Pontificia Universidad Católica de Valparaíso
Escuela de Ingeniería Eléctrica
Laboratorio de Sistemas Electrónicos e Instrumentación

Apuntes del Taller de Microcontroladores ARM7TDMI LPC2148

Profesor
Moisés Reyes Pardo

Alumnos
Agustín Valencia González
Camila Berríos Miranda
Claudio Carreño Bonilla

Apuntes preparados por
Agustín Valencia González

Valparaíso, 5 de octubre de 2012
www.labsei.ucv.cl

Resumen

Para salvaguardar los conocimientos adquiridos en el taller de programación de microcontroladores ARM7TDMI al que se asistió recientemente, se presentan a continuación apuntes de las charlas y experiencias realizadas con la mayor profundidad posible dentro de lo que permite la complejidad de la temática.

Esperamos que los capítulos siguientes sirvan como material de apoyo para futuras investigaciones realizadas por ayudantes del *LABSEI*, en próximos desarrollos de proyectistas que decidan usar la plataforma disponible en el laboratorio o a quien disponga de ella y desee iniciarse en el uso de esta tecnología.

El objetivo de estos apuntes no es entregar profundos fundamentos sobre la estructura del microcontrolador (para formar un experto en la materia), sino explicar en forma simple el uso de los módulos más comunes, esos que pueden encontrarse en cualquier microcontrolador contemporáneo y que también están presentes en el LPC2148, ya que debido a su velocidad y potencia de cálculo, lo hacen comparativamente superior a sus semejantes en estos ámbitos, sin olvidar que las aplicaciones de este procesador van más allá de las que se explicitarán acá, fundamentalmente en las que usan RTOS (Sistemas Operativos en Tiempo Real) en sistemas embebidos.

Todas las experiencias fueron realizadas con el IDE *CrossWorksfor ARM* en C/C++, y posteriormente implementados en la tarjeta de desarrollo *LPC-P2148 Prototypefor LPC2148 ARM7TDMI-S Microcontroller* de *Olimex* a través del programador JTAG *Olimex ARM-USB-OCD*.

Figura 1: Descargue formato digital y archivos complementarios

Índice general

Capítulo 1

Conceptos Generales

1.1. Comentarios Preliminares

Para el desarrollo de este taller se asume cierta experiencia básica en el trabajo con microcontroladores y su programación en lenguaje C. Es necesario comprender sentencias de operaciones lógicas, de concatenación, de desplazamiento y asignación de valores, tanto a nivel de registros como de bits, dominio de sistemas numéricos decimal, binario y hexadecimal y sus respectivos cambios de base, conceptos básicos sobre interrupciones y la forma en que un procesador las atiende, funcionamiento de un PLL, timer y contadores, conocimientos sobre el funcionamiento y uso de un conversor análogo/digital e implementación de protocolos de comunicación serial.

El trabajo se fundamenta en el manual citado como referencia [1] en la bibliografía, cuya redacción es en inglés técnico, por lo que es necesario poder leer e interpretar un manual en dicho idioma. Estos apuntes constituyen una síntesis de tal manual en las temáticas que se abordan, además de sugerencias y consejos obtenidos del trabajo práctico.

Si bien no se aborda el uso del 100 % de los recursos del microcontrolador, dejando fuera por ejemplo el conversor digital/análogo, lector de memorias SD, SPI, I2C, pipelinening, Módulo Acelerador de Memoria, etc., se entrega la información necesaria para que el lector pueda acceder a las referencias y entender la filosofía de trabajo al desarrollar aplicaciones sobre una plataforma basada en un núcleo ARM.

1.2. Historia, Licenciamiento y Aseguración de Proveedores y Desarrolladores

En 1985 *Acorn Computer Group*, desarrolla *ARM -Advanced Risc Machine-* basándose en el MOS6502 y continuando con la filosofía *RISC*[1], lanza en 1986 el ARM2, que podría considerarse el procesador de 32 bits más simple del mundo con tan sólo 30.000 transistores.

A finales de los '80, *Apple Computer* comienza a trabajar con Acorn para el desarrollo de nuevos núcleos de ARM. En Acorn se dieron cuenta que el hecho de que el fabricante de un procesador fuese también un fabricante de computadores podría tener resultados en desmedro de los clientes, por lo que se decidió crear una nueva compañía llamada *Advanced RISC Machines*, que sería la encargada del diseño y gestión de las nuevas generaciones de procesadores ARM.

Así, Apple utiliza ARM6 en su PDA *Apple Newton* y Acorn como procesador principal en su *RiscPC*. Hoy el procesador más usado es el ARM7TDMI en millones de aparatos de telefonía móvil y sistemas de videojuegos portátiles, presente en la mayoría de los teléfonos Nokia, iPods, Nintendo-DS, Dreamcast, placas NVidia, etc. Aunque su uso va en retirada dando paso a ARM10, ARM11 y Cortex.

ARM se ha posicionado de manera fuerte en el mercado, hoy son acreedores de licencias del core de ARM empresas como *DEC, Freescale -empresa que derivó de Motorola el 2004, Apple, IBM, Infineon Tecnologies,OKI, Texas Instruments, Nintendo, Toshiba, NVidia, Philips, VLSI, Atmel, Sharp, Samsung y ST-*

[1] Reduced Instruction Set Computer

Microelectronics, por nombrar algunas. Este proceso de licenciamiento ha llevado a que tres cuartos de los procesadores de 32 bits posean este chip en su núcleo. Esta forma de extender licencias garantiza por un tiempo indeterminado una gran cantidad de proveedores de esta tecnología para el sinnúmero de desarrolladores que los utilizan.

ARM tiene relativamente pocos componentes en el chip, por lo que el consumo de energía es muy bajo, lo que lo hace ideal para uso en sistemas embebidos.

Normalmente cuando un estudiante de ingeniería comienza a abrirse paso por el mundo de los microcontroladores y sistemas embebidos, es tendencia comenzar utilizando microcontroladores PIC de 8 bits, los cuales ya no están a la altura de ciertas aplicaciones. Por tanto si se desea aprender sobre el *estado del arte* de los microprocesadores, ARM7 es una excelente opción para luego, si se desea, emigrar a versiones más complejas como Cortex, o bien utilizar los IPCores que se han sintetizado en lenguaje de descripción de hardware para su implementación en FPGAs.

1.3. Evolución histórica sobre la arquitectura de ARM

A continuación se presenta la evolución e implementación de distintas tecnologías a través del tiempo en los cores de ARM:

- Versión 1: Direccionamiento de 26 bits, dentro del set de instrucciones no se incluyen multiplicaciones o un coprocesador matemático.
- Versión 2: Incluye un coprocesador matemático para multiplicaciones en 32 bits[2].
- Versión 3: Direccionamiento de 32 bits. Este core lo utilizan ARM6 y ARM7.
- Versión 4: Añade instrucciones de carga y almacenamiento. Como subversión se encuentra 4T, la que incluye el modo THUMB, el cual permite utilizar instrucciones de 16 bits, lo que otorga una mayor densidad de código en un mismo espacio de memoria. StrongARM -desarrollado por Intel- utiliza el core v4, ARM7TDMI implementa el core v4T.
- Versión 5T: Es básicamente la arquitectura 4T pero se añaden nuevas instrucciones.
- Versión 5TE: Añade un set de instrucciones para aplicaciones de DSP. ARM 9E-S es un ejemplo de este core.

1.4. Prestaciones de Arquitectura RISC y Pipelinening

Las principales características de la arquitectura RISC son:

1. Set de instrucciones reducido, simple y de tamaño fijo[3]
2. Sólo las instrucciones de carga y almacenamiento acceden a la memoria de datos.
3. Rol importante del compilador, el cual es mucho más complejo que uno para procesadores CISC.

El objetivo general de RISC es aumentar la velocidad de procesamiento reduciendo el acceso a la memoria -que es más lenta[4]-, y permitiendo segmentación y paralelismo mediante una ejecución de las instrucciones a través de pipelinening.

La manera más simple de entender lo que es el pipeline, es realizar una analogía a un proceso productivo en cadena. Por ejemplo supóngase una lavandería que dispone de una lavadora, una secadora y una plancha operada por una persona.

El tiempo que demoraría si se ejecuta el proceso lavadora-secadora-planchado, por lote de ropa esperando terminar de atender ese lote de ropa, para luego recién comenzar a atender el próximo lote de ropa sucia; es mucho mayor que si cuando pase la ropa lista de la lavadora a la secadora, y esta se vacíe, inmediatamente ingrese un nuevo lote, y así sucesivamente en un proceso contínuo y sincronizado.

[2]Esta es una de las principales características que lo diferencia de los microcontroladores PIC

[3]aunque el modo THUMB permite uso de instrucciones de 16 bits, estas se empaquetan de igual forma en 32 bits, usando menos memoria en mayor cantidad de instrucciones. A esto se refieren los textos como *mayor densidad de código*.

[4]Problema conocido como *limitación de memoria*.

Las propiedades de paralelismo dependen del grado de la máquina[5].
El core del microcontrolador a utilizar dispone de un pipeline de tres etapas:

1. **Fetch:** Se extrae de la memoria la instrucción.
2. **Decode:** Se decodifican los opcodes de las operaciones.
3. **Execution:** El (los) registro(s) es(son) leído(s) del banco de registros, se realizan las operaciones en la ALU y el shift -si es necesario-, el(los) registro(s) es(son) escrito(s) en el banco de registros nuevamente.

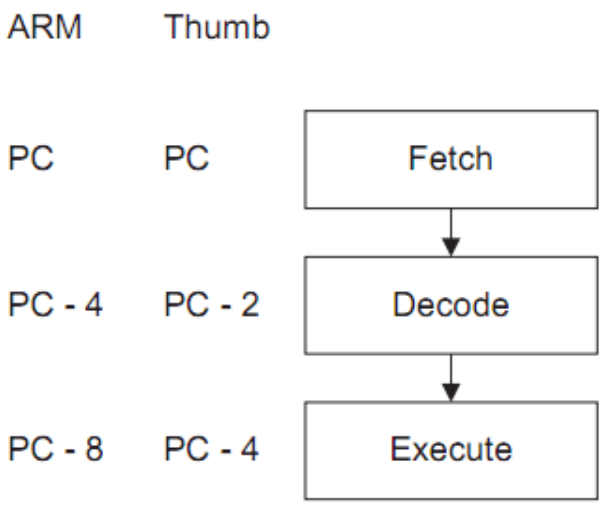


Figura 1.1: Pipeline del ARM7TDMI, figura extraída de la referencia [4]

Profundizar sobre el pipeline no es el objetivo en este caso, aunque es importante entender como funciona para lograr diseños óptimos, acá se obviarán pues son correspondientes a un curso de arquitectura de computadores o de sistemas embebidos.

1.5. Comparativo con procesadores contemporáneos

	PIC18FXX	**AVR**	**MSP430FXX**	**ARM7XX**
Arquitectura	8 bits	8 bits	16 bits	32 bits
Memoria Flash	Hasta 16 kB	Hasta 128 kB	Hasta 60 kB	Hasta 512kB
Conversor A/D	10 bits	10 bits	16 bits	10 bits
Frecuencia de Operación	40 MHz	16 MHz	25 MHz	60 MHz
Timers	1 de 8 bits 3 de 16 bits	2 de 8 bits 2 de 16 bits	2 de 8 bits 2 de 16 bits	2 de 32 bits 2 de 16 bits
RAM	1.5 kB	4 kB	16 kB	32 kB
MIPS	10	16	16	60
Consumo Nominal	0.5 mA/MHz	0.85 mA/MHz	0.25 mA/MHz	0.28 mA/MHz

Figura 1.2: Tabla comparativa PIC/AVR/MSP/ARM

[5]Grado del procesador superescalar.

Capítulo 2

Gestión de entradas y salidas digitales

2.1. Registro IODIR

El registro IODIR es quien contiene la configuración sobre qué pines -GPIO[1]- se comportarán como entrada o bien como salida.

Es un registro de 32 bits, cada bit corresponde a uno de los 32 pines del puerto en cuestión. Un 1 lógico indica una salida, y un 0 lógico, una entrada.

En resumen, si el bit 0 del registro IO1DIR se encuentra en alto, indica que el pin P1.0 (cero del puerto uno) es una salida.

Distintas formas de asignar como salidas a los pines P0.12 y P0.15 serían los siguientes códigos:

2.1.1. Ejemplo Asignación IODIR

Para asignar directamente P0.12 y P0.15 como salidas y el resto del puerto como entradas, escribir unos en los bits 12 y 15 y el resto cero como se ilustra en la Tabla de la Figura 2.1

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Cont	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Cont	**1**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**

Figura 2.1: Registro IO0DIR para el ejemplo 2.1.1

En resumen se tiene un registro de 32 bits cuyo valor es:
(En nomenclatura matemática)

$$00000000000000001001000000000000_2$$

(En lenguaje de programación)

```
IO0DIR = 0b 0000 0000 0000 0000 1001 0000 0000 0000;
```

Y realizando el cambio de base binaria a hexadecimal se obtiene que
(En nomenclatura matemática)

$$9000_{16}$$

(En lenguaje de programación)

```
IO0DIR = 0x9000;
```

Para mantener la configuración anterior presente en el registro y sólo modificar los bits 12 y 15 se puede recurrir a operaciones lógicas utilizando todo el registro:

```
IO0DIR = IO0DIR | 0x9000;
```

[1]General Porpouse Input/Output

Que sería lo mismo que escribir

```
IO0DIR |= 0x9000;
```

O en su efecto operar a nivel de bits:

```
IO0DIR = (1 << 12) | (1 << 15);
```

Una forma de asignar los mismos pines como entrada sería

```
IO0DIR &= ~(0x9000);
```

2.2. Registros IOSET y IOCLR

En conjunto el contenido de estos dos registros de 32 bits controlan el estado lógico de las salidas, de una manera bastante diferente de lo que se haría en un microcontrolador PIC de Microchip.

2.2.1. IOSET

Escribir un 1 en este registro indica la puesta en alto del pin al cual corresponde el bit del registro en que se escribe. Escribir un cero en tal registro no tendrá ningún efecto en las salidas.

2.2.2. IOCLR

Escribir un 1 en este registro produce que el pin correspondiente al bit del registro obtenga un estado bajo y además escribe un 0 en el bit respectivo en IOSET. Escribir ceros en este registro no tienen ningún efecto en el comportamiento de los puertos.

2.3. Registro IOPIN

El estado actual de los pines configurados como GPIO puede leerse siempre desde este registro de 32 bits, sin importar la dirección de este -input/output-.

Es decir, si el bit 10 del registro IO0PIN contiene un 1, entonces el pin P0.10 se encuentra a 3.3 V.

2.4. Aplicación Práctica

I. **Ejercicio 1**

Como se detalla en [2], en la tarjeta de desarrollo el Pin P0.10 posee un LED con la siguiente configuración:

Solución

Se aprecia que para encender el LEDs debe enviarse un 0 al puerto en que está conectado, así se escribe el siguiente código:

```
// Se importa librería del dispositivo y se define
// el clock en 12MHz
#include <LPC214x.h>
#define PLOCK 0x400

int main( void )
{

        // se asigna el pin P0.10 como salida
        IO0DIR |= 0x400;
        // se envía 1 para asegurar led apagado
```

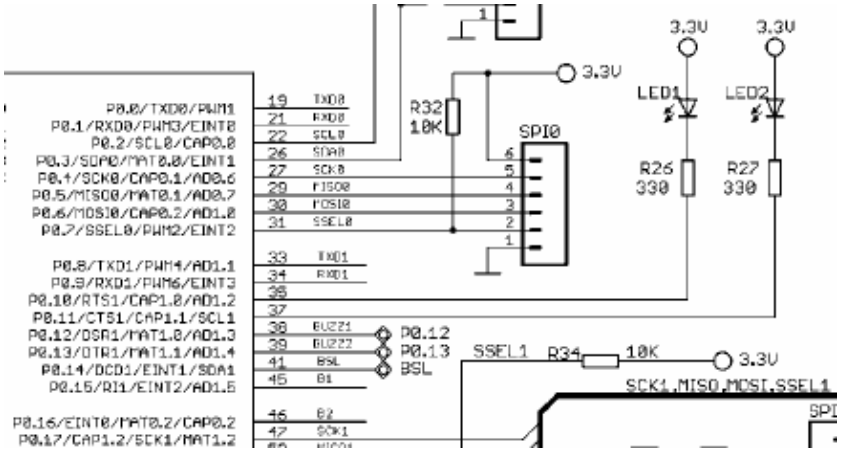


Figura 2.2: Conexión LEDs en la tarjeta de desarrollo

```
        IO0SET =  0x400;

        // se mantiene prendido el led
        IO0CLR = 0x400;
}
```

II. **Ejercicio 2**

Desarrollar un programa que encienda el LED conectado al pin P0.10 y P0.11 cada vez que se presionen los pulsadores conectados a los pines P0.15 y p0.16 respectivamente.

Solución

```
#include "LPC214x.h"
#define PLOCK 0x400

int main(void)
{
  IO0DIR |= 0xC00;
  IO0SET = 0xC00;

  while (1)
  {
    if ((IO0PIN & (1 << 15)) == 0)
    {
      IO0CLR = (1 << 10);
      IO0SET = (1 << 10);
    }
    if ((IO0PIN & (1 << 16)) == 0)
    {
      IO0CLR = (1 << 11);
      IO0SET = (1 << 11);
    }
  }
}
```

2.5. Cargar un programa al LPC2148 mediante Crossworks

Una vez ya creado el proyecto y escrito el código, corresponde, obviamente implementarlo. Para esto es necesario compilar y luego cargar el programa siguiendo los pasos a continuación detallados:

I. **Compilar**: Si se detectan errores, leer el prompt y solucionar de acuerdo a lo que indica.

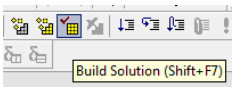


Figura 2.3: Compilar

II. **Conectar Programador**: Debe seleccionarse en *Targets* el programador *OLIMEX-ARM-OCD*, el divisor del reloj del JTAG en 4, y luego conectarlo.

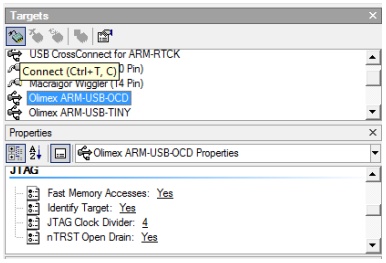


Figura 2.4: Conectar Programador

III. **Run**: Finalmente se empaqueta y corre el programa en el ARM.

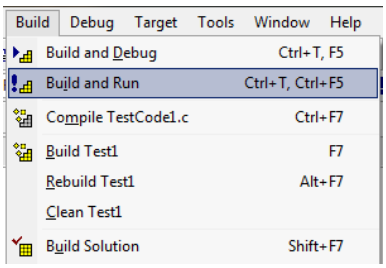


Figura 2.5: Run

2.6. Debug en Crossworks

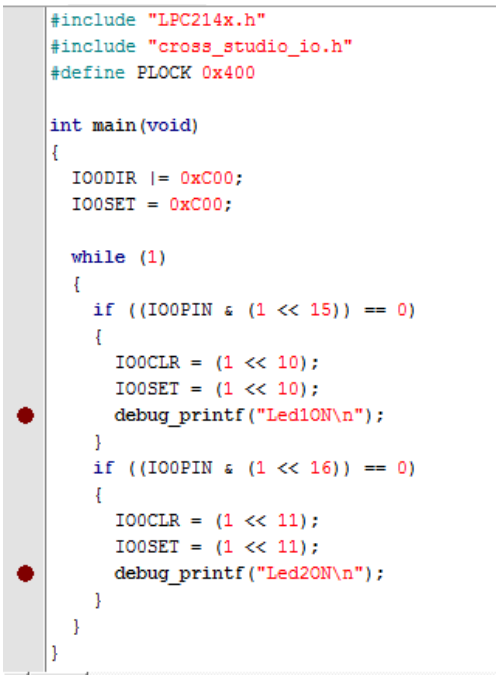
CrossWorks posee una modalidad bastante poderosa para el debugging de un programa.

Insertando breakpoints en el código, el programa se detendrá en tal línea, y podrán monitorearse el estado de ciertas direcciones de memoria o lo que el desarrollador necesite.

También puede utilizarse la función `debug_printf()` la cual permite que el microcontrolador escriba mensajes en pantalla, sin tener que dedicarse a programar el UART para lograr comunicación con el computador.

Para llevar a cabo esto a través de un ejemplo, se modifica el código del ejercicio 2.4.2 como se indica en la figura 2.6. En él se llama a la librería `cross_studio_io.h`, y se agrega dentro de cada condición if, un `debug_printf()` para que avise en pantalla que el LED debería estar encendido.

Para insertar los breakpoints basta hacer click en el sector gris del lado izquierdo del código. La manera de usar el `debug_printf()` es igual al `printf()` común en C. No confundirse, esto nada tiene que ver con comunicación serial, y tal línea de código sólo será funcional, si se usa Crossworks en modo debug.

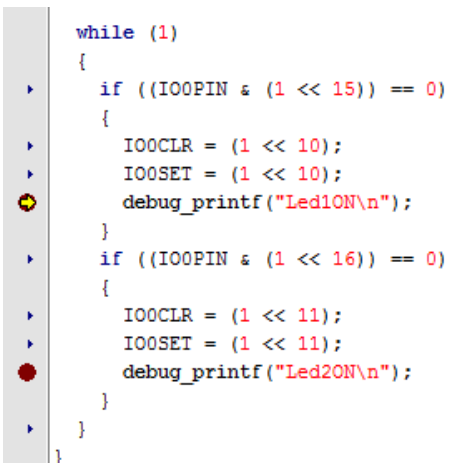


```
#include "LPC214x.h"
#include "cross_studio_io.h"
#define PLOCK 0x400

int main(void)
{
  IO0DIR |= 0xC00;
  IO0SET = 0xC00;

  while (1)
  {
    if ((IO0PIN & (1 << 15)) == 0)
    {
      IO0CLR = (1 << 10);
      IO0SET = (1 << 10);
      debug_printf("Led1ON\n");
    }
    if ((IO0PIN & (1 << 16)) == 0)
    {
      IO0CLR = (1 << 11);
      IO0SET = (1 << 11);
      debug_printf("Led2ON\n");
    }
  }
}
```

Figura 2.6: Código modificado del ejercicio 2.4.2 con breakpoints insertados.

Una vez realizado lo anterior basta con presionar F5 -*Build and Debug*- y luego presionar el simbolo *'play'* de la barra de tareas superior.

De esta forma, ya se estará corriendo el programa en modo debug, lo que implica que cada vez que este pase por una línea marcada como breakpoint el programa se detendrá inmediatamente -sin ejecutar a la misma línea-. Indicando con una flecha amarilla dentro del circulo rojo, cual es el breakpoint específico que detuvo la ejecución del programa. Para reanudarlo, debe presionarse la opción *'play'* nuevamente de forma manual.



```
  while (1)
  {
    if ((IO0PIN & (1 << 15)) == 0)
    {
      IO0CLR = (1 << 10);
      IO0SET = (1 << 10);
      debug_printf("Led1ON\n");
    }
    if ((IO0PIN & (1 << 16)) == 0)
    {
      IO0CLR = (1 << 11);
      IO0SET = (1 << 11);
      debug_printf("Led2ON\n");
    }
  }
}
```

Figura 2.7: Indicador de breakpoint activo.

La orden de impresión en pantalla arrojará su resultado en el prompt *'Debug I/O Terminal'* en tiempo real con la ejecución.

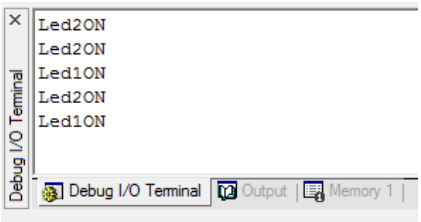


Figura 2.8: Salida en Debug I/O Terminal.

Capítulo 3

Interrupciones Externas

3.1. CTL SET ISR

CrossWorks permite un manejo de interrupciones de manera bastante sencilla, sin tener que lidiar con el análisis del vector de interrupciones, con la función `CTL_SET_ISR`, la cual necesita llamar a la cabecera `ctl_api.h` y tiene la siguiente estructura:

```
#include <ctl_api.h>

void ctl_set_isr( vector, prioridad, disparo, funcion, 0 );
```

En donde los argumentos de la función son:

3.1.1. Vector

Es un *unsigned int vector* que indica qué tipo de interrupción se está programando de acuerdo a la tabla de la Figura 3.1 -página 52 de [1]-, es decir, ha de ingresarse el número presente en la tabla para la interrupción respectiva.

Bit	23	22	21	20	19	18	17	16
Symbol	-	USB	AD1	BOD	I2C1	AD0	EINT3	EINT2
Access	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Bit	15	14	13	12	11	10	9	8
Symbol	EINT1	EINT0	RTC	PLL	SPI1/SSP	SPI0	I2C0	PWM0
Access	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Bit	7	6	5	4	3	2	1	0
Symbol	UART1	UART0	TIMER1	TIMER0	ARMCore1	ARMCore0	-	WDT
Access	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

Figura 3.1: Tabla Interrupciones por software

3.1.2. Prioridad

Corresponde a un número entero del 0 al 15, que define uno de los 16 niveles de prioridad disponibles correspondiente a la rutina a ejecutarse al ocurrir la interrupción en cuestión, en donde 0 es la máxima prioridad.

3.1.3. Disparo

Define la forma en que se activará la interrupción, las opciones disponibles son

- `CTL_ISR_TRIGGER_FIXED`: El tipo de la activación no es programable.

- `CTL_ISR_TRIGGER_LOW_LEVEL` Genera una interrupción cuando la señal está en estado bajo.
- `CTL_ISR_TRIGGER_HIGH_LEVEL` Genera una interrupción cuando la señal está en estado alto.
- `CTL_ISR_TRIGGER_NEGATIVE_EDGE` Genera una interrupción cuando la señal está en un canto de bajada.
- `CTL_ISR_TRIGGER_POSITIVE_EDGE` Genera una interrupción cuando la señal está en un canto de subida.
- `CTL_ISR_TRIGGER_DUAL_EDGE` Genera una interrupción cuando la señal está en un canto sin importar si su sentido es de subida o bajada.

3.1.4. Función

Indica el nombre que identifica la rutina (en C/C++) que debe ejecutarse cuando la interrupción se activa.

3.2. Sintaxis para el uso de Interrupciones en CrossWorks

Se presenta el código modelo para el trabajo con interrupciones en ARM7TDMI y CrossWorks en C/C++.

```
#include <ctl_api.h>

void IntFunction( void )
{
        // Se desactivan las otras interrupciones
        ctl_global_interrupts_re_enable_from_isr();

        // Limpiar flag de la interrupción correspondiente

        // ------------------------------------------------
        // CODIGO ATENCION
        // DE RUTINA USUARIO
        // ------------------------------------------------

        // Se vuelven a activar las otras interrupciones
        ctl_global_interrupts_un_re_enable_from_isr();
}
 int main( void )
{
        // Se habilitan la interrupciones
        ctl_global_interrupts_enable();

        // Se usa la api disponible para configurar la llamada
        ctl_set_isr(vector, prioridad, disparo, IntFunction, 0);

        // Se indica que no es enmascarable
        ctl_unmask_isr(vector)
}
```

3.3. PINSEL

Como es común en microcontroladores avanzados, un pin del package tiene variadas funciones, en el capítulo anterior sólo se usaron como GPIO, por tanto sólo se vio la configuración para ese uso.

`PINSEL` es un registro de 32 bits, que destina dos bits para la configuración de cada pin de un puerto, es decir serán necesarios 2 registros para configurar un puerto completo. Así `PINSEL0` controla los pines del P0.0 al P0.15, y `PINSEL1` los pines del P0.16 al P0.31. Para mayor información revisar las tablas 60, 61 y 62 de [1].

3.4. Limpiar Flag Interrupción Externa

El registro `EXTINT` es quien almacena los flags que indican el estado de las interrupciones externas.

Escribiendo 1 en los bits `EINT0` al `EINT3` en el registro `EXTINT` borra los respectivos bits. En el modo *level-sensitive* esta acción es eficaz sólo si el pin está en estado inactivo.

Es importante realizar esto en la rutina a la cual se ingresa cuando se activa la interrupción, de otro modo podría tener malfuncionamientos y perder el control del programa.

Table 9: External Interrupt Flag register (EXTINT - address 0xE01F C140) bit description

Bit	Symbol	Description	Reset value
0	EINT0	In level-sensitive mode, this bit is set if the EINT0 function is selected for its pin, and the pin is in its active state. In edge-sensitive mode, this bit is set if the EINT0 function is selected for its pin, and the selected edge occurs on the pin. Up to two pins can be selected to perform the EINT0 function (see P0.1 and P0.16 description in "Pin Configuration" chapter page 66.) This bit is cleared by writing a one to it, except in level sensitive mode when the pin is in its active state (e.g. if EINT0 is selected to be low level sensitive and a low level is present on the corresponding pin, this bit can not be cleared; this bit can be cleared only when the signal on the pin becomes high).	0
1	EINT1	In level-sensitive mode, this bit is set if the EINT1 function is selected for its pin, and the pin is in its active state. In edge-sensitive mode, this bit is set if the EINT1 function is selected for its pin, and the selected edge occurs on the pin. Up to two pins can be selected to perform the EINT1 function (see P0.3 and P0.14 description in "Pin Configuration" chapter on page 66.) This bit is cleared by writing a one to it, except in level sensitive mode when the pin is in its active state (e.g. if EINT1 is selected to be low level sensitive and a low level is present on the corresponding pin, this bit can not be cleared; this bit can be cleared only when the signal on the pin becomes high).	0
2	EINT2	In level-sensitive mode, this bit is set if the EINT2 function is selected for its pin, and the pin is in its active state. In edge-sensitive mode, this bit is set if the EINT2 function is selected for its pin, and the selected edge occurs on the pin. Up to two pins can be selected to perform the EINT2 function (see P0.7 and P0.15 description in "Pin Configuration" chapter on page 66.) This bit is cleared by writing a one to it, except in level sensitive mode when the pin is in its active state (e.g. if EINT2 is selected to be low level sensitive and a low level is present on the corresponding pin, this bit can not be cleared; this bit can be cleared only when the signal on the pin becomes high).	0
3	EINT3	In level-sensitive mode, this bit is set if the EINT3 function is selected for its pin, and the pin is in its active state. In edge-sensitive mode, this bit is set if the EINT3 function is selected for its pin, and the selected edge occurs on the pin. Up to three pins can be selected to perform the EINT3 function (see P0.9, P0.20 and P0.30 description in "Pin Configuration" chapter on page 66.) This bit is cleared by writing a one to it, except in level sensitive mode when the pin is in its active state (e.g. if EINT3 is selected to be low level sensitive and a low level is present on the corresponding pin, this bit can not be cleared; this bit can be cleared only when the signal on the pin becomes high).	0
7:4	-	Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA

Figura 3.2: Tabla EXTINT

3.5. Aplicación Práctica

Hacer un programa que encienda el LED conectado al pin P0.10 cada vez que se presiona el pulsador conectado al pin P0.15 utilizando interrupciones externas.

Solución.

Si se revisan las referencias [1] y [2] se observa que el pin P0.15 -que está conectado al pulsador- se encuentra en el pad 45, y este posee las funciones de *P0.15/RI1/EINT2/AD1.5*

Luego se sabe que para poder configurar tal pin debe revisarse la tabla de la Figura 3.4 sobre `PINSEL0` que contiene la información para la configuración de sus funciones, así se realiza lo siguiente.

De esta forma, se observa que para usar la interrupcion externa 2 disponible en el pin, a los bits 31 : 30 del registro `PINSEL0`, hay que cargarles un 1 y un 0 respectivamente, o bien -mirándolo desde la perspectiva práctica- tan solo un 1 en el bit 31.

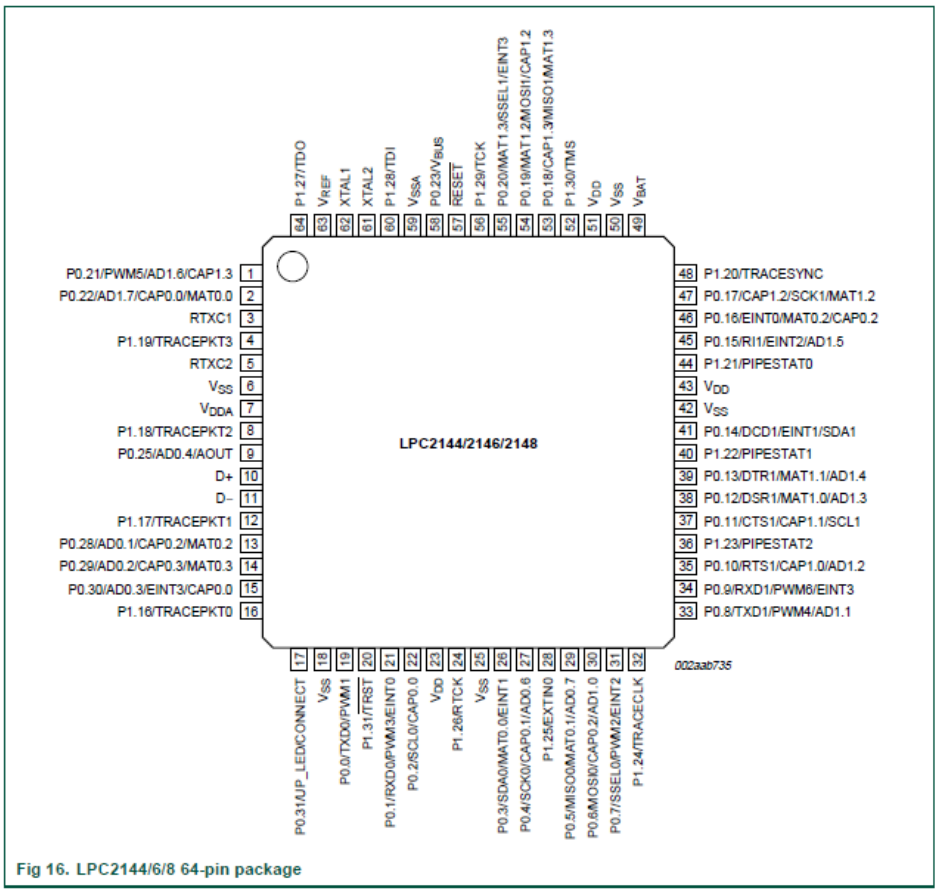


Figura 3.3: Package y pines del LPC2148

De esta forma se desarrolla el siguiente programa:

```
#include "LPC214x.h"
#include <cross_studio_io.h>
#include <ctl_api.h>
#define PLOCK 0x400

void pulsador ( void );

int main ( void )
{
      // Se configura el pin para usar la interrupción
        PINSEL0 |= ( 1 << 31 );

        // Pin P0.10 como salida
          IO0DIR |= 0x400;

        // Asegura que led esté apagado
        IO0SET = 0x400;

        // Config interrupción
```

Table 60: Pin function Select register 0 (PINSEL0 - address 0xE002 C000) bit description

Bit	Symbol	Value	Function	Reset value
19:18	P0.9	00	GPIO Port 0.9	0
		01	RxD (UART1)	
		10	PWM6	
		11	EINT3	
21:20	P0.10	00	GPIO Port 0.10	0
		01	Reserved[1][2] or RTS (UART1)[3]	
		10	Capture 1.0 (Timer 1)	
		11	Reserved[1][2] or AD1.2[3]	
23:22	P0.11	00	GPIO Port 0.11	0
		01	Reserved[1][2] or CTS (UART1)[3]	
		10	Capture 1.1 (Timer 1)	
		11	SCL1 (I^2C1)	
25:24	P0.12	00	GPIO Port 0.12	0
		01	Reserved[1][2] or DSR (UART1)[3]	
		10	Match 1.0 (Timer 1)	
		11	Reserved[1][2] or AD1.3[3]	
27:26	P0.13	00	GPIO Port 0.13	0
		01	Reserved[1][2] or DTR (UART1)[3]	
		10	Match 1.1 (Timer 1)	
		11	Reserved[1][2] or AD1.4[3]	
29:28	P0.14	00	GPIO Port 0.14	0
		01	Reserved[1][2] or DCD (UART1)[3]	
		10	EINT1	
		11	SDA1 (I^2C1)	
31:30	P0.15	00	GPIO Port 0.15	0
		01	Reserved[1][2] or RI (UART1)[3]	
		10	EINT2	
		11	Reserved[1][2] or AD1.5[3]	

[1] Available on LPC2141.
[2] Available on LPC2142.
[3] Available on LPC2144/6/8.

Figura 3.4: Extracto Table 60: PINSEL0 de [1]

```
        ctl_global_interrupts_enable();
        ctl_set_isr(16,0,CTL_ISR_TRIGGER_POSITIVE_EDGE,pulsador,0);
        ctl_unmask_isr( 16 );

        while ( 1 ) {
        IOOSET = ( 1 << 10 );
        }
}

void pulsador (void)
{
          ctl_global_interrupts_re_enable_from_isr();
          EXTINT = ( 1 << 2 );
          IOOCLR = ( 1 << 10 );
          ctl_global_interrupts_un_re_enable_from_isr();
}
```

Capítulo 4

PLL y Clocks en el Sistema

4.1. Diagrama de Clocks

Un PLL básicamente es un oscilador que realimenta frecuencia y/o fase de una señal de entrada para producir una señal de frecuencia mayor que la de entrada.

El LPC2148 posee 2 módulos de PLL. El PLL0 es usado para generar el *CCLK clock*, que es el clock del procesador, mientras el PLL1 provee al módulo USB la frecuencia de 48 MHz. Respecto de estructura estos dos PLLs son idénticos a excepción de que la interrupción de PLL trabaja sólo en PLL0.

El PLL0 y PLL1 aceptan un clock de entrada en un rango de frecuencia de 10 MHz a 25 MHz. La frecuencia de entrada se multiplica a un rango de frecuencia de 10 MHz a 60 MHz para el CCLK y 48 MHz para el USB. El multiplicador puede ser un *integer* de 1 a 32 -aunque en la práctica va de 1 a 6, porque después se obtienen valores muy altos que superan el límite del core-.

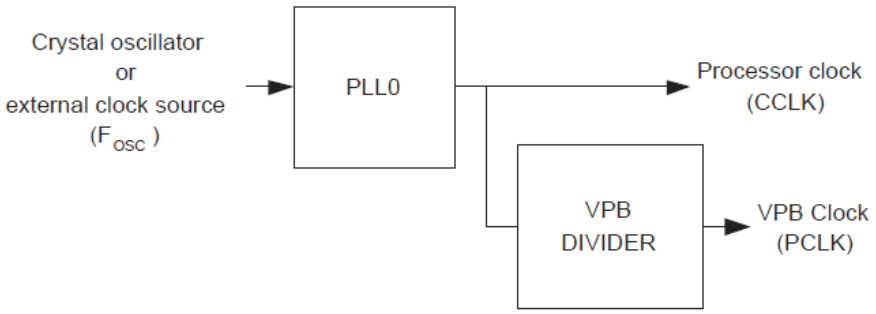


Figura 4.1: Diagrama de Clocks

El divisor de salida debe ser configurado para dividir por 2, 4, 8 o 16 veces, para producir el clock de salida para los periféricos.

4.2. Registros para la Configuración

Para entender en total profundidad este proceso debieran dominarse ciertos conceptos respecto de la arquitectura y optimización del funcionamiento del sistema, lo cual va allende el objetivo central de esta experiencia. Por lo que hay registros y sentencias que se dejan a estudio e investigación del lector.

4.2.1. PLLCON

Es el registro que controla la activación y conexión del PLL. Habilitar el PLL permite bloquear las configuraciones actuales del multiplicador y divisor (para evitar funcionamientos erráticos en caso que cambiasen debido a algún error). Conectar el PLL causa que el procesador y todas las funciones del chip funcionen a partir de la frecuencia del clock de salida del PLL.

Realizar cambios en PLLCON no surtirá ningún efecto hasta que se realice una secuencia correcta de PLL feed[1].

El bit 0 es PLLE, que habilita el PLL. Cuando su valor es 1, después de una secuencia PLL feed válida, este bit activará el PLL y posteriormente se permitirá bloquear el PLL a una frecuencia determinada.

El bit 1 es PLLC, que conecta el PLL. Cuando PLLE y PLLC son 1 simultáneamente, después de una secuencia de feed válida conecta el PLL como fuente del clock del microcontrolador. De otra forma el CCLK será el escilador externo.

4.2.2. PLLCFG

Registro que almacena los valores que configuran el multiplicador y el divisor del módulo de PLL. Los cambios en este registro no tendrán efecto sin una secuencia válida de PLL feed.

A su vez se divide en 2 subregistros MSEL y PSEL, de 5 bits y 2 bits respectivamente. Que determinarán mediante una relación matemática la frecuencia de trabajo del PLL.

Registro PLLCFG								
Bit	7	6	5	4	3	2	1	0
Uso	-	PSEL		MSEL				

Figura 4.2: Esquema PLLCFG

MSEL

Bits $< 0 : 4 >$: Valor del multiplicador. Determina el valor M para los cálculos de la frecuencia del PLL.

PSEL

Bits $< 5 : 6 >$: Valor del divisor. Determina el valor P para los cálculos de la frecuencia del PLL.

IMPORTANTE : El uso del bit 7 está reservado.

Cálculo de los valores en PSEL, MSEL y Frecuencia del PLL

El cálculo de los valores con los que se deben llenar los registros se explica con relaciones matemáticas simples y de fácil comprensión:

- **Contenido MSEL**

  Se tiene que la frecuencia del clock CCLK será un multiplo entero de la frecuencia de oscilación externa, entonces se plantea:

  $$cclk = M \cdot F_{OSC} \tag{4.1}$$

  y el contenido de MSEL será

  $$MSEL = M - 1 \tag{4.2}$$

- **Contenido PSEL**

  Por otra parte se cumple también que:

  $$cclk = \frac{F_{CCO}}{2 \cdot P} \tag{4.3}$$

  y la frecuencia del oscilador controlado por corriente[2] puede calcularse como

  $$F_{CCO} = 2 \cdot P \cdot cclk = 2 \cdot M \cdot P \cdot F_{OSC} \tag{4.4}$$

[1] En el punto 4.2.5 se explica qué es una secuencia de feed

[2] Consultar el Capítulo sobre el módulo PLL en [1]

Al realizar los cálculos para las configuraciones de estos registros debe tomarse en cuenta las limitaciones de la plataforma en la cual se trabaja, estos son los datos para el LPC2148:

I. F_{OSC} tiene un rango que va de 10 MHz a 25 MHz.

II. CCLK tiene un rango que va de 10 MHz al máximo permitido por el microcontrolador que en este caso son 60 MHz.

III. F_{CCO} está en el rango de los 156 MHz y 320 MHz.

Table 22: PLL Divider values

PSEL Bits (PLLCFG bits [6:5])	Value of P
00	1
01	2
10	4
11	8

Table 23: PLL Multiplier values

MSEL Bits (PLLCFG bits [4:0])	Value of M
00000	1
00001	2
00010	3
00011	4
...	...
11110	31
11111	32

Figura 4.3: Valores de P/PSEL y M/MSEL

4.2.3. PLLSTAT

Es un registro de sólo lectura que entrega los parámetros que están en efecto al momento de realizarse una lectura del mismo. Los valores de PLLSTAT podrían diferir de los de PLLCON y PLLCFG si no se ha realizado un PLL feed adecuado, en ese caso, PLLSTAT tendría los valores de la configuración del último feed exitoso.

4.2.4. PLOCK

Corresponde al bit 10 del registro PLLSTAT, e indica si el PLL se ha bloqueado. Cuando es cero el PLL no se ha bloqueado, cuando es uno el PLL se ha bloqueado a la frecuencia determinada por PLLCFG y se conecta al controlador de interrupciones [3].

4.2.5. PLLFEED

Este registro habilita la carga de las configuraciones en PLLCON y PLLCFG en los registros sombra que gobiernan la operación del PLL.

Para llevar a cabo esto, se requiere cargar en PLLFEED una secuencia de valores, esto es:

I. Escribir `0xAA` en PLLFEED

II. Escribir `0x55` en PLLFEED

Esto debe ser llevado a cabo en tal orden, y debe hacerse en clicos consecutivos del Bus VPB. Este último requerimiento implica que las interrupciones deben estar deshabilitadas en el ciclo de feed. Si falla cualquiera de estas condiciones las configuraciones en PLLCON y PLLCFG no serán efectivas.

[3] Consultar el capítulo sobre el VIC en [1]

Table 18: PLL Status register (PLL0STAT - address 0xE01F C088, PLL1STAT - address 0xE01F C0A8) bit description

Bit	Symbol	Description	Reset value
4:0	MSEL	Read-back for the PLL Multiplier value. This is the value currently used by the PLL.	0
6:5	PSEL	Read-back for the PLL Divider value. This is the value currently used by the PLL.	0
7	-	Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA
8	PLLE	Read-back for the PLL Enable bit. When one, the PLL is currently activated. When zero, the PLL is turned off. This bit is automatically cleared when Power-down mode is activated.	0
9	PLLC	Read-back for the PLL Connect bit. When PLLC and PLLE are both one, the PLL is connected as the clock source for the microcontroller. When either PLLC or PLLE is zero, the PLL is bypassed and the oscillator clock is used directly by the microcontroller. This bit is automatically cleared when Power-down mode is activated.	0
10	PLOCK	Reflects the PLL Lock status. When zero, the PLL is not locked. When one, the PLL is locked onto the requested frequency.	0
15:11	-	Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA

Figura 4.4: Contenido PLLSTAT

4.2.6. VPBDIV

No corresponde directamente a un registro que incide sobre el funcionamiento del PLL, mas bien es VPBDIV quien depende de la frecuencia del clock que se obtiene del PLL. Este registro controla el clock de Bus VPB[4]. Su contenido se detalla en la tabla de la figura 4.5.

Este módulo se explicita en el diagrama de clocks de la página 18.

Table 29: VPB Divider register (VPBDIV - address 0xE01F C100) bit description

Bit	Symbol	Value	Description	Reset value
1:0	VPBDIV	00	VPB bus clock is one fourth of the processor clock.	00
		01	VPB bus clock is the same as the processor clock.	
		10	VPB bus clock is one half of the processor clock.	
		11	Reserved. If this value is written to the VPBDIV register, it has no effect (the previous setting is retained).	
7:2	-	-	Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA

Figura 4.5: Contenido VPBDIV

4.3. Aplicación Práctica

La tarjeta de desarrollo con que se implementa la experiencia, posee un oscilador externo de 12 MHz, mediante uso del módulo del PLL escriba un programa que fije el clock de sistema su frecuencia máxima.

Solución

De forma previa a escribir el programa, se calculan los valores que deben cargarse en PLLCFG, así:

$$F_{OSC} = 12MHz$$
$$cclk = 60MHz$$

Utilizando la relación de la relación 4.1:

[4]Bus de periféricos. Consultar [1]

$$\begin{aligned} cclk &= M \cdot F_{OSC} \\ 60MHz &= M \cdot 12MHz \\ M &= 5 \end{aligned}$$

$$\therefore MSEL = 4$$

Como MSEL son los bits $< 4:0 >$, realizando cambio de base, se tiene que:

$$\text{MSEL} = 00100$$

Luego utilizando la relación de la relación 4.3 y teniendo en cuenta que los valores extremos para F_{CCO} son 156 MHz y 320 MHz, entonces se estudian ambos casos:

$$\begin{aligned} cclk &= \frac{F_{CCO}}{2 \cdot P} \\ P &= \frac{F_{CCO}}{2 \cdot cclk} \end{aligned}$$

$$\begin{aligned} caso\ \ i: & \\ P &= \frac{156}{2 \cdot 60} \\ P &= 1,3 \end{aligned}$$

$$\begin{aligned} caso\ \ ii: & \\ P &= \frac{320}{2 \cdot 60} \\ P &= 2,67 \end{aligned}$$

Luego, como se había mencionado anteriormente y se detalla en la tabla de la figura 4.3 el divisor puede trabajar en potencias de 2 -2, 4, 8 para este dispositivo, o bien 1 que mantiene la frecuencia sin variación alguna-. Por tanto el valor a elegir será 2, y de acuerdo a la tabla, el valor a cargar es 01.

Para terminar, al concatenar:

$$\begin{aligned} PLLCFG &= < - : PSEL[2] : MSEL[5] > \\ PLLCFG &= 00100100_2 \\ PLLCFG &= 24_{16} \end{aligned}$$

Después de esto, está todo dispuesto para escribir el código[5]. En el encabezado se dice que debe manipularse el clock del sistema, es decir el CCLK. Por ende debe trabajarse con el PLL0, así se escribe el siguiente código.

```
#include <LPC214x.h>
#define PLOCK 0x400

void initialize( void );
void feed( void );

int main( void )
{
        initialize();
        feed();
```

[5]Las líneas de código que hacen alusión al MAM -Módulo de Aceleración de Memoria- asumirlos como correctos pues es un tópico que está fuera del temario.

```
}

void initialize( void )
{
        //Se carga el PLL0CFG con el valor calculado
        PLL0CFG = 0x24;
        feed();

        // Habilitación del PLL
        PLL0CON = 0x1;
        feed();

        while( !(PLL0STAT & PLOCK) )
        {        // Espera hasta que el PLL fije la frecuencia
         }
        // Conecta al PLL como la fuente de CLOCK
        PLL0CON = 0x3;
        feed();

        // Enabling MAM and setting number of clocks
        // used for Flash memory fetch (4 cclks in this case)
        MAMCR = 0x2;
        MAMTIM= 0x3;

        // Fija el Clock (pclk) de los periféricos desde
        // el Clock del procesador (cclk), en este caso la mitad
        VPBDIV=0x2;
}

void feed( void )
{
        //Secuencia válida del feed
        PLL0FEED=0xAA;
        PLL0FEED=0x55;
}
```

Capítulo 5

Configuraciones e Interrupciones de Timer/Counter

5.1. Descripción

El LPC2148 posee dos timers de 32 bits con un prescaler de 32 bits también que pueden generar interrupciones -revisar la tabla de la figura 3.1-.

EL Timer/Counter está diseñado para contar ciclos del clock de periféricos PCLK o uno externo, además puede, opcionalmente, generar interrupciones o realizar otras acciones en tiempos especificados, basándose en 4 registros con el que se comparan valores y se realiza un match.

También posee 4 entradas de captura -que no se utilizarán, pero es bueno siempre nombrarlo- para obtener valores del timer cuando ocurre transiciones en las señales de entrada, opcionalmente, también se pueden generar interrupciones.

En el capítulo a continuación sólo se utilizará el módulo como timer, pero se entregan los conocimientos fundamentales para poder usarlo como contador en base a la información disponible en [1].

5.2. Registros Asociados

5.2.1. TCR

Timer Control Register, se utiliza para controlar la operación del Timer/Counter.

El bit 0 es *Counter Enable*, cuando su valor es 1 el timer y el prescaler están habilitados para contar. Cuando es 0 están deshabilitados.

El bit 1 es *Counter Reset*, cuando su valor es 1 el timer y el prescaler se resetean sincrónicamente con el próximo flanco de subida del PCLK. Los contadores permanecerán en cero mientras este bit no retorne a cero.

Los bits del $< 2 : 7 >$ están reservados.

5.2.2. IR

El registro de interrupción del Timer/Counter consiste en 4 bits para la activación de interrupciones de match y otro cuatro para las interrupciones de captura, si se produce alguna interrupción, el bit correspondiente a tal interrupción en el IR será 1, de otra forma el bit será 0.

Escribir un 1 en el bit respectivo a la interrupción la reseteará, escribir ceros no tiene ningún efecto.

5.2.3. TC

Timer Counter, este registro de 32 bits se incrementa cada PR+1 ciclos del PCLK -leer siguiente punto-. El Timer Counter es controlado desde el TCR.

Table 238: Interrupt Register (IR, TIMER0: T0IR - address 0xE000 4000 and TIMER1: T1IR - address 0xE000 8000) bit description

Bit	Symbol	Description	Reset value
0	MR0 Interrupt	Interrupt flag for match channel 0.	0
1	MR1 Interrupt	Interrupt flag for match channel 1.	0
2	MR2 Interrupt	Interrupt flag for match channel 2.	0
3	MR3 Interrupt	Interrupt flag for match channel 3.	0
4	CR0 Interrupt	Interrupt flag for capture channel 0 event.	0
5	CR1 Interrupt	Interrupt flag for capture channel 1 event.	0
6	CR2 Interrupt	Interrupt flag for capture channel 2 event.	0
7	CR3 Interrupt	Interrupt flag for capture channel 3 event.	0

Figura 5.1: Configuración Registro de Interrupciones

5.2.4. PR y PC

Prescaler Register y Prescaler Counter, Prescaler Counter es un registro que se incrementa en cada PCLK hasta el valor almacenado en Prescaler Register.

Cuando el valor de PR es alcanzado, el TC es incrementado y el PC es borrado.

5.2.5. MR

Match Register, puede ser habilitado desde el MCR para resetear el TC, detener el TC y el PC y/o generar una interrupción cada vez que alguno de los MR se iguale al TC.

5.2.6. MCR

Match Control Register, se usa para controlar qué operaciones se realizarán cuando uno de los cuatro MR se iguala al TC.

En la tabla de la figura 5.2 se cargan los valores para los comportamientos deseados de acuerdo a qué MR es el que se usa.

5.3. Aplicación Práctica

Desarrollar un programa, que utilizando interrupciones de Timer y PLL haga parpadear el LED conectado al Pin 0.10 en un ciclo de 1 segundo en alto, y otro segundo en bajo.

```
#include <LPC214x.h>
#include <ctl_api.h>
#define PLOCK 0x400

enum bool { true, false };
enum bool flag = true;

void initialize ( void );
void feed( void );
void delay( void );

int main (void )
{
        // Configuración del PLL
        initialize();
        feed();

        // Habilitación de Interrupciones
        ctl_global_interrupts_enable();
```

Table 241: Match Control Register (MCR, TIMER0: T0MCR - address 0xE000 4014 and TIMER1: T1MCR - address 0xE000 8014) bit description

Bit	Symbol	Value	Description	Reset value
0	MR0I	1	Interrupt on MR0: an interrupt is generated when MR0 matches the value in the TC.	0
		0	This interrupt is disabled	
1	MR0R	1	Reset on MR0: the TC will be reset if MR0 matches it.	0
		0	Feature disabled.	
2	MR0S	1	Stop on MR0: the TC and PC will be stopped and TCR[0] will be set to 0 if MR0 matches the TC.	0
		0	Feature disabled.	
3	MR1I	1	Interrupt on MR1: an interrupt is generated when MR1 matches the value in the TC.	0
		0	This interrupt is disabled	
4	MR1R	1	Reset on MR1: the TC will be reset if MR1 matches it.	0
		0	Feature disabled.	
5	MR1S	1	Stop on MR1: the TC and PC will be stopped and TCR[0] will be set to 0 if MR1 matches the TC.	0
		0	Feature disabled.	
6	MR2I	1	Interrupt on MR2: an interrupt is generated when MR2 matches the value in the TC.	0
		0	This interrupt is disabled	
7	MR2R	1	Reset on MR2: the TC will be reset if MR2 matches it.	0
		0	Feature disabled.	
8	MR2S	1	Stop on MR2: the TC and PC will be stopped and TCR[0] will be set to 0 if MR2 matches the TC.	0
		0	Feature disabled.	
9	MR3I	1	Interrupt on MR3: an interrupt is generated when MR3 matches the value in the TC.	0
		0	This interrupt is disabled	
10	MR3R	1	Reset on MR3: the TC will be reset if MR3 matches it.	0
		0	Feature disabled.	
11	MR3S	1	Stop on MR3: the TC and PC will be stopped and TCR[0] will be set to 0 if MR3 matches the TC.	0
		0	Feature disabled.	
15:12	-		Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA

Figura 5.2: Configuración del MCR

```
// Configuración de pines
IO0DIR = 0x400;
IO0CLR = (1 << 10);

// Reseteo del timer
T0TCR = 0x2;

// Se activa la interrupción de T0MR0 y TC se reseteará
// cuando ocurra un match
T0MCR = 0x3;

// El PR se carga con 0, por tanto el TC se incrementará
// directamente con el PCLK
T0PR = 0;

// Se producirá un match cada 1 segundo ya que la
// frecuencia del PCLK es 30 MHz debido a las configu-
// raciones de PLL
T0MR0 = 30000000;

//Se habilita el timer
T0TCR = 1 ;
```

```
        // Interrupción mediante API
        ctl_set_isr( 4, 0, CTL_ISR_TRIGGER_FIXED, delay, 0 );
        ctl_unmask_isr( 4 );
        while( 1 ){
        // espera }
}

void delay( void )
{
        // Limpia el flag de interrupcion de timer T0IR
        T0IR = 1 ;

        // Deshabilitación interrupciones
        ctl_global_interrupts_re_enable_from_isr();

        // Rutina de prendido y apagado
        if (flag)
        {
                IO0SET = (1 << 10);
                flag = !flag;
        }
        else
        {
                IO0CLR = (1 << 10);
                flag = !flag;
        }

        // Habilitación interrupciones
        ctl_global_interrupts_un_re_enable_from_isr();
}

void initialize( void )
{
        // Se carga el PLL0CFG con el valor calculado
        // para funcionar a 60 MHz
        PLL0CFG = 0x24;
        feed();

        // Habilitación del PLL
        PLL0CON = 0x1;
        feed();

        while( !(PLL0STAT & PLOCK) )
        {        // Espera hasta que el PLL fije la frecuencia
         }


        // Conecta al PLL como la fuente de CLOCK
        PLL0CON = 0x3;
        feed();

        // Enabling MAM and setting number of clocks
        // used for Flash memory fetch (4 cclks in this case)
        MAMCR = 0x2;
        MAMTIM= 0x3;
```

```
        // Fija el Clock (pclk) de los periféricos desde
        // el Clock del procesador (cclk), en este caso la mitad
        VPBDIV = 0x2;
}

void feed( void )
{
        //Secuencia válida del feed
        PLL0FEED = 0xAA;
        PLL0FEED = 0x55;
}
```

Capítulo 6

Conversor Análogo Digital

6.1. Características

El LPC2148 posee dos conversores análogo-digital de 10 bits que realiza la conversión mediante procesamiento por aproximaciones sucesivas, puede cambiarse la referencia a un V_{REF} que no exceda la magintud de V_{DDA}, de otra forma las lecturas serán erróneas incluso afectando a otros canales.

EL clocking de la conversión A/D está dado por el clock del Bus VPB[1], sin embargo se dispone de un divisor programable en cada conversor para obtener la frecuencia deseada para la conversión que tiene como máximo -y óptimo- 4.5 MHz.

6.2. Registro ADCR

Es el registro de Control del ADC de 32 bits. El ADCR debe estar escrito para seleccionar el modo de operación antes que la conversión A/D pueda ocurrir. Se divide en los siguientes subregistros:

6.2.1. $< 0 : 7 >$ SEL

Selecciona cual(es) canal(es) -AD0.7:0 / AD1.7:0- es(son) serán muestreados y convertidos. El bit 0 selecciona ADX.0 y el bit 7 selecciona a ADX.7. En modo *software-controlled* sólo uno de estos pines puede estar seleccionado, en modo *hardware scan*, pueden estar seleccionados todos los canales simultáneamente si así se desea. Si todos los bits están en 0, la selección equivaldrá a 0x01.

6.2.2. $< 8 : 15 >$ CLKDIV

El clock de los periféricos será divido por *este valor más uno* para producir el clock del conversor A/D, el cual debe ser siempre menor a 4.5 MHz.

Típicamente se busca trabajar con la frecuencia máxima, aunque hay casos -como una fuente de señal con alta impedancia- que quizás requiera un clock un poco menor.

$$ClockADC = \frac{PCLK}{CLKDIV + 1} \leq 4{,}5MHz \tag{6.1}$$

6.2.3. $< 16 >$ BURST

Si este bit está en alto se activa el *Burst mode*, en el cual el ADC realiza conversiones sucesivas de acuerdo a lo configurado en CLKS. La primera conversión corresponde al 1 menos significativo presente en SEL, y así hacia arriba.

No se utlizará en la actividad, si desea averiguar más, consulte la página 267 de [1].

Si este bit es cero, las conversiones son controladas por software, y requiere 11 clocks para completarlas.

[1] VLSI Peripheral Bus.

6.2.4. $< 17 : 19 >$ CLKS

Estos tres bits determinan la cantidad de bits de resolución con la que contará la conversión, o bien en *Burst mode*, la cantidad de ciclos usadas para cada conversión. Así se tiene que:

$< 19 : 17 >$			Burst Mode	Resolución
0	0	0	11 clocks	10 bits
0	0	1	10 clocks	9 bits
0	1	0	9 clocks	8 bits
0	1	1	8 clocks	7 bits
1	0	0	7 clocks	6 bits
1	0	1	6 clocks	5 bits
1	1	0	5 clocks	4 bits
1	1	1	4 clocks	3 bits

Figura 6.1: Tabla configuración CLKS

6.2.5. Bit $< 20 >$

Reservado.

6.2.6. $< 21 >$ PDN

Con un 1 el conversor A/D queda operacional. Con un 0, está desenergizado.

6.2.7. Bits $< 22 : 23 >$

Reservados.

6.2.8. $< 24 : 26 >$ START

Cuando el bit 24 está en alto y el ADC en funcionando normalmente -no en *Burst mode*- indica al core manualmente que puede iniciar las conversiones.

De otro modo son bits que configuran cuando comienza la conversión en *Burst mode*. No se detallará ello, más información en la página 268 de [1].

6.2.9. $< 25 >$ EDGE

Determina en qué canto de la señal de entrada comenzará la conversión en *Burst mode*.

6.2.10. Bits $< 28 : 31 >$

Reservados.

Como se ha visto, desde el bit 22 en adelante corresponden a bits reservados o de configuración para *Burst mode*, por tanto en esta actividad se trabajará con los 21 bits de menor peso.

6.3. ADxDRy

Es el registro de datos del canal y del conversor x. Es decir, para hacer referencia al canal 4 del ADC1, debería trabajarse con AD1DR4.

En la tabla anterior se puede apreciar la estructuración del contenido en ADDR, en donde se destacan el bit $< 31 >$ *DONE* que se setea a 1 cuando la conversión ha sido realizada, y los bits $< 6 : 15 >$ *RESULT* que cuando *DONE* está en alto, contiene el resultado de la conversión.

Table 260: A/D Data Registers (ADDR0 to ADDR7, ADC0: AD0DR0 to AD0DR7 - 0xE003 4010 to 0xE003 402C and ADC1: AD1DR0 to AD1DR7- 0xE006 0010 to 0xE006 402C) bit description

Bit	Symbol	Description	Reset value
5:0	-	Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA
15:6	RESULT	When DONE is 1, this field contains a binary fraction representing the voltage on the AIN pin, divided by the voltage on the V_{REF} pin (V/V_{REF}). Zero in the field indicates that the voltage on the AIN pin was less than, equal to, or close to that on V_{SSA}, while 0x3FF indicates that the voltage on AIN was close to, equal to, or greater than that on V_{REF}.	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

Figura 6.2: Contenido ADDR

6.4. PCONP

Power Control for Peripherals.

Este registro contiene bits de control que habilitan o deshabilitan funciones de periféricos individualmente, permitiendo así, eliminar consumo de energía de periféricos que no están siendo utilizados desconectandoles el clock específico del bloque de periféricos.

Algunos periféricos, particularmente los con funciones análogas, podrían tener consumos que no son dependientes de un clock, por lo tanto poseen otro control de activación, que desactivaría esa circuitería adicional para reducir aún más el consumo. En el caso del ADC sería el bit PDN del ADCR.

Para habilitar el ADC0 y ADC1 basta con levantar el bit 12 y 20 del registro PCONP respectivamente. Para mayor detalle del contenido de PCONP puede revisar la tabla 26 de [1].

6.5. Códigos Recomendados

A continuación se presentan códigos recomendados tanto para inicializar el conversor A/D como para realizar la conversión de la data propiamente tal.

6.5.1. Rutina de Inicialización

En el siguiente código X representa el canal del ADC0 que desea utilizarse, para ADC1 deben usarse los registros y configuraciones respectivos a tal periférico.

```
void ADC0INIT_X( void )
{
        PINSEL1 = ( 1 << ? );

        PCONP |= 0x1000;

        AD0CR = 0x?;
}
```

En la primera línea se configura el pin respectivo al canal que desea usarse para que responda a la función de ADC en el registro PINSEL[2].

En la segunda instrucción se habilita la alimentación para el ADC0 como se detalla en 6.4.

Y por último en la tercera línea ha de cargarse el valor correspondiente a la configuración del ADCR, éste paso es de suma importancia, pues es donde se determina como se comportará el ADC. En caso que en una experiencia práctica, el ADC no se comporte como se desea, al hacer un troubleshooting, el primer lugar donde buscar un error, sería acá.

[2]Si no entiende este paso, favor volver a la sección 3.3.

6.5.2. Rutina de Conversión

Nuevamente X representa el canal de ADC0 que desea utilizarse.

```
int ADC0READ_X( void )
{
        AD0CR |= 0x1000000;

        while( !( AD0DRX & 0x80000000 ) );

        return( AD0DRX );
}
```

En la primera orden se levanta el bit 24 para indicarle al ADC0 que puede iniciar las conversiones.

En el segundo comando se espera hasta que se levante el bit DONE de AD0DRX, es decir, espera que la conversión se realice.

Por último se pide que la función devuelva el valor de AD0DRX

6.6. Aplicación Práctica

La tarjeta de desarrollo posee un potenciómetro conectado al canal 3 del ADC0. Escriba un programa que realice conversiones de 10 bits de resolución cada 1 segundo gracias a interrupciones del Timer0 que indica acción de lectura con el parpadeo de un led e imprima el resultado en pantalla usando `debug_printf`.

Solución

De manera previa, se realizarán cálculos simples para determinar cuál es el valor que debe cargarse en AD0CR:

I. **SEL** Se utilizará el canal 3, por tanto debe cargarse un 1 al bit 3 de SEL, es decir, como SEL es un subregistro de 8 bits, se tiene que

$$SEL = 00001000$$

II. **CLKDIV** Típicamente se busca trabajar con 4.5MHz, y esto es lo que se hará.

Usando la relacion 6.1, se tiene que:

$$\begin{aligned} 4{,}5MHz &= \frac{PCLK}{CLKDIV+1} \\ 4{,}5MHz &= \frac{30MHz}{CLKDIV+1} \\ CLKDIV &= \frac{30}{4{,}5} - 1 \\ CLKDIV &= 5{,}6 \end{aligned}$$

Luego el valor debe ser 5 o 6:

$$\begin{aligned} ClockADC &= \frac{PCLK}{CLKDIV+1} \\ ClockADC &= \frac{30MHz}{5+1} \\ ClockADC &= 5MHz \end{aligned}$$

Lo cual sobrepasa el límite de 4.5 MHz, por tanto el valor definitivo debe ser 6, así el clock del ADC será:

$$\begin{aligned} ClockADC &= \frac{30MHz}{6+1} \\ ClockADC &= 4{,}29MHz \end{aligned}$$

Finalmente, tomando en cuenta que es un subregistro de 8 bits, el valor de CLKDIV es, :

$$\begin{aligned} CLKDIV &= 6_{10} \\ CLKDIV &= 00000110_2 \end{aligned}$$

III. **BURST** No se usará el Burst mode.

$$BURST = 0$$

IV. **CLKS** Como se utilizará una resolución de 10 bits, por tabla se tiene que:

$$CLKS = 000$$

V. **Bit 20** Reservado. No se escriben unos en él.

VI. **PDN** Se activa la conversión.

$$PDN = 1$$

VII. $< 31:22 >$ No se utilizarán.

Finalmente concatenando los resultados se tiene que:

$$\begin{aligned} AD0CR &= <-:PDN:-:CLKS:BURST:CLKDIV:SEL>_{32bits} \\ AD0CR &= <-:1:0:000:0:00000110:00001000>_{32bits} \\ AD0CR &= 00000000001000000000011000001000_2 \\ AD0CR &= 200608_{16} \end{aligned}$$

```
#include "LPC214x.h"
#include <ctl_api.h>
#include <cross_studio_io.h>
#define PLOCK 0x400

// Declaración variables
int results;
float Volts;
int val;

// Declaración funciones
void initialize( void );
void feed( void );
void interrupcion( void );
void adcInit0_3( void );
int adcRead0_3( void );



void main( void )
{
        //Configuración PLL
```

```
        Initialize();
        feed();

        // Se configura al pin 22 como output
        // y se asegura su estado en alto
        IO0DIR |= 0x400000;
        IO0SET |= 0x400000;

        // Inicializa el ADC 0, canal 3
        adcInit0_3();
        // Lee constantemente el resultado de ADC0.3

        // Habilita interrupciones
        ctl_global_interrupts_enable();

        // Reseteo del timer
        T0TCR = 0x2;

        // Se activa la interrupción de T0MR0 y TC se reseteará
        // cuando ocurra un match
        T0MCR = 0x3;

        // El PR se carga con 0, por tanto el TC se incrementará
        // directamente con el PCLK
        T0PR = 0;

        // Se producirá un match cada 1 segundo ya que la
        // frecuencia del PCLK es 30 MHz debido a las configu-
        // raciones de PLL
        T0MR0 = 30000000;

        //Se habilita el timer
        T0TCR = 1 ;

        // Interrupción mediante API
        ctl_set_isr( 4, 0, CTL_ISR_TRIGGER_FIXED, interrupcion, 0 );
        ctl_unmask_isr( 4 );
        while( 1 ){
        // espera }
 }

void interrupcion( void )
{

        // Limpia flag interrupcion timer
        T0IR = 1;

        // Deshabilitación interrupciones
        ctl_global_interrupts_re_enable_from_isr();

        // Low al pin 0.22
        IO0CLR = (1 << 22);

        // Lectura del ADC
        results = adcRead0_3();
```

```
        // Como el valor de la lectura está
        // en los bits <15:6>, se realiza un
        // deplazamiento en 6 lugares
        // y mediante un AND se salvan solo
        // los 10 primeros bits, es decir,
        // solo el valor de lectura
        val = (results >> 6) & 0x3FF;

        // Se convierte la lectura a un
        // valor interpretable en volts
        Volts = (val)*(3.3/1023.0);


        // Imprime en el debug el valor
        debug_printf("%1.2f Volts\n",Volts);

        // High al Pin 0.22
        IO0SET = (1 << 22);

        // Habilitación interrupciones
        ctl_global_interrupts_un_re_enable_from_isr();
}

void adcInit0_3(void)
{
        // Se configura a P0.30 para que cumpla
        // la función de ADC0.3
        PINSEL1 |= (1 << 28);

        //Habilita la alimentacion para ADC0
        PCONP |= 0x1000;

        // Configura e inicializa el conversor AD
        AD0CR = 0x200608;
}

int adcRead0_3(void)
{
        // Empieza la conversión de forma manual
        // escribiendo un 1 en el bit 24 (START)
        AD0CR |= 0x1000000;

        // Espera hasta que la conversión termine
        // Se levanta el bit DONE del AD0DR3
        while (!(AD0DR3 & 0x80000000))
        ;

        // Devuelve el resultado de la convesion
        return (AD0DR3);
}

void initialize( void )
{
        // Se carga el PLL0CFG con el valor calculado
        // para funcionar a 60 MHz
        PLL0CFG = 0x24;
```

```
        feed();


        // Habilitación del PLL
        PLL0CON = 0x1;
        feed();

        while( !(PLL0STAT & PLOCK) )
        {   // Espera hasta que el PLL fije la frecuencia
         }
        // Conecta al PLL como la fuente de CLOCK
        PLL0CON = 0x3;
        feed();

        // Enabling MAM and setting number of clocks
        // used for Flash memory fetch (4 cclks in this case)
        MAMCR = 0x2;
        MAMTIM= 0x3;

        // Fija el Clock (pclk) de los periféricos desde
        // el Clock del procesador (cclk), en este caso la mitad
        VPBDIV=0x2;
}

void feed( void )
{
        //Secuencia válida del feed
        PLL0FEED=0xAA;
        PLL0FEED=0x55;
}
```

Capítulo 7

Comunicación Serial

7.1. Descripción

El LPC2148 posee dos UARTs con registros de transimisión y recepción de 16 bits FIFO, ambas son idénticas con la excepción que en la UART1 se ha añadido una interfaz modem, en la cual no se centrará el análisis de este documento.

El disparo del receptor FIFO puede configurarse en 1, 4, 8, y 14 bytes.

Posee un mecanismo que habilita el control de flujo mediante sotware y hardware.

Para el uso de este módulo se utilizarán los pines RXD0 y TXD0, que corresponden a los pines de recepción y transmisión de la UART0 respectivamente.

7.2. Registros Asociados

7.2.1. UTHR

UART Transmit Holding Register. Es el byte que está al comienzo de la estructura FIFO del transmisor -*top*-, es decir, es el último caracter que ha ingresado al buffer. Puede ser escrito mediante la interfaz del bus. El LSB representa el primer bit a transmitir.

La transmisión de dicho byte se realizará cuando alcance el bottom del FIFO y la transimisión esté disponible.

El bit DLAB -*Divisor Latch Access Bit*[1] en ULCR debe estar en cero para poder acceder a UTHR.

UTHR es siempre un registro de sólo escritura.

7.2.2. URBR

UART Receiver Buffer Register. Es el byte en el top del FIFO de recepción, contiene el primer caracter recibido. Puede leerse mediante la interfaz del bus. El LSB representa el primer bit de información recibida -el más 'antiguo'-. Si la información recibida pesa menos de 8 bits, los MSB restantes se rellenarán con ceros.

El bit DLAB -*Divisor Latch Acces Bit*- en ULCR debe estar en cero para poder acceder a URBR. URBR es siempre un registro de sólo lectura.

7.2.3. UDLL y UDLM

UART Divisor Latch LSB y UART Divisor Latch MSB. Determinan como será el Baud Rate con el que trabajará el módulo UART[2].

Juntos forman un divisor de 16 bits, donde UDLL contiene los 8 bits inferiores, y UDLM los 8 bits de mayor peso. Un valor 0x0000 se interpretará como 0x0001 para evitar división por cero.

El bit DLAB debe estar en alto para poder acceder a estos registros.

[1] Revisar punto 7.2.6.

[2] Revisar el punto 7.3.

7.2.4. UFDR

UART Fractional Divider Register. Controla el prescaler del clock para la generación del baudrate y puede ser escrito y leído a la discresión del programador. Este prescaler toma el clock del VPB y genera una señal de salida de acuerdo a las especificaciones en UFDR.

Se divide en dos subregistros:

- $< 0 : 3 >$**DIVADDVAL:** Valor del divisor del prescaler. Si este valor es cero, el generador no afectará al baudrate del UART.
- $< 7 : 4 >$**MULVAL:** Valor del multiplicador del prescaler. Este valor debe ser igual o mayor a uno para una operación adecuada.
- $< 31 : 8 >$**Reservados:** No se deben escribir unos es estos bits.

7.2.5. UFCR

UART FIFO Control Register. Controla los FIFOs de recepción y transimisión.

- $< 0 >$**FIFO Enable:** Si es cero, los FIFOs están deshabilitados. Si es uno se habilitan los FIFOs de Rx y Tx y el acceso a los bits ¡7:1¿del UFCR. Cualquier transición en este bit borrará automáticamente los FIFOs de la UART.
- $< 1 >$**RX FIFO Reset:** Si se escribe un uno se borrarán todos los bytes del FIFO de Rx en el UART. Al activar este bit y ejecutarse la rutina de borrado del FIFO, este bit bajará automáticamente. Si es cero no tiene efecto alguno.
- $< 2 >$**TX FIFO Reset:** Si se escribe un uno se borrarán todos los bytes del FIFO de Tx en el UART. Al activar este bit y ejecutarse la rutina de borrado del FIFO, este bit bajará automáticamente. Si es cero no tiene efecto alguno.
- $< 5 : 3 >$**Reservados:** No se deben esribir unos en estos bits.
- $< 7 : 6 >$**Rx Trigger Level:** Estos dos bits determinan cuantos caracteres recibirá el FIFO de Rx antes de activar la interrupción:

< 7 : 6 >	**Descripción**
0 0	1 caracter o 0x01
0 1	4 caracteres o 0x04
1 0	8 caracteres o 0x08
1 1	14 caracteres o 0x0E

Figura 7.1: Configuración de Rx Trigger Level

7.2.6. ULCR

UART Line Control Register. Determina el formato de la data que será enviada o recibida.

- $< 1 : 0 >$**Word Length Select:** Longitud en bits de la palabra enviada o recibida.

< 1 : 0 >	**Descripción**
0 0	5 bits de largo
0 1	6 bits de largo
1 0	7 bits de largo
1 1	8 bits de largo

Figura 7.2: ULCR$< 1 : 0 >$: Word Length Select

- $< 2 >$**Stop Bit Select:** Si es cero se utilizará un bit de parada, si es uno, se utlizarán 2 bits de parada.Se utilizarán 1.5 bits de parada si se trabaja con 5 bits de largo.
- $< 3 >$**Parity Enable:** Si es cero se deshabilita la generación y checkeo de paridad, si es uno se habilita.
- $< 5 : 4 >$**Parity Select:** Configuran como se realizará el checkeo de paridad de la data.

< 5 : 4 >	**Descripción**
0 0	Paridad impar. El número de unos en el caracter transmitido y en el bit de paridad adjunto será impar.
0 1	Paridad par. El número de unos en el caracter transmitido y en el bit de paridad adjunto será par.
1 0	Se fuerza el bit de paridad a 1
1 1	Se fuerza el bit de paridad a 1

Figura 7.3: ULCR$< 5 : 4 >$: Parity Select

- $< 6 >$**Break Control:** Si es cero la condición de ruptura está deshanilitada. Si es uno, la salida de la UART se forzará a cero mientras el bit 6 de ULCR esté en alto.
- $< 7 >$**DLAB:** *Divisor Latch Access Bit.* Si es cero no permite el acceso a UDLL y UDLM, si es uno, lo hace.

7.2.7. ULSR

UART Line Status Register. Es un registro de sólo lectura que entrega información sobre el estado de los bloques de transmisión y recepción.

- $< 0 >$**RDR:** *Receiver Data Ready.* Se levanta cuando el buffer de recepción - URBR[3]- contiene un caracter que no ha sido leído, y se baja, cuando el FIFO del buffer se ha vaciado.

  Es decir, si RDR es 0, URBR está vacío; y si RDR es 1, URBR contiene data válida.

- $< 1 >$**OE:** *Overrun Error.* La condición de overrun se setea apenas ocurre. La acción de leer a ULSR, automáticamente baja este bit a cero.

  OE se levanta cuando en RSR hay un nuevo caracter y el FIFO de RBR está lleno. En este caso RBR no será sobreescrito y el caractar en RSR se perderá. Si es cero, no existe estado de error por overrun.

- $< 2 >$**PE:** *Parity Error.* Se dice que ocurre un error de paridad cuando el bit de paridad de un caracter recibido tiene un estado lógico incorrecto. Al igual que OE, una lectura a ULSR bajará este bit a cero. El tiempo de detección de un error de paridad depende de la configuración de UFCR.

  PE se levantará en caso de existir un error de paridad, si su estado es cero, no ha ocurrido error de paridad.

- $< 3 >$**FE:** *Framing Error.* Un error de framing ocurre cuando el bit de parada de un caracter es cero. Leer ULSR llevará este bit a cero. El tiempo de detección de un error de framing depende de las configuraciones de UFCR. En caso de ocurrir un error de framing, el receptor intentará resincronizarse asumiendo que el bit de parada erróneo es en verdad un bit de partida.

  Un estado alto en este bit indica error de framing, mientras un estado bajo indica que no existe error.

- $< 4 >$**BI:** *Break Interrupt.* Cuando en RXD0 llega un caracter que está contenido absolutamente por ceros -partida, data, paridad, parada- ocurre una interrupción de ruptura.

  Si BI está en alto, ha ocurrido una interrupción de quiebre, en caso contrario, no lo ha hecho.

- $< 5 >$**THRE:** *Transmitter Holding Register Empty.* Este bit se levanta inmediatamente cuando se ha detectado que el UTHR está vacío y se baja automáticamente cuando se escribe en UTHR data válida.

[3]Para mayor detalle sobre RBR y RSR revisar el esquema de la página 111 de [1].

- $< 6 >$**TEMT:** *Transmitter Empty.* Este bit de estado se levanta cuando están vacíos el UTHR y también el UTSR. Si está en estado bajo puede interpretarse como que el UTHR y/o el UTSR contienen data válida.
- $< 7 >$**RXFE:** *Error in RX FIFO.* Este bit se levanta cuando se carga al RBR un caracter que tiene algún problema, ya sea un error de paridad, de framing, o una interrupción de ruptura.

  Este bit se borrará si se lee ULSR y la situación no persiste.

7.3. Cálculo y configuración del Baud Rate

$$UART_{baudrate} = \frac{PCLK}{16 \cdot (16 \cdot UDLM + UDLL) \cdot \left(1 + \frac{DivAddVal}{MulVal}\right)} \tag{7.1}$$

En términos prácticos [1] recomienda utilizar también la siguiente relación

$$UART_{baudrate} = \frac{PCLK}{16 \cdot (16 \cdot UDLM + UDLL)} \cdot \frac{MulVal}{MulVal + DivAddVal} \tag{7.2}$$

Ha de tenerse en cuenta las siguientes restricciones:

I. $0 < MulVal \leq 15$

II. $0 \leq DivAddVal \leq 15$

Si UFDR no cumple con estas dos condiciones, entonces las salida del divisor fraccional es indefinida. Si DIVADDVAL es cero el divisor fraccional es deshabilitado y el clock no se dividirá.

Es importante que UFDR no se modiique *mientras* se transmite/recibe información, de ocurrir esto, la data se perderá o bien será corrupta.

7.3.1. Ejemplo

Determinar alguna configuración para que el UART1 funcione a un baudrate igual a 9600 cuando el clock de periféricos está en 20 MHz.

Solución.

De acuerdo a la relación 7.2 se tiene que:

$$\begin{aligned} UART_{baudrate} &= \frac{PCLK}{16 \cdot (16 \cdot UDLM + UDLL)} \cdot \frac{MulVal}{MulVal + DivAddVal} \\ 9600 &= \frac{20{,}000{,}000}{16 \cdot (16 \cdot UDLM + UDLL)} \cdot \frac{MulVal}{MulVal + DivAddVal} \end{aligned}$$

Para razones de simplificación del cálculo -y por experiencia práctica- se hace cero DIVADDVAL, así no se debe lidiar con tal factor, UDLM también pues en términos de magnitud basta con UDLL dividiendo para alcanzar el valor requerido

$$\begin{aligned} 9600 &= \frac{20{,}000{,}000}{16 \cdot (16 \cdot 0 + UDLL)} \cdot \frac{MulVal}{MulVal + 0} \\ 9600 &= \frac{20{,}000{,}000}{16 \cdot (UDLL)} \cdot 1 \\ UDLL &= \frac{20{,}000{,}000}{16 \cdot 9600} \\ UDLL &\approx 130_{10} \\ UDLL &= 82_{16} \end{aligned}$$

Luego debido a la aproximación, se comprueba el valor obtenido.

$$\begin{aligned} UART_{baudrate} &= \frac{PCLK}{16 \cdot (16 \cdot UDLM + UDLL)} \cdot \frac{MulVal}{MulVal + DivAddVal} \\ UART_{baudrate} &= \frac{20{,}000{,}000}{16 \cdot 130} \\ UART_{baudrate} &= 9615 \end{aligned}$$

Por tanto con tal configuración se obtiene un error del 0,16 % del valor del baudrate deseado, lo cual es aceptable.

Finalmente la configuración de los registros será:

```
U1FDR = 0x0;
U1DLM = 0x0;
U1DLL = 0x82;
```

Si bien la idea del presente item en estos apuntes, es simplemente no resignarse a asumir valores y entender de donde se obtienen estos. En la referencia [1], en la tabla 102 de las páginas 99 y 100, se entregan valores recomendados para ingresar en los registros, para diferentes velocidades de comunicación con sus respectivos errores, ya sea, llevando DIVADDVAL a cero, o bien una configuración óptima con errores casi nulos.

Table 102: Baudrates available when using 20 MHz peripheral clock (PCLK = 20 MHz)

Desired baudrate	MULVAL = 0 DIVADDVAL = 0			Optimal MULVAL & DIVADDVAL		
	U0DLM:U0DLL hex[2]	dec[1]	% error[3]	U0DLM:U0DLL dec[1]	Fractional pre-scaler value MULDIV / MULDIV + DIVADDVAL	% error[3]
7200	00AE	174	0.2240	124	5/(5+2)	0.0064
9600	0082	130	0.1600	93	5/(5+2)	0.0064
19200	0041	65	0.1600	31	10/(10+11)	0.0064
38400	0021	33	1.3760	12	7/(7+12)	0.0594
56000	0021	22	1.4400	13	7/(7+5)	0.0160
57600	0016	22	1.3760	19	7/(7+1)	0.0594
112000	000B	11	1.4400	6	7/(7+6)	0.1600
115200	000B	11	1.3760	4	7/(7+12)	0.0594
224000	0006	6	7.5200	3	7/(7+6)	0.1600
448000	0003	3	7.5200	2	5/(5+2)	0.3520

Figura 7.4: Fragmento de la tabla de valores recomendados para distintos valores de baudrate

7.4. Interrupciones de Puerta Serial

El registro que contiene las configuraciones que determinan como reaccionará el microcontrolador ante ciertas interrupciones por puerta serial es el UxIER, este análisis se centrará en el UART0, por tanto se utilizará el registro U0IER -*UART0 Interrupt Enable Register*. Para acceder a este registro el bit DLAB debe estar en estado bajo.

- $< 0 >$**RBR Interrupt Enable:** Habilita la interrupción RDA -*Data recibida disponible*-, también controla la interrupcion de time-out de caracter recibido.

  Si U0IER$< 0 >$, está en bajo, la interrupción por RDA está deshabilitada; en alto, está habilitada.

- $< 1 >$**THRE interrupt Enable:** Habilita la interrupción de THRE -*Transmitter Holding Register Empty, bit 5 del ULSR*-.

  Un 0 deshabilita la interrupción de THRE, en cambio un 1 la habilita.

- $< 2 >$**Rx Line Status Interrupt Enable:** Habilita la interrupción por estado de la línea de recepción, estado que puede leerse en los bits $< 4 : 1 >$ del ULSR.

  Si contiene un cero está deshabilitada, si contiene un uno está habilitada.

- $< 7 : 3 >$**Reservados:** No escribir unos en estos bits

- $< 8 >$**ABTOIntEn:** Habilita la interrupción de time-out del módulo de auto-baudrate.

  Habilita con 1 y deshabilita con 0.

- $< 9 >$**ABEOIntEn:** Habilita la interrupción de fin de auto-baud.

  Habilita con 1 y deshabilita con 0.

- $< 31 : 10 >$**Reservados:** No escribir unos en estos bits

El registro que indica el estado de las interrupciones pendientes y sus debidas prioridades es U0IIR. Cuando se lee U0IIR las interrupciones se congelan. No se entrará más en detalle respecto de este asunto, pues el lector puede consultar las tablas del manual en el capitulo destinado al UART0.
El flag de interrupción de puerta serial se baja con el simple hecho de leer U0IIR.

7.5. Envío de datos a bajo nivel, un acercamiento práctico a la estructura del UART

A continuación se presenta código a modo de recomendación para realizar la transmisión de un string mediante el UART1.

```
#include <LPC214x.h>
#define PLOCK 0x400
#define TEMT (1 << 6)
#define LINE_FEED 0x0A
#define RETORNO_CARRO 0x0D

void initilize( void );
void feed( void );
void conf_UART( void );
void escribir( void );

void main( void )
{
        initialize();
        feed();
        UART();
        escribir();
}

void conf_UART( void )
{
        PINSEL0 = 0x50000 ;
        U1FCR = 0x7 ;
        U1LCR = 0x83 ;
        U1DLL = 0x? ;
        U1DLM = 0x? ;
        U1LCR = 0x3 ;
}
```

En las cabeceras, aparte de llamarse a la librería del dispositivo y definirse la velocidad del clock del oscilador externo, también se definen tres variables. La primera, TEMT, es un vector con un 1 en el bit 6, esto se usará, en efecto, para realizar comparaciones referidas al bit TEMT del ULSR. La segunda y tercera corresponden a los códigos hexadecimales para los caracteres de control ascii correspondientes al line feed y retorno de carro.

Las funciones `initialize()` y `feed()` hacen referencia a la configuración del PLL y el CCLK, por lo que en el análisis se obviarán.

Luego de las definiciones y declaraciones de las funciones, en la primera línea de código dentro de la función UART, se indica mediante el registro PINSEL0, que los pines P0.8 y P0.9 responderán a sus funciones de TxD y RxD del UART1 respectivamente.

Al cargar U1FCR con 0x7, se le indica al core que habilite los FIFOs y los resetee. Posteriormente al escribir 0x83 -0b 1000 0011- en U1LCR se está configurando para que la comunicación sea mediante palabras de 8 bits y levanta DLAB para habilitar el acceso a U1DLL y U1DLM.

Finalmente se cargan los varlores calculados -o extraídos de la tabla- en U1DLL y UDLM y se baja DLAB, para quitar la autorización de escritura en dichos registros y poder acceder a U1THR.

```
void escribir( void )
{
        int i = 0;
        char c[] = "palabra";

        while( c[i] )
        {
                U1THR = c[i];
                i++;
        }

        U1THR = LINE_FEED;
        U1THR = RETORNO_CARRO;

        while( !(U1LSR & TEMT) )
        {}
}
```

En la función `escribir()` primeramente se define una variable de indice `i` y un string `c` que contiene el texto *palabra*, que desea enviarse por la puerta serial.

Mediante un ciclo while se ingresa al U1THR cada caracter del string a enviar, uno por vez, hasta que se termine. Posteriormente se envía un line feed y un retorno de carro por protocolo, esto es una 'buena costumbre', pues el programa correría perfectamente sin estas líneas, pero su visualización no sería la correcta.

Finalmente mediante una operación and se espera a que se levante el bit TEMT del U1LSR, que indicaría que la transmisión ha finalizado y los buffers están vacíos.

Esta es la manera de más bajo nivel -en C, obviando el assembler- de escribir en la puerta serial. Como se indicará más adelante existe una forma de hacerlo sin trabajar tanto a nivel de hardware, pero se considera esto, como la mejor forma de entender como funciona el UART en el LPC2148.

7.6. Printf, Putchar y Getchar

Si bien en el punto anterior se explica como escribir en la puerta serial del microcontrolador a nivel de hardware, quien está relativamente familiarizado con el uso de C para programar este tipo de dispositivos, conocerá funciones como `printf()`. Entonces no es de extrañar que se realice la pregunta *"¿Existe algo como printf para ARM?"*

La respuesta no es tan simple. Nativamente CrossWorks no posee esta función, entonces, en ese plano, la respuesta es no, no existe de manera nativa.

Pero en la comunidad de developers, han desarrollado código que permiten su uso y abierto al resto de los programadores mediante la web.

Para esto es necesario incluir las librerías `stdio.h` y `stdlib.h` en el código y almacenarlas en la carpeta del proyecto junto con `LPC214x.h`, y además añadir al código las funciones `putchar()` y `getchar()` -siendo

no necesario declarar estas funciones, debido a las librerías-, las cuales están disponibles en las cuales están disponibles en la página web del LABSEI (labsei.ucv.cl), sección "Publicaciones Internas".

Explicar de manera prosaica algo tan empírico, se hace complejo, por lo que se espera que con la experiencia práctica del ejercicio 7.7.2 quede claro la forma de utilizar tales funciones.

7.7. Aplicación Práctica

7.7.1. Ejercicio: Escritura de bajo nivel en la puerta serial

Escribir un programa que utilice la puerta serial 0 para enviar el mensaje *HOLA* con un baud rate de 115200.

Solución.

Lo primero es realizar los cálculos necesarios para la configuración de los registros que determinan el baudrate, entonces, se tienen dos opciones, una es ir y buscar los valores en la tabla 102 de la referencia [1], o bien, mediante las relaciones 7.1 y 7.2 determinar sus valores. La primera es simple y puede realizarla el lector por sí mismo, por lo tanto se optará, en este caso, por la segunda opción sólo por experimentar. Así:

$$\begin{aligned} UART_{baudrate} &= \frac{PCLK}{16 \cdot (16 \cdot UDLM + UDLL)} \cdot \frac{MulVal}{MulVal + DivAddVal} \\ 115200 &= \frac{30{,}000{,}000}{16 \cdot (16 \cdot UDLM + UDLL)} \cdot \frac{MulVal}{MulVal + DivAddVal} \end{aligned}$$

Lo anterior asumiendo que se trabaja con un clock de periféricos de 30 MHz, como se ha hecho en todo el curso para mantener la configuración del PLL. Luego por conveniencia se llevará a cero DIVADDVAL y UDLM, pues en este caso no son necesarios para alcanzar la tasa de transmisión requerida. De esta forma:

$$\begin{aligned} 115200 &= \frac{30{,}000{,}000}{16 \cdot (16 \cdot 0 + UDLL)} \cdot \frac{MulVal}{MulVal + 0} \\ 115200 &= \frac{30{,}000{,}000}{16 \cdot (UDLL)} \cdot 1 \\ UDLL &= \frac{30{,}000{,}000}{16 \cdot 9600} \\ UDLL &\approx 16_{10} \\ UDLL &= 10_{16} \end{aligned}$$

De esta forma se tiene que los valores a cargar para configurar el baudrate requerido son:

$$\begin{aligned} UDLL &= 0x10; \\ UDLM &= 0x0; \end{aligned}$$

De esta forma el código del programa será:

```
#include <LPC214x.h>
#define PLOCK 0x400
#define TEMT            (1<<6)
#define LINE_FEED         0xA
#define RETORNO_CARRO 0xD

void Initialize( void );
void feed( void );
void UART( void );

void main( void )
```

```
{
        initialize();
        feed();
        UART();

        int i = 0;
        char c[] = "HOLA";
        while( c[i] )
                { U0THR=c[i];
                   i++;        }
        U0THR=LINE_FEED;
        U0THR=RETORNO_CARRO;

        // Espera hasta que U0THR
        // y U0TSR esten vacios
        while(!(U0LSR & TEMT)){}
}

void initialize( void )
{
        // Se carga el PLL0CFG con el valor
        // calculado para funcionar a 60 MHz
        PLL0CFG = 0x24;
        feed();


        // Habilitación del PLL
        PLL0CON = 0x1;
        feed();

        while( !(PLL0STAT & PLOCK) )
        {   // Espera hasta que el PLL fije la frecuencia
         }
        // Conecta al PLL como la fuente de CLOCK
        PLL0CON = 0x3;
        feed();

        // Enabling MAM and setting number of clocks
        // used for Flash memory fetch (4 cclks in this case)
        MAMCR = 0x2;
        MAMTIM= 0x3;

        // Fija el Clock (pclk) de los periféricos desde
        // el Clock del procesador (cclk), en este caso la mitad
        VPBDIV=0x2;
}

void feed( void )
{
        // Secuencia válida del feed
        PLL0FEED=0xAA;
        PLL0FEED=0x55;
}

void UART( void )
{
```

```
        // Inicializa TX y RX
        PINSEL0=0x5;

        // Habilita fifo y lo resetea
        U0FCR=0x7;

        // Setea DLAB y el largo word a 8 bits
        U0LCR=0x83;

        // Setea baud rate a 115200
        // calculos anteriores
           U0DLL=0x10;
        U0DLM=0x0;

        // Limpia DLAB
        U0LCR=0x3;
}
```

Luego de cargar este programa en la tarjeta de desarrollo, la manera de comprobar que el trabajo que realiza es acorde a lo diseñado, puede conectarse al computador via Terminal de Bray u otro software que permita *sniffear* comunicación serial como se muestra en la figura 7.5.

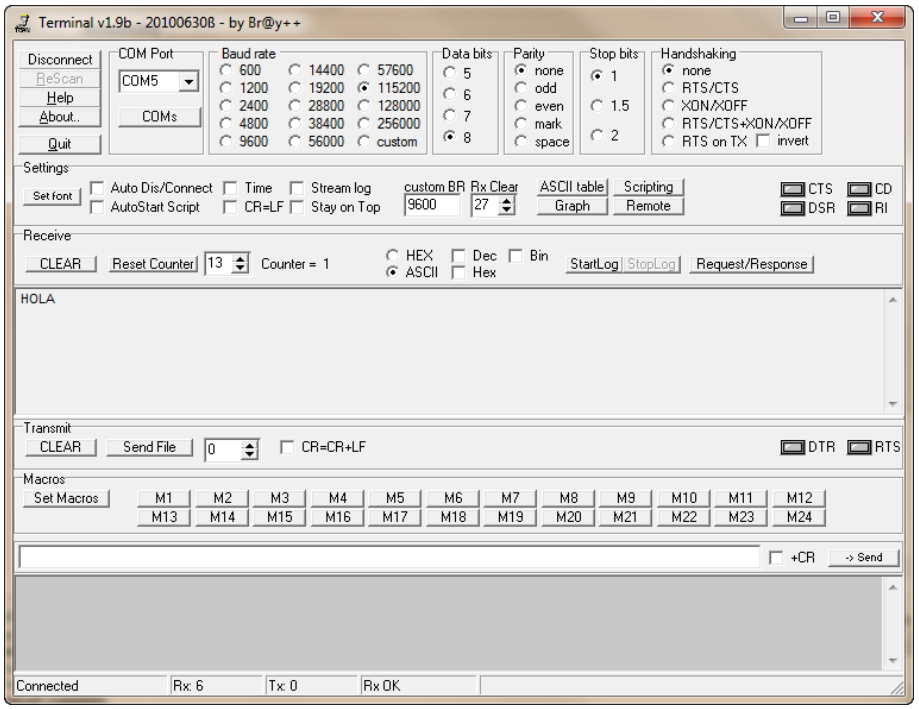


Figura 7.5: Respuesta en Terminal del ejercicio 7.7.1

7.7.2. Ejercicio: Lectura/Escritura con PUTCHAR y GETCHAR

Escribir un programa que utilice la puerta serial 0 para recibir un byte, si este es la letra 'A', deberá enviar una confirmación que diga 'CORRECTO', en caso de recibirse un byte diferente, se enviará un mensaje distinto. La comunicación se realizará con un PCLOCK de 30 MHz y un baudrate de 115200.

La recepción de datos (un byte) debe ser efectuada por medio de interrupciones de UART.

Solución.

Los cálculos de la configuración del baudrate están en el ejercicio anterior, por tanto se obviarán.

```
#include <LPC214x.h>
#include <stdio.h>
#include <stdlib.h>
#include <ctl_api.h>
#define PLOCK 0x400

char right[1] = "A";
char inbox[1];
int FlagRelease;

void Initialize ( void );
void feed( void );
void UART( void );
void interrupcion_UART( void );
```

Se incluyen las cabeceras y librerías necesarias, sobretodo `stdio.h` y `stdlib.h` para poder usar las funciones `putchar()` y `getchar()`. Se declara el caracter correcto y una variable de entrada, además de las funciones para evitar errores por cuestiones de orden al compilar.

```
int main (void)
{
  initialize();
  feed();
  UART();

  // interrupcion
  ctl_global_interrupts_enable();
  ctl_set_isr(6,0,CTL_ISR_TRIGGER_NEGATIVE_EDGE,interrupcion_UART,0);
  ctl_unmask_isr(6);
  U0IER = 1;
}
```

En el main, se realiza el proceso de *puesta a punto* del core y los periféricos como se ha visto en los capítulos anteriores.

Se configura la interrupción mediante el uso de la API destinada a ello, y se levanta el bit 0 de U0IER para habilitar la interrupción de recepción[4].

```
void interrupcion_UART( void )
{
  FlagRelease = U0IIR;
  inbox[0] = getchar();
  if ( strcmp(inbox[0],right[0]) == 0 )
  {
    printf("CORRECTO\n");
  }
  else
  {
    printf("NO VALIDO\n");
  }
}
```

En la atención de la interrupción primero se lee U0IIR para bajar el flag de la interrupción, y pueda accederse a ella nuevamente. Luego se lee el caracter mediante `getchar()` y se realiza una comparación para determinar si es igual al caracter almacenado en la variable `right[]` -la comparación es de tipo string.

```
void UART( void )
{
```

[4]Revisar sección 7.4 sobre interrupciones de UART.

```
  //inicializa tx y rx
  PINSEL0 = 0x5;

  //habilita fifo y lo resetea
  U0FCR = 0x7;

  // setea DLAB y el largo word a 8 bits
  U0LCR = 0x83;

  // setea baud rate
  U0DLL = 0x10;
  U0DLM = 0x0;

  //limpia DLAB
  U0LCR = 0x3;
}

void initialize( void )
{
  // Se carga el PLL0CFG con el valor calculado
  // para funcionar a 60 MHz
  PLL0CFG = 0x24;
  feed();

  // Habilitación del PLL
  PLL0CON = 0x1;
  feed();

  while( !(PLL0STAT & PLOCK) )
  {   // Espera hasta que el PLL fije la frecuencia
   }
  // Conecta al PLL como la fuente de CLOCK
  PLL0CON = 0x3;
  feed();

  // Enabling MAM and setting number of clocks
  // used for Flash memory fetch (4 cclks in this case)
  MAMCR = 0x2;
  MAMTIM= 0x3;

  // Fija el Clock (pclk) de los periféricos desde
  // el Clock del procesador (cclk), en este caso la mitad
  VPBDIV=0x2;
}

void feed(void)
{
  PLL0FEED=0xAA;
  PLL0FEED=0x55;
}
```

Las cuales son funciones ya conocidas utilizadas en capítulos anteriores. Para terminar se inserta el código respectivo a las funciones de `getchar()` y `putchar()` que se nombra en la sección 7.6.

```
/****************************/
/* Write Character To UART0 */
/****************************/
```

```
int putchar (int ch)
{
  if (ch == '\n')
  {
    //Wait TXD Buffer Empty
    while (!(U0LSR & 0x20));
    // Write CR
    U0THR = 0x0D;
  }
  // Wait TXD Buffer Empty
  while (!(U0LSR & 0x20));
  // Write Character
  return (U0THR = ch);
}

/*****************************/
/* Read Character From UART0 */
/*****************************/
int getchar (void)
{
  // Wait RXD Receive Data Ready
  while (!(U0LSR & 0x01));
  // Get Receive Data & Return
  return (U0RBR);
}
```

Finalmente se comprueba el funcionamiento correcto enviando caracteres con Terminal Bray y revisando la respuesta enviada por el microcontrolador.

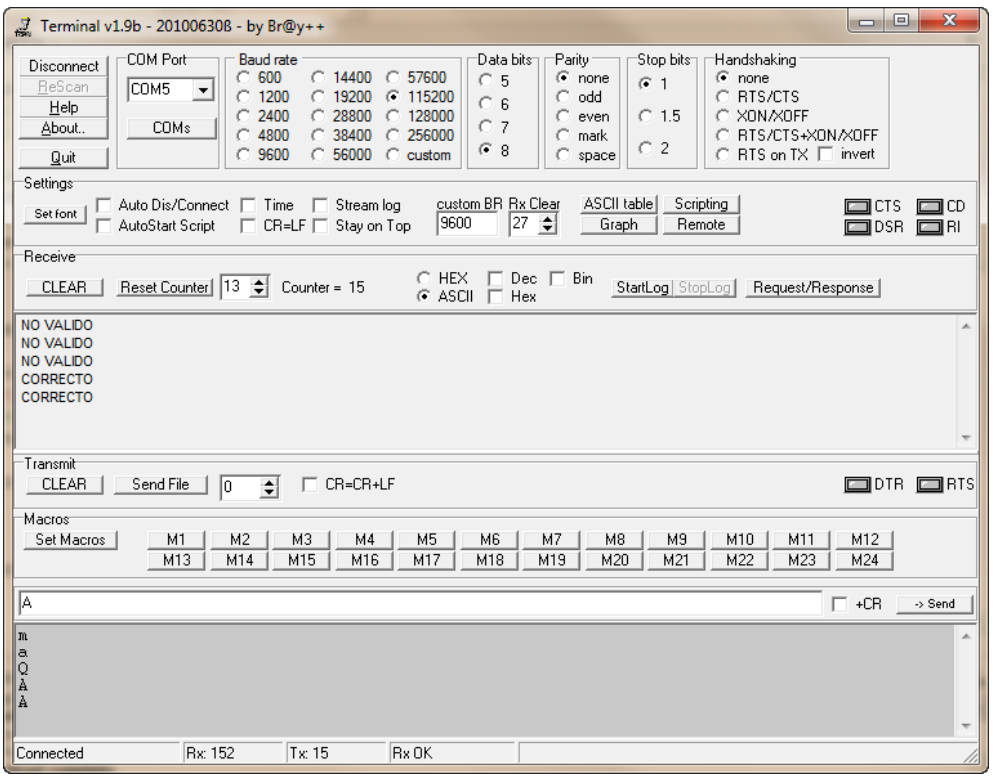


Figura 7.6: Respuesta en Terminal del ejercicio 7.7.2. Envío de caracteres y respuestas del microcontrolador.

Anexo A

Instalación del paquete para la familia LPC2000 y configuración del Programador

Para seguir estas instrucciones es necesario tener instalado y licenciado Crossworks, por razones legales, obviamente en este texto no se resolverá tal problemática.

Ir a la pestaña *Tools* > *install Package* y seleccionar el archivo NXP_LPC2000.hzq

Luego ir a la pestaña *Targets* > *Olimex ARM-USB-OCD*, seleccionar en el recuadro targets la opción Olimex ARM-USB-OCD, luego en el recuadro Propiedades buscar opción JTAG Clock Divider y cambiar el valor 1 (por defecto) al valor 4 (el valor elegido según lo leído en algunos foros debe ser cercano a $\frac{1}{4}$ del valor del clock de la tarjeta, este valor se debe variar hasta que el programa detecte el programador)

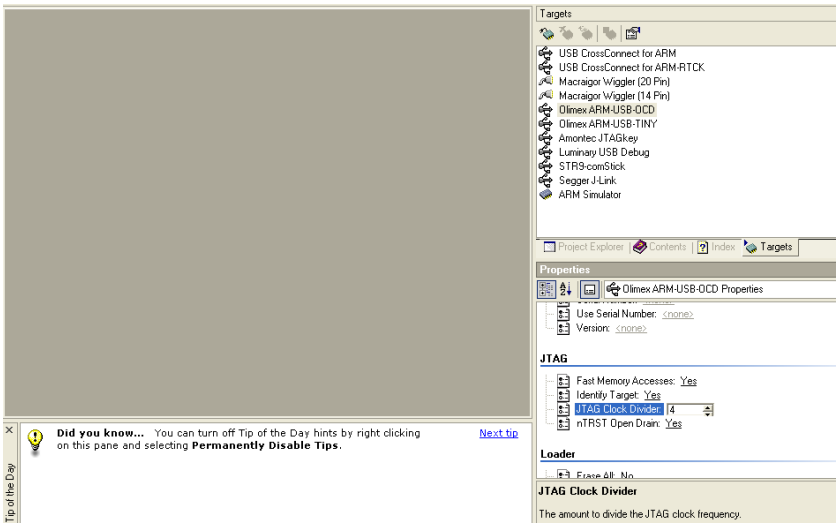


Figura A.1: Instalación paquete LPC2000

Anexo B

Creación de un Proyecto en Crossworks

Para crear un proyecto en Crossworks, basta con presionar: *file* > *New* > *New project*, luego un wizard ayudará con las opciones de configuración del proyecto. Así, aparecerá la siguiente ventana emergente de la figura B.1.

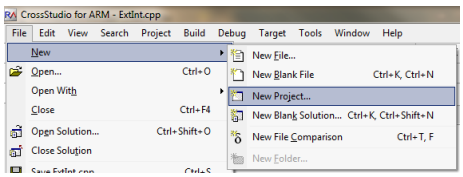


Figura B.1: Crear Proyecto Paso 1

En la figura B.2 se indica que se trabajará con un core de la familia LPC21xx, que el proyecto a realizar será del tipo ejecutable, y posteriormente el path de la carpeta que alojará al proyecto.

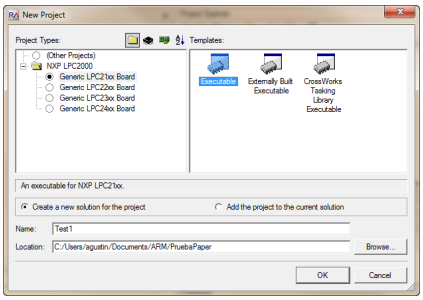


Figura B.2: Crear Proyecto Paso 2

En la siguiente ventana del wizard que se muestra en la figura B.3 se le indica al software que el microprocesador en que se implementará el proyecto, corresponde al LPC2148 y que el clock externo al cual está conectado oscila a una frecuencia de 12MHz.

La ventana siguiente corresponde a los archivos que se añadirán al proyecto, se recomienda tener todas activas (figura B.4).

Luego en la ventana que se muestra en la figura B.5 se pregunta que tipos de configuraciones utilizará el proyecto. Esto se refiere a los modos de debug Flash/RAM, modos THUMB, etc. No se ahondará al respecto, por lo que se recomienda dejar todas las opciones activas.

Finalmente el proyecto está creado, pero no posee ninguna fuente en él. Entonces sólo resta agregar un código y el proyecto estará listo. Para esto:

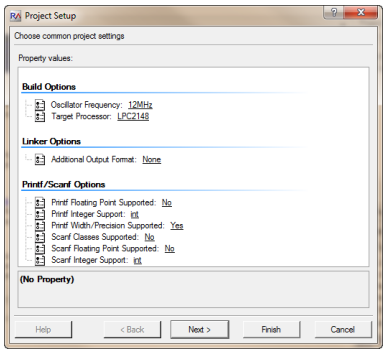


Figura B.3: Crear Proyecto Paso 3

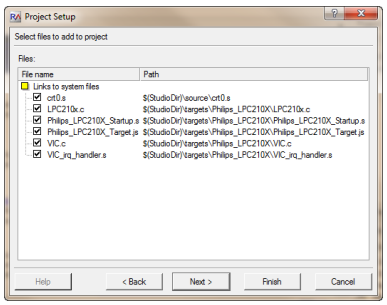


Figura B.4: Crear Proyecto Paso 4

A diferencia de MPLAB, escribir en C/C++ en Crossworks es nativo y no es necesario añadir compiladores externos, por lo que ofrece inmediatamente las opciones para hacerlo.

Se puede comprobar que la fuente se ha agregado al proyecto en el *Project Explorer* como se muestra a continuación.

Si bien, la creación del proyecto ha finalizado mediante el wizard, una compilación no resultaría exitosa, pues en la misma carpeta en que esté el archivo.hzp que maneja Crossworks debe guardarse también el archivo LPC214x.h, el cual se puede descargar desde la página web del LABSEI (labsei.ucv.cl), sección "Publicaciones Internas".

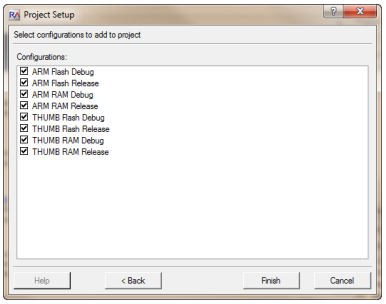


Figura B.5: Crear Proyecto Paso 5

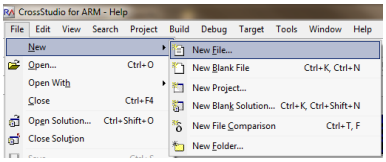


Figura B.6: Agregar Fuente Paso 1

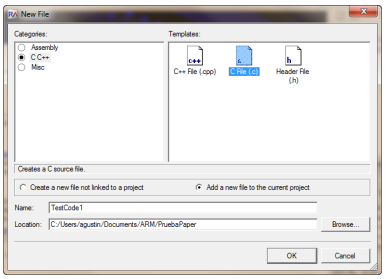


Figura B.7: Agregar Fuente Paso 1

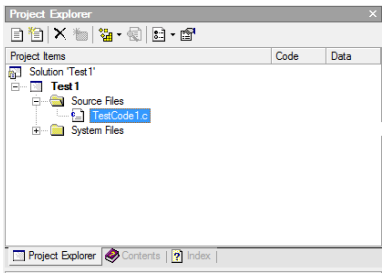


Figura B.8: Fuente agregada al proyecto

Bibliografía

[1] UM10139 Volume 1: LPC214x User Manual Rev. 01.
Koninklijke Philips Electronics
2005

[2] MCI Electronics Esquemático LPC-P2148

[3] Un vistazo a la arquitectura ARM.
Mauro Parra Miranda
UNAM, 2001.
http://usuarios.multimania.es/luiscastillo/bash/arm.pdf

[4] ARM7TDMI-S (Rev 3). Technical Reference Manual
ARM Limited
1998-2000
http://infocenter.arm.com/help/topic/com.arm.doc.ddi0084f/DDI0084.pdf

[5] Embedded Systems: Lecture-5 ARM Processor.
Dr. Santanu Chaudhury
Department of Electrical Engineering, Indian Institute of Technology Delhi
National Program on Technology Enhanced Learning
http://nptel.iitm.ac.in/video.php?subjectId=108102045

[6] Embedded Systems: Lecture-6 More ARM Instructions.
Dr. Santanu Chaudhury
Department of Electrical Engineering, Indian Institute of Technology Delhi
National Program on Technology Enhanced Learning
http://nptel.iitm.ac.in/video.php?subjectId=108102045

[7] Embedded Systems: Lecture-7 ARM: Interrupt Processing.
Dr. Santanu Chaudhury
Department of Electrical Engineering, Indian Institute of Technology Delhi
National Program on Technology Enhanced Learning
http://nptel.iitm.ac.in/video.php?subjectId=108102045

[8] Artículo Arquitectura ARM.
Wikipedia
http://es.wikipedia.org/wiki/Arquitectura_ARM.

Índice de figuras