Introducción a lenguaje C para embedidos

Breve reseña

Sistema embebido

El sistema embebido se puede describir mejor como un sistema que tiene tanto el hardware como el software y está diseñado para realizar una tarea específica. Un buen ejemplo de un sistema integrado, que tienen muchos hogares, es un lavarropas.

Los sistemas embebidos no siempre son stand-alone como el lavarropas pero son parte de un sistema mucho más grande. Por ejemplo, el auto. Muchos modulitos stand alone que hacen tareas específicas y envían resultados de estados a otra principal.

Reseña

En un Sistema Embebido simple, el Módulo de Hardware principal es el Procesador. El procesador es el corazón del sistema integrado y puede ser cualquier cosa como un microprocesador, microcontrolador, DSP, CPLD (dispositivo lógico programable complejo) o un FPGA (arreglos de puertas lógicas programables en campo).

Todos estos dispositivos tienen una cosa en común: son programables, es decir, podemos escribir un programa (que es la parte de software del sistema integrado) para definir cómo funciona realmente el dispositivo.

El software o programa integrado permite al hardware monitorear eventos externos (entradas / sensores) y controlar dispositivos externos (salidas) en consecuencia. Durante este proceso, el programa para un sistema integrado puede tener que manipular directamente la arquitectura interna del hardware integrado (generalmente el procesador), como temporizadores, interfaz de comunicaciones en serie, manejo de interrupciones y puertos de E / S, etc.

A partir de la declaración anterior, queda claro que la parte de software de un sistema integrado es igualmente importante que la parte de hardware. No tiene sentido tener componentes de hardware avanzados con programas mal escritos (software). Por eso necesitamos escribir un buen código.

Usando C para nuestro LPC.

¿En qué nos basamos para elegir C?

Size: The memory that the program occupies is very important as Embedded Processors like Microcontrollers have a very limited amount of ROM (Program Memory).

Speed: The programs must be very fast i.e., they must run as fast as possible. The hardware should not be slowed down due to a slow running software.

Portability: The same program can be compiled for different processors. Cross compilation.

Ease of Implementation

Ease of Maintenance

Readability

Los sistemas embebidos se desarrollaron principalmente utilizando lenguaje ensamblador. Aunque el lenguaje ensamblador es el más cercano a las instrucciones del código de máquina real y produce archivos hexadecimales de tamaño pequeño, la falta de portabilidad y la gran cantidad de recursos (tiempo y mano de obra) dedicados a desarrollar el código dificultan el trabajo con el lenguaje ensamblador.

Ventajas de usar C.

Algunos de los beneficios de utilizar C como lenguaje de programación principal:

Significativamente fácil de escribir código en C

Consume menos tiempo en comparación con el assembler.

El mantenimiento del código (modificaciones y actualizaciones) es muy sencillo

Hacer uso de las funciones de la biblioteca (libraries) para reducir la complejidad del código principal.

Puede transferir fácilmente el código a otra arquitectura con muy pocas modificaciones

Embedded C o C. Aclaración.

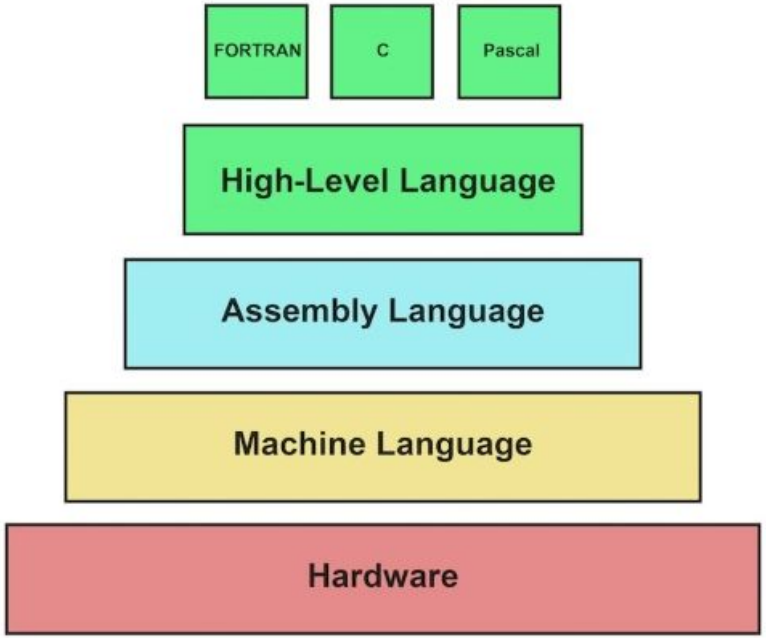
Son muy similares.

Son ISO estándar los dos o sea tienen la misma sintaxis, tipos de datos y funciones.

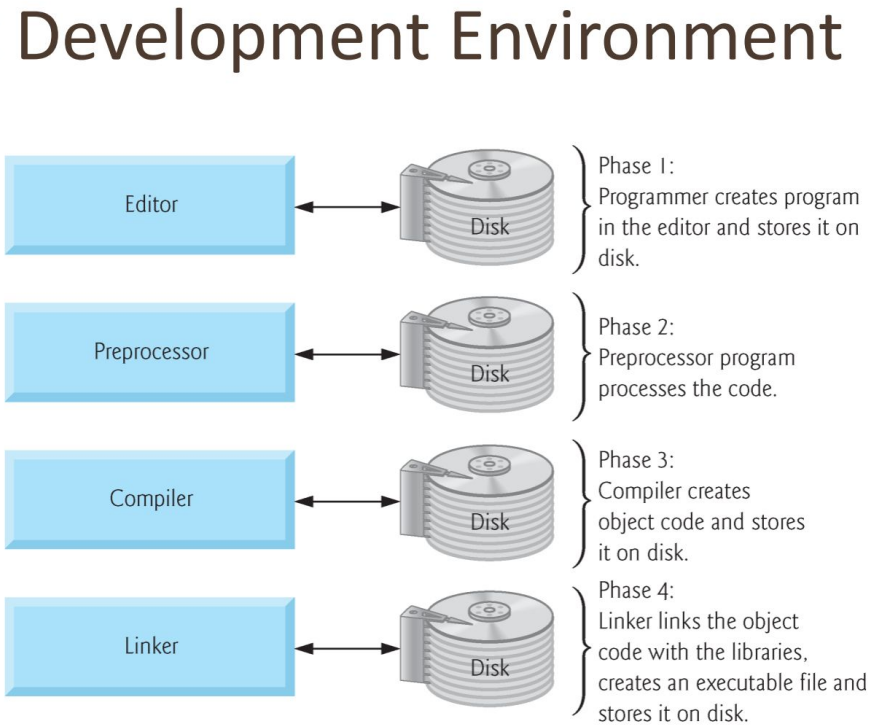
Embedded C es básicamente una extensión del lenguaje de programación C estándar con características adicionales como direccionamiento de E / S, aritmética de punto fijo, etc.

El lenguaje de programación C se usa generalmente para desarrollar aplicaciones de escritorio, mientras que el C embebido se usa en el desarrollo de aplicaciones basadas en microcontroladores. Nosotros vamos a usar C porque está más estandarizado en los LPC.

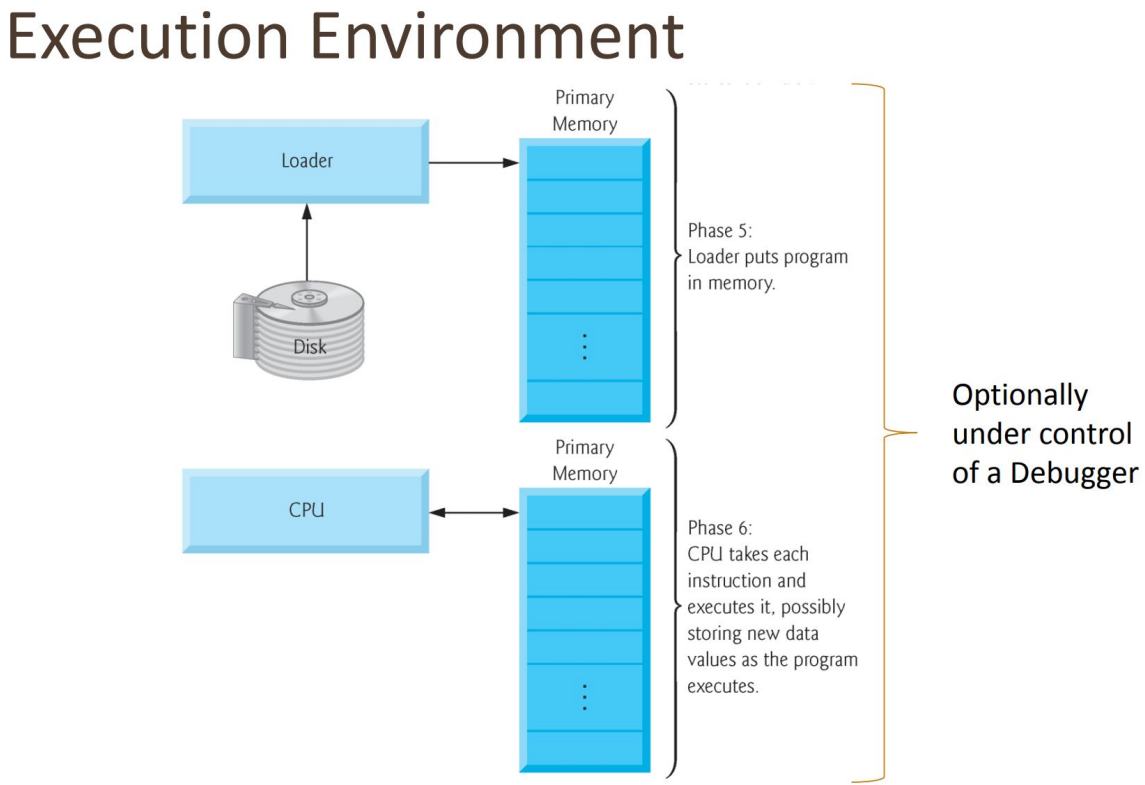
Capas del lenguaje



Desarrollo del entorno.



Ejecución


Execution Environment
Primary Memory
Loader
Disk
Phase 5: Loader puts program in memory.
Primary Memory
CPU
Phase 6: CPU takes each instruction and executes it, possibly storing new data values as the program executes.
Optionally under control of a Debugger

IDE

IDE Integrated Development Environment Editor Compiler Debugger Ex: MS Visual C++ Xcode

¿Cuál vamos a usar nosotros?

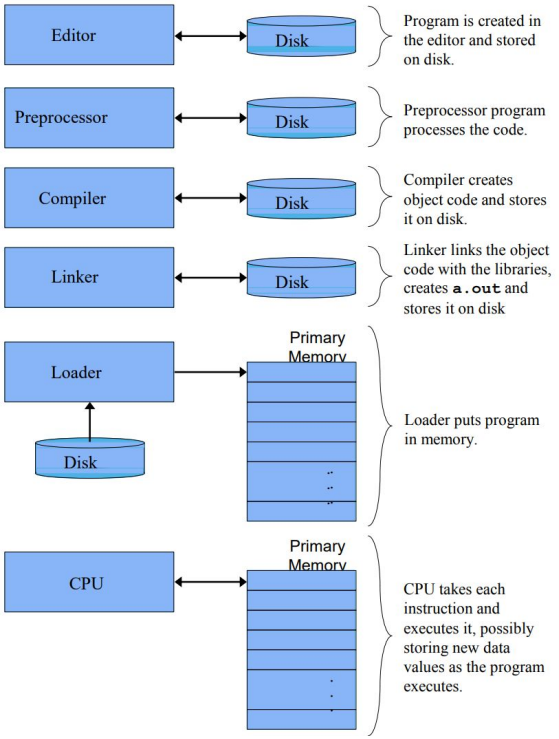
LPC expresso o MCU expresso. El que más se adapte a la placa que tengamos.

Fases de nuestra programación.

Basics of a Typical C Environment

Phases of C Programs:

1. Edit
2. Preprocess
3. Compile
4. Link
5. Load
6. Execute



Estructura de nuestros programas en C

Tienen una main function.Es un “MUST have”

Tienen llamadas a múltiples funciones para hacer diferentes cosas.

Function1

Function 2...etc.

Cómo son las funciones?

Todas las funciones tienen un header seguido de un bloque de código que la define. (se puede definir tmb qué hace en el header .h :))

Definiendo una función

```
#include <iostream>
#include <string>


int main()
{
  std::string name;
  std::cout << "What is your name? ";
  getline (std::cin, name);
  std::cout << "Hello, " << name << "!\n";
}
```

Podemos hacer que “devuelva” algo.

Return expression

1.Configura lo que va a devolver.

2. Devuelve el valor al que la llamó o invocó.

Ejemplo:

```
// Example program
#include <iostream>
#include <string>

int main()
{
  std::string name;
  std::cout << "What is your name? ";
  getline (std::cin, name);
  std::cout << "Hello, " << name << "!\n";
  return 0;///  Tiene sentido?
}
```

Directivas de preprocesamiento.

Una línea de programa en C que comienza con # le dice al compilador antes de que comience la traducción al lenguaje ob del micro que va a usar las funciones y defines que estén en ese archivo .h. por ejemplo # include ‘stdio.h’; El archivo se denomina header.

<> indican que es un archivo de encabezado estándar del compilador. “” indica que es un archivo de usuario o de un directorio específico.

Punteros en C.

El concepto básico es simple:

- Es una variable que guarda una dirección de una posición de memoria.
- El truco está en entender cómo es manejada la memoria en un programa en C.

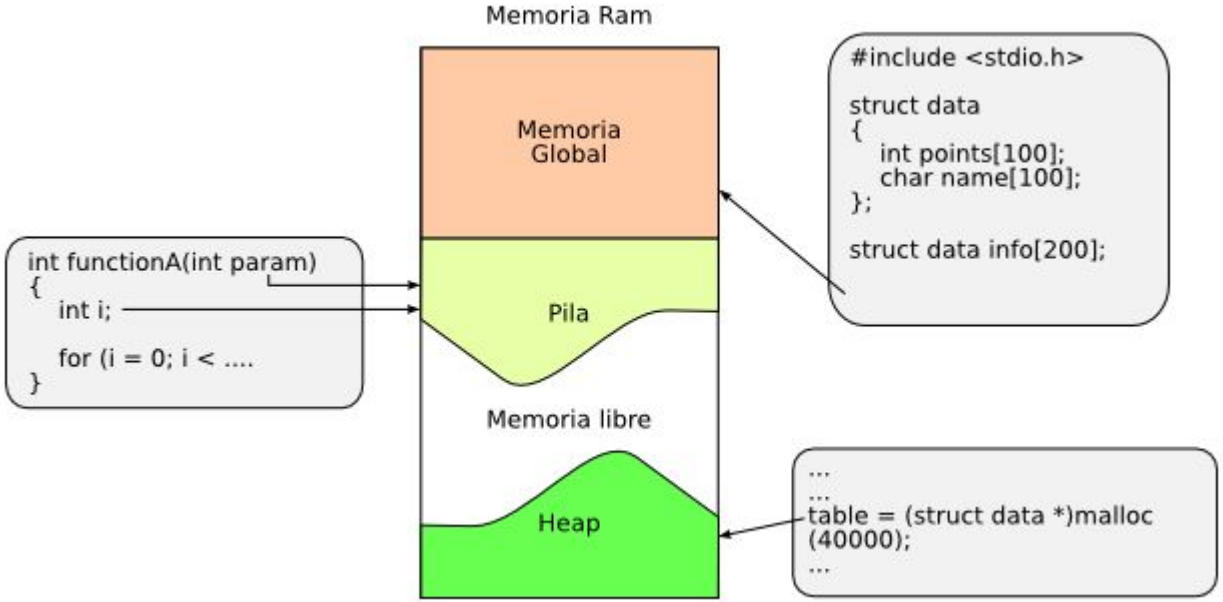
Punteros en C.

¿Cómo trabajan los punteros y memoria?

Cuando se compila un programa, este funciona con 3 tipos de memorias:

- Estática/Global:
  - Son declaradas estáticamente.
  - Variables globales usan esta región de memoria.
  - Son ubicadas y permanecen mientras exista el programa o este termine. Todas las funciones tienen acceso a variables globales.
- Automática: Son declaradas dentro de una función y son creadas cuando una función es llamada. Está restringida al alcance de la función y su ‘tiempo de vida’ al tiempo de la función que se está ejecutando.
- Dinámica: La memoria se asigna desde una pila y se puede ir liberando según la necesidad. Un puntero hace referencia a la memoria asignada. El alcance se limita al puntero que hace referencia la memoria. Esta ‘existe’ hasta que se libera.

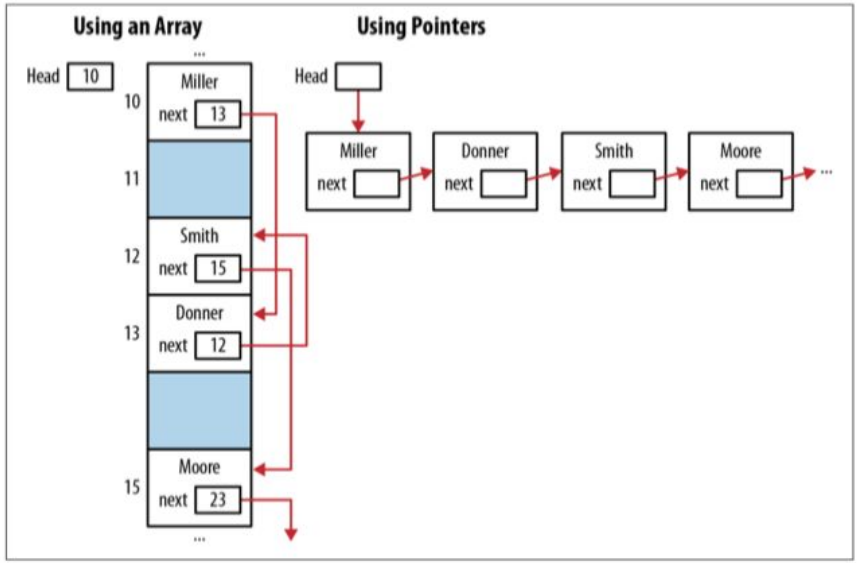
Memoria en general:



1. Memoria global. Es el área en la que están almacenadas las variables que se declaran globales o estáticas y las constantes de tipo cadena de caracteres. Es decir, en esta zona de memoria se almacenan todos aquellos datos que están presentes desde el comienzo del programa hasta que termina.
2. La pila. Es donde las variables aparecen y desaparecen en un momento puntual de la ejecución de un programa. Se utiliza principalmente para almacenar variables locales a las funciones. Estas variables tienen un ámbito reducido, sólo están disponibles mientras se está ejecutando la función en la que han sido definidas. En la pila se encuentran todas estas variables, y por tanto, en esa zona se está continuamente insertando y borrando variables nuevas.
3. El *heap*. Esta zona contiene memoria disponible para que se reserve y libere en cualquier momento durante la ejecución de un programa. No está dedicada a variables locales de las funciones como la pila, sino que es memoria denominada “dinámica” para estructuras de datos que no se saben si se necesitan, e incluso tampoco se sabe su tamaño hasta que el programa está ejecutando.

Ejemplo gráfico.

Lista linkeada.



Ejemplo continuación.

La figura anterior muestra como puede una lista de empleados linkeada puede ser visualizada con arrays o punteros.

La variable head indica que el primer elemento de la lista linkeado está en el index 10 del array. Cada elemento del array tiene una estructura que representa un empleado. El ‘next’ de la estructura tiene el índice del array del siguiente empleado. Lo que está en celeste es memoria sin uso.

Usando punteros, vemos que:

- La variable head tiene un puntero al nodo del primer empleado. Cada nodo tiene los datos del empleado y también un puntero al siguiente empleado de la lista.

Importante.

Esto impone igualmente restricciones en la cantidad de elementos que puede contener. Los punteros no tienen esta limitación porque un nuevo nodo puede ser “dinámicamente” ubicado como sea necesario.

La asignación de memoria dinámica se efectúa en C mediante el uso de punteros. Las funciones malloc y free se utilizan para asignar y liberar memoria dinámica, respectivamente. La asignación de memoria dinámica permite estructuras de datos y matrices de tamaño variable, como listas enlazadas y colas.

Declarando punteros.

Punteros a variables son declarados usando un tipo de datos seguido por un asterisco y el nombre del puntero a la variable.

int num;

int *pi;

es lo mismo que:

int* pi;

int * pi;

int*pi;

Declarando punteros.

El asterisco declara la variable como puntero. Es un símbolo de sobrecarga, se usa para multiplicar y ‘dereferenciar’un puntero.

Como vimos:

El contenido de la var pi debería ser la dirección de memoria de un variable del tipo integer.

Al no ser inicializadas pueden contener cualquier cosa.

Usando punteros...

int num;

int *pi;

pi = num; ----> ¿?

“invalid conversion to int to int” Porque la variable pi es del tipo puntero a integer y num es del tipo integer. Lo correcto es:

int num;

pi = #

Sin embargo se puede ‘castear’:

pi = (int *) num;

No tira error pero puede pasar cualquier cosa. :)

Mostrando punteros.

int num = 0; int *pi = #

printf("Address of num: %d Value: %d\n",&num, num); printf("Address of pi: %d Value: %d\n",&pi, pi);

Address of num: 4520836 Value: 0

Address of pi: 4520824 Value: 4520836

Estructuras

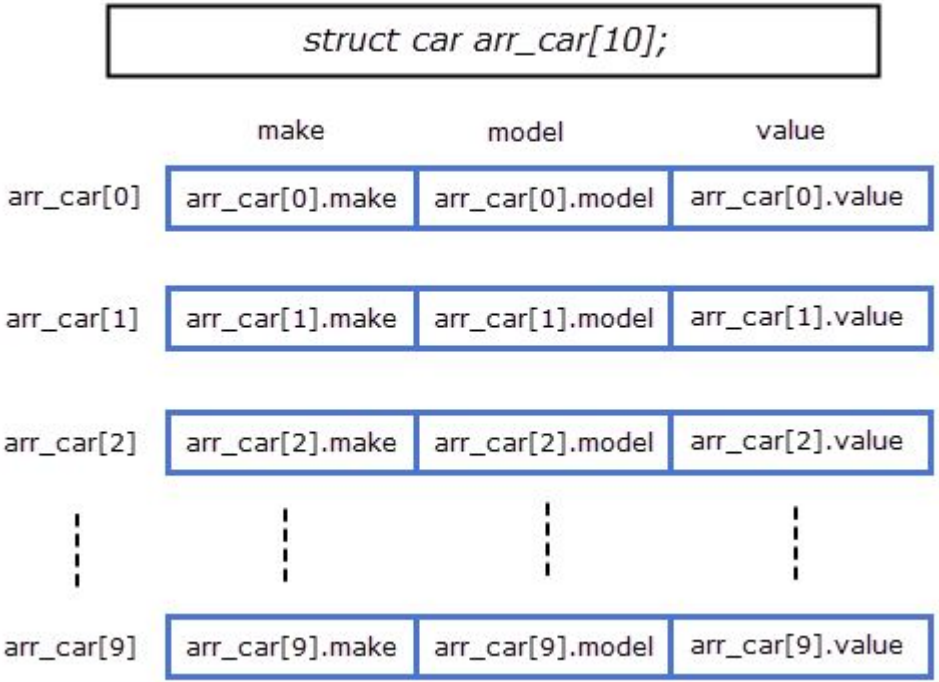
La estructura es una colección de variables de diferentes tipos de datos con un solo nombre.

Por ejemplo crear una estructura que sea ‘car’

```
struct car
{
    char make[20];
    char model[30];
    int year;
};
```

Array de estructuras.

struct car arr_car[10];



Arrays y estructuras

arr_stu[0] : apunta al elemento 0 del arreglo

arr_stu[1] : apunta al elemento 1 del arreglo.

arr_stu[0].name : refiere al nombre del elemento 0 del arreglo.

arr_stu[0].roll_no : refiere al roll_no del elemento 0 del arreglo.

arr_stu[0].marks : refiere a las marcas del elemento 0 del arreglo.

Inicializando estructuras.

```
struct car
{
    char make[20];
    char model[30];
    int year;
};
struct car arr_car[2] = {
                            {"Audi", "TT", 2016},
                            {"Bentley", "Azure", 2002}
                        };
```

Estructuras anidadas

```
1  structure tagname_1
2  {
3      member1;
4      member2;
5      member3;
6      ...
7      membern;
8
9      structure tagname_2
10     {
11         member_1;
12         member_2;
13         member_3;
14         ...
15         member_n;
16     }, var1
17
18 } var2;
```

Ejemplo

stu.p.name - se refiere al nombre de la persona

stu.p.age - se refiere a la edad de la persona.

stu.p.dob - se refiere a la fecha de nacimiento.

```
1  struct student
2  {
3      struct person
4      {
5          char name[20];
6          int age;
7          char dob[10];
8      } p ;
9
10     int rollno;
11     float marks;
12 } stu;
```

Es importante tener en cuenta que la estructura persona no existe por sí sola. No podemos declarar una variable de tipo struct person en ningún otro lugar del programa.

Puntero a estructuras

Esto declara un puntero ptr_dog que puede almacenar la dirección de la variable de tipo struct dog. Ahora podemos asignar la dirección de la variable spike a ptr_dog usando el operador &.

```c
struct dog
{
    char name[10];
    char breed[10];
    int age;
    char color[10];
};

struct dog spike;

// declaring a pointer to a structure of type struct dog
struct dog *ptr_dog
```

ptr_dog = &spike;

ptr_dog apunta a la

variable spike de estructura dog.

Operador * y .

(*ptr_dog).name - se refiere al name de dog.

(*ptr_dog).breed - se refiere a breed de dog.

Los paréntesis de * ptr_dog son necesarios porque la precedencia del operador punto (.) Es mayor que la del operador indirecto (*).

Usando operador ->

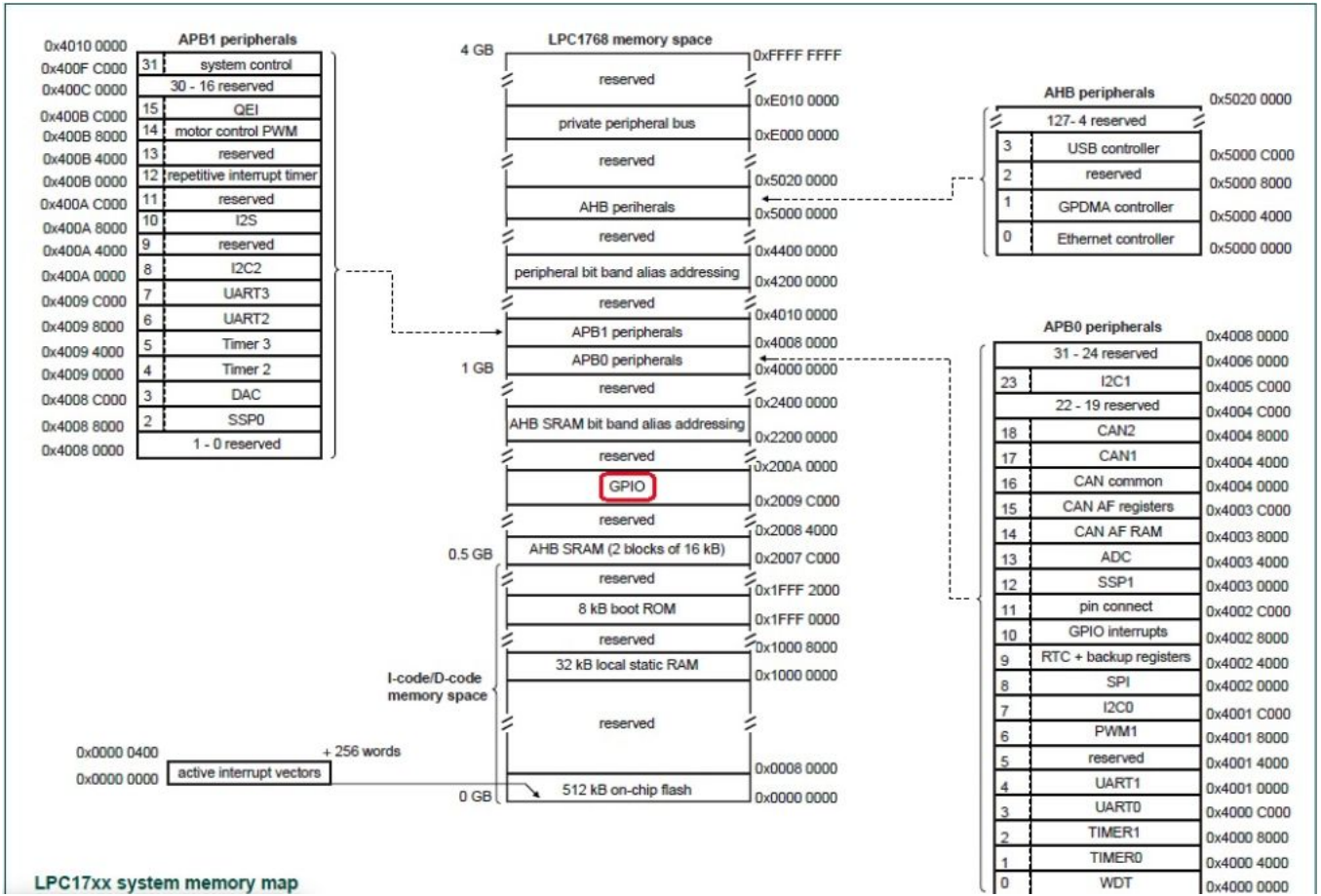
ptr_dog->name // se refiere al name de dog

ptr_dog->breed // se refiere a breed de dog

Estructura del HW y Memorias (Addresses)

El LPC176x tiene una arquitectura de 32bits.Puede mapear 2^32 posiciones (4GB). Estos 4Gb están divididos para los bloques de HW: ROM,RAM,GPIO,AHB Peripherals.

Address Map


APB1 peripherals
0x4010 0000
0x400F C000
0x400C 0000
0x400B C000
0x400B 8000
0x400B 4000
0x400B 0000
0x400A C000
0x400A 8000
0x400A 4000
0x400A 0000
0x4009 C000
0x4009 8000
0x4009 4000
0x4009 0000
0x4008 C000
0x4008 8000
0x4008 0000
31 system control
30 - 16 reserved
15 QEI
14 motor control PWM
13 reserved
12 repetitive interrupt timer
11 reserved
10 I2S
9 reserved
8 I2C2
7 UART3
6 UART2
5 Timer 3
4 Timer 2
3 DAC
2 SSP0
1 - 0 reserved
LPC1768 memory space
4 GB
0xFFFF FFFF
reserved
0xE010 0000
private peripheral bus
0xE000 0000
reserved
0x5020 0000
AHB periherals
0x5000 0000
reserved
0x4400 0000
peripheral bit band alias addressing
0x4200 0000
reserved
0x4010 0000
APB1 peripherals
0x4008 0000
APB0 peripherals
1 GB
0x4000 0000
reserved
0x2400 0000
AHB SRAM bit band alias addressing
0x2200 0000
reserved
0x200A 0000
GPIO
0x2009 C000
reserved
0x2008 4000
0.5 GB
AHB SRAM (2 blocks of 16 kB)
0x2007 C000
reserved
0x1FFF 2000
8 kB boot ROM
0x1FFF 0000
reserved
0x1000 8000
32 kB local static RAM
0x1000 0000
I-code/D-code memory space
reserved
0x0000 0400
+ 256 words
0x0000 0000
active interrupt vectors
0x0008 0000
0 GB
512 kB on-chip flash
0x0000 0000
AHB peripherals
0x5020 0000
127- 4 reserved
3 USB controller
0x5000 C000
2 reserved
0x5000 8000
1 GPDMA controller
0x5000 4000
0 Ethernet controller
0x5000 0000
APB0 peripherals
0x4008 0000
31 - 24 reserved
0x4006 0000
23 I2C1
0x4005 C000
22 - 19 reserved
0x4004 C000
18 CAN2
0x4004 8000
17 CAN1
0x4004 4000
16 CAN common
0x4004 0000
15 CAN AF registers
0x4003 C000
14 CAN AF RAM
0x4003 8000
13 ADC
0x4003 4000
12 SSP1
0x4003 0000
11 pin connect
0x4002 C000
10 GPIO interrupts
0x4002 8000
9 RTC + backup registers
0x4002 4000
8 SPI
0x4002 0000
7 I2C0
0x4001 C000
6 PWM1
0x4001 8000
5 reserved
0x4001 4000
4 UART1
0x4001 0000
3 UART0
0x4000 C000
2 TIMER1
0x4000 8000
1 TIMER0
0x4000 4000
0 WDT
0x4000 0000
LPC17xx system memory map

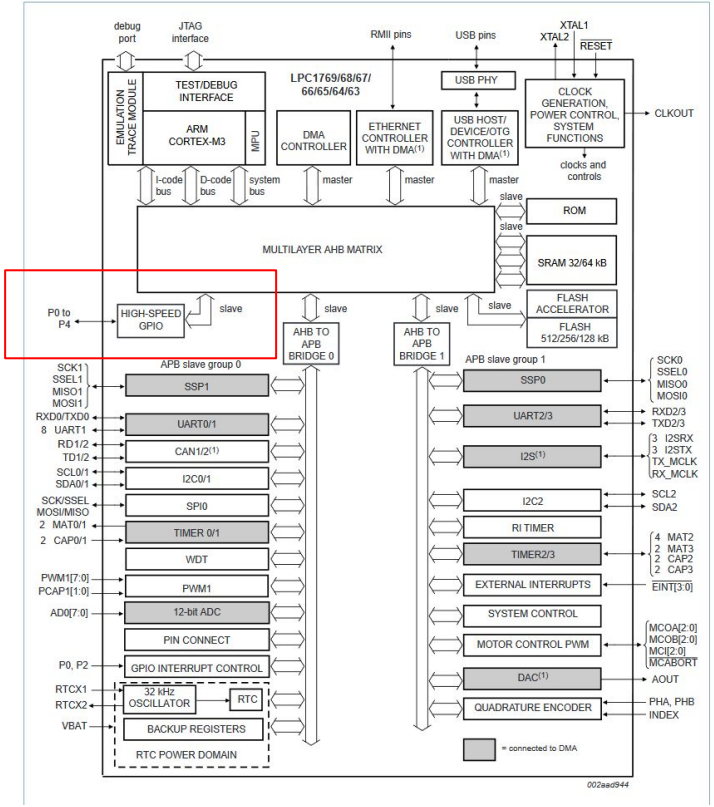
GPIO LPC1769

1. Entender qué son los GPIOS.
    a. Esquema conceptual.
2. ¿Cómo se configuran? Manual.
3. Ejemplo.

¿GPIO qué son?

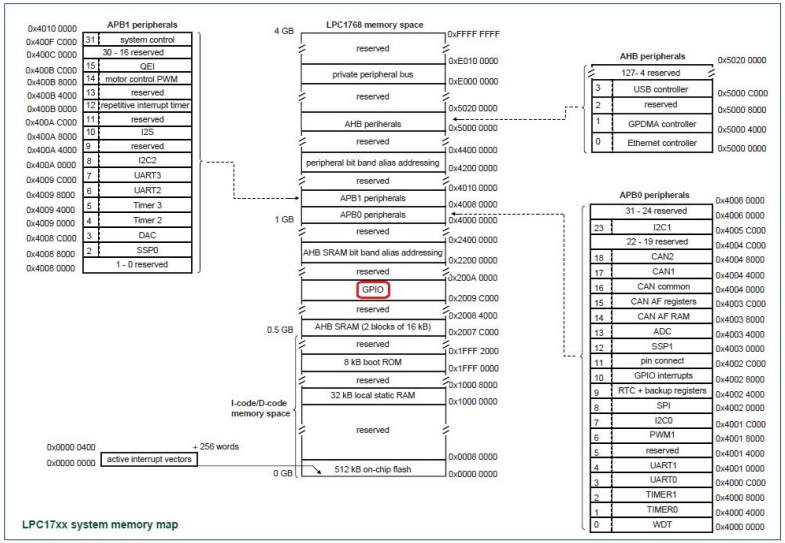
General Purpose Input Output.
O sea Entradas- Salidas, para usarlas como queramos.

Los pines pueden ser dinámicamente configurados como entradas o salidas.



¿Cómo se configuran?

1. Están mapeados en estas memorias:
   0x2009 C000 - 0x2009 FFFF.
2. Hay que decirle al PINSEL que se va a usar como GPIO y no como otra cosa. Porque comparten layout de pines con otras funciones. Entonces hay un multiplexor que configurar para que configure el GPIO a las memorias.
3. Se hace con el PINSEL registers.
4. Hay que hacerlo antes de ser activado. Si no queda como indefinido.



LPC17xx system memory map

Table 75. Summary of PINSEL registers

Register	Controls	Table
PINSEL0	P0[15:0]	Table 80
PINSEL1	P0 [31:16]	Table 81
PINSEL2	P1 [15:0] (Ethernet)	Table 82
PINSEL3	P1 [31:16]	Table 83
PINSEL4	P2 [15:0]	Table 84
PINSEL5	P2 [31:16]	not used
PINSEL6	P3 [15:0]	not used
PINSEL7	P3 [31:16]	Table 85
PINSEL8	P4 [15:0]	not used
PINSEL9	P4 [31:16]	Table 86
PINSEL10	Trace port enable	Table 87

Para GPIO ¿qué se escribe en PINSELX

Si el programa no usó o cambió las funciones de PINSEL, el estado por defecto es el 00 o sea el GPIO.

Si no, vamos a tener que leer el estado y reconfigurarlo.

Table 76. Pin function select register bits

PINSEL0 to PINSEL9 Values	Function	Value after Reset
00	Primary (default) function, typically GPIO port	00
01	First alternate function	
10	Second alternate function	
11	Third alternate function	

GPIO PINMODE

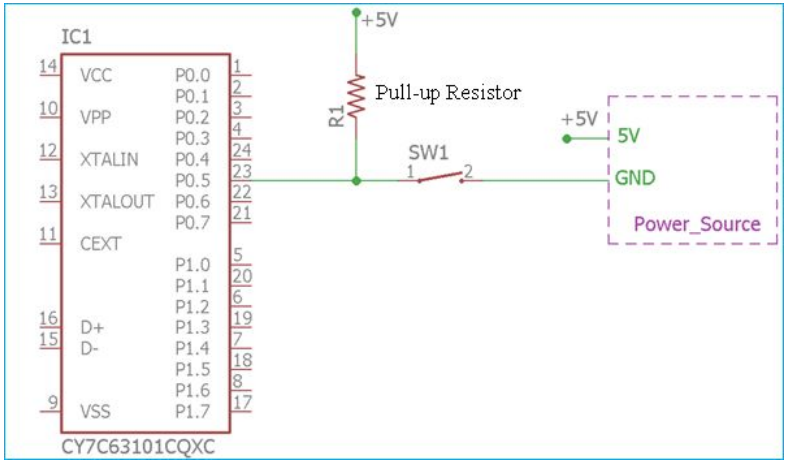
Configura los modos de entrada de todos los puertos.

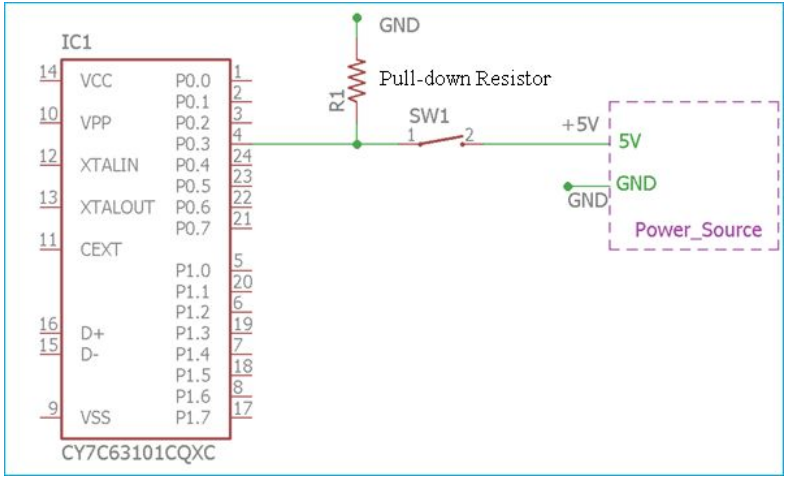
1. On Chip pull up. 0b00
2. Repeater mode. 0b01
3. Ni Pull up ni Pull down. 0b10
4. Pull down. 0b11

Table 77. Pin Mode Select register Bits

PINMODE0 to PINMODE9 Values	Function	Value after Reset
00	Pin has an on-chip pull-up resistor enabled.	00
01	Repeater mode (see text below).	
10	Pin has neither pull-up nor pull-down resistor enabled.	
11	Pin has an on-chip pull-down resistor enabled.	

Pull up - Pull down - Repeater mode





Cuando un GPIO está configurado en modo repetidor, el pull-up se habilita cuando el pin se coloca alto y el pull-down se habilita cuando el pin se coloca bajo. Si nada cambia, éste conservará su último estado conocido. Se usa para evitar que el pin quede flotante y evitar corrientes entrantes y salientes de leakage.

Importante:

Las resistencias on-chip pull-up/pull-down se pueden seleccionar para cada pin independientemente de la función seleccionada para ese pin con la excepción de los pines I2C, I2C0 y USB

Ver tablas de 121-125

Ejemplo:

8.5.6 Pin Function Select Register 7 (PINSEL7 - 0x4002 C01C)

The PINSEL7 register controls the functions of the upper half of Port 3. The direction control bit in the FIO3DIR register is effective only when the GPIO function is selected for a pin. For other functions, direction is controlled automatically.

Table 85. Pin function select register 7 (PINSEL7 - address 0x4002 C01C) bit description

PINSEL7	Pin name	Function when 00	Function when 01	Function when 10	Function when 11	Reset value
17:0	-	Reserved	Reserved	Reserved	Reserved	0
19:18	P3.25[1]	GPIO Port 3.25	Reserved	MAT0.0	PWM1.2	00
21:20	P3.26[1]	GPIO Port 3.26	STCLK	MAT0.1	PWM1.3	00
31:22	-	Reserved	Reserved	Reserved	Reserved	0

[1] Not available on 80-pin package.

8.5.7 Pin Function Select Register 9 (PINSEL9 - 0x4002 C024)

The PINSEL9 register controls the functions of the upper half of Port 4. The direction control bit in the FIO4DIR register is effective only when the GPIO function is selected for a pin. For other functions, direction is controlled automatically.

Table 86. Pin function select register 9 (PINSEL9 - address 0x4002 C024) bit description

PINSEL9	Pin name	Function when 00	Function when 01	Function when 10	Function when 11	Reset value
23:0	-	Reserved	Reserved	Reserved	Reserved	00
25:24	P4.28	GPIO Port 4.28	RX_MCLK	MAT2.0	TXD3	00
27:26	P4.29	GPIO Port 4.29	TX_MCLK	MAT2.1	RXD3	00
31:28	-	Reserved	Reserved	Reserved	Reserved	00

8.5.8 Pin Function Select Register 10 (PINSEL10 - 0x4002 C028)

Only bit 3 of this register is used to control the Trace function on pins P2.2 through P2.6.

Table 87. Pin function select register 10 (PINSEL10 - address 0x4002 C028) bit description

Bit	Symbol	Value	Description	Reset value
2:0	-	-	Reserved. Software should not write 1 to these bits.	NA
3	GPIO/TRACE		TPIU interface pins control.	0
		0	TPIU interface is disabled.	
		1	TPIU interface is enabled. TPIU signals are available on the pins hosting them regardless of the PINSEL4 content.	
31:4	-	-	Reserved. Software should not write 1 to these bits.	NA

Resumen:


LPC176x GPIO
PINSEL
FIODIR
FIOSET, FIOCLR, FIOPIN
Selects the Pin Function
Configures Pin Direction
Access the Port Pin

Ejemplo 1

```
#include <lpc17xx.h>

void delay(unsigned int count)
{
  unsigned int i,j;
  for(i=0;i<count;i++)
  {
    for(j=0;j<5000;j++);
  }
}

/* start the main program */
void main()
{
    SystemInit();                   //Clock and PLL configuration
    LPC_PINCON->PINSEL4 = 0x000000; //Configure the Pins for GPIO;
    LPC_GPIO2->FIODIR = 0xffffffff;  //Configure the PORT pins as OUTPUT;

  while(1)
    {

     /* Turn ON all the leds and wait for one second */
       LPC_GPIO2->FIOSET = 0xffffffff;     // Make all the Port pins as high
       delay(1000);

     /* Turn OFF all the LEDs and wait for one second */
       LPC_GPIO2->FIOCLR = 0xffffffff;     // Make all the Port pins as low
       delay(1000);
    }
}
```

¿Qué hace?

Parpadea un led.

Configura los pines del PORT2 como GPIO usando PINSEL.

Configura FIODIR como salida.

EL led se enciende con un pulso en alto que se hace escribiendo FIOSET en 1

Después de un delay el led se apaga con un pulse en bajo que se hace escribiendo el registro FIOCLR en 1.

Ejemplo 2.

```
#include <lpc17xx.h>

void delay(unsigned int count)
{
  unsigned int i,j;
  for(i=0;i<count;i++)
  {
    for(j=0;j<5000;j++);
  }
}

/* start the main program */
void main()
{
    SystemInit();                   //Clock and PLL configuration
    LPC_PINCON->PINSEL4 = 0x000000; //Configure the Pins for GPIO;
    LPC_GPIO2->FIODIR = 0xffffffff;  //Configure the PORT pins as OUTPUT;

  while(1)
    {

     /* Turn ON all the leds and wait for one second */
       LPC_GPIO2->FIOPIN = 0xffffffff;     // Make all the Port pins as high
       delay(1000);

     /* Turn OFF all the LEDs and wait for one second */
       LPC_GPIO2->FIOPIN = 0x00;     // Make all the Port pins as low
       delay(1000);
    }
}
```

¿Qué hace?

No se los voy a anotar.

Ejercicio propuesto.

Generar un archivo header y c para control de puerto GPIO.

¿Cómo sería?

1-Crear funciones de configuración. Paso el puerto, paso el pin, y entrada-salida.

2- Crear funciones de seteo.

3- Crear funciones de status.

4- Probarlo.

Interrupciones

- GPIO.
- EINTx

Ver slides separadas

Systick timer

El System Tick Timer es un bloque integrado dentro del core del micro M3. El System Tick Timer está diseñado para generar una interrupción configurable por el usuario para que la use un sistema operativo u otro software de administración del sistema.

El System Tick Timer es un temporizador de **24 bits** que cuenta hasta cero y genera una interrupción. La intención es proporcionar un intervalo de tiempo fijo entre interrupciones.

Systick timer config

Registers

Register	Address	Description
STCTRL	0xE000 E010	System Timer Control and status register
STRELOAD	0xE000 E014	System Timer Reload value register
STCURR	0xE000 E018	System Timer Current value register
STCALIB	0xE000 E01C	System Timer Calibration value register

STCTRL					
31-17	16	15-3	2	1	0
RESERVED	COUNTFLAG	RESERVED	CLKSOURCE	TICKINT	ENABLE

STCURR	
31-24	23 - 0
RESERVED	CURRENT

Timers en LP1769

- Tiene 4 bloques de Timer de 32 bit.
- Cada timer puede ser usado para ‘interrumpir’ a un tiempo definido o si se configura como contador (counter) puede ser usado para demodular señales PWM.
- Cada timer tiene su registro TC y PR (timer counter y Prescale Register)asociado.

TC y PR registers.

- Cuando el timer fue reseteado y habilitado (enabled) el TC se pone en cero y se incrementa según:
  - (1 + PR) ciclos de reloj.
  - PR es prescale register con el valor que hayamos puesto.
  - Cuando TC alcanza su máximo valor, se reseta a 0 y comienza a contar de nuevo.

Match Registers

- Contiene un valor especificado por usuario.
- Cuando el contador inicia, cada vez que TC es incrementado el valor de TC es comparado con este registro.
- Si TC == Match Register, se puede generar una interrupción o resetear el contador.
- Se pueden usar para:
  - Detener el contador y disparar interrupción.
  - Resetear el timer e interrumpir.
  - Seguir contando e interrumpir cuando matchee.

Capture Register.

- Se usa para capturar señales de entrada.
- Cuando una transición ocurre en el pin de Captura, se puede capturar el valor de TC en cualquiera de los 4 registros o generar una interrupción.
- Se usa principalmente para demodulación PWM.
- Lo vemos más adelante.:)

Registros para configurar TIMER

Mirando la struct LPC_TIMx con x de 0-3.

LPC_TIM0 tiene:

- **PR:** Almacena el valor de Prescale Counter.
- **PC:** Se incrementa en cada PCLK(pulso de clock). Controla la resolución del timer. Cuando PC alcanza el valor in PR , PC se pone en 0 and Timer Counter se incrementa 1. Por eso cuando PR=0 el TC se incrementa cada pulso de clock. Si PR=9 TC se incrementa en el ciclo 10 de PCLK.
- **TC:** Timer Counter Register (32 bit) –Registro principal. Timer Counter se incrementa en 1 cuando PC alcanza su máximo valor especificado en PR. Si timer no se resetea explicitamente cuando interrumpe va a trabajar como free running counter. O sea cuenta siempre hasta que se resetea a 0 cuando alcance 0xFFFFFFFF.
- **TCR: Timer Control Register:** Se usa para habilitar, deshabilitar o resetear el timer. Con bit 0 = 1 el timer se habilita con bit 0= 0 se desactiva. Cuando bit 1 = 1 ->TC y PC se ponen en cero simultáneamente juntos en el siguiente flanco ascendente de PCLK. Todos los otros bits son reservados.
- **CTCR:** Count Control register – Selecciona entre Timer or Count mode. CTCR = 0x0 se selecciona Timer Mode.
- **MCR: Match Control register –** Controla lo que va a hacer cuando el valor en MR coincide con TC. Bits 0,1,2 son para MR0 , Bits 3,4,5 para MR1.
    - Para MR0:
        - bit0:Interrumpe cuando MR0 = TC. Interrupción se habilita con 1 y se deshabilita con 0.
        - bit1: Resetea a MR0 cuando se escribe 1, TC se resetea cuando coincide con MR0. Se desactiva con 0.
        - bit2: Se detiene en MR0. Cuando se escribe 1 -> TC y PC se detienen cuando MR0 coincide con TC.
- **IR:Interrupt Register –** Contiene los flags de interrupción para 4 matchs and 4 capture interrupciones. Bit0 to Bit3 son para MR0 a MR3 respectivamente. And similarmente los siguientes 4 para interrupciones CR0-3. Cuando una interrupción se levanta, el correspondiente bit en IR se pone en 1 y 0 si no. Escribiendo 1 al correspondiente bit se resetea la interrupción – which is used to acknowledge the completion of the corresponding ISR execution.
- **EMR: External Match Register –** Nos da el estado y control de External Match Output Pins. Los primeros 4 bits son EM0 y EM3. Los siguientes 8 bits son para EMC0 a EMC3 en pares de dos.
    - Bit 0 – EM0: External Match 0. Cuando coincide TC y MR0, dependiendo de los bits[5:4] por ejemplo: EMC0 de este registro puede toglear, ponerse en alto, ponerse en bajo o hacer nada. Maneja MATx.0 con x es el número del timer.
    - Bits[5:4] – EMC0: External Match 0. Seleccionan la funcionalidad como sigue:
        - 0 No hace nada.
        - 1 Limpia el correspondiente External Match output poniendolo en 0. MATx.m pin puesto en LOW
        - 2 Setea 1 en External Match output. MATx.m pin puesto en HIGH.
        - 3 Toglea Externa Match output.
    - DE igual forma Bits[7:6] – EMC1, Bits[9,8] – EMC2, Bits[11:10] – EMC3.
    - Timer0/1/3 tienen sólo 2 Match outputs Pines mientras que Timer2 tiene 4 Match output Pines. De ahí que EM2, EM3 y EMC2,EMC3 no aplican para Timer0/1/3.

Configurando el timer:

Secuencia recomendada:

1. Poner un valor en `LPC_TIMx->CTCR`
2. Definir `LPC_TIMx->PR`
3. Cargar en Match Register(s) de ser necesario.
4. Cargar el valor in `LPC_TIMx->MCR` si usa Match registers / Interrumpe.
5. Reset timer – PR y TC.
6. Cargar `LPC_TIMx->TCR` to `0x01` para habilitar el timer.
7. Reset `LPC_TIMx->TCR` con `0x00` para desactivar el Timer si se quiere.

Configurando el clock para los timers.

Se usa PCLKSEL0 & PCLKSEL.

- Timer0 bits[3:2] usa PCLKSEL0 .
- Timer1 bits[5:4] usa PCLKSEL0.
- Timer2 bits[13:12] usa PCLKSEL1.
- Timer3 bits[15:14] usaPCLKSEL1.

Podemos 4 diferentes divisores para el CCLK que va al PCLK.

- [00] – PCLK = CCLK/4 (Default o reset)
- [01] – PCLK = CCLK
- [10] – PCLK = CCLK/2
- [11] – PCLK = CCLK/8

Cálculos:

The delay or time required for 1 clock cycle when PCLK = 'X' Mhz is given by :

$$T_{PCLK} = \frac{1}{PCLK_{Hz}} = \frac{1}{X * 10^6} \text{ Seconds}$$

It is also the maximum resolution Timer block can proved at a given PCLK frequency of X Mhz. The general formula for Timer resolution at X Mhz PCLK and a given value for prescale (PR) is as given below:

$$T_{RES} = \frac{PR+1}{PCLK_{Hz}} = \frac{PR+1}{X * 10^6} \text{ Seconds}$$

Hence, we get the **Formula for Prescaler (PR) for required Timer resolution (T_{RES} in Secs)** at given **PCLK(in Hz)** frequency as:

$$PR = (PCLK_{Hz} * T_{RES}) - 1$$

$$PR = ((X * 10^6) * T_{RES}) - 1$$

Note that here, the resolution is also the time delay required to increment TC by 1.

Hence, Prescaler value for 1 micro-second resolution/ 1us time delay at 25 Mhz PCLK is,

$$PR_{1us} = (25Mhz * 1uS) -1 = (25*10^6 * 10^{-6}) -1 = 24$$

Prescale for 1 mS (milli-second) resolution at 25Mhz PCLK is,

$$PR_{1ms} = (25Mhz * 1ms) -1 = (25*10^6 * 10^{-3}) -1 = 24999$$

The maximum resolution of all the timers is 10 nano-seconds when using PCLK = CCLK = 100Mhz and PR=0 which is as follows,

$$T_{MAXRES} = [1 / (100Mhz)] = 10ns$$

Funciones

```
#define PRESCALE (25000-1)


void initTimer0(void)
{
    /* PLL0 configuró CCLK = 100Mhz and PCLK = 25Mhz.*/
    LPC_SC->PCONP |= (1<<1); //Power up TIM0. By default TIM0 and TIM1 habilitados
    LPC_SC->PCLKSEL0 &= ~(0x3<<3); //Set PCLK for timer = CCLK/4 = 100/4 (default)
    LPC_TIM0->CTCR = 0x0;
    LPC_TIM0->PR = PRESCALE; //Increment TC at every 24999+1 clock cycles
    //25000 clock cycles @25Mhz = 1 mS

    LPC_TIM0->TCR = 0x02; //Reset Timer
}
```

Funciones

```
void delayMS(unsigned int milliseconds) // Timer0
{
    LPC_TIM0->TCR = 0x02; //Reset Timer

    LPC_TIM0->TCR = 0x01; //Enable timer

    while(LPC_TIM0->TC < milliseconds); //Espera hasta que timer alcance el tiempo.

    LPC_TIM0->TCR = 0x00; //Disable timer
}
```

Ejemplo de Timer con Match Outputs

```
#include <lpc17xx.h>

#define PRESCALE (25000-1) //25000 PCLK clock cycles to increment TC by 1

void initTimer0(void);

int main (void) {
		//SystemInit(); //called by Startup Code before main(), hence no need to call again.
		initTimer0(); //Initialize Timer0
		LPC_PINCON->PINSEL3 |= (1<<24) | (1<<25) | (1<<27) | (1<<26); //config MAT0.0(P1.28) and MAT0.0(P1.29) outputs
		initTimer0();


		while(1)
		{
			//Idle loop
		}
}
```

```
void initTimer0(void)
{
	/* PLL0 configurado con CCLK = 100Mhz and PCLK = 25Mhz.*/
	LPC_SC->PCONP |= (1<<1); //Power up TIM0. By default TIM0 and TIM1 habilitados
	LPC_SC->PCLKSEL0 &= ~(0x3<<3); //Set PCLK for timer = CCLK/4 = 100/4 (default)

	LPC_TIM0->CTCR = 0x0;
	LPC_TIM0->PR = PRESCALE; //Increment LPC_TIM0->TC at every 24999+1 clock cycles
	//25000 clock cycles @25Mhz = 1 mS

	LPC_TIM0->MR0 = 500; //
	LPC_TIM0->MCR = (1<<1); //Reset on MR0 Match
	LPC_TIM0->EMR |= (1<<7) | (1<<6) | (1<<5) | (1<<4); //Toggle Match output for MAT0.0(P1.28),
MAT0.1(P1.29)

	LPC_TIM0->TCR = 0x02; //Reset Timer
	LPC_TIM0->TCR = 0x01; //Enable timer
}
```

Ejemplo interrupción.

Timer con Captura

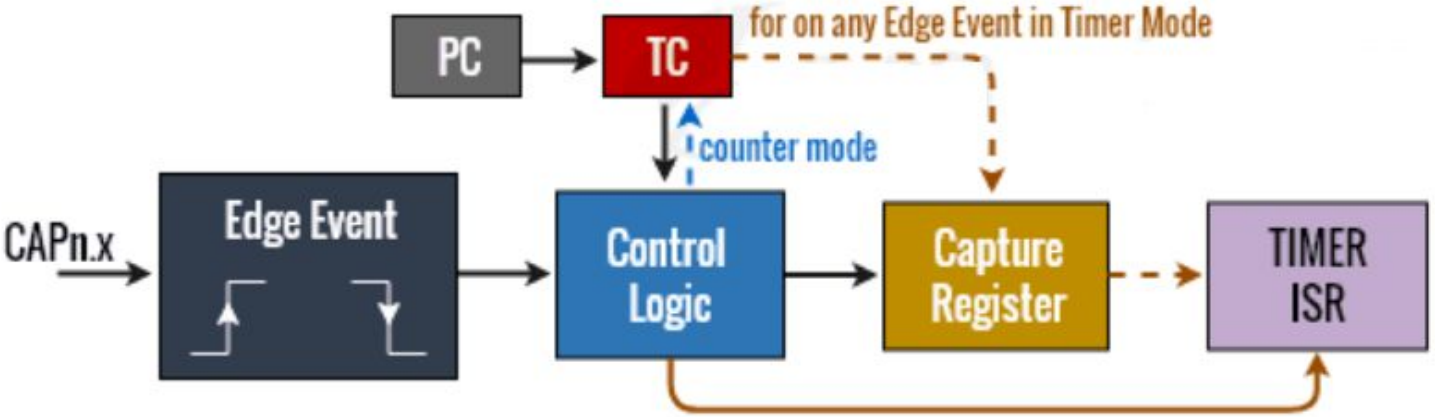
Cada timer tiene 2 canales de captura: CAPn.0 y CAPn.1 con n=número del timer.

Usando estos canales podemos “sacar una foto”del valor actual del TC cuando un flanco de la señal es detectado.

Timer0		Timer1		Timer2		Timer3	
Ch.	Pin	Ch.	Pin	Ch.	Pin	Ch.	Pin
CAP0.0	P1.26	CAP1.0	P0.18	CAP2.0	P0.4	CAP3.0	P0.23
CAP0.1	P1.27	CAP1.1	P0.19	CAP2.1	P0.5	CAP3.1	P0.24

Cómo funcionan las capturas:

- Timer Mode.
- Counter Mode.



Timer mode and Counter Mode.

1-En **Timer Mode** el clock del micro es usado como fuente para incrementar el timer counter (TC) cada ‘PR+1’ clock cycles (Prescale register). Cuando viene una señal ya sea por flanco asc o desc o ambos se detecta ese evento y se hace un timestamp del valor actual del TC y se carga en Capture Register(CRx) y puede generar una interrupción cuando Capture Register se carga con un valor nuevo. Esto se hace con CCR.

2- **Counter Mode**, una señal externa es usada para incrementar TC cada vez que un flanco (asc desc o ambos) es detectado. se configura usando CTCR. Los bits correspondientes para Capture deben ser seteado en cero en el CCR.

Registros para captura.

1) CCRx – Capture Control Register: Selecciona el tipo de flanco. (asc desc o ambos) que es usado para Capture Registers (CR0-CR1) y puede una generar interrupción cuando ocurre.

Para el CCRx:

- **Bit 0:** Capture on CAPn.0 rising edge. When set to 1, a transition of 0 to 1 on CAPn.0 will cause CR0 to be loaded with the contents of TC. Disabled when 0.
- **Bit 1:** Capture on CAPn.0 falling edge. When set to 1, a transition of 1 to 0 on CAPn.0 will cause CR0 to be loaded with the contents of TC. Disabled when 0.
- **Bit 2:** Interrupt on CAPn.0 event. When set to 1, a CR0 load due to a CAPn.0 event will generate an interrupt. Disabled when 0.

Similarly bits 3-5, son para **CR1**.

Seteando en 1 ambos flancos queda con doble trigger.

2) CR0 & CR1 – Capture Registers: Cada registro de captura está asociado con un 'Capture Pin'. Dependiendo de lo que pongas en CCR, CRn puede ser cargado con el valor actual de TC cuando ocurra un evento específico..

3) CTCR Count Control Register: Selecciona entre timer mode o counter mode

Bits[1:0] – Used to select Timer mode or which Edges can increment TC in counter mode.

- [00]: Timer Mode. PC is incremented every Rising edge of PCLK.
- [01]: Counter Mode. TC is incremented on Rising edges on the CAP input selected by Bits[3:2].
- [10]: Counter Mode. TC is incremented on Falling edges on the CAP input selected by Bits[3:2].
- [11]: Counter Mode. TC is incremented on Both edges on the CAP input selected by Bits[3:2].

Bits[3:2] – Count Input Select. Sólo aplica cuando CTCR bits son distintos de [00].

- [00]: Used to select CAPn.0 for TIMERn as Count input.
- [01]: Used to select CAPn.1 for TIMERn as Count input.
- [10] & [11]: Reserved.

Ejemplo:

```c
#include <lpc17xx.h>
#include <stdio.h> /


unsigned int period = 0;
unsigned int previous = 0;
unsigned int current = 0 ;
int limitFlag = 0;
#define TIMER_RES 0.02 //.
#define PRESCALE (25-1)
int main(void)
{
    SystemInit();

    initTimer0();

    /*Using CCLK = 100Mhz y PCLK_TIMER2 = 100Mhz.*/
    LPC_SC->PCONP |= (1<<22); //
    LPC_SC->PCLKSEL1 |= (1<<12);
    LPC_PINCON->PINSEL0 |= (1<<9) | (1<<8);
    LPC_TIM2->CTCR = 0x0;
    LPC_TIM2->PR = 1;
    LPC_TIM2->TCR = 0x02; /
    LPC_TIM2->CCR = (1<<0) | (1<<2);
    LPC_TIM2->TCR = 0x01;

    NVIC_EnableIRQ(TIMER2_IRQn);

    while(1)
    {
        if(limitFlag)
        {
            NVIC_EnableIRQ(TIMER2_IRQn);
            delayMS(500);
        }
        else
        {
            delayMS(500);
        }
    }


}
```

```c
void TIMER2_IRQHandler(void)
{
    LPC_TIM2->IR |= (1<<4);
    current = LPC_TIM2->CR0;
    if(current < previous)
    {
        period = 0xFFFFFFFF + current - previous;
    }
    else
    {
        period = current - previous;
    }
    previous = current;

    if(period < 60)
    {
        NVIC_DisableIRQ(TIMER2_IRQn);
        limitFlag = 1;
    }
    else limitFlag = 0;
}

void initTimer0(void) //PCLK = 25Mhz!
{
    LPC_TIM0->CTCR = 0x0;
    LPC_TIM0->PR = PRESCALE;
    LPC_TIM0->TCR = 0x02;
}

void delayMS(unsigned int milliseconds)
{
    delayUS(milliseconds * 1000);
}
```

Ejemplo de Match.

```c
#include "LPC17xx.h"

//void confGPIO(void); // Prototipo de la funcion de conf. de puertos
void confTimer(void);
int main(void) {

//  confGPIO();
    confTimer();
    while(1){}
    return 0;
}

/*void confGPIO(void){
    LPC_GPIO0->FIODIR |= (1<<22);
    return;
}
*/

void confTimer(void){
    LPC_SC->PCONP       |= (1<<1);  // pag. 65
    LPC_SC->PCLKSEL0    |= (1<<2);  // pag. 59
    LPC_PINCON->PINSEL3 |= (3<<24); // pag. 120
    LPC_TIM0->EMR       |=(3<<4);   // pag. 509
    LPC_TIM0->MR0        = 70000000;//
    LPC_TIM0->MCR       |= (1<<1);       // pag. 507
    LPC_TIM0->MCR       &= (1<<0);       // pag. 507
    LPC_TIM0->TCR        = 3;        // pag. 505
    LPC_TIM0->TCR       &= ~(1<<1);
    // NVIC_EnableIRQ(TIMER0_IRQn);
    return;
}

/*
void TIMER0_IRQHandler(void) //ISR del timer0
{
static uint8_t i = 0;
if (i==0){
LPC_GPIO0->FIOSET = (1<<22);
i = 1;
}
else if (i==1){
LPC_GPIO0->FIOCLR = (1<<22);
i = 0;
}
LPC_TIM0->IR|=1; //Limpia bandera de interrupci?n
return;
}
 */
```

Ejercicio Propuesto

1. Generar con timer0 una señal de freq. variable.
2. Usando el capture “medir”el periodo usando otro timer.
3. Prender leds tipo vúmetro según la frecuencia.
4. Con un pulsador cambiar la frecuencia de pasos de 100khz. Actualizar el vúmetro.
5.

ADC en LPC1769

Analog to Digital Converter.

Resolución 12 bits SAR multiplexado en 8 canales.

La referencia de tensión en el ADC es medida entre VREFN a VREFP, o sea este sería el rango de conversión. Casi siempre VREFP está conectada a VDD y VREFN está conectada a GND.

Como el LPC1769 está conectado a 3.3 volts,este será la tensión de referencia del ADC.

La resolución del ADC = 3.3/(2^{12}) = 3.3/4096 =0.000805 = **0.8mV**

Frecuencia de Operación: **13Mhz.**

Tasa de conversión (convertion rate) = **200Khz**. Toma 65 ciclos de reloj en obtener el dato. 13000Khz/65 =200Khz.

ADC on LPC1769 Channels

Adc Channel	Port Pin	Pin Functions	Associated PINSEL Register
AD0	P0.23	0-GPIO, 1-**AD0[0]**, 2-I2SRX_CLK, 3-CAP3[0]	14,15 bits of PINSEL1
AD1	P0.24	0-GPIO, 1-**AD0[1]**, 2-I2SRX_WS, 3-CAP3[1]	16,17 bits of PINSEL1
AD2	P0.25	0-GPIO, 1-**AD0[2]**, 2-I2SRX_SDA, 3-TXD3	18,19 bits of PINSEL1
AD3	P0.26	0-GPIO, 1-**AD0[3]**, 2-AOUT, 3-RXD3	20,21 bits of PINSEL1
AD4	P1.30	0-GPIO, 1-VBUS, 2- , 3-**AD0[4]**	28,29 bits of PINSEL3
AD5	P1.31	0-GPIO, 1-SCK1, 2- , 3-**AD0[5]**	30,31 bits of PINSEL3
AD6	P0.3	0-GPIO, 1-RXD0, 2-**AD0[6]**, 3-	6,7 bits of PINSEL0
AD7	P0.2	0-GPIO, 1-TXD0, 2-**AD0[7]**, 3-	4,5 bits of PINSEL0

ADC en LPC1769 Registros

Registro	Descripción
ADCR	A/D Control Register: Configura el ADC.
ADGDR	A/D Global Data Register: Contiene el bit de ADC done (que ya completó el ciclo) y el resultado de la última conversión.
ADINTEN	A/D Interrupt Enable Register.
ADDR0 - ADDR7	A/D Channel Data Register: Contiene el valor reciente del ADC para cada respectivo canal.
ADSTAT	A/D Status Register: Contiene el DONE y el flag de OVERRUN para todos los canales del ADC.

Registro ADCR

31:28	27	26:24	23:22	21	20:17	16	15:8	7:0
Reserved	EDGE	START	Reserved	PDN	Reserved	BURST	CLCKDIV	SEL

Descripción:

Bit 7:0 – SEL : Channel Select

Seleccionan un canal particular para la conversión del ADC. Un bit selecciona cada canal (por eso hay 7). Seteando Bit-0 samplea el AD0[0]. De esta forma seteando bit-7 will samplea el AD0[7].

Bit 15:8 – CLCKDIV : Clock Divisor

El APB clock (PCLK_ADC0) está dividido por este valor +1 para producir el clock del conversor A/D. Debe ser menor o igual a 13 Mhz.

Bit 16 – BURST: Este bit se usa para setear la configuración en modo BURST. Si este bit se pone en alto el módulo del ADC hará la conversión para todos los canales que estén seleccionado in SEL bits. Poniéndolo en bajo desactiva el modo BURST.

Bit 21 – PDN : Power Down Mode: Poniendo este bit en alto saca del modo power down al ADC haciéndolo operacional. Poniéndolo en cero apaga el ADC.

Bit 24:26 – START: Estos bits controlan la conversión del ADC. En modo burst estos bits están en 0.

000 - Conversion Stopped

001- Start Conversion Now

Ver página 588 para los otros valores. Se puede comenzar a convertir por pin externo, entonces podemos elegir flanco

Bit 27 - EDGE: This bit is significant only when the START field contains 010-111. It starts conversion on selected CAP or MAT input.

0 - On Falling Edge

1 - On Rising Edge

Registro AD0GDR

	31	27	26:24	23:16	15:4	3:0
Bit 15:4 - RESULT	DONE	OVERRUN	CHN	Reserved	RESULT	Reserved

Contiene el valor de la conversión en el canal que se seleccionó con **ADCR.SEL**

Este valor hay que leerlo una vez que la conversión está en estado “DONE”.

Bit 26:24 - CHN : Channel

Estos bits contienen el número del canal para el cual se hizo la conversión y el valor está disponible en RESULT.

Bit 27 - OVERRUN

Este bit se pone en alto cuando está trabajando en modo BURST y se ha sobre escrito un valor de conversión. O sea, se hizo la captura, no se leyó y se perdió con el valor de la siguiente captura.

Bit 31 - DONE

Este bit se pone en alto cuando se completó la conversión.Se pone en cero cuando es leído y cuando se cambia el ADCR. Si se modifica el ADCR mientras está haciendo una conversión, se pone en alto y comienza una nueva conversión.

Otra forma de obtener los datos.

ADDR0 to ADDR7 – A/D Data registers : Estos registros contienen el resultado de la más reciente conversión que fue completada para cada canal. Hay que tener cuidado de usar estos o el AD0GDR.

Table 535: A/D Data Registers (AD0DR0 to AD0DR7 - 0x4003 4010 to 0x4003 402C) bit description

Bit	Symbol	Description	Reset value
3:0	-	Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA
15:4	RESULT	When DONE is 1, this field contains a binary fraction representing the voltage on the AD0[n] pin, as it falls within the range of V_{REFP} to V_{REFN}. Zero in the field indicates that the voltage on the input pin was less than, equal to, or close to that on V_{REFN}, while 0xFFF indicates that the voltage on the input was close to, equal to, or greater than that on V_{REFP}.	NA
29:16	-	Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA
30	OVERRUN	This bit is 1 in burst mode if the results of one or more conversions was (were) lost and overwritten before the conversion that produced the result in the RESULT bits.This bit is cleared by reading this register.	
31	DONE	This bit is set to 1 when an A/D conversion completes. It is cleared when this register is read.	NA

Registro ADSTAT

ADSTAT – A/D Status register : Este registro contiene el DONE y el OVERRUN flags para todos los canales del ADC.

1. **Bits[7:0] – DONE[7 to 0]:** Acá están los DONE de cada canal por bit. Bit 7 canal 7, bit X canal X.
2. **Bits[15:8] – OVERRUN[7 to 0]:** Acá están los OVERRUN de cada canal por bit. Bit 7 canal 7, bit X canal X.
3. **Bit 16 – ADINT:** Este es el flag de interrupción del ADC. Se pone en alto cuando cualquier canal indivual del ADC tiene un DONE y disparó una interrupción.

Modos de operación.

-Por Software: En este modo sólo se puede una conversión por vez. Para hacer otra conversión necesitaremos reiniciar el proceso. Únicamente 1 bit en SEL field del ADCR puede estar en alto. Se puede configurar cualquier canal pero uno por vez.

-Burst: Las conversiones son hechas continuamente en los canales seleccionados haciendo “round-robin” de los canales. En este modo no se puede, controlar por Soft. Se pueden perder datos -> “Overrun” se llama. Se da cuando un dato no fue leído y es reemplazado por otro más nuevo. Generalmente se dispara una interrupción en Burst mode para ir a leer el último dato. Esta interrupción se dispara cuando la conversión en cualquiera de los canales seleccionados termina.

¿Cómo trabajar con interrupciones, polling o timer?

Podemos estar preguntando todo el tiempo al ADC si está en DONE o podemos hacer que dispare una interrupción cada vez que tenga un dato.

Otra opción podría ser disparar un timer a la frecuencia de sampleo e ir leyendo los datos. Esto podría ser en modo BURST o en modo por software.

También se puede activar el modo BURST y bajarle la frecuencia al ADC en caso de no querer sobremuestrear. Inconvenientes: Baja muchísimo (65 veces) por cada pulso de clock la frecuencia. Conviene siempre dejarlo al máximo, y ponerle un timer, o hacer decimado (descartar muestras).

Particularmente habría que ver la aplicación en particular qué requiere.

Pasos para configurar el ADC.

1- Configurar GPIO pin para la función de ADC con PINSEL.

2- Habilitar el Clock del ADC.

3- Deseleccionar todos los canales del ADC y darle el Power on con el bit ADCR.PDN.

4- Seleccionar el canal para la conversión con los bits en ADCR.SEL

5- Setear el bit ADCR.START para dar comienzo a la conversión en el canal seleccionado.

6- Esperar por la conversión con el bit ADGR.DONE

7- Leer el valor de ADGR.RESULT.

Ejemplo de Software control:

```c
#include <lpc17xx.h>
#include <stdio.h>

//Armo defines
#define VREF       3.3 //Reference Voltage a VREFP pin y VREFN = 0V(GND)
#define ADC_CLK_EN (1<<12)
#define SEL_AD0_0  (1<<0) //selecciono el AD0.0
#define CLKDIV     1 //Configuro el ADC clock-divider (ADC_CLOCK=PCLK/CLKDIV+1) = 12.5Mhz @ 25Mhz PCLK
#define PWRUP      (1<<21) //Seteando este a cero se apaga el bloque.
#define START_CNV  (1<<24) //001 Para dar comienzo a la conversi'on.
#define ADC_DONE   (1U<<31) //define it as unsigned value or compiler will throw #61-D warning
#define ADCR_SETUP_SCM ((CLKDIV<<8) | PWRUP)

int main(void)
{
    SystemInit(); //sets CCLK=100Mhz, PCLK=25Mhz


    LPC_SC->PCONP |= ADC_CLK_EN; //Enable ADC clock
    LPC_ADC->ADCR =  ADCR_SETUP_SCM | SEL_AD0_0;
    LPC_PINCON->PINSEL1 |= (1<<14) ; //select AD0.0 for P0.23
    int result = 0;
    float volts = 0;

    while(1)
    {
        LPC_ADC->ADCR |= START_CNV; //Start new Conversion

        while((LPC_ADC->ADDR0 & ADC_DONE) == 0); //Wait untill conversion is finished

        result = (LPC_ADC->ADDR0>>4) & 0xFFF; //12 bit Mask to extract result

        volts = (result*VREF)/4096.0; //Convert result to Voltage

    }
}
```

Ejercicio propuesto 1.

1- Dada una señal de 100khz de componente espectral máxima que ingresa por un pin del ADC, se necesita procesar. El rango dinámico de amplitud es de 3.3v.

Proponer otra señal de frecuencia definida para que samplee a la señal conectada al ADC y poder reconstruirla. 300 samples.

Escriba el programa de la manera más conveniente. Definir si burst o software, tiempos de sampleo, tipo de variables.

Ejercicio propuesto 2.

2- Dada dos señal de 50khz periódicas de componente espectral máxima que ingresan por pines del ADC, se necesita convolucionarlas en el dominio del tiempo (las señales son de secuencias sincronizadas) y almacenarlas. El rango dinámico de amplitud es de 3.3v.

Escriba el programa de la manera más conveniente. Definir si burst o software, tiempos de sampleo, tipo de variables.

Encontrar el patrón de cada secuencia y hacer la convolución para un período.

DMA. Direct Memory access.

Uno de los principales problemas que se nos planteó cuando vimos ADC fue la pérdida de datos cuando estamos cercanos a las frecuencias de sampleos máximas y el cpu tiene que hacer operaciones con punto flotante. La solución en ese momento fue chequear el overrun o procesar de otra forma los datos.

El LPC ofrece algo interesante que se puede utilizar en muchas aplicaciones no sólo ADC → DMA:

> Es una función que permite a los periféricos acceder a la memoria directamente o transferir datos a otros periféricos sin la ayuda de la CPU. Entonces, siempre que usemos DMA, podemos hacer cálculos en la CPU mientras se realiza la transferencia de datos. **Realmente hacer uso del microcontrolador como multitarea.**

DAC

1- 10 Bits de resolución

2- Update rate de 1Mhz.

3- Power/Rate configurable.

4- Salida con buffer de corriente.

5- Amplitud de salida:

DAC Output = (Vref * InputValue)/MaxValue = (3.3 * InputValue)/1024 = InputValue * 3.22 mv.

Configuración de DAC

DAC Channel	Port Pin	Pin Functions	Associated PINSEL Register
Aout	P0.26	0-GPIO, 1-AD0[3], 2-**AOUT** , 3-RXD3	20,21 bits of PINSEL1

DACCTRL register : P/ usar con DMA

DACR register: Contiene el valor a ser convertido a tensión analógica y un bit de control de power.

DACCNTVAL register : P/ usar con DMA

DAC config cont.

DACR - 0x4008C000

Bits 0:5: Son reservados.No escribir.

Bits 6:15: Value.

Bit 16: Bias power. 0 -> 1us update rate (1Mhz); 1-> 2.5us update rate 400Khz. Carga capacitiva máxima de 100pf.

Bits 17:31: Reservados.

Pasos para configurar:

Configurar pin for DAC output in PINSEL (no PCONP)

Dato para ser convertido es escrito en VALUE bits (DACR[15:6])

Vo = VALUE × ((VREFP - VREFN)/1024) + VREFN

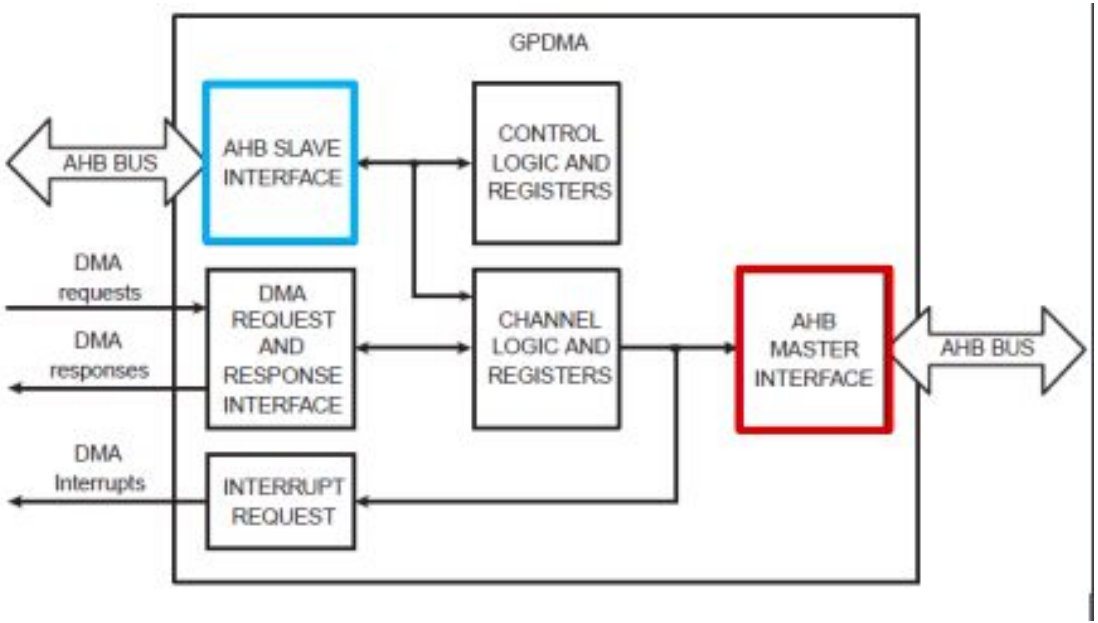
Configurar BIAS bit (DACR[16]) para 1Mhz o 400Khz.

DMA

Las transferencias de datos pueden requerir mucho tiempo de CPU para ejecutar instrucciones de carga y almacenamiento. Cuando se utiliza un controlador de acceso directo a memoria (DMA), la CPU se libera para ejecutar otras instrucciones.

LPC17xx tiene un DMA de uso general (GPDMA) que tiene 8 canales de los cuales cada uno puede realizar transferencias de datos. Se admiten todos los tipos de transferencias de datos como memoria-memoria, memoria-periférico, periférico-periférico. Y podemos priorizar las transferencias DMA como queramos. También puede realizar transacciones de 8 bits, 16 bits y 32 bits. Si queremos transferir una serie datos que no están almacenados de manera contigua, entonces podemos usar una función del DMA que usa lista enlazada. (linked lists)

GPDMA block.


GPDMA
AHB BUS
AHB SLAVE INTERFACE
CONTROL LOGIC AND REGISTERS
DMA requests
DMA responses
DMA REQUEST AND RESPONSE INTERFACE
CHANNEL LOGIC AND REGISTERS
AHB MASTER INTERFACE
AHB BUS
DMA Interrupts
INTERRUPT REQUEST

Ejemplo 1 DMA Memory to DAC.

La idea es sacar una forma de onda por el DAC con estos “detalles”:

1- No usamos CPU para la transferencia de memoria (array con los samples) al DAC.

2- Para verificar el punto 1 se puede hacer que el uC togglee un led, mientras saca datos por el DAC. Se habilita una interrupción EINT0 y el uC se pone en modo “sleep” se va a verificar que el DAC sigue sacando datos aún con el uC dormido.

3- Para el punto 2 el que lo quiera probar tiene que usar la AHB-SRAM.(0x20004000-0x2007C000) porque es la única que queda encendida cuando entra en sleep mode. La otra se apaga.

DMA Mem-DAC

4- Suponemos que tenemos la forma de onda a sacar en **sin[]** con la amplitud que va de 0 a 512 y de 0 a -512. Es multiplicar x512 el array.

5- **sin[total_samples]** : está desplazada 512 (metemos offset de 512 porque no podemos poner valores negativos en DACR) y <<6 posiciones | 1<<16 (BIAS=1). UINT32. Ver tabla 541.

DMA Mem-DAC.

El DMA en el LPC1700 tiene la capacidad de usar una estructura de lista enlazada para que se puedan transferir bloques de memoria no contiguos. Cada canal DMA individual puede admitir una transferencia de datos configurada de forma independiente.

Cada canal contiene algunos set de registros que pertenecen a su configuración/operación:

- Channel Source address
- Channel Destination address
- Channel Linked List Item
- Channel Control
- Channel Configure

En celeste son los que forman una lista linkeada. Si Linked List Item =0 el DMA desactiva el canal después de haber transferido toda la data.

Si el registro LLI no está en 0, el DMA cargará **nuevos valores de origen, destino, LLI** y control desde la dirección indicada por la LLI.

Se completa con la estructura GPDMA_LLI_Type nombre_de_la_estruct.

Channel configure se completa con GPDMA_Channel_CFG_Type GPDMACfg;

Pasos básicos de configuración:

- Seleccionar el canal a usar para la transferencia de datos según la prioridad (los canales DMA tienen prioridades fijas. Debemos elegir el canal de prioridad requerida para nuestro uso).
- Configurar el ancho de transferencia del bus (8 bits, 16 bits o 32 bits).
- Configurar la source y el destination.
- Establecer la señal de solicitud de DMA o sea la señal que iniciará la transferencia de DMA.
- Establecer el comportamiento Little endian o Big endian.
- Establecer el tamaño del stream de datos o sea el número de bytes que deben transferirse continuamente en una transacción DMA.
- Establecer el tipo de transferencia Mem-Mem, Mem-Peripheral o Peripheral-Peripheral.

Ejemplo paso a paso:

Vamos a prender y apagar un led :) pero acá vamos a almacenar una serie de unos y ceros alternos en un array y luego lo transferimos al puerto GPIO donde está conectado el LED y a **través del canal 0 del DMA.**

Ejemplo paso a paso

1- ¿Quién va a hacer o disparar la solicitud de DMA?

Vamos a configurar un timer que define la frecuencia de la salida de datos.

```
LPC_SC->PCONP |= 1 << 2; // Power up

LPC_SC->PCLKSEL0 |= 0x01 << 4; // CCLK

LPC_TIM1->MR0 = 1 << 25;

LPC_TIM1->MCR = 1 << 1; // reset on Match Compare 0
```

Ejemplo paso a paso.

2- Configuramos el DMA.

LPC_SC->PCONP |= 1 << 29; //Configuramos ver tabla 46, pg 64.

3. Habilitamos GPDMA (tabla 558, pg 610)

```
LPC_GPDMA->DMACConfig |= 1 << 0;
```

4. Poner el Match Compare 0 (MAT1.0 signal) como DMA request signal.

LPC_GPDMA->DMACSync &= ~(1 << 10); // No es estrictamente necesario DMACSync register está en cero con el reset. Habilita la syncronización del DMA request signals.

LPC_SC->DMAREQSEL |= 1 << 2; // Timer1 Match Compare 0 como DMA request tabla 560 pag 611

Ejemplo paso a paso.

Clear the Interrupt Terminal Count Request and Interrupt Error Status register. (Writing 1 to the clear registers will clear the entry in the main register. i.e writing 0xff to DMACIntErrClr register will clear the DMACIntErrStat register)

LPC_GPDMA->DMACIntErrClr |= 0xff; // table 550, pg 607 limpia errores de interrupción todos los canales.

LPC_GPDMA->DMACIntTCClear |= 0xff; // table 551, pg 607

Ejemplo paso a paso.

Configuramos la fuente y el destination address.

LPC_GPDMACH0->DMACCDestAddr = (uint32_t) &(LPC_GPIO1->FIOPIN3); // LED está conectado P1.29

LPC_GPDMACH0->DMACCSrcAddr = (uint32_t) &data[0]; // data[] tiene los datos a sacar por el puerto.

LPC_GPDMACH0->DMACCLLI = 0;//No usamos listas linkeadas.

Ejemplo paso a paso.

Configuramos burst transfer size, source burst and destination burst size, source and destination ancho, source incremento, destination incremento (LPC17xx manual table 564, pg 614)

En nuestro caso, el tamaño de transferencia es de 200 bytes, los tamaños del stream de datos de origen y destino son 1, el ancho de origen y destino son 8 bits y necesitamos hacer un incremento de origen y no hay necesidad de un incremento de destino porque los datos salen siempre por el P1.29.

LPC_GPDMACH0->DMACCControl = 200 | (1 << 26) //tabla 564

Ejemplo paso a paso.

Configuramos el tipo de transferencia como Memory to Peripheral y el request signal MAT1.0 tabla 566.

LPC_GPDMACH0->DMACCConfig = (10 << 6) | (1 << 11); // 10 MAT1.0 y es puesta como destino del request del periférico.(tabla 544, 566,**565**)

Habilitamos el canal:

LPC_GPDMACH0->DMACCConfig |= 1; //enable ch0

Se puede configurar una interrupción para cuando termine la transferencia DMA de modo que después de que se complete la transferencia DMA y se genere la solicitud de interrupción, en la rutina de interrupción, deshabilitar el canal escribiendo:

LPC_GPDMACH0->DMACCConfig = 0; // stop ch0 dma

Ejemplo de código de DAC y DMA.

```
#define DMA_SIZE 60

#define NUM_SINE_SAMPLE 60

#define SINE_FREQ_IN_HZ 50

#define PCLK_DAC_IN_MHZ 25 //CCLK divided by 4

void confPin(void);

void confDMA(void);

void confDac(void);

GPDMA_Channel_CFG_Type GPDMACfg;

uint32_t dac_sine_lut[NUM_SINE_SAMPLE];
```

```
int main(void)
{
    uint32_t i;
    uint32_t sin_0_to_90_16_samples[16]={\
            0,1045,2079,3090,4067,\
            5000,5877,6691,7431,8090,\
            8660,9135,9510,9781,9945,10000\
    };
    confPin();
    confDac();
    //Prepare DAC sine look up table
    for(i=0;i<NUM_SINE_SAMPLE;i++)
    {
        if(i<=15)
        {
            dac_sine_lut[i] = 512 + 512*sin_0_to_90_16_samples[i]/10000;
            if(i==15) dac_sine_lut[i]= 1023;
        }
        else if(i<=30)
        {
            dac_sine_lut[i] = 512 + 512*sin_0_to_90_16_samples[30-i]/10000;
        }
        else if(i<=45)
        {
            dac_sine_lut[i] = 512 - 512*sin_0_to_90_16_samples[i-30]/10000;
        }
        else
        {
            dac_sine_lut[i] = 512 - 512*sin_0_to_90_16_samples[60-i]/10000;
        }
        dac_sine_lut[i] = (dac_sine_lut[i]<<6);
    }
    confDMA();
    // Enable GPDMA channel 0
    GPDMA_ChannelCmd(0, ENABLE);
    while (1);
    return 0;
}
```

Funciones

```
void confPin(void){
    PINSEL_CFG_Type PinCfg;
    /*
     * Init DAC pin connect
     * AOUT on P0.26
     */
    PinCfg.Funcnum = 2;
    PinCfg.OpenDrain = 0;
    PinCfg.Pinmode = 0;
    PinCfg.Pinnum = 26;
    PINSEL_ConfigPin(&PinCfg);
    return;
}
```

```
void confDMA(void){
    GPDMA_LLI_Type DMA_LLI_Struct;
    //Prepare DMA link list item structure
    DMA_LLI_Struct.SrcAddr= (uint32_t)dac_sine_lut;
    DMA_LLI_Struct.DstAddr= (uint32_t)&(LPC_DAC->DACR);
    DMA_LLI_Struct.NextLLI= (uint32_t)&DMA_LLI_Struct;
    DMA_LLI_Struct.Control= DMA_SIZE
            | (2<<18) //source width 32 bit
            | (2<<21) //dest. width 32 bit
            | (1<<26) //source increment
            ;
    /* GPDMA block section -------------------------------------
    /* Initialize GPDMA controller */
    GPDMA_Init();
    // Setup GPDMA channel --------------------------------
    // channel 0
    GPDMACfg.ChannelNum = 0;
    // Source memory
    GPDMACfg.SrcMemAddr = (uint32_t)(dac_sine_lut);
    // Destination memory - unused
    GPDMACfg.DstMemAddr = 0;
    // Transfer size
    GPDMACfg.TransferSize = DMA_SIZE;
    // Transfer width - unused
    GPDMACfg.TransferWidth = 0;
    // Transfer type
    GPDMACfg.TransferType = GPDMA_TRANSFERTYPE_M2P;
    // Source connection - unused
    GPDMACfg.SrcConn = 0;
    // Destination connection
    GPDMACfg.DstConn = GPDMA_CONN_DAC;
    // Linker List Item - unused
    GPDMACfg.DMALLI = (uint32_t)&DMA_LLI_Struct;
    // Setup channel with given parameter
    GPDMA_Setup(&GPDMACfg);
    return;
}
```

Funciones

```
void confDac(void){
    uint32_t tmp;
    DAC_CONVERTER_CFG_Type DAC_ConverterConfigStruct;
    DAC_ConverterConfigStruct.CNT_ENA =SET;
    DAC_ConverterConfigStruct.DMA_ENA = SET;
    DAC_Init(LPC_DAC);
    /* set time out for DAC*/
    tmp = (PCLK_DAC_IN_MHZ*1000000)/(SINE_FREQ_IN_HZ*NUM_SINE_SAMPLE);
    DAC_SetDMATimeOut(LPC_DAC,tmp);
    DAC_ConfigDAConverterControl(LPC_DAC, &DAC_ConverterConfigStruct);
    return;
}
```

Ejercicio:

Usar ADC con DMA y FIFO de interfaz entre ADC samples y lectura de los valores.

Leo valores a un cierto clock. T0.

ADC va convirtiendo a otro clock.

Overflow- Underflow.

Procesar datos y sacarlos por el DAC por otro canal de DMA.