Sistemas Electrónicos Digitales Avanzados

Repaso de conocimientos de cursos anteriores

Programación en C

Tipos de datos en C

- Tipos básicos de datos
  - **int**: define números enteros
  - **char**: define caracteres ASCII
  - **float**: define números en punto flotante de precisión normal
  - **double**: define números en punto flotante en doble precisión
- Calificadores
  - **short int** o **short**, **long int** o **long**
  - **unsigned**
- Definición de constantes
  - Un número como **1234** se toma como **int**
  - Un número mayor que el que cabe en un int se toma como **long**
  - **1234L** también es un **long**. **1234UL** es un **unsigned long**.
  - **0xFF** en un **int** de valor 255. **0xFFL** es un **long** de valor 255.
  - **'0'** corresponde con el valor **ASCII** del carácter '0', es decir: **48**
  - '\n' corresponde con el avance de línea (10) y '\r' es retorno de carro (13)
  - **'\0'** corresponde con un **0**.

Tipos de datos en C

- Pero .... ¿Cuánto bytes ocupa cada tipo de datos?
  - Depende del procesador (CPU) y del compilador.
  - En Keil ARM

Type	Size in bits	Natural alignment in bytes
char	8	1 (byte-aligned)
short	16	2 (halfword-aligned)
int	32	4 (word-aligned)
long	32	4 (word-aligned)
long long	64	8 (doubleword-aligned)
float	32	4 (word-aligned)
double	64	8 (doubleword-aligned)
long double	64	8 (doubleword-aligned)
All pointers	32	4 (word-aligned)

Tipos de datos en C

- Rangos de valores de variables
  - **signed char**: -128 a 127
  - **unsigned char**: 0 a 255 (0x00 a 0xFF)
  - **short**: -32.768 a 32.767
  - **unsigned short**: 0 a 65.535 (0x0000 a 0xFFFF)
  - **int**: -2.147.483.648 a 2.147.483.647
  - **unsigned int**: 0 a 4.294.967.295 (0x00000000 a 0xFFFFFFFF)
- Los números negativos se almacenan en complemento a 2
  - El bit más significativo es el signo.
  - Para representar un número en complemento a 2
    - Complemento a 1 (se cambian los unos por los ceros y viceversa)
    - Se suma uno.

Tipos de datos en C

- En el estándar C99 (1999) se definen tipos de tamaño fijo
  - Las definiciones están en el fichero "stdint.h"
  - **int8_t, int16_t, int32_t, int64_t**
  - **uint8_t, uint16_t, uint32_t, uint64_t**

```
    /* exact-width signed integer types */
typedef   signed          char int8_t;
typedef   signed short     int int16_t;
typedef   signed           int int32_t;
typedef   signed      long long int64_t;

    /* exact-width unsigned integer types */
typedef unsigned          char uint8_t;
typedef unsigned short     int uint16_t;
typedef unsigned           int uint32_t;
typedef unsigned      long long uint64_t;
```


Tipos de datos en C

- ¿Qué pasa cuando una variable supera su valor?
  - Ejemplo 1
    - unsigned char a = 255;
    - signed char b;
    - a++; → ¿Cuánto vale a?
      - 0
    - b = a; → ¿Cuánto vale b?
      - -1
  - Ejemplo 2
    - Unsigned char tiempo_1 = 250;
    - Unsigned char tiempo_2 = 10;
    - ¿Cuánto vale tiempo_2 – tiempo_1?
      - Tiempo_1 = 250 = 0b 1111 1010
      - - Tiempo_1 → 0b 0000 0101 + 1 = 0b 0000 0110
      - Tiempo_2 = 10 → 0b 0000 1010
      - Tiempo_2 + (-Tiempo_1) → 0b 0001 0000 = 16
      - Si tuviéramos un temporizador de 8 bits y leyéramos en un instante un 250 y en otro instante un 10. ¿Cuántos pulsos de reloj habría habría avanzado el temporizador?
        - 251, 252, 253, 254, 255, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 → 16

Tipos de datos en C

- Constantes de enumeración
  - Lista de valores enteros constantes.
  - Si no se especifica nada los valores que toman son consecutivos empezando en cero pero se les puede dar valores específicos.
  - **enum boolean {NO, SI};**
    - Si se escribe NO es como si se pusiera un 0
    - Si se escriba SI es como si se pusiera un 1
  - **enum estados {INICIAL, PARADO, BAJANDO, SUBIENDO};**
  - **enum ASCII {NULL='\0', CR = '\r', LF = '\n'};**

Tipos de datos en C

- Declaración de variables
  - int a;
    - Se reserva espacio en memoria volátil para la variable y se inicializa con valor 0
    - Estas variables se sitúan en la zona ZI-data (Zero Initialized Data) en RAM (zona .bss en el fichero .map).
    - Por optimización del compilador, las variables de menos de 8 bytes que deberían ir en ZI-data, se almacenan en RW-data.
      - http://www.keil.com/forum/10624/
  - int a = 10;
    - Se reserva espacio en memoria volátil para la variable y se le asigna un valor concreto en el proceso de inicialización.
    - También se almacena el valor de inicialización en memoria no volátil para ser copiado a la memoria volátil tras la inicialización.
    - Estas variables se sitúan en RW-data y ocupan tanto memoria RAM como memoria no volátil (donde se almacena el valor de inicialización).
  - const int a = 10;
    - Se reserva espacio en memoria en memoria no volátil y se le asigna un valor.
    - Estas variables se sitúan en RO-data (Read Only) y ocupan memoria FLASH.
  - Program Size: Code=99444 RO-data=4556 RW-data=192 ZI-data=25644

Tipos de datos en C

- Declaración de variables
  - Muchos microcontroladores suelen tener memoria RAM limitada
    - El LPC1768 tiene dos bancos de memoria separados de 32K bytes.
  - Las constantes, especialmente las cadenas de caracteres deben situarse en memoria FLASH.
  - Utilizar un tipo variable de tamaño adecuado para la aplicación.
  - Utilizar nombres de variables con sentido.
    - entradaPWM, gananciaAmplificador, estadoSistema

Tipos de datos en C

- Números en punto flotante
  - **float y double**
  - Se almacena mantisa y exponente.
  - Permite trabajar con números con decimales de rangos muy elevados
  - Si el procesador no tiene unidad hardware para trabajar con números en punto flotante, las operaciones aritméticas pueden tardar mucho.

```
int i_a, i_b, i_c;
i_a = 10;
i_b = 1000;

i_c = i_a + i_b;   // 0,01 us
i_c = i_a - i_b;   // 0,01 us
i_c = i_a * i_b;   // 0,01 us
i_c = i_a / i_b;   // 0,06 us
```

```
float f_a, f_b, f_c;
f_a = 10.2;
f_b = 1000.5;

f_c = f_a + f_b;   // 0,40 us
f_c = f_a - f_b;   // 0,36 us
f_c = f_a * f_b;   // 0,41us
f_c = f_a / f_b;   // 0,64us

f_c = exp(f_c);    //  2,06 us
f_c = sin(f_c);    // 20,07 us
f_c = log10(f_c);  // 25,45 us
```

```
double i_a, i_b, i_c;
d_a = 10.2;
d_b = 1000.5;

d_c = d_a + d_b;   // 0,59 us
d_c = d_a - d_b;   // 0,56 us
d_c = d_a * d_b;   // 0,85 us
d_c = d_a / d_b;   // 1,46 us

d_c = exp(d_c);    // 18,88 us
d_c = sin(d_c);    // 19,63 us
d_c = log10(d_c);  // 25,39 us
```

Tipos de datos en C

- Operaciones aritméticas
  - Operaciones con enteros
    - 5/2 → 2 (cociente de la división) 5%2 → 1 (resto de la división)
    - ¿Qué valores va tomando la variable j cuando se va incrementando i?
      - ....
      - i++;
      - j = i % 8;
      - ...
  - Conversiones de tipos implícitas
    - En operaciones aritméticas de dos tipos diferentes
      - El tipo 'menor' es promovido al tipo 'superior' antes de hacer la operación.
      - El resultado es del tipo 'superior'
    - unsigned short a = 1000;
    - unsigned short b = 1000;
    - float c;
    - c = (a * b) / 3.0;
      - 1000 * 1000 = 1000000 que supera tamaño de short → 16960/3.0=5653.33
    - c = ((long)a * b) / 3.0;
      - 1000 * 1000 = 1000000 → 1000000 / 3.0 = 333333,333

Tipos de datos en C

- Operaciones de manejo de bits
  - Operadores
    - & → AND bit a bit
    - | → OR bit a bit
    - ^ → XOR bit a bit
    - << → desplazamiento a la izquierda (introduciendo ceros)
    - >> → desplazamiento a la derecha (introduciendo ceros)
    - ~ → complemento a 1 (inversión de todos los bits)
  - Para poner un bit a uno
    - LPC_GPIO0->FIOPIN = LPC_GPIO0->FIOPIN | 0x00000080;
    - LPC_GPIO0->FIOPIN = LPC_GPIO0->FIOPIN | (1 << 7);
    - LPC_GPIO0->FIOPIN |= (1 << 7);
  - Para poner un bit a cero
    - LPC_GPIO0->FIOPIN = LPC_GPIO0->FIOPIN & 0xFFFFFF7F;
    - LPC_GPIO0->FIOPIN = LPC_GPIO0->FIOPIN & (~ 0x00000080);
    - LPC_GPIO0->FIOPIN = LPC_GPIO0->FIOPIN & (~(1 << 7));
    - LPC_GPIO0->FIOPIN &= ~(1 << 7);

Tipos de datos en C

- Estructura de control SWITCH
  - Representa una decisión múltiple que comprueba si una expresión coincide con determinadas constantes enteras.
  - La sentencia break provoca una salida del switch
  - Es conveniente poner siempre un default aunque no vaya a entrar nunca (salvo error)
  - Ejemplo 1

```
switch (entrada) {
  case 0:
     // Acción 0
     break;
  case 1:
     // Acción 1
  case 2:
     // Acción 1 y 2
     break;
  default:
     // Si no vale 0, 1 o 2
     break;
}
```

  - Ejemplo 2

```
switch (rxData) {
  case '0':
     // Acción para '0'
     break;
  case 's':
     // Acción para 's'
     break;
  default:
     // Si no vale '0' o 's'
     break;
}
```

  - Ejemplo 3

```
switch (estado) {
  case ON:
     // Acción para ON
     break;
  case OFF:
     // Acción para OFF
     break;
  default:
     // Si ninguna
     break;
}
```


Variables globales y locales

- Variables globales
  - No se declaran dentro de una función.
  - Se reserva espacio para ellas en memoria RAM desde el principio hasta el final de la ejecución.
  - Cuando sólo hay un fichero .C pueden ser leídas y escritas por cualquier función.
  - Deben usarse variables globales sólo cuando sea necesario.
- Variables locales a las funciones
  - Se declaran en el interior de una función.
  - Se reserva espacio para ellas en la Pila (Stack) al iniciar la función y desaparecen al finalizar la función.
  - Sólo se pueden leer y escribir desde dentro de la función.
  - Deben usarse variables locales cuando sólo se van a usar dentro de la función.

Variables globales y locales

- Variables locales estáticas
  - Se pone delante el calificador "**static**"
  - Sólo son visibles en el interior de una función.
  - Se inicializan la primera vez que se ejecuta la función
  - Al salir de la función no desaparecen sino que mantienen el valor hasta la siguiente ejecución.
  - **Se reserva espacio para ellas en memoria RAM** desde el principio hasta el final de la ejecución.
  - Deben usarse cuando se quiere mantener el valor de una variable pero con visibilidad local.
  - Ejemplo

```
int32_t numEjecuciones(void) {
    static int32_t ejecuciones = 0;
    ejecuciones++;
    return ejecuciones;
}
```


Variables volatile

- Variables volatile
  - La optimización de código del compilador puede ser errónea si no se tiene cuidado
    - Por ejemplo: eliminar variables que sólo se leen y no se escriben en el programa principal.
      - En una interrupción periódica se incrementa la variable clk.
      - En el programa principal:
        - clk = 0:
        - whle(1) {
        - if (clk > 1000) ...
      - El complidador podría no compilar el contenido del if por suponer que siempre clk=0
  - Con **volatile** se indica que el contenido de una variable puede sufrir modificaciones que no estén explícitas en el código.
    - Registros asociados al hardware
      - Puertos de entrada
      - Temporizadores
      - Registros con flags de aviso
    - Variables globales modificadas en interrupciones.
  - Deben declararse como volatile las variables que corresponden a registros del HW y las variables globales que son modificadas por una interrupción.

División de un programa en varios ficheros

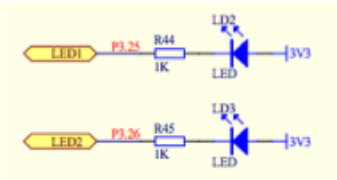
- Variables globales estáticas
  - Se pone delante el calificador "static" en una variable global.
  - Sólo son visibles en el interior del fichero .C donde se definen. No pueden verse desde otros ficheros .C
- Funciones estáticas
  - Son funciones a las que sólo pueden llamar otras funciones del mismo fichero.
  - No son visibles al código situado en otros ficheros.
- Variables globales externas
  - Cuando se declara una variable como "extern" el compilador supone que la variable está declarada completamente en otro fichero .C

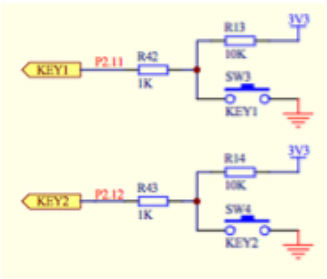
División de un programa en varios ficheros

- Ficheros de cabecera
  - Normalmente para cada fichero .C se crea un .H con las funciones, definiciones y variables globales a las que se quiere tengan acceso el programa desde otros ficheros.
  - El fichero .H correspondiente se incluye en el fichero donde se usen las variables.
  - En los ficheros .H no se pone código, sólo definiciones.
  - Los ficheros .H no se incluyen en el proyecto de Keil.
  - Para evitar múltiples inclusiones de un mismo fichero se suelen introducir sentencias de compilación condicionada.

División de un programa en varios ficheros

- Ejemplo de conmutación de los LEDs de la tarjeta MiniDK2
  - V1: Programa que conmuta los LEDs cada 100ms
  - V2: Programa que conmuta los LEDs cada 100ms mejorado
  - V3: Se utilizan funciones para manejar los LEDs
  - V4: Se paran los LEDs al pulsar KEY1
  - V5: Se divide el programa en varios ficheros.





Estructuras: campos de bits

- struct
  - Una estructura engloba variables que puedes ser de tipos diferentes.
  - Los campos de las estructuras mantienen el alineamiento de sus tipos de datos. Pueden dejar espacios libres entre campos.
  - Para pasar a una función como parámetro se pasa un puntero

Estructuras: campos de bits

- struct
  - Se pueden definir campos de bits
    - struct ADCR_t {
    - uint32_t SEL:7;
    - uint32_t CLKDIV:7;
    - uint32_t BURST:1;
    - uint32_t reserved1:4;
    - uint32_t PDN:1;
    - uint32_t reserved2:2;
    - uint32_t START:3;
    - uint32_t EDGE:1;
    - };

    - struct ADCR_t* ADCR;

    - ADCR = (struct ADCR_t*)&(LPC_ADC->ADCR);
    - ADCR->SEL = 3;
    - ADCR-> CLKDIV = 24;
    - ADCR->EDGE = 1;

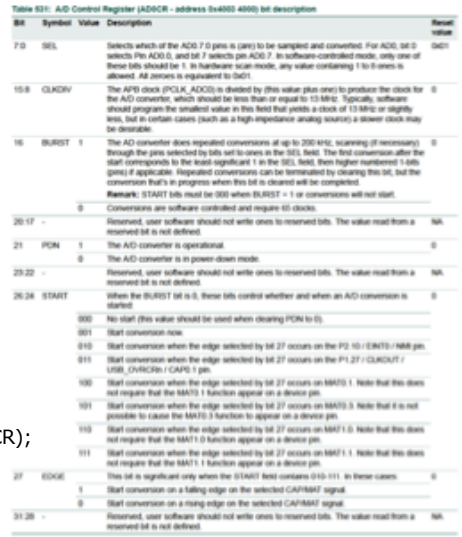


Table 531: A/D Control Register (AD0CR - address 0x4003 4000) bit description

Bit	Symbol	Value	Description	Reset value
7:0	SEL		Selects which of the AD0.7:0 pins is (are) to be sampled and converted. For AD0, bit 0 selects Pin AD0.0, and bit 7 selects pin AD0.7. In software-controlled mode, only one of these bits should be 1. In hardware scan mode, any value containing 1 to 8 ones is allowed. All zeroes is equivalent to 0x01.	0x01
15:8	CLKDIV		The APB clock (PCLK_ADC0) is divided by (this value plus one) to produce the clock for the A/D converter, which should be less than or equal to 13 MHz. Typically, software should program the smallest value in this field that yields a clock of 13 MHz or slightly less, but in certain cases (such as a high-impedance analog source) a slower clock may be desirable.	0
16	BURST	1	The A/D converter does repeated conversions at up to 200 kHz, scanning (if necessary) through the pins selected by bits set to ones in the SEL field. The first conversion after the start corresponds to the least-significant 1 in the SEL field, then higher numbered 1-bits (pins) if applicable. Repeated conversions can be terminated by clearing this bit, but the conversion that's in progress when this bit is cleared will be completed. **Remark:** START bits must be 000 when BURST = 1 or conversions will not start.	0
		0	Conversions are software controlled and require 65 clocks.	
20:17	-		Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA
21	PDN	1	The A/D converter is operational.	0
		0	The A/D converter is in power-down mode.	
23:22	-		Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA
26:24	START		When the BURST bit is 0, these bits control whether and when an A/D conversion is started:	0
		000	No start (this value should be used when clearing PDN to 0).	
		001	Start conversion now.	
		010	Start conversion when the edge selected by bit 27 occurs on the P2.10 / EINT0 / NMI pin.	
		011	Start conversion when the edge selected by bit 27 occurs on the P1.27 / CLKOUT / USB_OVRCRn / CAP0.1 pin.	
		100	Start conversion when the edge selected by bit 27 occurs on MAT0.1. Note that this does not require that the MAT0.1 function appear on a device pin.	
		101	Start conversion when the edge selected by bit 27 occurs on MAT0.3. Note that it is not possible to cause the MAT0.3 function to appear on a device pin.	
		110	Start conversion when the edge selected by bit 27 occurs on MAT1.0. Note that this does not require that the MAT1.0 function appear on a device pin.	
		111	Start conversion when the edge selected by bit 27 occurs on MAT1.1. Note that this does not require that the MAT1.1 function appear on a device pin.	
27	EDGE		This bit is significant only when the START field contains 010-111. In these cases:	0
		1	Start conversion on a falling edge on the selected CAP/MAT signal.	
		0	Start conversion on a rising edge on the selected CAP/MAT signal.	
31:28	-		Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA

Estructuras: campos de bits

- struct
  - Se pueden definir campos de bits
    - struct {
    - uint32_t flag1:1;
    - uint32_t flag2:1;
    - uint32_t pulsado_KEY1:1;
    - uint32_t pulsado_KEY2:1;
    - uint32_t flag5:1;
    - } flags;

    - flags.flag1 = 1;

    - flags.flag1 = 0;

    - if (LPC_GPIO2->FIOPIN & (1<<11))
    - flags.pulsado_KEY1 = 1;
    - else
    - flags.pulsado_KEY1 = 0;

    - if (flags.pulsado_KEY1) …

Uniones

- Union
  - Tipo de datos que define campos que comparten el mismo espacio de memoria.

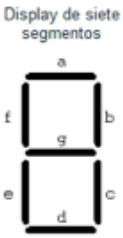
Arrays

- array
  - Es un conjunto ordenado de elementos del mismo tipo.
  - El nombre del array es un puntero al primer elemento del array.
  - Es necesario definir el tamaño del array
  - En C no se comprueba si se accede a un elemento fuera del array
    - int8_t datos[5];
    - int8_t datos[5] = {32, 24, 2, 17, 33};
      - datos[0] es 32
      - datos[2] es 2
      - datos[8] = 17;
        - Se escribe fuera del array
        - Lo más probable es que modifique el contenido de otra variable.
        - Podría provocar un Hardware Fault (o no)
    - int8_t* pdatos[10]; // es un array de punteros a int8_t
      - pdatos[0] = datos;
      - pdatos[0][3] es 17
    - int8_t matriz[3][4] = {{12,1,23,12}, {1,2,3,4}, {2,45,23,1}};

Arrays

- array
  - Los arrays se pueden utilizar como decodificadores.
    - Se colocan los 7 leds de un display de 7 segmentos de P2.0 a P2.6 ('a' a 'g')
      - uint8_t BCD_7seg[] = {0x3F,0x06,0x5B,0x4F,0x66,0x6D,0x7D,0x07,0x7F,0x6F,
      - 0x77, 0x7C, 0x39, 0x5E, 0x79, 0x71}
      - LPC_GPIO2->FIOPIN = (LPC_GPIO2->FIOPIN & 0xFFFFFF80) | BCD_7seg[numero];

Cifra	–gfedcba	Display
0000	00111111	0
0001	00000110	1
0010	01011011	2
0011	01001111	3
0100	01100110	4
0101	01101101	5
0110	01111101	6
0111	00000111	7
1000	01111111	8
1001	01100111	9
1010	01110111	A
1011	01111100	b
1100	00111001	C
1101	01011110	d
1110	01111001	E
1111	01110001	F



https://programacionsiemens.com/display-de-7-segmentos-step-7/

Cadenas de caracteres (Stings)

- strings
  - Las cadenas de caracteres son arrays de caracteres.
  - Se utilizan para almacenar caracteres de texto.
  - En el array se almacenan los códigos ASCII de cada carácter.
  - Las cadenas de caracteres terminan con un '\0'
    - char cadena[] = "Hola";
      - En un array de 5 elementos: 'H' 'o' 'l' 'a' '\0'
  - No se comprueban los tamaños. Siempre debe haber reservado más espacio del necesario.

https://programacionsiemens.com/display-de-7-segmentos-step-7/

Cadenas de caracteres (Stings)

- Funciones de cadena
  - #include <string.h>
  - strlen(<cadena>)
    - Devuelve la longitud de la cadena sin tomar en cuenta el caracter de final de cadena.
  - strcpy(<cadena_destino>, <cadena_origen>)
    - Copia el contenidode <cadena_origen>en<cadena_destino>.
  - strcat(<cadena_destino>, <cadena_origen>)
    - Concatena el contenidode <cadena_origen>al final de<cadena_destino>.
  - strcmp(<cadena1>, <cadena2>)
    - Compara las dos cadenas y devuelve un 0 si las dos cadenas son iguales, un número negativo si <cadena1> es menor que (precede alfabéticamente a) <cadena2> y un número positivo (mayor que cero) si <cadena1> es mayor que <cadena2>.

https://programacionsiemens.com/display-de-7-segmentos-step-7/

Salida con formato

- int printf (char *format, arg1, arg2, ...)
  - printf("dato: %d", numero);

Modificador de Escape	Descripción
%c	Un único carácter
%hd	Un entero corto
%hu	Un entero corto sin signo
%d	Un entero con signo, en base decimal
%ld	Un entero largo
%e	Un número real en coma flotante, con exponente *3e2*
%E	Un número real en coma flotante, con exponente en mayúscula *3E2*
%f	Un número real en coma flotante, sin exponente
%o	Un entero en base octal
%p	Un puntero o dirección de memoria
%s	Una cadena de caracteres
%u	Un entero sin signo, en base decimal
%x	Un entero en base hexadecimal *aa289*
%X	Un entero en base hexadecimal en mayúsculas *AA289*

printf(":%4d:\n", 1234);	:1234:
printf(":%4d:\n", 12345);	:12345:
printf(":%4d:\n", -1);	: -1:
printf(":%4d:\n", -12);	: -12:
printf(":%4d:\n", -123);	:-123:
printf(":%4d:\n", -1234);	:-1234:
printf(":%4d:\n", -12345);	:-12345:
printf(":%-4d:\n", -12);	:-12 :
printf(":%e:\n", 1234567.89);	:1.234568e+006:
printf(":%e:\n", +1234567.89);	:1.234568e+006:
printf(":%e:\n", -1234567.89);	:-1.234568e+006:
printf(":%E:\n", 1234567.89);	:1.234568E+006:
printf(":%f:\n", 1234567.89);	:1234567.890000:
printf(":%g:\n", 1234567.89);	:1.23457e+006:
printf(":%G:\n", 1234567.89);	:1.23457E+006:

- Pero .... ¿por dónde saca los caracteres?

http://kripkenprogestructurada.blogspot.com.es/2011/01/tarea-2.html
http://slideplayer.es/slide/1549020/

Salida con formato

- int sprintf (char* string, char *format, arg1, arg2, ...)
  - Igual que printf pero la salida es en una cadena de caracteres.
  - Es muy importante que el tamaño del string sea suficientemente grande para almacenar la salida.
    - sprintf(cadena, "Valor de salida: %d", valor);
    - sprintf(estado, "ON");
    - sprintf(cadena, "Estado = %s", estado);
    - char formato[] = "Estado = %s";
    - char estado[] = "ON";
    - sprintf(cadena, formato, estado);
    - char formato[] = "Formato: Estado = %s";
    - char estado[] = "ON";
    - sprintf(cadena, &formato[9], estado);

http://kripkenprogestructurada.blogspot.com.es/2011/01/tarea-2.html
http://slideplayer.es/slide/1549020/

Stack y Heap

- Stack
  - Se almacena el contexto y la dirección de retorno al saltar a subrutinas (funciones en C).
  - Se almacenan las variables locales
  - Está situada en memoria RAM
  - Se reserva espacio en el fichero startup_LPC17xx.s
  - Si la pila excede el espacio reservado puede modificar el valor de variables o un cambio de una variable podría modificar la pila (y hacer que el programa retorne a una dirección descontrolada).
- Heap
  - Espacio reservado para almacenar memoria reservada dinámicamente con malloc().
  - Se reserva espacio en el fichero startup_LPC17xx.s
  - Si se utilizan funciones de reserva dinámica de memoria y no se ha reservado espacio de Heap, el programa no funciona correctamente (no da error ni de compilación ni de linker).
  - Las librerías de TCP/IP y de acceso a la tarjeta SD necesitan Heap.

Stack y Heap

- Situación en memoria del Stack y Heap
  - El tamaño del Heap y del Stack se fijan en el fichero startup_LPC17xx.s
  - La localización en memoria puede verse en el fichero .MAP
    - Execution Region RW_IRAM1 ó Execution Region RW_IRAM2
  - El orden en memoria es el siguiente:
    - Primero se reserva espacio para las variables estáticas y globales
      - Primero la zona .data (rw-data)
      - Después la zona .bss (zi-data)
    - Posteriormente se coloca el Heap
    - Por último el Stack
      - **El puntero de pila se inicializa al final del Stack avanzando hacia el Heap.**

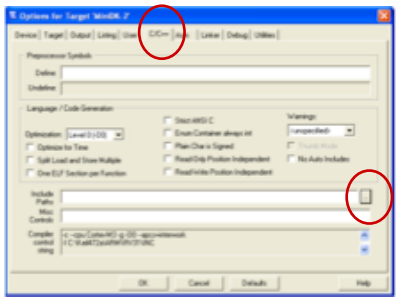
```
Execution Region RW_IRAM1 (Base: 0x10000000, Size: 0x00000368, Max: 0x00008000, ABSOLUTE)

Base Addr    Size         Type   Attr      Idx    E Section Name        Object

0x10000000   0x00000004   Data   RW         16    .data               system_lpc17xx.o
0x10000004   0x00000001   Data   RW         59    .data               temp5.o
0x10000005   0x00000003   PAD
0x10000008   0x00000060   Zero   RW        174    .bss                c_w.l(libspace.o)
0x10000068   0x00000100   Zero   RW          2    HEAP                startup_lpc17xx.o
0x10000168   0x00000200   Zero   RW          1    STACK               startup_lpc17xx.o
```


Proceso de compilación

- Proyecto
  - El proyecto contiene todos los ficheros del programa:
    - Ficheros .C que contienen código C y que suelen incluir a varios ficheros .H
    - Ficheros .S que contienen código en ensamblador.
    - Ficheros .LIB que contienen bibliotecas de funciones ya compiladas.
  - En las opciones del proyecto se puede configurar:
    - Espacio de memoria de programa y de datos disponible para ser utilizada.
    - Los directorios donde se almacenan los ficheros intermedios
    - Los directorios donde el compilador debe buscar los ficheros de cabecera
    - El nivel de optimización del compilador.
    - La configuración del hardware de depuración.
    - ....

Proceso de compilación

- Preprocesador
  - Se ejecuta antes de compilar el programa
  - #include "nombre" o #include <nombre>
    - Sustituye la línea por todo el contenido del fichero "nombre".
    - "nombre" busca el fichero en el directorio del proyecto
    - <nombre> busca los archivos en los directorios del PATH (por orden)
      - Opciones del proyecto → C/C++ → Include Paths



Proceso de compilación

- Preprocesador
  - Definición de macros
    - Las macros se escriben normalmente en mayúsculas
      - #define TICK 50
      - #define LED1_ON LPC_GPIO3->FIOCLR = (1<<25)
      - Y luego se llama con: LED1_ON;
    - Para definir una macro no es necesario asignar un valor
      - #define DEBUG
    - Las macros pueden tener parámetros:
      - #define PIN_P0_ON(n) LPC_GPIO0->FIOSET = (1<<n)
  - Compilación condicionada
    - Permite compilar o no determinadas zonas de código
      - #ifdef DEBUG
        - El código en esta parte se compila si está definida DEBUG
      - #else
        - El código de esta parte se compila si no está definida DEBUG
      - #endif

Proceso de compilación

- Preprocesador
  - Antes de compilar se ejecuta el preprocesador:
    - Incluye en cada fichero .C los ficheros de cabecera correspondientes.
    - Sustituye las macros por su valor.
    - Elimina el código que no va a ser compilado (compilación condicional).
    - Se eliminan los comentarios.
    - Como resultado se genera un fichero .i
- Compilador
  - A partir de un fichero .C genera código máquina reubicable (fichero .o)
    - No está asociado a una zona de memoria específica
  - Se realiza el análisis sintáctico del código.
  - Cuando hay un error indica la línea donde se ha producido
  - Es conveniente leer los warnings y si es posible eliminarlos.
  - Optimización del compilador
    - El compilador intenta generar el código más rápido y corto posible.
    - Hay veces que toma decisiones que no esperamos.
  - Se puede generar un fichero .txt con el ensamblador asociado a cada línea

Proceso de compilación

- Optimización del compilador
  - El compilador intenta generar el código más rápido y corto posible.
    - Hay veces que toma decisiones que no esperamos.
  - Se pueden configurar varios niveles de optimización

Optimization

Control compiler code optimization for the generated code. Sets the compiler command-line option -Onum:

- **Default:** Use the compiler default or the setting of a higher Target or Group level.
- **Level 0 (-O0)**: Turn off all optimization, except some simple source transformations.
- **Level 1 (-O1)**: Turn off optimizations that seriously degrade the debug view.
- **Level 2 (-O2)**: High optimization (default level). The debug view might be less satisfactory because the mapping of object code to source code is not always clear.
- **Level 3 (-O3)**: Maximum optimization. Note that **Level 3** in combination with **Optimize for Time** may generate more code that **Level 2** since it may unroll loops.

Optimize for Time

Reduce execution time at the possible expense of a larger code size. Sets the compiler command-line option -Otime. If not enabled, the compiler assumes -Ospace.

- Algunos ejemplos de optimización
  - Se pueden eliminar las líneas de código donde se escribe en variables que luego no se utilizan o que se vuelve a escribir un valor sin leerse.
    - Esto puede ser un problema si la variable es modificada por una interrupción.
    - Si una variable puede cambiar fuera del flujo normal de programa → **volatile**

Proceso de compilación

- Ensamblador
  - A partir de ficheros con código en ensamblador (.S) genera código máquina reubicable (.O)
- Linker
  - El Linker enlaza la información de varios ficheros .O y ficheros .LIB para generar un fichero final de código máquina ubicado en unas zonas de memoria concretas.
  - Los errores del Linker no se asocian a una línea de código
    - Suelen ser más difíciles de resolver que los errores de compilación.
    - Los errores más típicos son:
      - Se excede el espacio de memoria reservado para variables
      - Hay variables que se ponen como extern en uno o varios sitios pero no se declaran completamente en ninguno.
      - Variables que se declaran con el mismo nombre en varios ficheros .C (o .H)
  - El fichero .MAP contiene la localización en memoria de los diferentes elementos (variables y funciones)
  - En vez de generar un fichero ejecutable se podría generar un .LIB para ser utilizado en otro proyecto.

Organización del proyecto

- Organización del proyecto
  - En el proceso de compilación se generan muchos ficheros que es conveniente mantener ordenados.
- Organización de los ficheros fuente
  - En un proyecto suelen utilizarse ficheros .C propios y ficheros .C a modo de bibliotecas de funciones
    - Suelen ser ficheros .C elaborados por terceras personas o desarrollados para otro proyecto
  - Es conveniente colocarlos en directorios separados para facilitar su portabilidad a otros proyectos.
    - En las opciones del proyecto → C/C++ → Include Paths hay que incluir el directorio donde estén los ficheros .H asociados.
    - Los .C se incluyen en el proyecto para ser compilados.

Organización del proyecto

- Organización de los ficheros de salida
  - Opciones del proyecto -> Output -> Select Folder for Objets
    - Indica dónde guardar los ficheros .O e intermedios como .crf, .d
    - Normalmente se colocan en una carpeta llamada OBJ
- Organización de los ficheros de listados intermedios
  - Opciones del proyecto -> Listing -> Select Folder for Objets
    - Se indica dónde guardar los ficheros .I, .TXT y .MAP
    - Normalmente se colocan en una carpeta llamada LST
- Recomendación de organización de directorios
  - Source o User
  - Libraries
  - MDK-ARM o Project
    - OBJ
    - LST

Fichero LPC17xx.h

- Ejemplo: Acceso a los registros de acceso a los puertos
  - Situación de los registros en memoria

Registro	Port0	Port1	Port2	Port3	Port4
FIODIR	0x2009 C000	0x2009 C020	0x2009 C040	0x2009 C060	0x2009 C080
FIOMASK	0x2009 C010	0x2009 C030	0x2009 C050	0x2009 C070	0x2009 C090
FIOPIN	0x2009 C014	0x2009 C034	0x2009 C054	0x2009 C074	0x2009 C094
FIOSET	0x2009 C018	0x2009 C038	0x2009 C058	0x2009 C078	0x2009 C098
FIOCLR	0x2009 C01C	0x2009 C03C	0x2009 C05C	0x2009 C07C	0x2009 C09C

```
#define LPC_GPIO_BASE          (0x2009C000UL)
#define LPC_GPIO0_BASE         (LPC_GPIO_BASE + 0x00000)
#define LPC_GPIO1_BASE         (LPC_GPIO_BASE + 0x00020)
#define LPC_GPIO2_BASE         (LPC_GPIO_BASE + 0x00040)
#define LPC_GPIO3_BASE         (LPC_GPIO_BASE + 0x00060)
#define LPC_GPIO4_BASE         (LPC_GPIO_BASE + 0x00080)
```

Fichero LPC17xx.h

- Ejemplo: Acceso a los registros de acceso a los
  - Situación de los registros en memoria

Registro	Port0
FIODIR	0x2009 C000
FIOMASK	0x2009 C010
FIOPIN	0x2009 C014
FIOSET	0x2009 C018
FIOCLR	0x2009 C01C

```
#define __I  volatile const /*!< Defines 'read only' permissions    */
#define __O  volatile       /*!< Defines 'write only' permissions   */
#define __IO volatile       /*!< Defines 'read / write' permissions */
```

```
typedef struct
{
  union {
    __IO uint32_t FIODIR;
    struct {
      __IO uint16_t FIODIRL;
      __IO uint16_t FIODIRH;
    };
    struct {
      __IO uint8_t  FIODIR0;
      __IO uint8_t  FIODIR1;
      __IO uint8_t  FIODIR2;
      __IO uint8_t  FIODIR3;
    };
  };
  uint32_t RESERVED0[3];
  union {
    __IO uint32_t FIOMASK;
    struct {
      __IO uint16_t FIOMASKL;
      __IO uint16_t FIOMASKH;
    };
    struct {
      __IO uint8_t  FIOMASK0;
      __IO uint8_t  FIOMASK1;
      __IO uint8_t  FIOMASK2;
      __IO uint8_t  FIOMASK3;
    };
  };
```


Fichero LPC17xx.h

- Ejemplo: Acceso a los registros de acceso a los
  - Situación de los registros en memoria

Registro	Port0
FIODIR	0x2009 C000
FIOMASK	0x2009 C010
FIOPIN	0x2009 C014
FIOSET	0x2009 C018
FIOCLR	0x2009 C01C

```
  union {
    __IO uint32_t FIOSET;
    struct {
      __IO uint16_t FIOSETL;
      __IO uint16_t FIOSETH;
    };
    struct {
      __IO uint8_t  FIOSET0;
      __IO uint8_t  FIOSET1;
      __IO uint8_t  FIOSET2;
      __IO uint8_t  FIOSET3;
    };
  };
  union {
    __O  uint32_t FIOCLR;
    struct {
      __O  uint16_t FIOCLRL;
      __O  uint16_t FIOCLRH;
    };
    struct {
      __O  uint8_t  FIOCLR0;
      __O  uint8_t  FIOCLR1;
      __O  uint8_t  FIOCLR2;
      __O  uint8_t  FIOCLR3;
    };
  };
} LPC_GPIO_TypeDef;
```

Fichero LPC17xx.h

- Ejemplo: Acceso a los registros de acceso a los puertos
  - Situación de los registros en memoria

Registro	Port0	Port1	Port2	Port3	Port4
FIODIR	0x2009 C000	0x2009 C020	0x2009 C040	0x2009 C060	0x2009 C080
FIOMASK	0x2009 C010	0x2009 C030	0x2009 C050	0x2009 C070	0x2009 C090
FIOPIN	0x2009 C014	0x2009 C034	0x2009 C054	0x2009 C074	0x2009 C094
FIOSET	0x2009 C018	0x2009 C038	0x2009 C058	0x2009 C078	0x2009 C098
FIOCLR	0x2009 C01C	0x2009 C03C	0x2009 C05C	0x2009 C07C	0x2009 C09C

```
#define LPC_GPIO0        ((LPC_GPIO_TypeDef      *) LPC_GPIO0_BASE    )
#define LPC_GPIO1        ((LPC_GPIO_TypeDef      *) LPC_GPIO1_BASE    )
#define LPC_GPIO2        ((LPC_GPIO_TypeDef      *) LPC_GPIO2_BASE    )
#define LPC_GPIO3        ((LPC_GPIO_TypeDef      *) LPC_GPIO3_BASE    )
#define LPC_GPIO4        ((LPC_GPIO_TypeDef      *) LPC_GPIO4_BASE    )
```

LPC_GPIO0->FIOSET = ...

Recomendaciones

- Comentarios
  - El código debe ir comentado. No comentar obviedades.
  - Los comentarios deben escribirse a la vez que se escribe el código.
- Uso de Macros
  - Es conveniente utilizar macros para definir constantes que luego se utilizan en el código y comentar el significado. Evitar los "números mágicos"
- Siempre **identar** el código correctamente.
- Poner a las variables nombres con sentido.
- Utilizar una regla de estilo como la disponible en Blackboard.
- Si es posible utilizar gestor de versiones como CVS, subversion, Git ...
  - Los repositorios pueden ser locales o en red.
  - https://es.wikipedia.org/wiki/Programas_para_control_de_versiones
- Si los comentarios se ponen de una determinada manera se pueden utilizar programas de documentación automática como doxigen
  - http://www.stack.nl/~dimitri/doxygen/

Ejercicios

- Ejercicios con bits
  - Suponiendo que hay 8 LEDs conectados a los bitrs P0.0 a P0.7 hacer que se vaya encendiendo un LED cada vez, desplazándose y rebotando en los extremos cambiando de led cada 100ms
    - Sin utilizar esperas activas (Delays)
    - Utilizando el SysTick
    - Utilizar el simulador para comprobar el funcionamiento.
  - Modificar el programa para que sólo se desplacen los LEDs al pulsar KEY1
  - Modificar el programa para que al pulsar KEY1 se mueva un LED y al pulsar KEY2 se inicie otra secencia.
    - Si no se pulsa ninguno no se mueven los LEDs
    - Si se pulsan KEY1 y KEY2 se moverán dos LEDs a la misma velocidad pero con desfase dependiendo de cuándo se pulsaron.
- Arrays
  - Realiza un programa que almacene en un array de 32 muestras de 16 bits un ciclo completo de un seno. Posteriormente este array se utilizará para enviar muestras a un conversor digital analógico y generar una señal sinusoidal.

Cálculo del plazo máximo de respuesta

- Referencias
  - Kernighan, B. W., & Ritchie, D. M. (1991). *El lenguaje de programación C*. Pearson Educación.
  - Documentación on-line de Keil – ARM
    - http://infocenter.arm.com/help/index.jsp
  - Retargeting en Keil-ARM
    - http://www.keil.com/pack/doc/compiler/RetargetIO/html/index.html