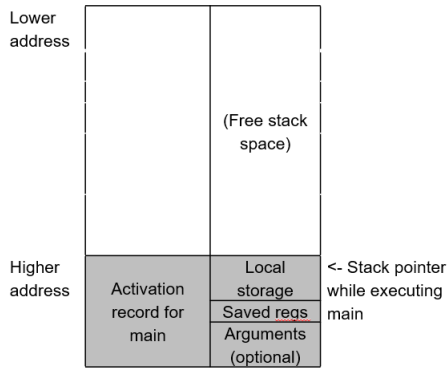




Pila de llamado a funciones

```
int main(void) {
    auto vars
    a();
}

void a(void) {
    auto vars
    b();
}

void b(void) {
    auto vars
    c();
}

void c(void) {
    auto vars
    …
}
```

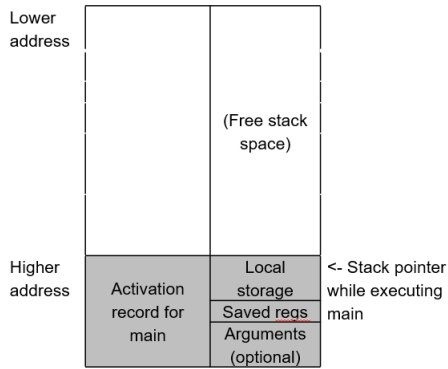
main



Pila de llamado a funciones

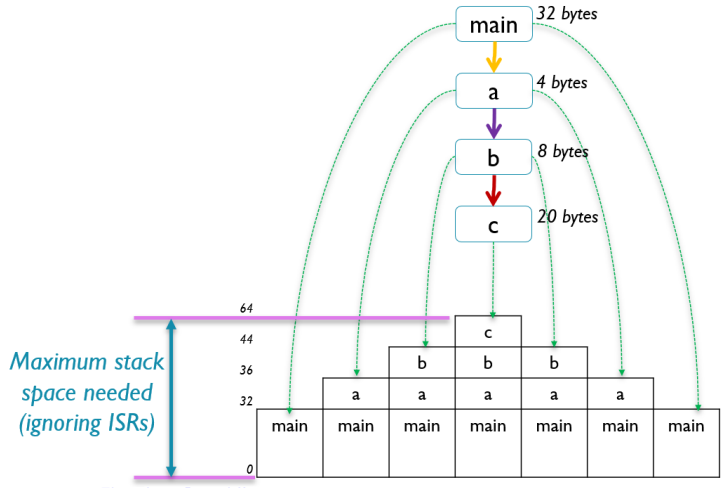
```
int main(void) {
    auto vars
    a();
}

void a(void) {
    auto vars
    b();
}

void b(void) {
    auto vars
    c();
}

void c(void) {
    auto vars
    …
}
```

main

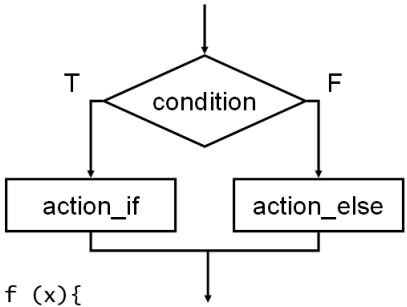


Pila de llamado a funciones

- El stack incrementa con el llamado a subrutinas. Este desaparece una vez se ejecuten.
- La profundidad determina el tamaño del stack
- Peligro con recursividad.



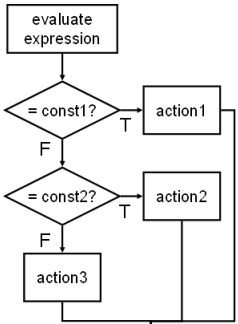
Flujo de control - If-else




```
if (x){
    y++;
} else {
    y--;
}
```

```
;;;39       if (x){
000056  2900 CMP  r1,#0
000058  d001 BEQ  |L1.94|
;;;40         y++;
00005a  1c52 ADDS r2,r2,#1
00005c  e000 B    |L1.96|

   |L1.94|
;;;41       } else {
;;;42         y--;
00005e  1e52 SUBS r2,r2,#1

   |L1.96|
;;;43       }
```

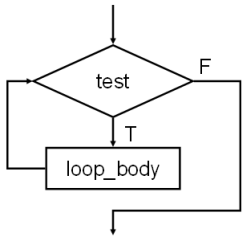
Flujo de control - Switch




```
;;;45        switch (x) {
000060  2901    CMP   r1,#1
000062  d002    BEQ   |L1.106|
000064  291f    CMP   r1,#0x1f
000066  d104    BNE   |L1.114|
000068  e001    B     |L1.110|
    |L1.106|
```

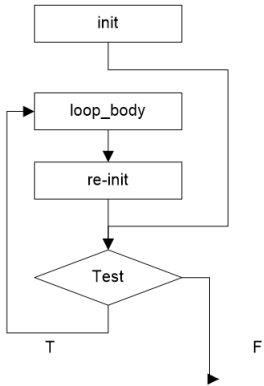
```
;;;46        case 1:
;;;47          y += 3;
00006a  1cd2    ADDS  r2,r2,#3
;;;48          break;
00006c  e003    B     |L1.118|
|L1.110|
;;;49        case 31:
;;;50          y -= 5;
00006e  1f52    SUBS  r2,r2,#5
;;;51          break;
000070  e001    B     |L1.118|
|L1.114|
;;;52        default:
;;;53          y--;
000072  1e52    SUBS  r2,r2,#1
;;;54          break;
000074  bf00    NOP
|L1.118|
000076  bf00    NOP
;;;55        }
```

Flujo de control - While




```
;;;57        while (x<10) {
000078  e000 B      |L1.124|
                |L1.122|
;;;58          x = x + 1;
00007a  1c49 ADDS r1,r1,#1
                |L1.124|
00007c  290a CMP  r1,#0xa
;57
00007e  d3fc BCC  |L1.122|
;;;59        }
```

Flujo de control - For

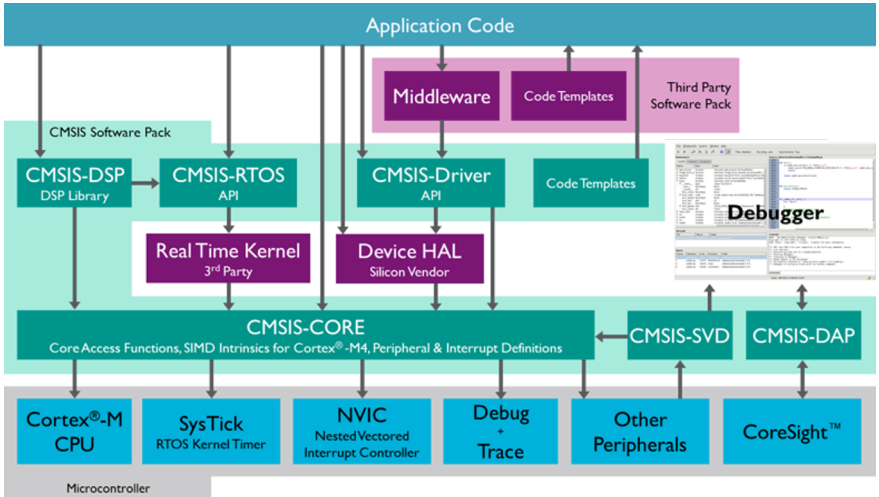



```
;;;61       for (i = 0; i <
10; i++){
000080  2300 MOVS  r3,#0
000082  e001 B     |L1.136|

                |L1.132|
;;;62          x += i;
000084  18c9 ADDS  r1,r1,r3
000086  1c5b ADDS  r3,r3,#1
;61

                |L1.136|
000088  2b0a CMP   r3,#0xa
;61
00008a  d3fb BCC   |L1.132|
;;;63       }
```

CMSIS: Cortex Microcontroller Software Interface Standard

- Que es CMSIS?
  - Capa de abstracción de hardware: capa de software entre el programa de aplicación y el hardware.
  - Convenciones y estándares para la interfaz de software, estructuras y nombres.
- Problemas - hay muchas opciones
  - Algunas arquitecturas Cortex-M ofrecen mayores instrucciones que otras.
  - Cuando se está trabajando con Cortex-M4, el compilador tiene instrucciones adicionales como por ejemplo operaciones de punto flotante.
  - Muchos proveedores de MCU crean MCU basados en Cortex-M. Los procesadores son consistentes pero los perifericos cambian y su acceso a ellos de la misma forma.
  - Muchos compiladores disponibles. Diferentes características de optimización.
  - Mucho RTOS disponibles.
- Beneficios
  - Recorta el tiempo de aprendizaje en nuevo hardware.
  - Simplifica el desarrollo de software.
  - Simplifica el reuso de software.

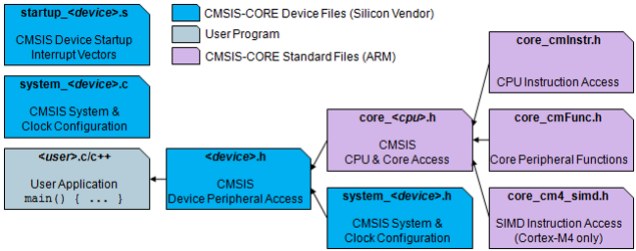
Estructura de CMSIS



Convenciones de CMSIS

- General
  - Cumple con estandar ANSI C y C++
  - Usa ANSI C <stdint.h> para tipos de datos.
  - Esta estructurado bajo el estándar MISRA 2004.
- Convenciones para los nombres
  - Nombres en mayúsculas para los registros principales, registros de periféricos, instrucciones de CPU.
  - Minúsculas para las interrupciones y el nombre de funciones.
  - Prefijo de espacio entre nombre '_' para grupos.
- Documentación utilizando Doxygen.
- Licenciamiento
  - Gratis.
  - Puede ser utilizado para todos los dispositivos Cortex-M.

CMSIS-CORE



- Los archivos de dispositivo CMSIS-CORE (azul) implementan la funcionalidad específica del dispositivo
- La aplicación de usuario debe incluir <device.h> para usar las funciones CMSIS-CORE.
  - Acceso a periféricos.
  - Excepciones e interrupciones.
  - Entre otras.
- Características adicionales
  - API estándar para periféricos y núcleo del procesador Cortex-M
  - Organización de archivos de encabezado y convenciones.
  - Esqueletos de funciones de inicialización del sistema, implementados por proveedores de MCU.

Registros Ejemplo

- Archivo header MK64F12.h.
  - Este archivo define tipos de datos estructuras para acceder a los registros del sistema.

```
/** GPIO - Register Layout Typedef */
typedef struct {
__IO uint32_t PDOR; /**< Port Data Output Register, offset: 0x0 */
__O  uint32_t PSOR; /**< Port Set Output Register, offset: 0x4 */
__O  uint32_t PCOR; /**< Port Clear Output Register, offset: 0x8 */
__O  uint32_t PTOR; /**< Port Toggle Output Register, offset: 0xC */
__I  uint32_t PDIR; /**< Port Data Input Register, offset: 0x10 */
__IO uint32_t PDDR; /**< Port Data Direction Register, offset: 0x14 */
} GPIO_Type;
```

Registros Ejemplo

- Archivo header MK64F12.h.
  - Declara puntero a los registros.

```
/* GPIO - Peripheral instance base addresses */
/** Peripheral PTA base address */
#define PTA_BASE    (0x400FF000u)
/** Peripheral PTA base pointer */
#define PTA    ((GPIO_Type *)PTA_BASE)

PTA->PDOR = my_data;
```

INTRODUCCIÓN A LOS MICROCONTROLADORES

GRACIAS