Universidad de Valladolid

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS DE TELECOMUNICACIÓN

Grado en Ingeniería de Tecnologías Específicas de Telecomunicación

Trabajo Fin de Grado

Diseño de una interfaz para la programación de la memoria interna de los microcontroladores C8051Fxxx de Silicon Labs

Autor:
Jesús García Barajas
Tutor:
Dr. Jesús Arias Álvarez

Abstract

This Final Degree Project consists of programming a microcontroller without the dedicated software and hardware, and facilitate its use to anybody with a minimum knowledge on programming. To achieve this aim, we dispose of a microcontroller with an alternative system of programming its flash memory, another microcontroller which has a developed firmware on a RAM memory to control the mentioned programming system, and a computer as the user interface that provides the data to the firmware.As can be seen, there are three connected devices by different communication systems with advantages and disadvantages. The thing which has in common is the programming language C.

Key Words: Firmware, Microcontroller Programmer, Micontroller Target, User Interface, Bit Banging.

Resumen

En este Trabajo Fin de Grado consiste en lograr programar un microcontrolador sin disponer del software ni hardware dedicado a ello y poder permitir su uso a cualquier persona sin grandes conocimientos de programación. Para lograr esta finalidad se dispone de un microcontrolador con un sistema alternativo de programar su memoria flash, otro microcontrolador que aloja en su memoria RAM el firmware, desarrollado al efecto, que controla el sistema de programación alternativo, y un ordenador que sirve como interfaz de usuario y proporciona los datos necesarios al firmware. Como se puede ver, hay tres dispositivos que están interconectados con sistemas de comunicación diferentes, y cada uno de los cuales presenta sus propias ventajas e inconvenientes. Una cosa les une y es común para todos ellos, el lenguaje de programación C.

Palabras Clave: Firmware, Microcontroller Programmer, Microcontroller Target, Interfaz de Usuario, Bit Banging

Índice general

1. Introducción **15**

1.1. Motivación 15

1.2. Objetivos 15

1.3. Estructura del trabajo 16

1.4. Vista General del trabajo 16

2. Hardware del Proyecto **19**

2.1. Introducción 19

2.2. Dispositivos 19

2.3. Interfaz C2 21

3. Software del Proyecto **24**

3.1. Introducción 24

3.1.1. Interfaz C2 25

3.1.2. Registros de Programación 25

3.1.3. Interfaz de Programación 26

3.2. Firmware 26

3.2.1. Primitivas C2 27

3.2.2. Interfaz de programación de aplicaciones *(APIs)* 35

4. Aplicación **50**

4.1. Introducción 50

4.2. Transmisión Serie 50

4.3. Interfaz de usuario 51

4.4. Formato de trama 53

4.5. Interfaz del Firmware 53

5. Conclusiones **55**

5.1. Conclusiones y Conocimientos adquiridos 55

Anexos **58**

I. Firmware para la Familia C8051Fxxx de Silicon Labs **60**

II. Interfaz de Usuario por Línea de Comandos **80**

III. Interfaz del Firmware **93**

IV. Manuales de Usuario para el Interfaz C2 **99**

V. DataSheetLPC2103 **131**

VI. SFRs C8051F4xxx **145**

VII. Compilación en Linux **151**

VIII. Ficheros .HEX **158**

IX. Configuración de la conexión serie **162**

Lista de Figuras

1.1. Diagrama Hardware 17
1.2. Diagrama Software 18

2.1. Memoria de los microcontroladores C8051F41x 20
2.2. Interconexión de los dispositivos LPC2103 y C8051413 22
2.3. Ejemplo de conexión en laboratorio 22
2.4. Esquema Hardware del Interfaz C2 23

3.1. Esquema C2 para microcontroladores C8051F41x 24
3.2. Bits de estado 25
3.3. Comandos de programación para la memoria flash o EPROM 26
3.4. InitClk() 27
3.5. PulseClk() 28
3.6. Reset() 29
3.7. WriteByte() 30
3.8. ReadByte() 31
3.9. AddressWrite 32
3.10. AddressRead() 33
3.11. Check() 33
3.12. DataWrite() 34
3.13. DataRead() 35
3.14. InitDevice() 36
3.15. InitPI() 37
3.16. WriteFlashBlock() 39
3.17. ReadFlashBlock() 41
3.18. EraseFlashPage() 43
3.19. EraseDevice() 45
3.20. WriteSFR() 47

3.21. ReadSFR() 49

4.1. Interfaz de Usuario 52
4.2. Interfaz del Firmware 54

Lista de Tablas

4.1. Comandos y argumentos para la interfaz de usuario. 51
4.2. Formato de trama para las distintas instrucciones 53

Capítulo 1

Introducción

En este capítulo se detallarán la motivación y los objetivos del presente trabajo.

1.1. Motivación

Los microcontroladores día a día nos van sorprendiendo con nuevas características, nuevos periféricos y nuevas estructuras. En este comienzo de milenio, realmente han dado un gran adelanto con la tecnología Flash en la memoria de programa, que permite programar y borrar la memoria en la propia placa de nuestro sistema (ISP). También permite la reprogramación de la misma sin parar la aplicación (IAP). Con la incorporación de un circuito PLL en el oscilador, permite poder utilizar un cristal de baja frecuencia, así como programar la frecuencia del Bus. Los supervisores de funcionamiento tanto a nivel software como a nivel hardware, es otra mejora relevante, lo que permite reducir el número de componentes externos en nuestro hardware. Las nuevas tecnologías del silicio permiten aumentar la velocidad del bus y disminuir el consumo, así como utilizar encapsulados más pequeños, reduciendo el costo. También la incorporación de un módulo de depuración interno, ha permitido crear nuevas herramientas de desarrollo mucho más económicas, donde se puede tener una emulación en tiempo real.

Sin embargo en muchas ocasiones tenemos que recurrir a la electrónica *"más antigua y convencional"* y al manejo de microcontroladores antiguos de los que no disponemos de entornos de desarrollo integrado (IDE) o estos no funcionan en las versiones más modernas de sistemas operativos. Para estos casos debemos utilizar todo nuestro ingenio para poder seguir dando vida útil a estos microcontroladores que normalmente siempre ofrecen nuevas formas de programación.

Por todos estos motivos, se pretende seguir utilizando la familia de microcontroladores C8051 de Silicon Labs por medio de su memoria flash a la que podemos acceder y realizar labores de escritura y lectura a través del interfaz C2 sin utilizar el hardware ni el software habilitados por la compañía para ello.

1.2. Objetivos

Con la ayuda de los conocimientos adquiridos durante estos años de estudio, manuales, datasheet y libros de programación en C, se pretende conseguir los siguientes objetivos:

- Diseño hardware de un circuito de interfaz que permita la programación de los citados micros a través de su bus de depuración, C2.
- Desarrollo de un firmware para dicha interfaz.
- Una aplicación de línea de comandos para entornos Linux que permita cargar en la memoria flash interna de los micros C8051Fxxx ficheros en formatos binario e Intel-Hexadecimal y ejecutar otras aplicaciones del firmware desarrollado para estos dispositivos

1.3. Estructura del trabajo

Este Trabajo Fin de Grado constara de los siguientes apartados:

- **Capitulo 1. Introducción**
  Se describe el porqué de este proyecto, los objetivos y una visión general del mismo.
- **Capitulo 2. Hardware del Proyecto**
  En este capítulo se verán los dispositivos usados y se concretara en el medio fundamental para acceder a la memoria flash del C8051Fxxx que es el Interfaz C2.
- **Capitulo 3. Software de Proyecto**
  Se describe el funcionamiento del Interfaz C2 y como se debe realizar la programación en el *MCu Programmer* para su correcta utilización. Ademas se detalla el firmware desarrollado para el acceso a la memoria flash del C8051Fxxx.
- **Capitulo 4. Aplicación**
  Se trata lo referente a la aplicación en linea de comandos, el menú de opciones mediante el paso de argumentos, la comunicación entre el ordenador y el *MCu Programmer*, y los protocolos creados para ello.
- **Capitulo 5. Conclusiones**
  Se hablará de las conclusiones finales del trabajo y los conocimientos adquiridos.
- **Anexos**
  Contendrá los códigos y manuales aquí mencionados.

1.4. Vista General del trabajo

Para tener una visión general del trabajo y comprender la finalidad del mismo se analizaran tanto a nivel hardware, como de software todos los medios participantes para así en capítulos posteriores desarrollarlos de manera más particular.

Hardware

En este aspecto vamos a interconectar el microcontrolador C8051F431 (*MCu Target*) con el microcontrolador LPC2103 (*Mcu Programmer*) para ello se empleara el hardware C2 presente en el C8051Fxxx y que proporciona acceso directo a la memoria Flash de dicho microcontrolador, que es el objetivo final de este proyecto. El hardware C2 consiste principalmente en una serie de registros pero lo que interesa

son las dos entradas, una para la señal de reloj (C2CK) y otra para la señal de datos (C2D), que actuara en su caso también como salida. Tanto la linea C2CK como la C2D deberán ir conectadas a alguno de los pines de la GPIO del LPC2103. Ahora bien, los archivos y comandos se pasarán desde el ordenador por lo que este ira conectado por medio del USB a la UART del LPC2103 por lo que se hace necesario el uso del convertidor USB-Serie FT232RL, que esta integrado en la placa del LPC2103. La aplicación en linea de comandos deberá instalarse en el ordenador, que al disponer de muchos recursos es aquí donde se deben realizar las máximas operaciones posibles, mientras que el firmware y main del C8051Fxxx ira en la memoria SRAM de LPC2103, que esta limitada a 8kB. Otra razón para que el firmware este alojado en el LPC2103, es la portabilidad, pues cualquier persona con alguna noción de programación podría usarla en otros ordenadores sin tener la aplicación de linea de comandos.

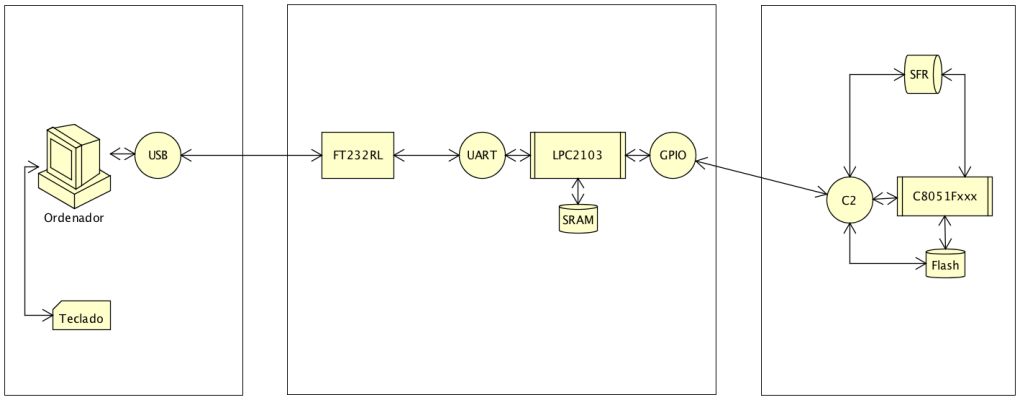


Figura 1.1: Diagrama Hardware

Software

La programación sera completamente en el lenguaje C y se utilizara el compilador GCC para ARM "arm-none-eabi-gcc" desde el sistema operativo GNU/Linux UBUNTU. En el ordenador va alojada la aplicación de linea de comandos que ofrece distintas opciones a través del paso de parámetros. En el ordenador es donde se convierten todos los datos a binario y se construye una trama con estos, de acuerdo a un protocolo establecido. Esta trama se envía por medio del protocolo serie a la UART del LPC2103. En este punto la trama que llega es interpretada por medio de un programa desarrollado para ello y con estos datos ya se puede ejecutar una instrucción del firmware determinada. Este firmware esta diseñado para la comunicación con el dispositivo C2 del C8051Fxxx y se realizara mediante Bit Banging entre los pines de la GPIO del LPC2103 y las entradas C2 (C2CK y CD2). Ahora es cuando el interfaz C2 actúa sobre la Memoria flash o los Registros de Funciones Especiales (*SFR*) mediante una serie de lecturas y escrituras en los registros de la misma.

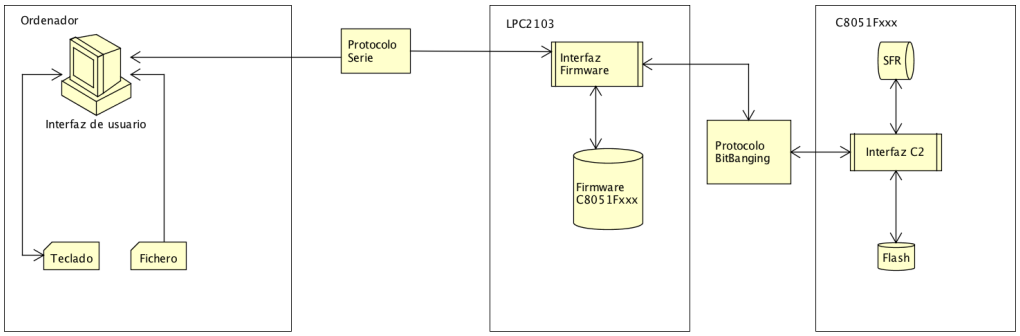


Figura 1.2: Diagrama Software

Capítulo 2

Hardware del Proyecto

2.1. Introducción

En este capítulo se verán los dispositivos usados y se concretara en el medio fundamental para acceder a la memoria flash del C8051Fxxx que es el Interfaz C2.

2.2. Dispositivos

C8051F413 de Silicon Labs

Los microcontroladores de la familia C8051F41x de Silicon Labs tienen un bus de datos de 8 bits, arquitectura Harvard, conjunto de instrucciones CISC, además de otras características que pueden ser consultadas en el datasheet del dispositivo.

La estructura de la memoria de un microcontrolador de la familia C8051F41x es la siguiente:

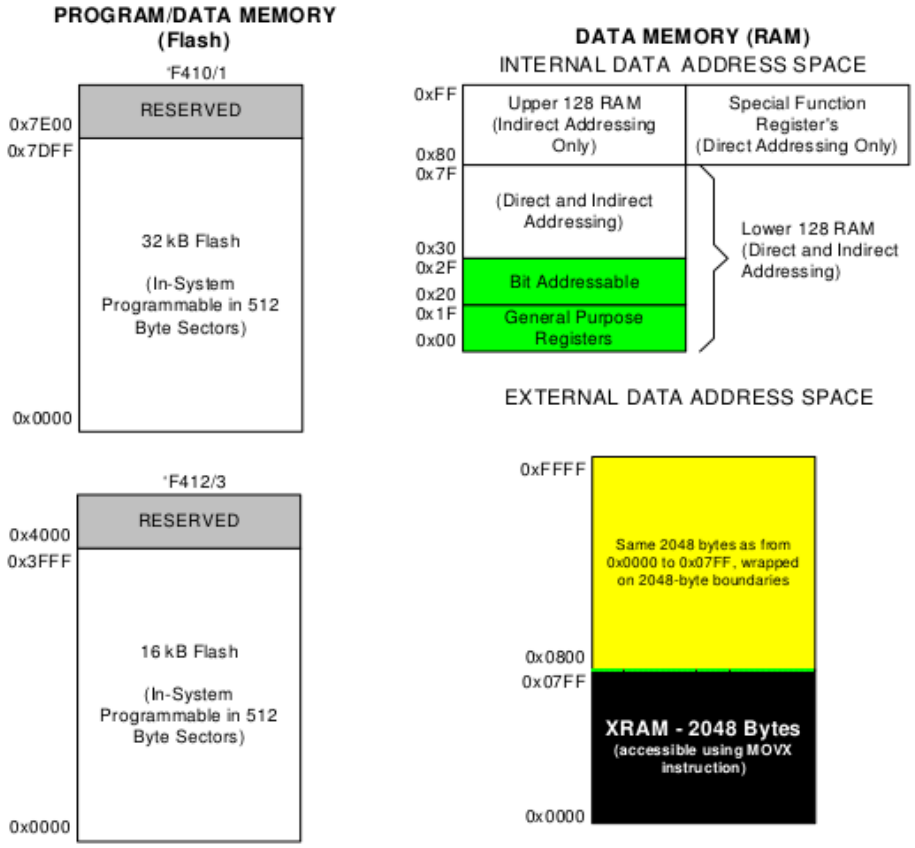


Figura 2.1: Memoria de los microcontroladores C8051F41x

Presenta 256 bytes de datos RAM, de los 128 bytes superiores se puede acceder con direccionamiento indirecto, o con direccionamiento directo al espacio de direcciones SFR de propósito especifico. Los 128 bytes inferiores de RAM son accesibles mediante direccionamiento directo e indirecto. Los primeros 32 bytes son direccionables como cuatro bancos de registros de propósito general y los siguientes 16 bytes pueden ser direccionables por bytes o por bit. El C8051F41x puede seleccionar uno de los 4 bancos de registros de los que dispone, mediante los bits RS0 y RS1 del registro de estado PSW. En ocasiones puede incluir 2048 bytes de RAM, mapeados en el espacio de memoria de datos externo.

La memoria Flash del programa consta de 32 kB (F410 y F411) o 16 kB (F412 y F413). Esta memoria puede ser reprogramada en el sistema en sectores de 512 bytes cada uno y no requiere ningún voltaje especial de programación fuera de chip.

Los *SFRs* se encuentran en la RAM interna del microcontrolador, entre la dirección 80H y 0FFH, permitiendo el acceso directo exclusivamente. Los SFRs son registros destinados, en su mayoría, al control de los periféricos integrados en el C8051F41x. Por medio de distintos *SFRs* se puede: acceder a los puertos de entrada/salida del; leer o escribir en el puerto serie del microcontrolador; controlar los temporizadores y contadores; configurar el sistema de interrupciones, etc. Más información de los *SFR* en el anexo VI.

LPC2103

Es el microcontrolador que almacenara en su memoria SRAM de 8kB el firmware. Algunas características del LPC2103 de NXP son: conjunto de instrucciones RISC, memoria Von Neumann, 32-bit

de datos y un core de alto rendimiento y bajo consumo de la familia ARM7TDMI-S. Para mas información consultar el datasheet del dispositivo en el anexo V.

El microcontrolador LPC2103 cumplira el papel de *MCu Programmer* y sera el encargado de realizar las tareas necesarias para actuar sobre la memoria flash del *MCu Target* por medio del interfaz C2. La técnica empleada en la comunicación sera la de *Bit Banging* por la cual mediante el software diseñado al efecto, este establece y muestrea directamente el estado de los pines en el microcontrolador, y es responsable de todos los parámetros de la señal: sincronización, niveles y temporización. Con ello obtenemos una comunicación serie bit a bit a muy bajo coste pues no necesitamos del uso de ningún otro hardware adicional.

Todo esto es posible porque ambos microcontroladores utilizan los mismos niveles eléctricos CMOS de 0V *Low* y 3.3V *High*.

PC

Es donde se visualizan los resultados y utiliza la linea de comandos para ejecutar las distintas funciones del firmware. Debe estar provisto de un sistema operativo GNU/Linux y del compilador cruzado de C gcc-arm-none-eabi el cual se utiliza junto con las librerías del LPC2103. El PC se une al LPC2103 a través del USB y es transformado a una comunicación serie en la placa del LPC2103 por medio de un convertidor USB-Serie FT232RL.

2.3. Interfaz C2

Los dispositivos de la familia C8051 incluyen una interfaz de depuración de 2 hilos (C2) integrada, para permitir la programación de la memoria flash o EEPROM y la depuración del sistema, lo cual no es tratado en este proyecto. El Interfaz C2 utiliza una señal de reloj (C2CK) y una señal de datos bidireccional (C2D) para transferir información entre el dispositivo y otro sistema electrónico. Los principales puntos a tratar sobre esta interfaz son la conexión, los registros a utilizar y la memoria flash. Más información la tenemos en el datasheet del dispositivo y los manuales de uso del C2 incluidos en el anexo IV.

Conexión

La interfaz C2 de Silicon Labs es un protocolo de comunicación en serie de dos hilos diseñado para permitir la programación en el sistema y eliminar errores en los dispositivos Silicon Labs de bajo número de pines. La comunicación C2 implica un maestro de interfaz (el programador, depurador o probador) y un objetivo de interfaz (el dispositivo que se va a programar, depurar o probar). Los dos cables utilizados en la comunicación C2 son los Datos (C2D) y el Reloj (C2CK).

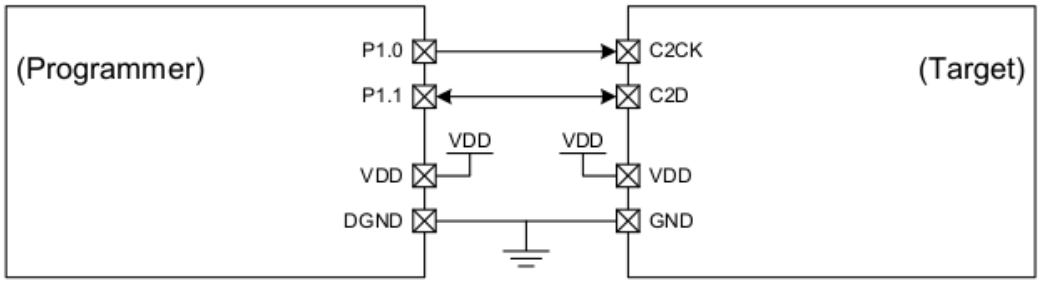


Figura 2.2: Interconexión de los dispositivos LPC2103 y C8051413

Como es lógico hay que alimentar el *MCu Target* que en la fotografía lo hace el mismo *MCu Programmer.*

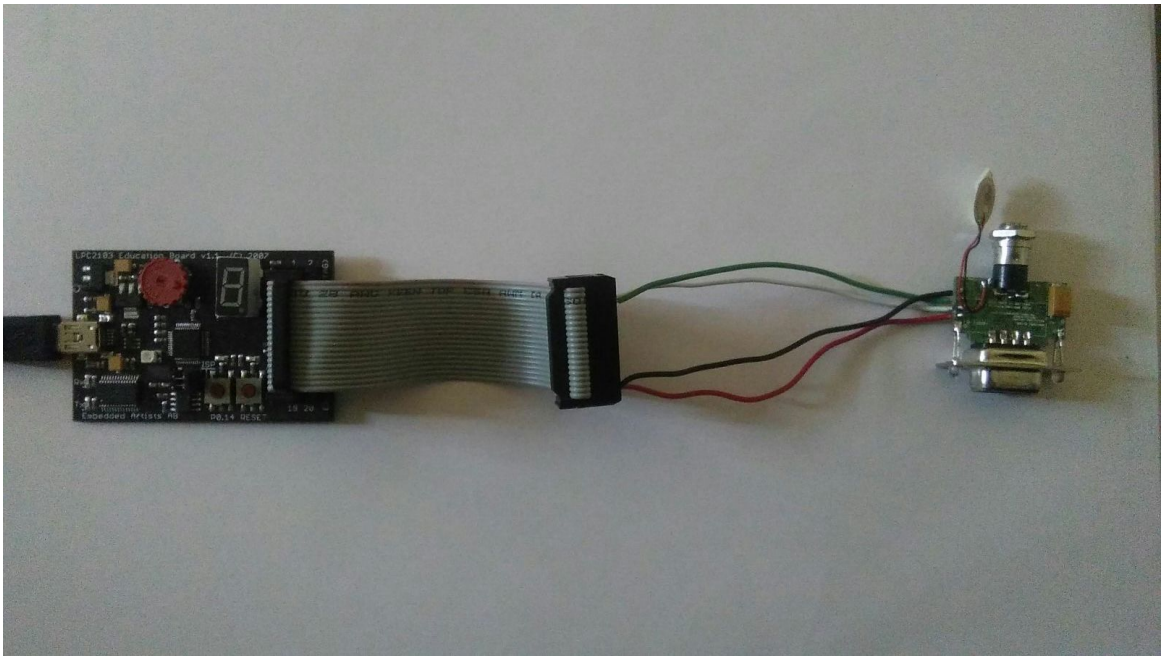


Figura 2.3: Ejemplo de conexión en laboratorio

Esquema Hardware

Para la programación vía interfaz C2 se utilizan una serie de puertas lógicas, desplazadores y registros:

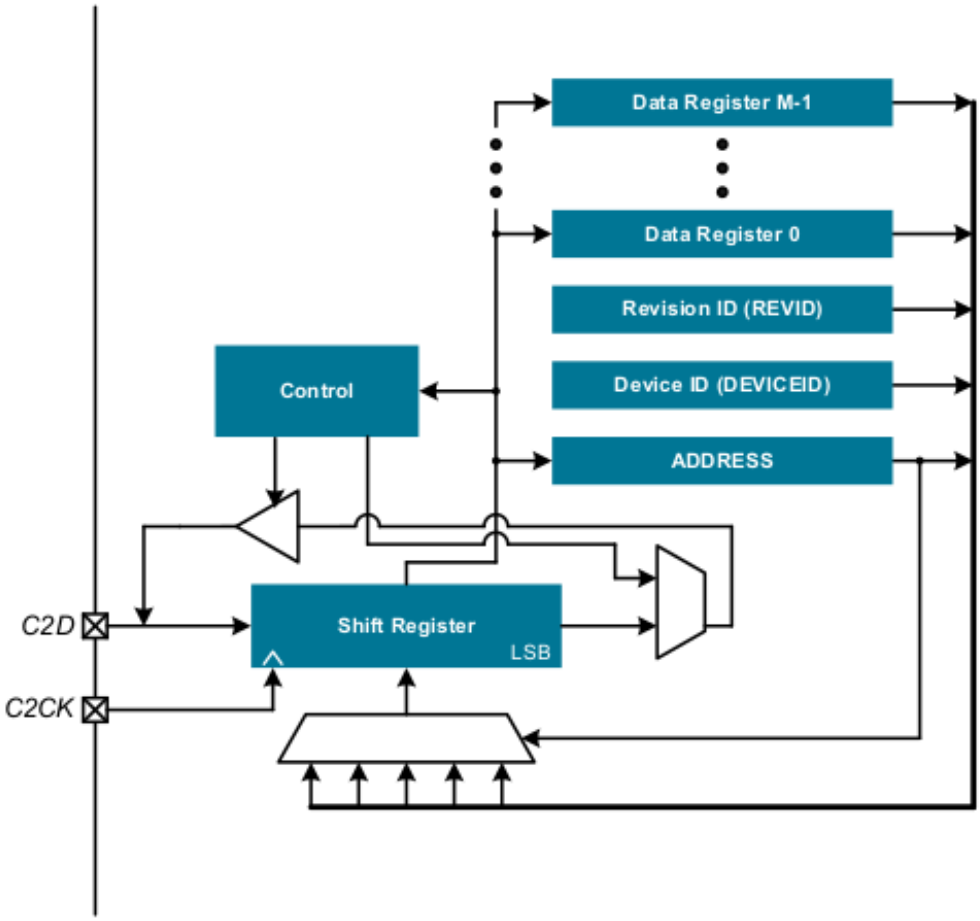


Figura 2.4: Esquema Hardware del Interfaz C2

Memoria Flash

La memoria Flash ,integrada y programable, sobre la que actúa el interfaz C2 se utiliza para el código de programa y el almacenamiento de datos no volátil. Una vez ha sido programada al 0 lógico, se debe borrar de nuevo para volverla al 1 lógico antes de ser reprogramada, pues los bytes Flash adquieren el valor 0xFF cuando son borrados. Las operaciones de escritura y borrado son temporizadas automáticamente por el hardware para una ejecución adecuada y no es necesario el sondeo de datos para determinar su final. También hay que tener en cuenta que la ejecución de código se bloquea durante las operaciones de escritura y borrado de la memoria Flash.

Capítulo 3

Software del Proyecto

3.1. Introducción

Los dispositivos C8051Fxxx tienen una Interfaz de programación (*PI*) que se accede a través de la interfaz C2 y un conjunto de registros de programación. El diagrama de bloques de acceso Flash se muestra a continuación.

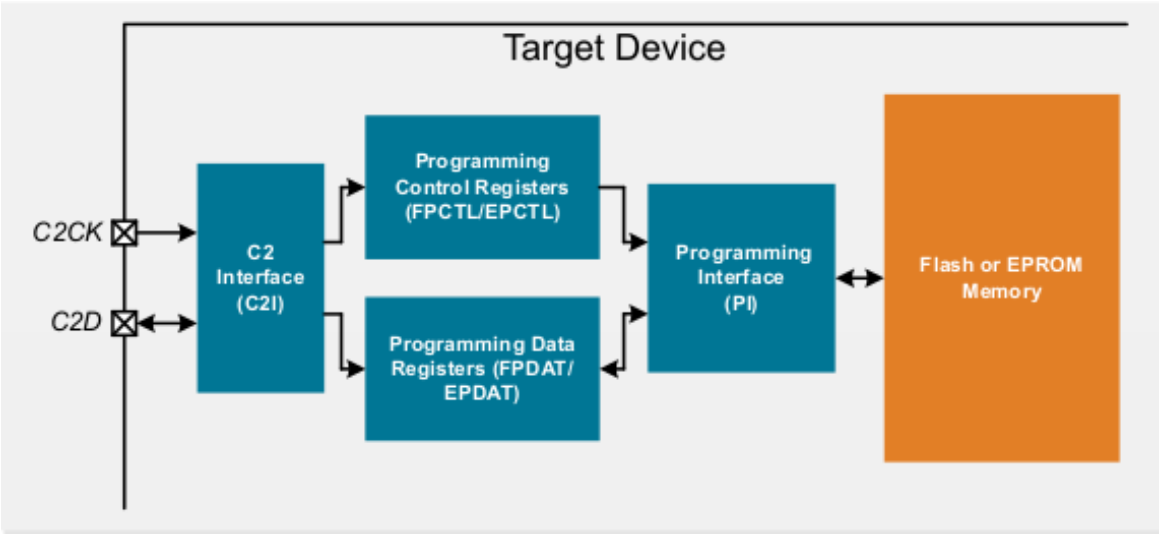


Figura 3.1: Esquema C2 para microcontroladores C8051F41x

Podemos dividir todo lo necesario para acceder a la memoria flash vía interfaz C2 en tres puntos:

- Interfaz C2.
- Registros de programación (*RI*).
- Interfaz de programación (*PI*).

3.1.1. Interfaz C2

La interfaz C2 consta de una registro de dirección y acceso a hasta 256 registros de datos. Este acceso es proporcionado por la capa física C2, visto en la Figura 2.4:

En esta parte se utilizan los registros:

- **C2 Address(*ADDRESS*):**
  El registro de direcciones C2 (*ADDRESS*) cumple dos funciones en la programación C2 Flash:
  - Selecciona a qué registro de datos C2 se accede durante los ciclos de lectura y escritura de datos C2.
  - Proporciona información de estado del *PI*.

  Las lecturas de direcciones, comando *AddressRead()*, se usa frecuentemente durante la programación C2 como un esquema de establecimiento de contactos entre el programador y el *PI*, retornando un código de estado de 8 bits, formateado como se muestra en la tabla siguiente:

Bit	Name	Description
7	EBusy or FLBusy	This bit indicates when the EPROM or Flash is busy completing an operation.
6	EError	This bit is set to 1 when the EPROM encounters an error.
5:2	—	Unused
1	InBusy	This bit is set to 1 by the C2 Interface following a write to FPDAT. It is cleared to 0 when the PI acknowledges the write to FPDAT.
0	OutReady	This bit is set to 1 by the PI when output data is available in the FPDAT register.

Figura 3.2: Bits de estado

- **C2 Device ID (*DEVICEID*):**

  El registro identificador de dispositivo (*DEVICEID*) es un registro de datos de sólo lectura que contiene un identificador de dispositivo de 8 bits el cual indica la familia de microcontroladores a la que pertenece. La dirección para el registro *DEVICEID* es 0x00.

- **Revision ID Register (*REVID*):**

  El registro identificador de revisión (*REVID*) es un registro de datos C2 de sólo lectura que contiene el identificador de revisión de 8 bits del dispositivo C2. La dirección del registro *REVID* es 0x01.

3.1.2. Registros de Programación

La comunicación en el Interfaz de Programación y el Interfaz C2 tiene lugar por medio de los Registros de Programación:*FPCTL* y *FPDAT*.

- **C2 Flash Programming Control (*FPCTL*):**

Este registro se utiliza para activar la programación de la memoria Flash a través de la interfaz C2. Para activar la programación C2 Flash, los siguientes códigos deben escribirse en este orden: 0x02, 0x01, incluido el reset correspondiente al inicio. Hay que tener en cuenta que una vez que la programación C2 Flash esté activada, se debe emitir un reset del sistema para reanudar el funcionamiento normal. La dirección del registro *FPCTL* es 0x02.

- **C2 Flash Programming Data (*FPDAT*):**

  Este registro se utiliza para pasar comandos, direcciones y datos a la memoria Flash durante los accesos del *PI*.La dirección del registro *FPDAT* depende del dispositivo pero pueden ser 0xAD o 0xB4.

3.1.3. Interfaz de Programación

El Interfaz de Programación Flash o *PI* es un conjunto de comandos de programación que se ejecuta utilizando una secuencia de lecturas y escrituras del registro *FPDAT*.Los comandos válidos se muestran a continuación y son los únicos que actúan directamente en la memoria flash

Command	Code	Supported Devices
Block Write	0x07	all (flash or EPROM)
Block Read	0x06	all (flash or EPROM)
Page Erase	0x08	flash devices only
Device Erase	0x03	flash devices only
Get Version	0x01	all (flash or EPROM)
Get Derivative	0x02	all (flash or EPROM)
Direct Read	0x09	all (flash or EPROM)
Direct Write	0x0A	all (flash or EPROM)
Indirect Read	0x0B	all (flash or EPROM)
Indirect Write	0x0C	all (flash or EPROM)

Figura 3.3: Comandos de programación para la memoria flash o EPROM

3.2. Firmware

La programación en el microcontrolador se realiza mediante el lenguaje de programación C sobre el microcontrolador LPC2103 que mediante su GPIO actuara mediante *Bit Banging* en el *MCu Target* C8051Fxxx de Silicon Labs. Como compilador se ha utilizado el GCC para ARM "arm-none-eabi-gcc" desde el sistema operativo GNU/Linux UBUNTU.

La programación utiliza 2 niveles, una serie de comandos para transmitir y recibir las señales eléctricas del *MCu programmer* al *MCu Target* y que este interpretara como distintas funciones esenciales de lectura y escritura de los registros *ADDRESS, DEVICEID y REVID*, y un segundo nivel donde propiamente se actuara sobre la memoria Flash, realizado por el PI, en el que se emplean los registros *FPDAT* y *FPCTL*.

Los registros de funciones especiales (*SFR*), que se encargan del control y configuración de la CPU y sus periféricos, son accesibles a los 2 niveles, pero hay que tener en cuenta que algunos bits de

estos pueden ser fijos por lo que la operación de escritura no tiene efecto, en todo caso siempre puede realizarse la lectura.

Para el desarrollo del Firmware se ha utilizado el propio manual que el fabricante suministra para ello y que puede ser consultado en el anexo IV, mientras que el código C se encuentra en el anexo I.

3.2.1. Primitivas C2

En este apartado se analizan las operaciones básicas para actuar en el C8051Fxxx e inicializar el *PI.*

- **Macros C2:**
  Mediante una serie de macros estos pondrán como salida o entrada la línea de datos C2D y si esta es salida, a baja o alta, así como generar los pulsos de la señal de reloj llevando a alta o baja la señal C2CK.

```
#define C2D (1<<5)
#define C2CK (1<<4)
#define C2D_INPUT IO0DIR &=~(C2D);
#define C2D_OUTPUT IO0DIR |=(C2D);
#define C2D_READ ((IO0PIN & (C2D))>>5);
#define C2D_1 IO0SET |=C2D;
#define C2D_0 IO0CLR |=C2D;
#define C2CK_H IO0SET |=C2CK;
#define C2CK_L IO0CLR |=C2CK;
```

- **void InitClk():**
  Para inicializar la señal de reloj C2CK , poner el puerto como salida y permanecer en alta.

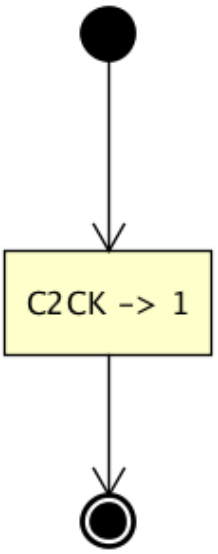


Figura 3.4: InitClk()

- **void PulseClk():**
  Para generar un pulso de reloj, este debe ser de menos de 5 µs en baja y después llevado de nuevo a alta.

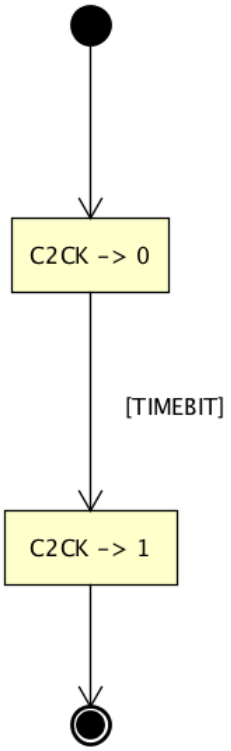


Figura 3.5: PulseClk()

- **void Reset():**
  Antes de comenzar cualquier operación de programación hay que realizar un reset y este se genera mediante la señal de reloj con al menos 20 µs en baja y llevado a alta.

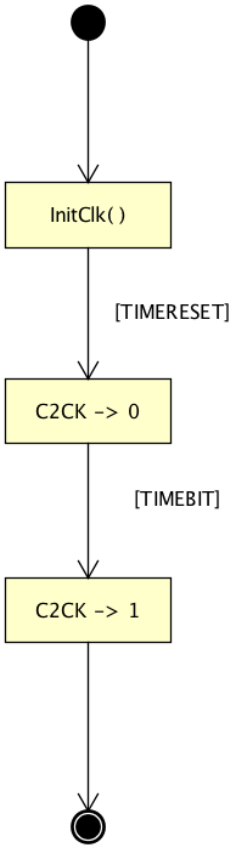


Figura 3.6: Reset()

- **void WriteByte(int byte):**
  Esta función genera la secuencia de bits del byte pasado como parámetro en el puerto C2D. Sera utilizada por otras funciones para cualquier operación de escritura.

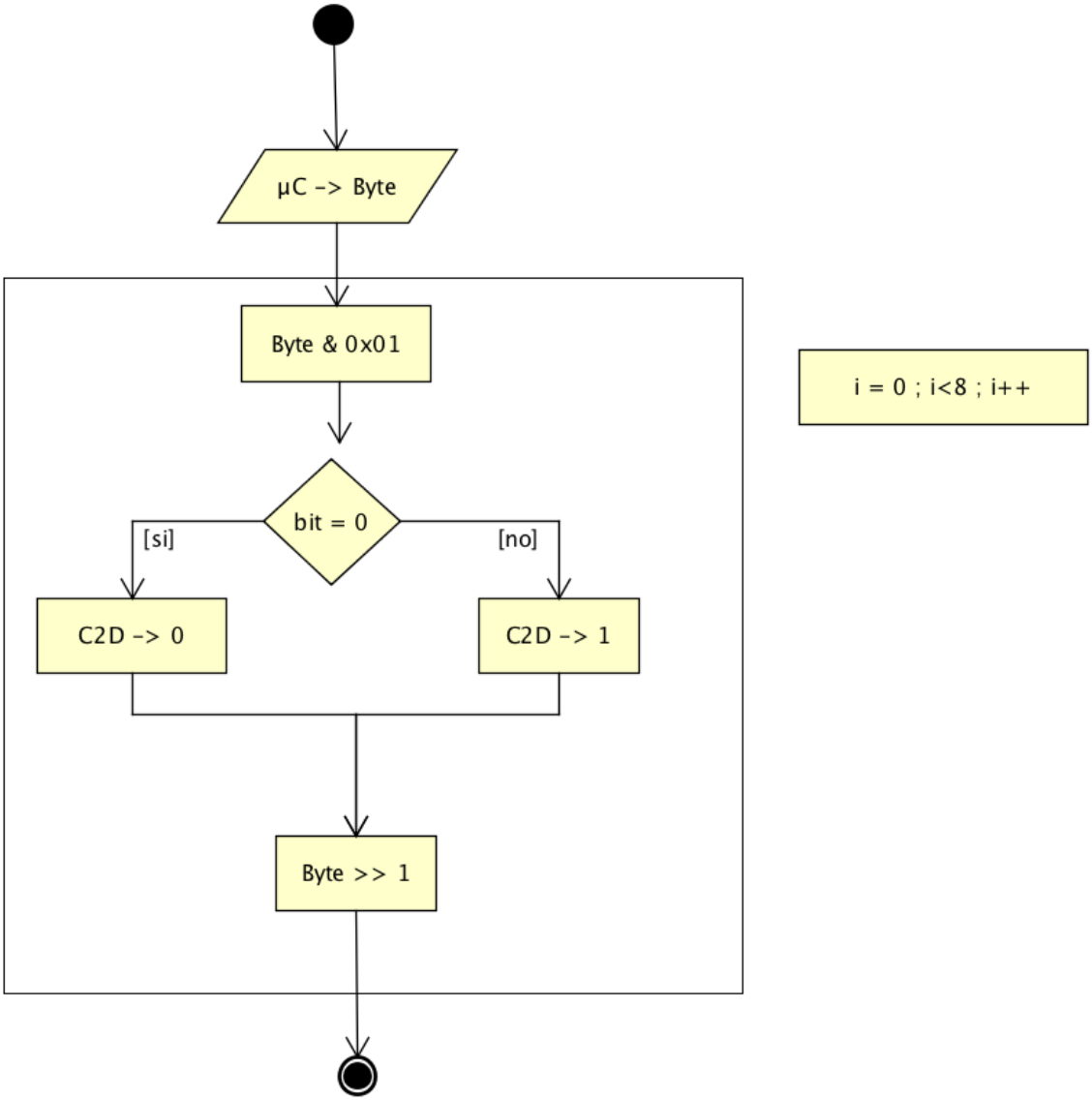


Figura 3.7: WriteByte()

- **int ReadByte():**
  Esta función lee la secuencia de bits que proporciona el C8051 y genera un byte con dichos bits para ser retornado como resultado de la llamada a esta función. Sera utilizada por otras funciones para cualquier operación de lectura.

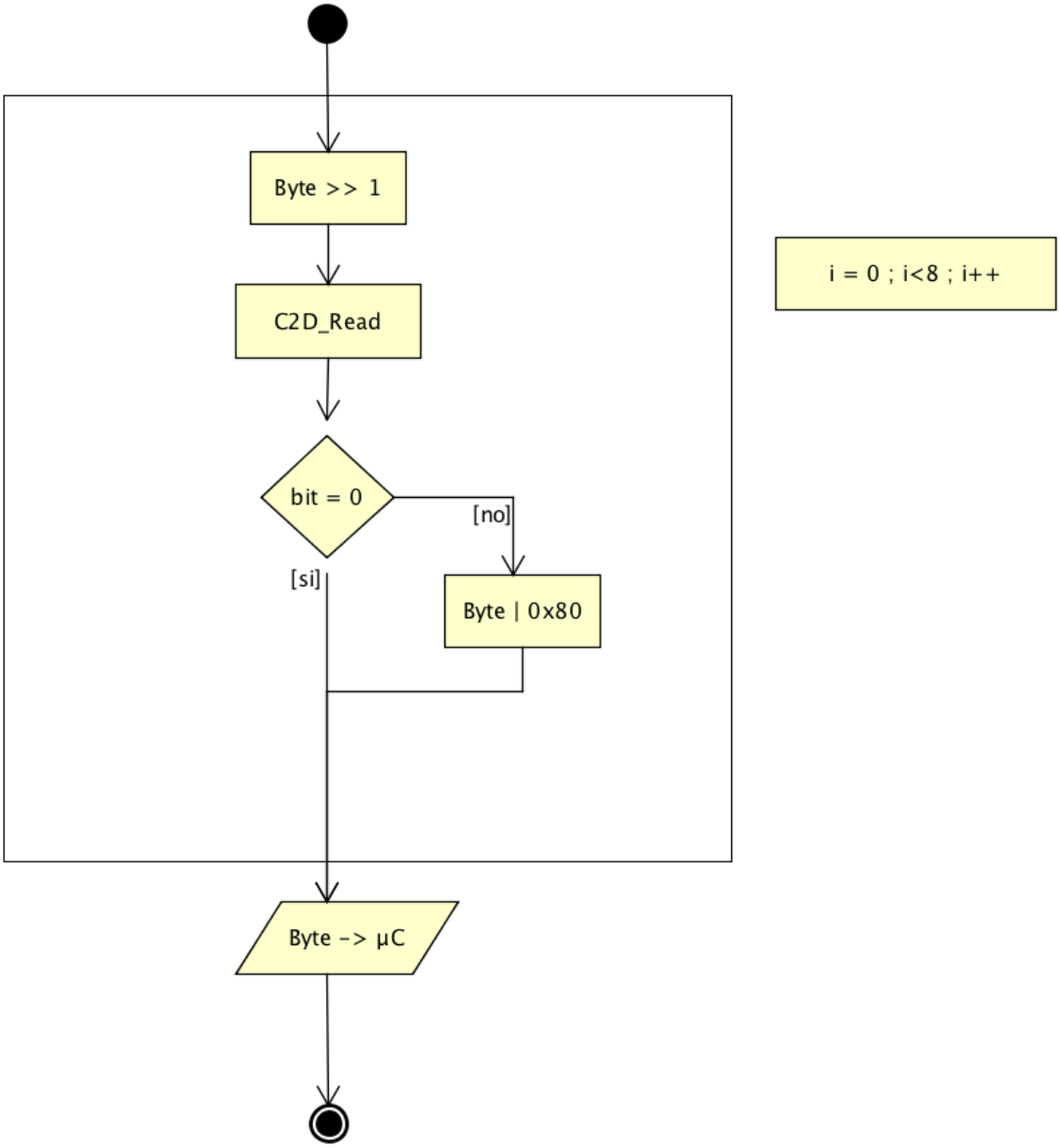


Figura 3.8: ReadByte()

- **void AddressWrite(int byte):**
  Para escribir un byte en el registro C2 *ADDRESS* y proporcionar la dirección del registro de datos al cual queremos acceder. Sera utilizada por las funciones tanto de lectura como de escritura de datos.

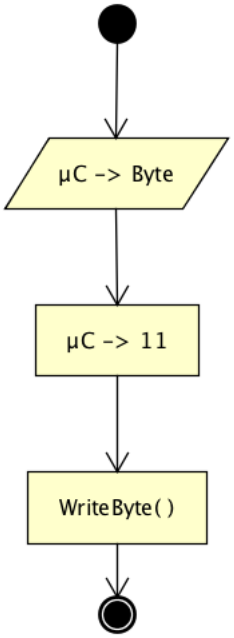


Figura 3.9: AddressWrite

- **int AddressRead():**
  Para leer el registro C2 *ADDRESS* y retorna los 8 bits del registro de estado, lo cual es frecuentemente utilizado durante la programación C2 para la comprobación de la salida y llegada de datos.

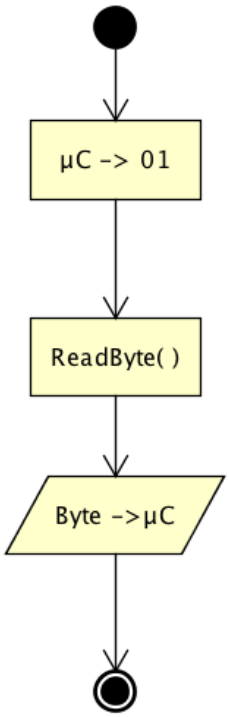


Figura 3.10: AddressRead()

- **int Check():**
  Realiza la lectura del puerto C2D y retorna el valor presente en el puerto. Sera utilizada por las funciones de lectura y escritura de datos para proporcionar la espera necesaria al procesamiento de los mismos por parte del C8051Fxxx.

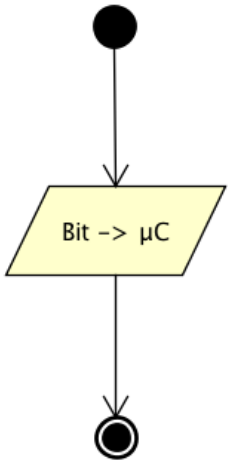


Figura 3.11: Check()

- **void DataWrite(int byte):**
  Escribe el byte pasado como parámetro al registro de datos cuya dirección ha sido indicada por el registro *ADDRESS*.

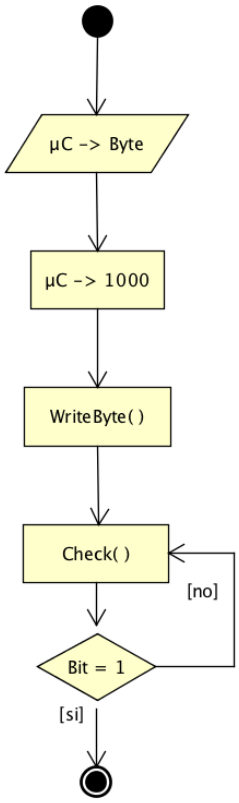


Figura 3.12: DataWrite()

- **int DataRead():**
  Retorna el valor del registro de datos cuya dirección ha sido indicada por el registro ADDRESS.

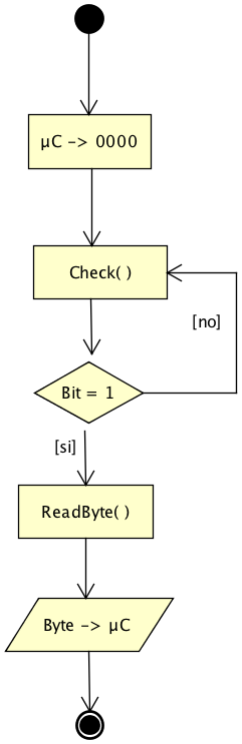


Figura 3.13: DataRead()

3.2.2. Interfaz de programación de aplicaciones *(APIs)*

El interfaz de programación proporciona un conjunto de comandos para permitir el acceso a la memoria FLASH del C8051Fxxx. Todos estos comandos utilizaran las primitivas C2 anteriores así como los registros de programación C2 (*ADDRESS, DEVICEID, FPCTL y FPDAT*). Además estos comandos están provistos de contadores para evitar bucles infinitos y errores de datos por lo que retornan un 0 si todo es correcto, o un 1 si se ha producido un error.

- **int InitDevice():**
  Retorna la dirección del registro *FPDAT* según el valor proporcionado por la lectura de un comando *DataRead()* después de un reset. Dicho valor depende del modelo de dispositivo de la familia C8051Fxxx.

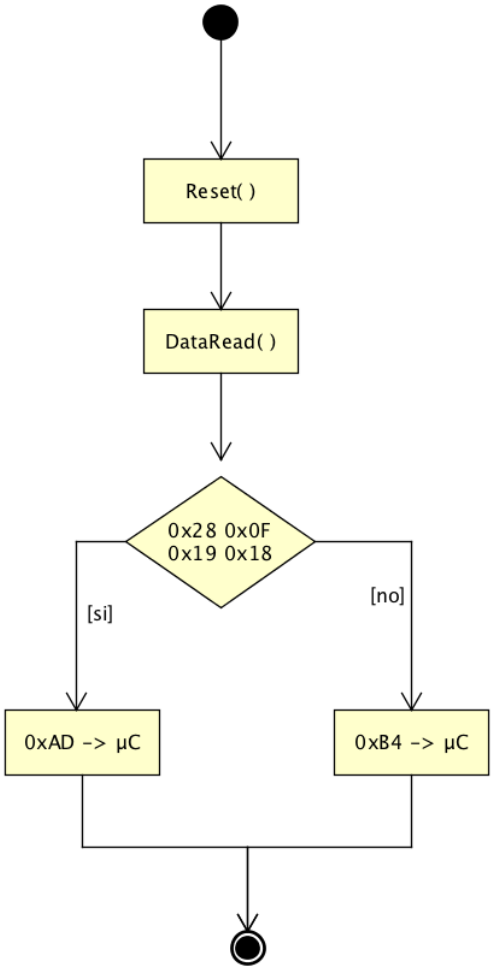


Figura 3.14: InitDevice()

- **void initPI():**
  Antes de utilizar cualquier comando se debe inicializar el modulo PI mediante una secuencia de instrucciones y un tiempo de espera de 20 ms.

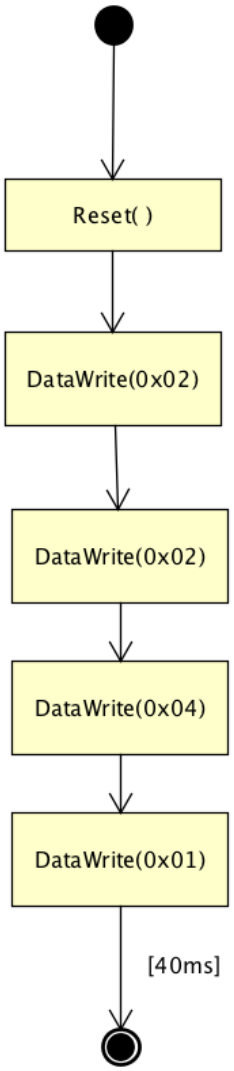


Figura 3.15: InitPI()

- **int WriteFlashBlock(int FPDAT, int Block[ ],int AddressC, int Lenght):**
  Toda escritura en la memoria flash debe hacerse mediante este comando en el cual se pasa como parámetro la dicción del registro *FPDAT*, la dirección inicial, un vector con los datos a escribir y el número de datos. El tamaño del bloque de datos va desde 1 a 256, con lo que la escritura de tamaños superiores debe hacerse de manera secuencial.

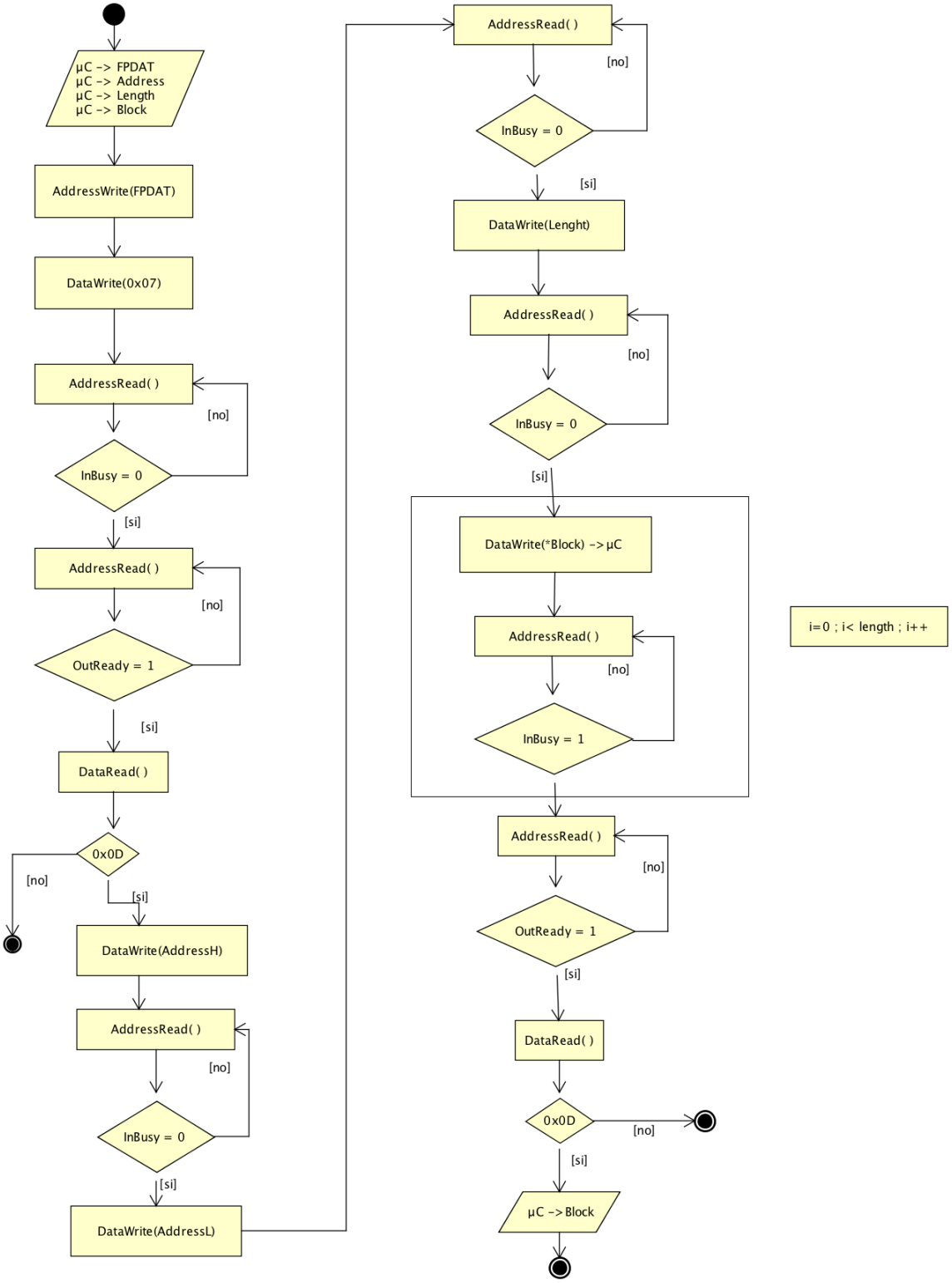


Figura 3.16: WriteFlashBlock()

- **int ReadFlashBlock(int FPDAT, int Block[ ],int AddressC, int Lenght):**
  Toda lectura en la memoria flash debe hacerse mediante este comando en el cual se pasa como parámetro la dicción del registro *FPDAT*, la dirección inicial, un vector donde serán escritos los datos a leer de la memoria flash y el número de datos. El tamaño del bloque de datos va desde 1 a 256, con lo que la lectura de tamaños superiores debe hacerse de manera secuencial.

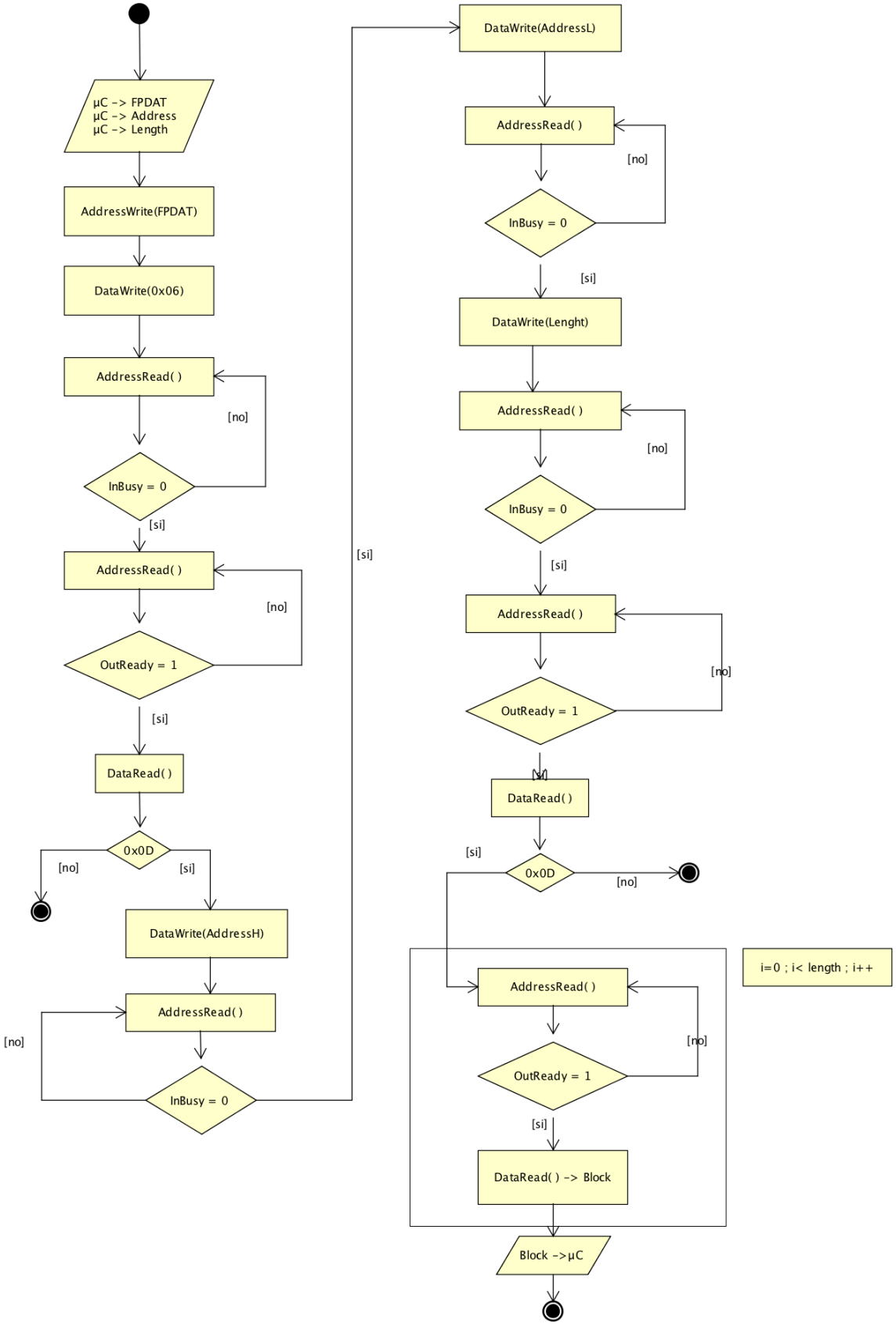


Figura 3.17: ReadFlashBlock()

- **int EraseFlashPage (int FPDAT, int NumPage):**
  La memoria flash está dividida en páginas, de tamaños de 512 a 1024 bytes según el dispositivo. Para realizar un borrado de la memoria en páginas se utiliza dicho comando indicando la dirección del *FPDAT* y el número de pagina.

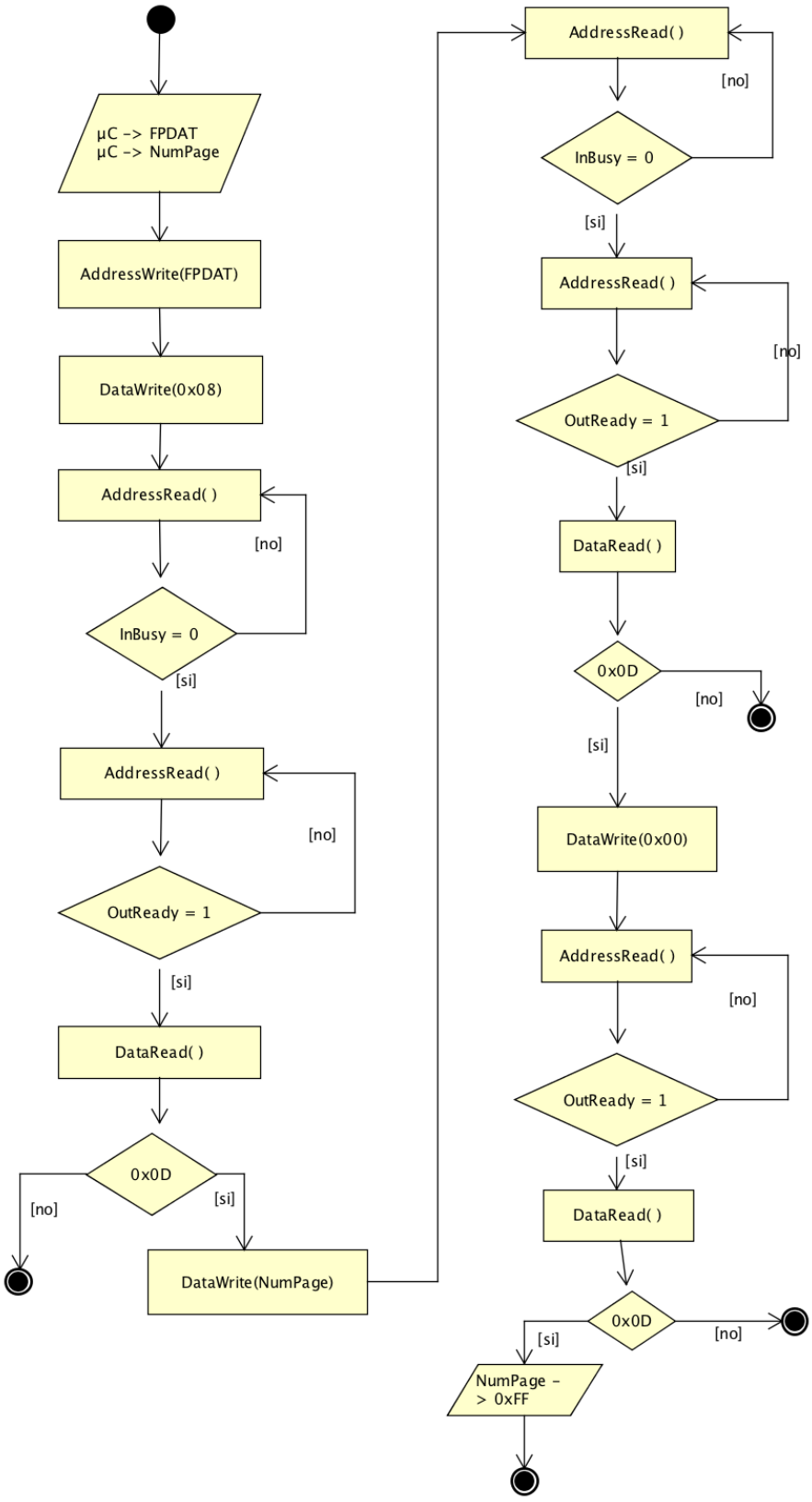


Figura 3.18: EraseFlashPage()

- **int EraseDevice(int FPDAT):**
  Para realizar un borrado completo de la memoria flash se utiliza dicho comando indicando la dirección del *FPDAT* del dispositivo.

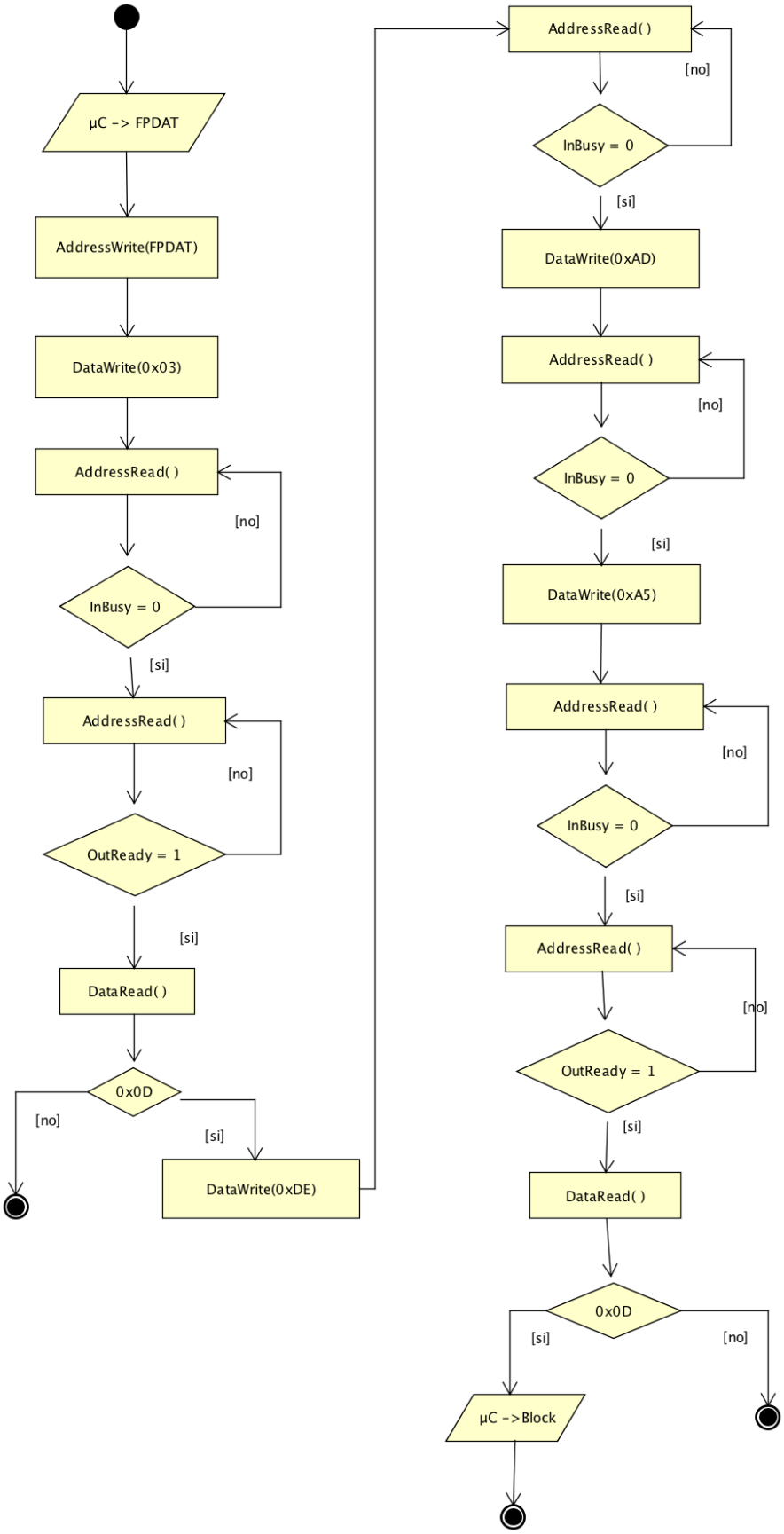


Figura 3.19: EraseDevice()

- **int WriteSFR(int FPDAT, int AddressSFR, int NewSFR):**
  Esta función permite modificar cualquier registro de funciones especiales que este habilitado para ello. Se pasan como parámetro la dirección del registro *FPDAT*, la dirección del *SFR* a modificar y el nuevo contenido a escribir en el registro.

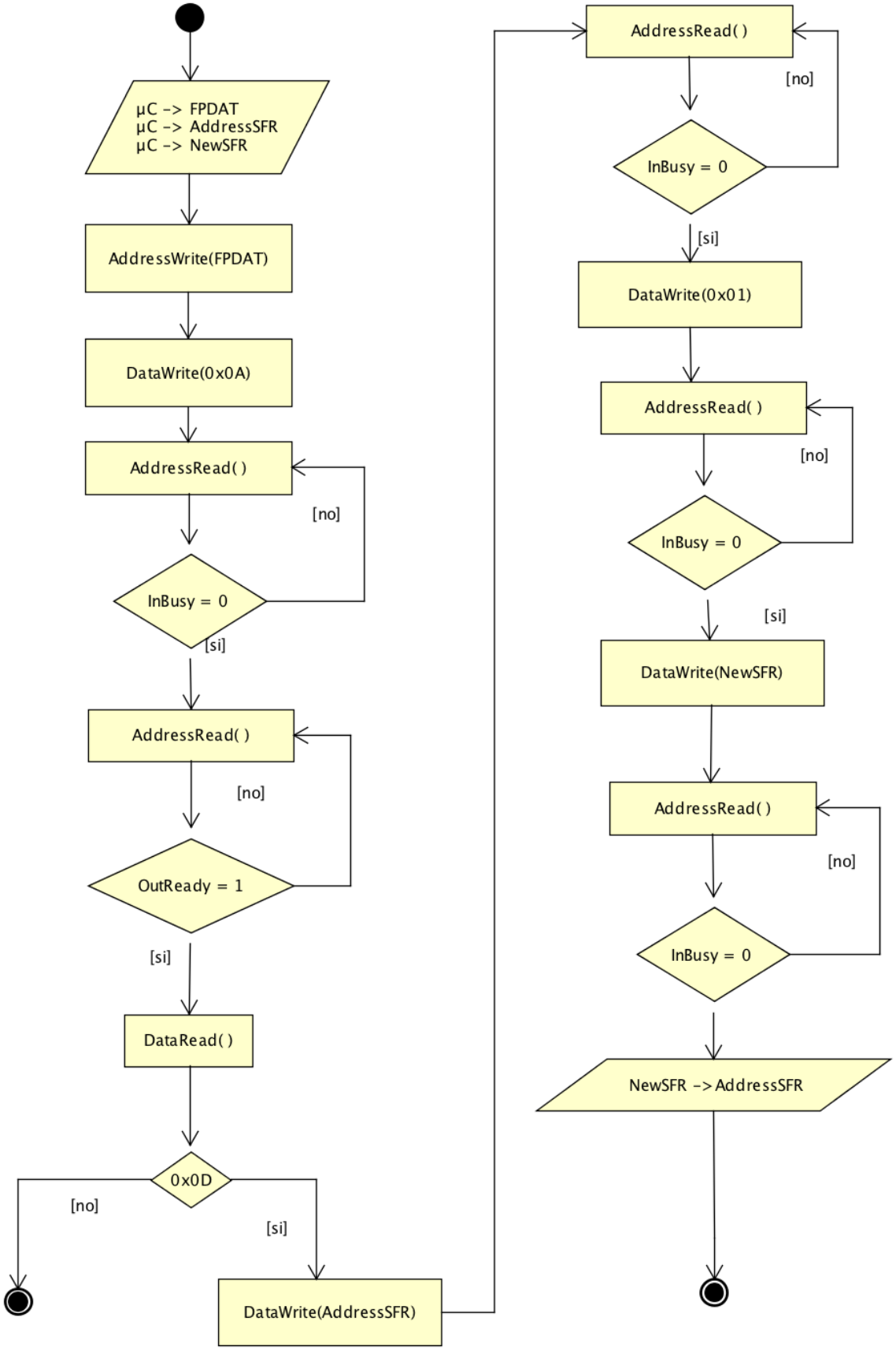


Figura 3.20: WriteSFR()

- **int ReadSFR(int FPDAT, int AddressSFR,):**
  Esta función permite leer cualquier registro de funciones especiales. Se pasan como parámetro la dirección del registro *FPDAT* y la dirección del *SFR* a leer, mostrando el contenido del *SFR* direccionado.

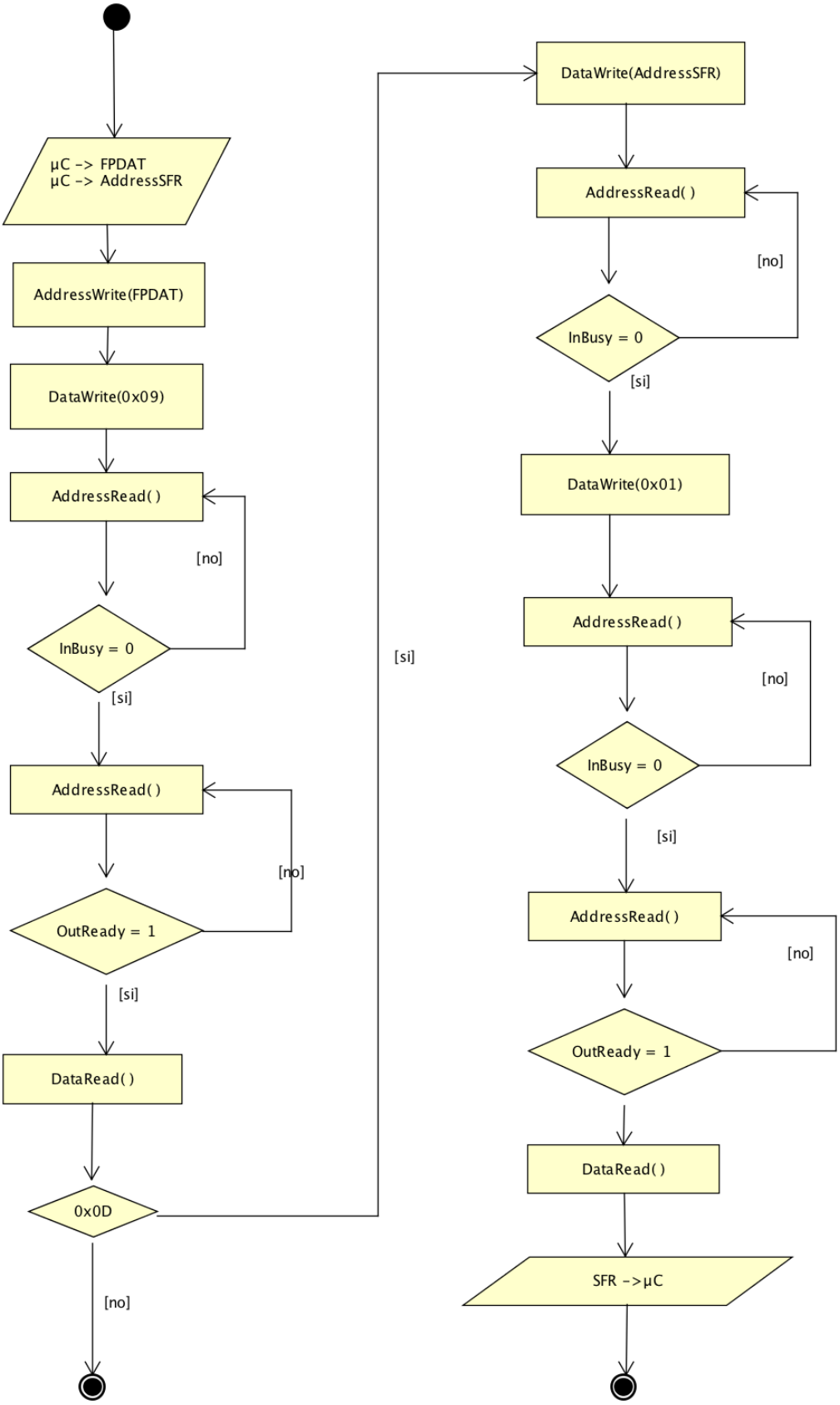


Figura 3.21: ReadSFR()

Capítulo 4

Aplicación

4.1. Introducción

En este capítulo se detallarán las cuestiones relativas al diseño de la aplicación en linea de comandos de Linux, funciones empleadas, la intercomunicación PC-LPC2103 y los diagramas de secuencia. La idea que subyace en esta aplicación es que a partir de un comando con unos argumentos determinados el programa deberá tratar estos datos para que puedan ser decodificados por el *MCu Programmer* de manera rápida y sin errores, y este actuar en consonancia con los datos recibidos sobre el Interfaz C2 del *MCu Target*. Para la transmisión de dichos datos se debe utilizar un protocolo de comunicación preestablecido y correctamente configurado que en este caso sera la transmisión serie.

4.2. Transmisión Serie

El medio físico para realizar la transmisión del ordenador al LPC2103 sera un cable USB que ira conectado la patilla A al puerto USB del ordenador y la Mini-B al LPC2103, aunque esto es lo menos interesante pues hoy en día es común utilizar USB para la conexión de periféricos. Aunque el medio sea el cable USB, el protocolo de comunicaciones que se usa es el Protocolo Serie por lo que antes de llegar a la UART0 del LPC2103 hay que realizar la conversión USB-Serie mediante el circuito integrado FT232RL. El LPC2103 esta equipado con dos UART y ambas permiten la variación de los parámetros de la transmisión serie por medio de los bancos de registros asociados a la UART, lo cual es muy importante pues los recursos del LPC2103 son muy limitados y no es necesario utilizar librerías y funciones de C que consuman gran cantidad de memoria. Este problema no le tiene el PC, luego desde aquí es donde se realiza la configuración de la conexión y donde se utilizan librerías y funciones. Podemos consultar los manuales en anexo IX y IX. La comunicación utilizada en el proyecto tiene las siguientes características de configuración :

- **Velocidad Serie: 115200 baudios**
- **Longitud de dato: 8 bits**
- **Sin bit de paridad**
- **Bit de Stop**
- **Buffer FIFO de 16 byte (LPC2103)**

4.3. Interfaz de usuario

Como se ha ido comentando el interfaz de usuario irá alojado en el ordenador, y sera del tipo de línea de comandos. En el a partir de un archivo ejecutable y una serie de argumentos podremos realizar todas las funciones desarrolladas en el firmware del C8051Fxxx. La obtención del archivo ejecutable esta explicada en el documento del anexo VII, así como el código C de la aplicación II. Los argumentos válidos y su sintaxis están en la siguiente tabla:

Tabla 4.1: Comandos y argumentos para la interfaz de usuario.

Descripción	**Ejecutable**	**Comando**	**Argumento 1**	**Argumento 2**
Ayuda	$./a.out	-h		
Copiar Archivo.hex	$./a.out	-w	<Archivo.hex>	
Copiar Archivo.bin	$./a.out	-b	<Archivo.bin>	
Leer Flash	$./a.out	-r	<Dirección>	<Número de Bytes>
Borrar Página	$./a.out	-d	<Número de Página>	
Borrar Dispositivo	$./a.out	-e		
Escribir SFR	$./a.out	-s	<Dirección SFR>	<Nuevo SFR>
Leer SFR	$./a.out	-l	<Dirección SFR>	

Después de escribir correctamente el ejecutable y sus argumentos, se guardan los datos proporcionados, inicializa la conexión serie con el LPC2103, y según el comando recibido seguirá un hilo u otro del programa, pero en cualquier caso la finalidad de todos ellos es la formación de una trama de datos, ya todos en binario, para transmitirla por medio de la conexión serie. Por supuesto una vez realizada la operación se espera la señal de retorno (ACK) para indicar el éxito de la ejecución del programa o el fracaso de la misma (NACK). En caso de ser un comando recurrente como ocurre en la copia de archivos .bin y .hex, se volvería a formar otra trama con nuevos datos , así hasta completar el archivo o llenar la memoria flash. El diagrama de flujo del interfaz de usuario se observa a continuación

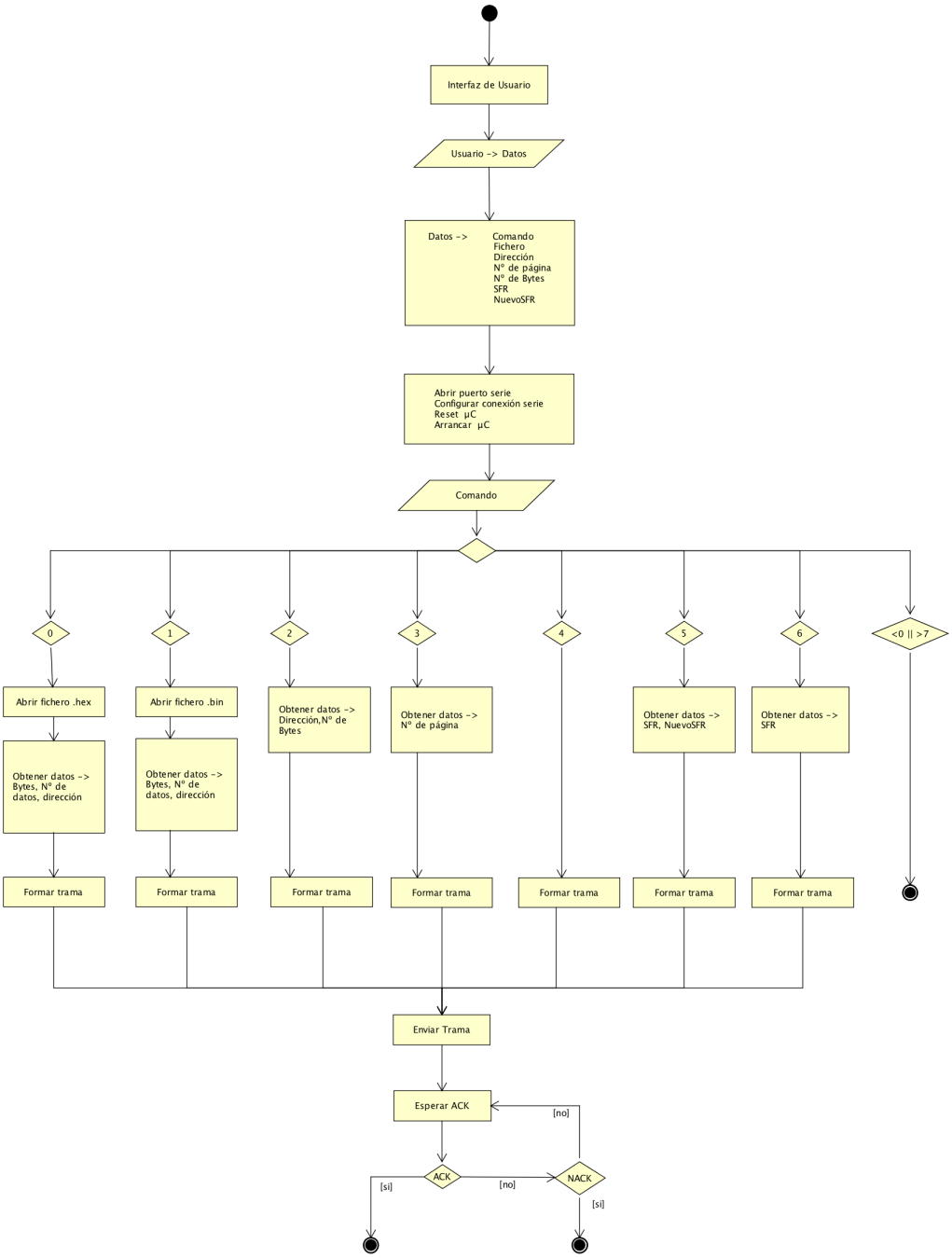


Figura 4.1: Interfaz de Usuario

Para llevar a cabo todas las operaciones se utilizan las siguientes funciones auxiliares:

- **void safewrite(int fd,char *buf,int n):**
  Permite garantizar el envió de un buffer de datos a través de la UART. Se pasan como parámetros el identificador de fichero, un puntero al buffer de datos y el número de datos a transmitir.
- **int writeSerial(int fd, unsigned int *buffer, int ilength):**
  Envía un buffer de datos por la UART. Se pasan como parámetros el identificador de fichero, un puntero al buffer de datos y el número de datos a transmitir.

- **static int ascii2hex(char c):**
  Transforma un carácter ASCII a su valor hexadecimal.
- **int TransStrHex(unsigned char a):**
  Transforma los caracteres de un archivo.hex a su valor hexadecimal, la especial naturaleza de este tipo de archivos se explica en el anexo VIII.

4.4. Formato de trama

Los formatos de trama serán diferentes para cada comando, pero en todos ellos el primer byte enviado indicara el comando y todas finalizan con 0x0D, 0x0A.

Tabla 4.2: Formato de trama para las distintas instrucciones

Descripción	**Byte 0**	**Byte 1**	**Byte 2**	**Byte 3**	**Byte 4**	**Byte 5**	**Byte n-1**	**Byte n**
Ayuda								
C. Archivo.hex	0x00	Nº de Bytes	Dirección H	Dirección L	Dato 0	.....	0x0D	0x0A
C. Archivo.bin	0x01	Nº de Bytes	Dirección H	Dirección L	Dato 0	.....	0x0D	0x0A
Leer Flash	0x02	Nº de Bytes	Dirección H	Dirección L	0x0D	0x0A		
Borrar Página	0x03	Nº de Página	Nº de Página	0x0D	0x0A			
B. Dispositivo	0x04	0x0D	0x0A					
Escribir SFR	0x05	Dirección SFR	Nuevo SFR	0x0D	0x0A			
Leer SFR	0x06	Dirección SFR	0x0D	0x0A				

4.5. Interfaz del Firmware

Este programa recibirá la trama, la interpretara y realizara una de las funciones desarrolladas en el firmware. Una vez terminada la operación contestamos al PC mediante un ACK o NACK según el resultado obtenido. Su código se encuentra en el anexo III y su diagrama de flujo a continuación:

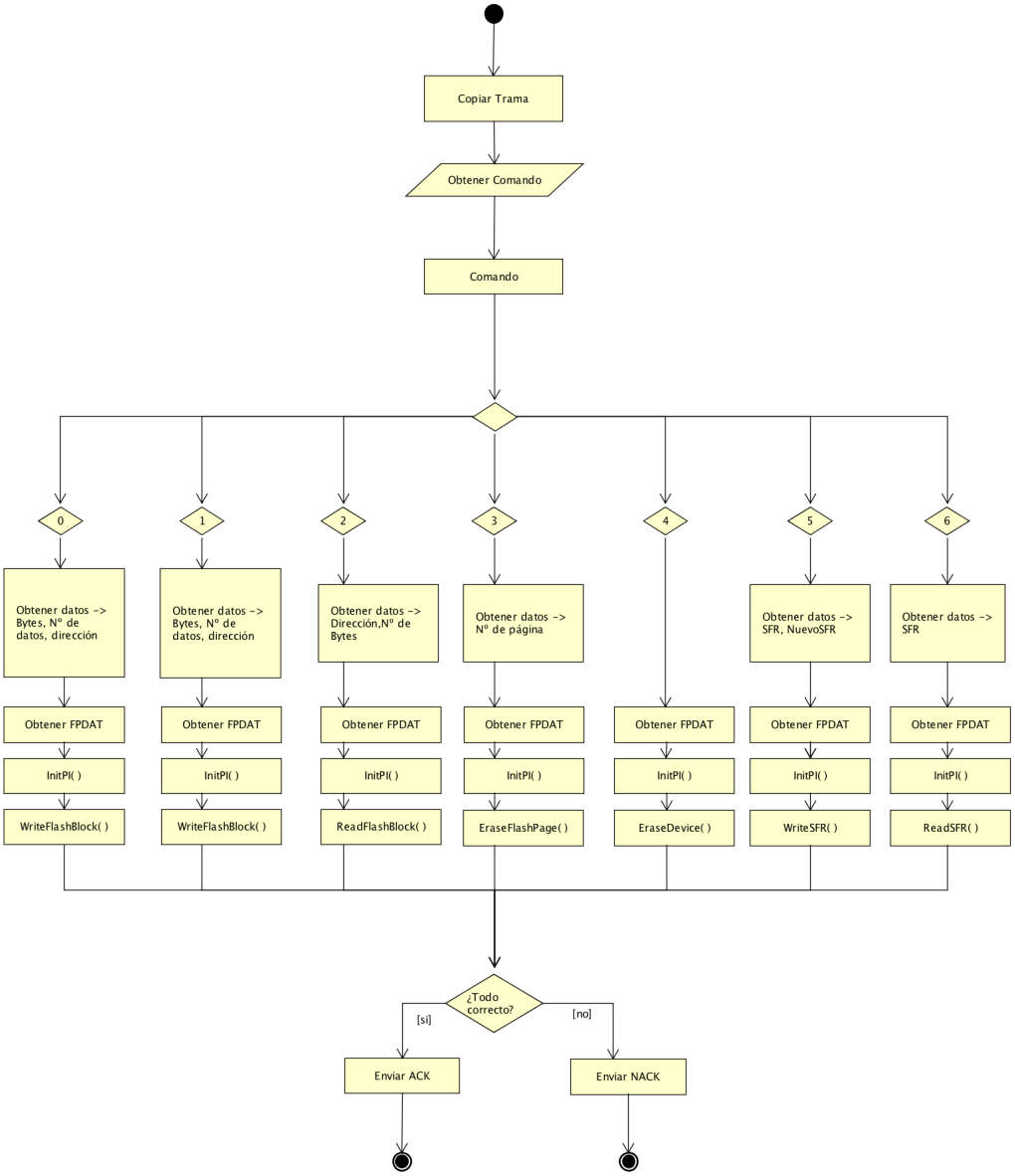


Figura 4.2: Interfaz del Firmware

Se utilizan las siguientes funciones auxiliares:

- **void UART0putch(char c):**
  Permite enviar un carácter a través de la UART. Se pasa como parámetro el carácter a enviar
- **char UART0getch():**
  Lee un carácter cuando llega a la UART y es retornado por la función.

Capítulo 5

Conclusiones

5.1. Conclusiones y Conocimientos adquiridos

Los objetivos para este proyecto se han cumplido de manera satisfactoria. Se ha logrado desarrollar un firmware y una interfaz de usuario sencillos pero con toda la funcionalidad y tolerantes frente a fallos.

El Diseño hardware se ha explicado de manera clara y se han expuesto todos los pasos que siguen los datos entre los dispositivos y los diferentes tipos de protocolos de comunicación que se emplean en sus enlaces. Además hemos visto como cada dispositivo ofrecía distintos recursos y se ha tenido que adaptar la programación a ello.

El firmware es completamente funcional, esta bien explicado y su código es fácilmente comprensible. Hay que añadir que es exportable a cualquier otro microcontrolador, realizando pequeños cambios en el código como son la selección de las patillas y el puerto serie.

La interfaz de usuario es sencilla, puede originar errores con el paso de argumentos, pero por ello se emplea un comando de ayuda y se muestran antes de su ejecución los datos recogidos.

Durante la realización de este proyecto se han adquirido nuevos conocimientos del desarrollo de firmware , del lenguaje de programación C y del funcionamiento interno de los microcontroladores. Además de ha tenido que consultar gran cantidad de manuales y datasheet, de manera que se han manejado otros aspectos que sin aparecer en este documento han sido de gran valor didáctico. Otro matiz que merece ser destacado es la utilización de LaTeX. para la elaboración de la memoria, que ha servido para conocer todo el potencial de esta herramienta.

Bibliografía

[1] J. L. Antonakos and K. C. J. Mansfield, *Programación estructurada en C.* Madrid: PRENTICE HALL IBERIA, 1997.

[2] B. S. Gottfried, *Programación en C.* 28023 Aravaca (Madrid): McGraw Hill, 2005.

[3] Silicon Labs, "An127: Flash programming via the c2 interface,"

[4] Silicon Labs, "C8051f410/1/2/3 Datasheet,"

[5] NXP, "LPC2101/02/03 Datasheet," 2009.

[6] D. Márquez, "El fichero hex explicado," *Primer Congreso Virtual de Microcontroladores*, 2009.

[7] F. d. I. Universidad Complutense de Madrid, "Cómo compilar y ejecutar programas en línea de comandos,"

[8] P. H. Baumann, "Cómo programar el puerto serie en linux," 1997.

[9] F. P. Rivera, "Lnxcomm: Linux communication," 2011.

Anexos

Anexo I

Firmware para la Familia C8051Fxxx de Silicon Labs

```
1  //-----------------------------------------------------------------------------
2  // FILE NAME: FirmwareC8051.c
3  // TARGET DEVICE:          C8051Fxxx
4  // PROGRAMMER DEVICE:         LPC2103
5  // CREATED ON:           21/08/2017
6  // CREATED BY:                  JESUS GARCIA BARAJAS
7  //
8  // Code to program a C8051Fxxx device through its C2 interface.
9  //
10 // This software targets a LPC2103 device, though it could be easily
11 // ported to any device.
12 //
13 // Code assumes the following Port pin connections:
14 // LPC2103: | C8051Fxxx:
15 // P0.4     -> C2CK
16 // P0.5     -> C2D
17 // VDD      -> VDD
18 // GND      -> GND
19 //
20 // Program includes:
21 //- FLASH programming routines (Block Write, Block Read, Page Erase,
22 //  Device Erase,SFR White and SFR Read)
23 //- Primitive C2 communication routines (Address Write, Address Read,
24 //  Data Write, Data Read, Reset, and other support routines)
25 //- Test software for the above routines
26 //
27 // Calling sequences for FLASH Programming Routines:
28 //
29 // C2 Initialization Procedure:
30 // 1) Call InitDevice()
31 // 2) Call InitPI()
32 //
```

```
33 // Note: All following routines assume the C2 Initialization Procedure
34 // has been executed prior to their calling
35 //
36 // Block Read Procedure:
37 // 0) Set parameter <FPDAT> provided by InitDevice()
38 // 1) Set parameter <AddressC> to the target FLASH starting address
39 // 2) Set parameter <Length> to the number of bytes to read
40 // 3) Set the pointer <Block> to the location to store the read data
41 // 4) Call ReadFlashBlock() (will return a '0' if successful,'1' otherwise)
42 //
43 // Block Write Procedure:
44 // 0) Set parameter <FPDAT> provided by InitDevice()
45 // 1) Set parameter <Address> to the target FLASH starting address
46 // 2) Set parameter <Length> to the number of bytes to write
47 // 3) Set the pointer <Block> to the location of the data to be written
48 // 4) Call WriteFlashBlock() (will return a '0' if successful,'1' otherwise)
49 //
50 // Page Erase Procedure:
51 // 0) Set parameter <FPDAT> provided by InitDevice()
52 // 1) Set parameter <NumPage> to a location within the page to be erased
53 // 2) Call EraseFlashPage() (will return a '0' if successful,'1' otherwise)
54 //
55 // Device Erase Procedure:
56 // 0) Set parameter <FPDAT> provided by InitDevice()
57 // 1) Call EraseDevice() (will return a '0' if successful,'1' otherwise)
58 //
59 // SFR Write Procedure:
60 // 0) Set parameter <FPDAT> provided by InitDevice()
61 // 1) Set parameter <AddressSFR> to the location of the SFR
62 // 2) Set parameter <NewSFR> to be written
63 // 3) Call WriteSFR() (will return a '0' if successful,'1' otherwise)
64 //
65 // SFR Read Procedure:
66 // 0) Set parameter <FPDAT> provided by InitDevice()
67 // 1) Set parameter <AddressSFR> to the location of the SFR
68 // 3) Call ReadSFR() (will return a '0' if successful,'1' otherwise)
69 //
70 //-------------------------------------------------------------------------
71
72 /*********************************************************
73                               Header files
74  *********************************************************/
75 //#include <c8051f000.h>    // SFR declarations
76 #include "lpc21xx.h"
77 #include "minilib.h"
78 #include <stdio.h>
79 #include <string.h>
80 //#include "8051.h"
81 //#include "lpc2103.h"
```

```
82
83  /*************************************************************
84                          Define SECTION
85  *************************************************************/
86
87  // Times
88  #define TIMERESET 100
89  #define TIMEBIT 2
90
91  // Macros
92  #define C2D (1<<5)
93  #define C2CK (1<<4)
94  #define C2D_INPUT IOODIR &=~(C2D);
95  #define C2D_OUTPUT IOODIR |=(C2D);
96  #define C2D_READ ((IOOPIN & (C2D))>>5)
97  #define C2D_1 IOOSET |=C2D;
98  #define C2D_0 IOOCLR |=C2D;
99  #define C2CK_H IOOSET |=C2CK;
100 #define C2CK_L IOOCLR |=C2CK;
101
102 //Register STATUS ADDRESS
103 #define FLBussy    (1<<7)
104 #define EError     (1<<6)
105 #define InBusy     (1<<1)
106 #define OutReady   (1<<0)
107
108 // FLASH information
109 #define FLASH_SIZE 8192          // FLASH size in bytes
110 #define NUM_PAGES FLASH_SIZE/512 // Number of 512-byte FLASH pages
111
112
113
114 /*************************************************************
115                         GLOBAL SECTION
116 *************************************************************/
117 //static unsigned char FPDAT=0xB4;
118
119
120 /*************************************************************
121                         FUNTION SECTION
122 *************************************************************/
123
124 //-------------------------------------------------------------
125 // Primitive C2 Command Routines
126 //-------------------------------------------------------------
127 // These routines perform the low-level C2 commands
128 //-------------------------------------------------------------
129
130 //-----------------------------------------------------------------------------------
```

```
131 // InitClk()
132 //-------------------------------------------------------------------------------
133 //  -Inicialize C2CK
134 //-------------------------------------------------------------------------------
135
136 void InitClk()
137 {
138         IO0DIR |=(C2CK);
139         C2CK_H;
140 }
141
142 //-------------------------------------------------------------------------------
143 // PulseClk()
144 //-------------------------------------------------------------------------------
145 // - Strobe C2CK
146 //-------------------------------------------------------------------------------
147
148 void PulseClk()
149 {
150         C2CK_L;
151         _delay_us(TIMEBIT);
152         C2CK_H;
153         _delay_us(TIMEBIT);
154 }
155
156 //-------------------------------------------------------------------------------
157 // Reset()
158 //-------------------------------------------------------------------------------
159 // - Performs a target device reset by pulling the C2CK pin low for >20us
160 //-------------------------------------------------------------------------------
161
162 void Reset()
163 {
164      InitClk();
165      C2CK_L
166      _delay_us(TIMERESET);
167      C2CK_H
168          _delay_us(TIMEBIT);
169   }
170
171 //-------------------------------------------------------------------------------
172 // whiteByte()
173 //-------------------------------------------------------------------------------
174 // Transmits the write byte in C2D output
175 //-------------------------------------------------------------------------
176
177 void WriteByte(int byte)
178 {
179      int i,j;
```

```
180       C2D_OUTPUT;
181       for (i=0;i<8;i++)
182       {
183
184           j=byte&0x01;                                    // Shift out 8 bit byte field.LSB-first
185           //_printf("j%d=%08x\n",i,j);//DEBUG_INFO
186           if (j==1) C2D_1;
187                     if (j==0)        C2D_0;
188            //  else (_puts("\r\n\nERROR DE ESCRITURA\r\n\n"));//DEBUG_INFO
189           PulseClk();
190           byte = (byte>>1);
191       }
192   }
193   //-------------------------------------------------------------------------------------
194   // AddressWrite()
195   //-------------------------------------------------------------------------------------
196   // - Performs a C2 Address register write (writes the <Byte> input to Address register)
197   //-------------------------------------------------------------------------------------
198
199   void AddressWrite(int byte)
200   {
201
202         C2D_INPUT;
203         PulseClk();                                   //Start
204         C2D_OUTPUT;
205         C2D_1;
206             PulseClk();                                   // Instruction field (11b -> Address Writ
207             PulseClk();
208         WriteByte(byte);
209         C2D_INPUT;
210         PulseClk();                                   //Stop
211   }
212
213   //-------------------------------------------------------------------------------------
214   // ReadByte()
215   //-------------------------------------------------------------------------------------
216   // Returns the read byte in C2D input
217   //-------------------------------------------------------------------------------------
218
219   int ReadByte()
220   {
221       int i,j,byte=0x00000000;
222
223
224       for (i=0;i<8;i++)
225        {
226           PulseClk();
227           byte=(byte>>1);                           // Shift in 8 bit byte field.LSB-first
228                   j=C2D_READ;
```

```
229           //_printf("dato%d=%d\n",i,j);//DEBUG_INFO
230           if (j==1) byte=byte|0x80;
231           //else byte=byte|0x00;//DEBUG_INFO
232
233       }
234       return byte;
235   }
236
237   //-------------------------------------------------------------------------------
238   // AddressRead()
239   //-------------------------------------------------------------------------------
240   // - Performs a C2 Address register read
241   // - Returns the 8-bit register content
242   //-------------------------------------------------------------------------------
243
244   int AddressRead()
245   {
246
247       int Address;
248       C2D_INPUT
249       PulseClk();                         //Start
250       C2D_OUTPUT
251       C2D_0
252       PulseClk();                         // Instruction field (10b -> Address Read)
253       C2D_1
254       PulseClk();
255       C2D_INPUT
256       Address=ReadByte();
257       PulseClk();                         //Stop
258       return Address;
259   }
260
261   //-------------------------------------------------------------------------------
262   // Check()
263   //-------------------------------------------------------------------------------
264   // Returns the read bit in C2D input
265   //-------------------------------------------------------------------------
266
267   int Check()
268   {
269         int i=0;
270         PulseClk();
271         i=C2D_READ;
272
273         return i;
274   }
275
276
277   //-------------------------------------------------------------------------------
```

```
278 // DataWrite()
279 //-------------------------------------------------------------------------------
280 // - Performs a C2 Data register write (writes <byte> input to data register)
281 //-------------------------------------------------------------------------
282
283 void DataWrite(int byte)
284 {
285       int j;
286
287       C2D_OUTPUT;
288       PulseClk();                                   //Start
289       C2D_1;
290           PulseClk();                                   // Instruction field (01b -> Data Write)
291           C2D_0;
292           PulseClk();                                   // Length field (00b -> 1 byte)
293           PulseClk();
294           PulseClk();
295       WriteByte(byte);
296       C2D_INPUT;
297
298       do {
299             j=Check();
300         }while (j==0);
301
302       PulseClk();                                   //Stop
303
304 }
305
306 //-------------------------------------------------------------------------------
307 // DataRead()
308 //-------------------------------------------------------------------------------
309 // - Performs a C2 Data register read
310 // - Returns the 8-bit register content
311 //-------------------------------------------------------------------------
312
313 int DataRead()
314 {
315     int j,byte;
316     //InitClk;
317     PulseClk();                                 //Start
318     C2D_OUTPUT;
319     C2D_0;
320     PulseClk();                                 // Instruction field (00b-> Data Read)
321     PulseClk();
322     PulseClk();                                 // Length field (00b -> 1 byte)
323     PulseClk();
324
325     C2D_INPUT;
326
```

```
327         do {
328               j=Check();
329           }while (j==0);
330 
331       byte=ReadByte();
332       PulseClk();                                   //Stop
333 
334       return byte;
335 }
336 
337 //-----------------------------------------------------------------------------------
338 // FLASH Programming Routines (High Level)
339 //-----------------------------------------------------------------------------------
340 //
341 // These high-level routines perform the FLASH Programming Interface (FPI)
342 // command sequences.
343 //-----------------------------------------------------------------------------
344 
345 
346 //-----------------------------------------------------------------------------------
347 // InitDevice()
348 //-----------------------------------------------------------------------------------
349 // - Reads the target Devcie ID register and Return FPDAT
350 //-----------------------------------------------------------------------------
351 int InitDevice()
352 {
353     int DeviceID=0x00;
354     Reset();
355     _delay_ms(50);
356     DeviceID=DataRead();
357     _printf("ID=%08x\n",DeviceID);//DEBUG_INFO
358     if (DeviceID==0x0F) return 0x00AD;
359     if (DeviceID==0x28) return 0x00AD;
360     if (DeviceID==0x18) return 0x00AD;
361     if (DeviceID==0x19) return 0x00AD;
362     else return 0x00B4;
363 }
364 
365 //-----------------------------------------------------------------------------------
366 // InitPI()
367 //-----------------------------------------------------------------------------------
368 // - Initializes the C2 Interface for FLASH programming
369 //-----------------------------------------------------------------------
370 
371 void InitPI()
372 {
373       Reset();
374       AddressWrite(0x02);//Control register (FPCTL) for C2 Data register accesses
375       DataWrite(0x02);   //Write the first key code to enable C2 FLASH programming
```

```
376       DataWrite(0x04);   //Write the second key code to enable C2 FLASH programming
377       DataWrite(0x01);        //Write the third key code to enable C2 FLASH programming (Start)
378       _delay_ms(40);            // Delay for at least 20ms to ensure the target is ready for C2
379   }
380
381   //---------------------------------------------------------------------------------
382   // ReadFlashBlock()
383   //---------------------------------------------------------------------------------
384   // - Reads a block of FLASH memory starting at <AddressC>
385   // - The size of the block is defined by <Length>
386   // - Stores the read data at the location targeted by the pointer <*Block>
387   // - Assumes that FLASH accesses via C2 have been enabled prior to the function call
388   // - Function call returns a '0' if successful; returns a '1' if unsuccessful
389   //---------------------------------------------------------------------------------
390
391   int ReadFlashBlock (int FPDAT, int *Block, int AddressC, int Length)
392   {
393           if (Length>256) return 1;
394           int AddressH=AddressC>>8;
395           int AddressL=AddressC&0xFF;
396       //_printf("AH:%d\n",AddressH);//DEBUG_INFO
397       //_printf("AL:%d\n",AddressL);//DEBUG_INFO
398
399       int i,Data,Address,j;
400       j=0;
401       i=0;
402       AddressWrite(FPDAT);                          //Select the FLASH Programming Data register
403       DataWrite(0x06);                                  //Send 0x06 Command Read Block
404
405       do {                                          //Wait for input acknowledge
406           Address=AddressRead()&InBusy;
407           //_printf("Condicion:%08x\n",Address);//DEBUG_INFO
408           i++;
409           if (i>1000) return 1;
410           }while (Address);
411
412
413       do {                                          //Wait for status information
414           Address=AddressRead()&OutReady;
415           //_printf("Condicion:%08x\n",Address);//DEBUG_INFO
416           i++;
417           if (i>1000) return 1;
418           }while (!(Address));
419
420       Data=DataRead();                                  //Checking the command
421       //_printf("Elemento 0D=%08x\n",Data);//DEBUG_INFO
422       if (Data==0x0D) Data=0;
423       else return 1;
424
```

```
425     DataWrite(AddressH);                            //Send address high byte
426
427     do {                                            //Wait for input acknowledge
428         Address=AddressRead()&InBusy;
429         //_printf("Condicion:%08x\n",Address);//DEBUG_INFO
430         i++;
431         if (i>1000) return 1;
432         }while (Address);
433
434     DataWrite(AddressL);                            //Send address low byte
435
436     do {                                            //Wait for input acknowledge
437         Address=AddressRead()&InBusy;
438         //_printf("Condicion:%08x\n",Address);//DEBUG_INFO
439         i++;
440         if (i>1000) return 1;
441         }while (Address);
442
443     DataWrite(Length);                                //Send block size
444
445     do {                                            //Wait for input acknowledge
446         Address=AddressRead()&InBusy;
447         //_printf("Condicion:%08x\n",Address);//DEBUG_INFO
448         i++;
449         if (i>1000) return 1;
450         }while (Address);
451
452     do {                                            //Wait for status information
453         Address=AddressRead()&OutReady;
454         //_printf("Condicion:%08x\n",Address);//DEBUG_INFO
455         i++;
456         if (i>1000) return 1;
457         }while (!(Address));
458
459     Data=DataRead();                                  //Checking the command
460     //_printf("Elemento 0D=%08x\n",Data);//DEBUG_INFO
461     if (Data==0x0D) Data=0;
462     else return 1;
463
464     for(j=0;j<(Length);j++){
465
466         do {                                          //Wait for status information
467             Address=AddressRead()&OutReady;
468             //_printf("Condicion:%08x\n",Address);//DEBUG_INFO
469             i++;
470             if (i>1000) return 1;
471             }while (!(Address));
472
473             Block[j]=DataRead();                    // Read data from the FPDAT register
```

```
474             //_printf("Elemento leido%d=%08x\n",j,Block[j]);//DEBUG_INFO
475                             }
476     return 0;
477 }
478 
479 //-------------------------------------------------------------------------------------
480 // WhiteFlashBlock()
481 //-------------------------------------------------------------------------------------
482 // - Writes a block of FLASH memory starting at <Address>
483 // - The size of the block is defined by <Length>
484 // - Writes the block stored at the location targetted by <*Block>
485 // - Assumes that FLASH accesses via C2 have been enabled prior to the function call
486 // - Function call returns a '0' if successful; returns a '1' if unsuccessful
487 //-------------------------------------------------------------------------------------
488 
489 int WriteFlashBlock(int FPDAT, int *Block, int AddressC, int Length)
490 {
491         if (Length>256) return 1;
492         int AddressH=AddressC>>8;
493         int AddressL=AddressC&0xFF;
494     int Address,Data,i,j;
495     i=0;
496     AddressWrite(FPDAT);                            //Select the FLASH Programming Data register
497     DataWrite(0x07);                                    // 0x07 Command Block Write
498 
499     do {                                            //Wait for input acknowledge
500         Address=AddressRead()&InBusy;
501         _printf("Condicion:%08x\n",Address);//DEBUG_INFO
502         i++;
503         if (i>1000) return 1;
504         }while (Address);
505 
506 
507 
508     do {                                            // Wait for status information
509         Address=AddressRead()&OutReady;
510         _printf("Condicion:%08x\n",Address);//DEBUG_INFO
511         i++;
512         if (i>1000) return 1;
513         }while (!(Address));
514 
515 
516     Data=DataRead();                                    //Checking the command
517     _printf("Elemento 0D=%08x\n",Data);//DEBUG_INFO
518     if (Data==0x0D) Data=0;
519     else return 1;
520 
521     DataWrite(AddressH);                            //Send address high byte
522 
```

```
523     do {                                        //Wait for input acknowledge
524         Address=AddressRead()&InBusy;
525         _printf("Condicion:%08x\n",Address);//DEBUG_INFO
526         i++;
527         if (i>1000) return 1;
528         }while (Address);
529 
530     DataWrite(AddressL);                        //Send address low byte
531 
532     do {                                        //Wait for input acknowledge
533         Address=AddressRead()&InBusy;
534         _printf("Condicion:%08x\n",Address);//DEBUG_INFO
535         i++;
536         if (i>1000) return 1;
537         }while (Address);
538 
539     DataWrite(Length);                            //Send block size
540 
541     do {                                        //Wait for input acknowledge
542         Address=AddressRead()&InBusy;
543         i++;
544         if (i>1000) return 1;
545         }while (Address);
546 
547     for(j=0;j<(Length);j++){
548         DataWrite(Block[j]);                    // Write data to the FPDAT register
549         //_printf("Elemento escrito%d=%08x\n",j,Block[j]);
550 
551         do {                                    //Wait for input acknowledge
552             Address=AddressRead()&InBusy;
553             _printf("Condicion:%08x\n",Address);//DEBUG_INFO
554             i++;
555             if (i>1000) return 1;
556             }  while (Address);
557     }
558 
559     do {                                        //Wait for status information
560         Address=AddressRead()&OutReady;
561         _printf("Condicion:%08x\n",Address);//DEBUG_INFO
562         i++;
563         if (i>1000) return 1;
564         }while (!(Address));
565 
566     Data=DataRead();                            //Checking the command
567     _printf("Elemento 0D=%08x\n",Data);//DEBUG_INFO
568     if (Data==0x0D) return 0;
569     else return 1;
570 }
571 
```

```
572 //-------------------------------------------------------------------------------------
573 // EraseFlashPage()
574 //-------------------------------------------------------------------------------------
575 // - Erases a 512-byte FLASH page
576 // - Targets the FLASH page containing the address <NumPage>
577 // - Assumes that FLASH accesses via C2 have been enabled prior to the function call
578 // - Function call returns a '0' if successful; returns a '1' if unsuccessful
579 //-------------------------------------------------------------------------------------
580
581 int EraseFlashPage ( int FPDAT, int NumPage)
582 {
583     int Address,Data;
584     int i=0;
585     AddressWrite(FPDAT);                          //Select the FLASH Programming Data register
586     DataWrite(0x08);                                  //0x08 command Erase Page
587
588     do {                                          //Wait for input acknowledge
589         Address=AddressRead()&InBusy;
590         _printf("Condicion:%08x\n",Address);//DEBUG_INFO
591         i++;
592         if (i>1000) return 1;
593         }  while (Address);
594
595     do {                                          //Wait for status information
596         Address=AddressRead()&OutReady;
597         _printf("Condicion:%08x\n",Address);//DEBUG_INFO
598         i++;
599         if (i>1000) return 1;
600         }while (!(Address));
601
602     Data=DataRead();                                  //Checking the command
603     _printf("Elemento 0D=%08x\n",Data);//DEBUG_INFO
604     if (Data==0x0D) Data=0;
605     else return 1;
606
607     DataWrite(NumPage);                                 //Send FLASH page number
608
609     do {                                          //Wait for input acknowledge
610         Address=AddressRead()&InBusy;
611         _printf("Condicion:%08x\n",Address);//DEBUG_INFO
612         i++;
613         if (i>1000) return 1;
614         }  while (Address);
615
616     do {                                          //Wait for status information
617         Address=AddressRead()&OutReady;
618         _printf("Condicion:%08x\n",Address);//DEBUG_INFO
619         i++;
620         if (i>1000) return 1;
```

```
621         }while (!(Address));
622 
623     Data=DataRead();                                    //Checking the command
624     if (Data==0x0D) Data=0;
625     else return 1;
626 
627     DataWrite(0x00);                                    //Dummy write to initiate erase
628 
629     do {                                               //Wait for input acknowledge
630         Address=AddressRead()&InBusy;
631         _printf("Condicion:%08x\n",Address);//DEBUG_INFO
632         i++;
633         if (i>1000) return 1;
634         }while (Address);
635 
636     do {                                               //Wait for status information
637         Address=AddressRead()&OutReady;
638         _printf("Condicion:%08x\n",Address);//DEBUG_INFO
639         i++;
640         if (i>1000) return 1;
641         }while (!(Address));
642 
643     Data=DataRead();                                    //Checking the command
644     _printf("Elemento 0D=%08x\n",Data);//DEBUG_INFO
645     if (Data==0x0D) return 0;
646     else return 1;
647 }
648 
649 //-------------------------------------------------------------------------------------
650 // WriteSFR()
651 //-------------------------------------------------------------------------------------
652 // - Write in the SFR register located in the direction <AddressSFR>
653 // - The new SFR data will be <NewSFR>
654 // - Assumes that FLASH accesses via C2 have been enabled prior to the function call
655 // - Function call returns a '0' if successful; returns a '1' if unsuccessful
656 //-------------------------------------------------------------------------------------
657 
658 int WriteSFR(int FPDAT, int AddressSFR, int NewSFR)
659 {
660     int Address,Data;
661     int i=0;
662     AddressWrite(FPDAT);                               //Select the FLASH Programming Data register
663     DataWrite(0x0A);                                        //0x0A command Write SFR
664     do {                                               //Wait for input acknowledge
665         Address=AddressRead()&InBusy;
666         _printf("Condicion:%08x\n",Address);//DEBUG_INFO
667         i++;
668         if (i>1000) return 1;
669         }while (Address);
```

```
670
671     do {                                        //Wait for status information
672         Address=AddressRead()&OutReady;
673         _printf("Condicion:%08x\n",Address);//DEBUG_INFO
674         i++;
675         if (i>1000) return 1;
676         }while (!(Address));
677
678     Data=DataRead();                              //Checking the command
679     _printf("Elemento 0D=%08x\n",Data);//DEBUG_INFO
680     if (Data==0x0D) Data=0;
681     else return 1;
682
683     DataWrite(AddressSFR);                        //Send Address SFR
684
685     do {                                        //Wait for input acknowledge
686         Address=AddressRead()&InBusy;
687         _printf("Condicion:%08x\n",Address);//DEBUG_INFO
688         i++;
689         if (i>1000) return 1;
690         }while (Address);
691
692     DataWrite(0x01);                              //Send command Start
693
694     do {                                        //Wait for input acknowledge
695         Address=AddressRead()&InBusy;
696         _printf("Condicion:%08x\n",Address);//DEBUG_INFO
697         i++;
698         if (i>1000) return 1;
699         }while (Address);
700
701     DataWrite(NewSFR);                            //Send New SFR
702
703     do {                                        //Wait for input acknowledge
704         Address=AddressRead()&InBusy;
705         _printf("Condicion:%08x\n",Address);//DEBUG_INFO
706         i++;
707         if (i>1000) return 1;
708         }while (Address);
709
710     return 0;
711 }
712
713 //-------------------------------------------------------------------------------
714 // ReadSFR()
715 //-------------------------------------------------------------------------------
716 // - Read in the SFR register located in the direction <AddressSFR>
717 // - Assumes that FLASH accesses via C2 have been enabled prior to the function call
718 // - Function call returns a '0' if successful; returns a '1' if unsuccessful
```

```
719 //-------------------------------------------------------------------------------
720
721 int ReadSFR(int FPDAT, int AddressSFR)
722 {
723     int Address,Data;
724     int i=0;
725     AddressWrite(FPDAT);                          //Select the FLASH Programming Data register
726     DataWrite(0x09);                                   //0x09 command Read SFR
727
728     do {                                          //Wait for input acknowledge
729         Address=AddressRead()&InBusy;
730         _printf("Condicion:%08x\n",Address);//DEBUG_INFO
731         i++;
732         if (i>1000) return 1;
733         }while (Address);
734
735     do {                                          //Wait for status information
736         Address=AddressRead()&OutReady;
737         _printf("Condicion:%08x\n",Address);//DEBUG_INFO
738         i++;
739         if (i>1000) return 1;
740         }while (!(Address));
741
742     Data=DataRead();                                 //Checking the command
743     _printf("Elemento 0D=%08x\n",Data);//DEBUG_INFO
744     if (Data==0x0D) Data=0;
745     else return 1;
746
747     DataWrite(AddressSFR);                         //Send Address SFR
748
749     do {                                          //Wait for input acknowledge
750         Address=AddressRead()&InBusy;
751         _printf("Condicion:%08x\n",Address);//DEBUG_INFO
752         i++;
753         if (i>1000) return 1;
754         }while (Address);
755
756     DataWrite(0x01);                                 //Send command start
757
758     do {                                          //Wait for input acknowledge
759         Address=AddressRead()&InBusy;
760         _printf("Condicion:%08x\n",Address);//DEBUG_INFO
761         i++;
762         if (i>1000) return 1;
763         }while (Address);
764
765     do {                                          //Wait for status information
766         Address=AddressRead()&OutReady;
767         _printf("Condicion:%08x\n",Address);//DEBUG_INFO
```

```
768         i++;
769         if (i>1000) return 1;
770         }while (!(Address));
771 
772     Data=DataRead();
773     _printf("Contenido del SFR 0x%02x=%02x\n",AddressSFR,Data);
774 
775     return 0;
776 }
777 
778 //-------------------------------------------------------------------------------------
779 // EraseDevice()
780 //-------------------------------------------------------------------------------------
781 // - Erases the entire FLASH memory space
782 // - Assumes that FLASH accesses via C2 have been enabled prior to the function call
783 // - Function call returns a '0' if successful; returns a '1' if unsuccessful
784 //-------------------------------------------------------------------------------------
785 
786 int EraseDevice(int FPDAT)
787 {
788     int Address,Data;
789     int i=0;
790     AddressWrite(FPDAT);                        //Select the FLASH Programming Data register
791     DataWrite(0x03);                                 //0x03 command erase device
792 
793     do {                                        //Wait for input acknowledge
794     Address=AddressRead()&InBusy;
795     _printf("Condicion:%08x\n",Address);//DEBUG_INFO
796     i++;
797     if (i>1000) return 1;
798     }while (Address);
799 
800     do {                                        //Wait for status information
801         Address=AddressRead()&OutReady;
802         _printf("Condicion:%08x\n",Address);//DEBUG_INFO
803         i++;
804         if (i>1000) return 1;
805         }while (!(Address));
806 
807     Data=DataRead();                                //Checking the command
808     _printf("Elemento 0D=%08x\n",Data);//DEBUG_INFO
809     if (Data==0x0D) Data=0;
810     else return 1;
811 
812     DataWrite(0xDE);                                //Arming sequence command 1
813 
814     do {                                        //Wait for input acknowledge
815     Address=AddressRead()&InBusy;
816     _printf("Condicion:%08x\n",Address);//DEBUG_INFO
```

```
817     i++;
818     if (i>1000) return 1;
819     }while (Address);
820 
821     DataWrite(0xAD);                                    //Arming sequence command 2
822 
823     do {                                                //Wait for input acknowledge
824     Address=AddressRead()&InBusy;
825     _printf("Condicion:%08x\n",Address);//DEBUG_INFO
826     i++;
827     if (i>1000) return 1;
828     }while (Address);
829 
830     DataWrite(0xA5);                                    //Arming sequence command 3
831 
832     do {                                                //Wait for input acknowledge
833     Address=AddressRead()&InBusy;
834     _printf("Condicion:%08x\n",Address);//DEBUG_INFO
835     i++;
836     if (i>1000) return 1;
837     }while (Address);
838 
839     do {                                                //Wait for status information
840         Address=AddressRead()&OutReady;
841         _printf("Condicion:%08x\n",Address);//DEBUG_INFO
842         i++;
843         if (i>1000) return 1;
844         }while (!(Address));
845 
846     Data=DataRead();                                    //Checking the command
847     _printf("Elemento 0D=%08x\n",Data);//DEBUG_INFO
848     if (Data==0x0D) return 0;
849     else return 1;
850 }
851 
852 /*************************************************************
853                         MAIN SECTION
854 *************************************************************/
855 
856 void main (void)
857 {
858 
859     int SFR=0x00;
860     int i,j,NUM;
861     int FPDAT=0x0000;
862     int Aux=0x00000000;
863 
864     int BufferWrite[256];
865     int BufferRead[256];
```

```
866     int *PTR_Write=BufferWrite;
867     int *PTR_Read=BufferRead;
868
869     _puts("\r\n\nHello World\r\n\n");
870
871     for(i=0;i<(256);i++){
872
873                         BufferWrite[i]=i;
874
875                       }
876
877
878
879
880     Reset();
881
882     Aux=AddressRead();
883
884     _printf("Status bit=%08x\n",Aux);//DEBUG_INFO
885     FPDAT=InitDevice();
886     _printf("FPDAT=%08x\n",FPDAT);//DEBUG_INFO
887
888     InitPI();
889
890
891
892     j=ReadSFR(FPDAT,0x98);         //SFR SCON0 reset value:0x40
893     if (j==(1)) _puts("\r\nERROR \r\n\n");
894     if (j==(0)) _puts("\r\nLectura de SFR correcta  \r\n\n");
895
896            j=WriteSFR(FPDAT,0x98,0xC8);//SFR SCON0 new value 0xC8 9-bit UART with Variable Bau
897     if (j==(1)) _puts("\r\nERROR \r\n\n");
898         if (j==(0)) _puts("\r\nEscritura de SFR correcta \r\n\n");
899
900      j=ReadSFR(FPDAT,0x98);
901     if (j==(1)) _puts("\r\nERROR \r\n\n");
902     if (j==(0)) _puts("\r\nLectura de SFR correcta  \r\n\n");
903
904
905
906     j=EraseDevice(FPDAT);
907     if (j==1) _puts("\r\nERROR \r\n\n");
908     if (j==0) _puts("\r\nBorrado del dispositivo correcto\r\n\n");
909     _delay_ms(40);
910
911     j=EraseFlashPage (FPDAT,0);
912     if (j==1) _puts("\r\nERROR \r\n\n");
913     if (j==0) _puts("\r\nBorrado por Página correcto \r\n\n");
914
```

```
915     j=WriteFlashBlock(FPDAT,PTR_Write,0x0100,50);
916         if (j==(1)) _puts("\r\nERROR \r\n\n");
917         if (j==(0)) _puts("\r\nEscritura correcta \r\n\n");
918 
919     j=ReadFlashBlock(FPDAT,PTR_Read,0x00FF,256);
920     if (j==1) _puts("\r\nERROR \r\n\n");
921     if (j==0) _puts("\r\nLectura correcta \r\n\n");
922     _delay_ms(40);
923 
924     for(i=0;i<(256);i++){
925 
926               _printf("Elemento%d=%08x\n",i,BufferRead[i]);//DEBUG_INFO
927 
928                       }
929 
930 
931 
932 
933 
934         while (1);//PCON=1;                // CPU idle
935 
936 }
```

Anexo II

Interfaz de Usuario por Línea de Comandos

```
1  //------------------------------------------------------------------------------
2  // FILE NAME: terminalPC.c
3  // TARGET DEVICE:          LPC2103
4  // PROGRAMMER DEVICE:         GNU/Linux
5  // CREATED ON:           21/08/2017
6  // CREATED BY:                  JESUS GARCIA BARAJAS
7  //
8  // Code to program a C8051Fxxx device through its C2 interface.
9  //
10 // This software targets a PC device, though it could be easily
11 // ported to any device.
12 //------------------------------------------------------------------------------
13 /*****************************************************
14                                  Header files
15  *****************************************************/
16 #include <sys/time.h>
17 #include <sys/types.h>
18 #include <unistd.h>
19 #include <asm/termios.h>
20 #include <fcntl.h>
21 #include <stdio.h>
22
23 /*****************************************************
24                        Define SECTION
25 *****************************************************/
26 #define DEV "/dev/ttyUSB0"
27 #define BAUDRATE B115200
28 /*****************************************************
29                       FUNTION SECTION
30 *****************************************************/
31
32 //------------------------------------------------------------------------------
```

```
33 // safewrite()
34 //---------------------------------------------------------------------------------------
35 // - Retry until you write the entire buffer
36 //---------------------------------------------------------------------------------------
37 // ------------------------------------------------------
38 // write seguro.
39 void safewrite(int fd,char *buf,int n)
40 {
41   int i,j;
42   i=0;
43   do {
44     j=write(fd,&buf[i],n);
45     usleep(6000);
46     i+=j; n-=j;
47   } while (n);
48 }
49 //---------------------------------------------------------------------------------------
50 // writeSerial()
51 //---------------------------------------------------------------------------------------
52 // - Write buffer
53 //---------------------------------------------------------------------------------------
54 int writeSerial( int fd, unsigned int *buffer, int ilength )
55 {
56     int written;
57     int result;
58     int i;
59     int o;
60     int length;
61     unsigned int out[256];
62
63
64     printf("writeSerial ilength = %d\n", ilength);
65
66
67     o = 0;
68     for( i = 0; i < ilength; i++ )
69                       {
70                         out[o++] = buffer[i];
71                       }
72     out[o++] = '\r';
73     out[o++] = '\n';
74
75     length = o;
76
77     written = 0;
78     result = 0;
79
80     while( length - written > 0 ){
81
```

```
                /*
                printf("writeSerial length = %d: ", length - written);//DEBUG
                for( i = 0; i < length - written; i++ ){

                                printf("%02x", * (out + written + i) );
                }
                printf("\n");
                */
                result = write( fd, out + written, length - written );
                if ( result >= 0 ){

                            written = written + result;
                }
                else
                {
                break;
                }
    }

    if ( result < 0 )
    {
      perror( "writing SERIAL failed:");
      goto exit;
    }

    result = 0;

    exit:
    return result;
}

//------------------------------------------------------------------------------
// ascii2hex( )
//------------------------------------------------------------------------------
// - Pass a character to its hex value
//------------------------------------------------------------------------------
static int ascii2hex( char c )

{
    int ret = -1;

    if ( ( c >= '0' ) && ( c <= '9' ) )
    {
  ret = c - '0';
    }
    else if ( ( c >= 'A' ) && ( c <= 'Z' ) )
    {
  ret = c - 'A' + 10;
    }
```

```
131       else if ( ( c >= 'a') && ( c <= 'z' ) )
132       {
133     ret = c - 'a' + 10;
134       }
135       return ret;
136   }
137
138   //---------------------------------------------------------------------------
139   // TransStrHex( )
140   //---------------------------------------------------------------------------
141   // - Pass a character from a file.hex to its hexadecimal value
142   //---------------------------------------------------------------------------
143   int TransStrHex(unsigned char a)
144   {
145     switch (a)
146     {
147       case '0':
148       return 0b0000;
149       case '1':
150       return 0b0001;
151       case '2':
152       return 0b0010;
153       case '3':
154       return 0b0011;
155       case '4':
156       return 0b0100;
157       case '5':
158       return 0b0101;
159       case '6':
160       return 0b0110;
161       case '7':
162       return 0b0111;
163       case '8':
164       return 0b1000;
165       case '9':
166       return 0b1001;
167       case 'A':
168       return 0b1010;
169       case 'B':
170       return 0b1011;
171       case 'C':
172       return 0b1100;
173       case 'D':
174       return 0b1101;
175       case 'E':
176       return 0b1110;
177       case 'F':
178       return 0b1111;
179       case ':':
```

```
180     return 0x3A;
181     case 10:
182     return 0x0A;
183     case 13:
184     return 0x0D;
185 
186     default:
187     printf("Dato erroneo, elimina los comentarios del archivo .hex\n");
188   }
189 }
190 /*************************************************************
191                         MAIN SECTION
192 *************************************************************/
193 
194 main(int argc,char **argv)
195 {
196 
197         int i,j,fd,fdin,baud=115200,result,hex=0,crlf=0,sl;
198         struct termios term;
199 
200         int y,w,z,h;
201     unsigned int Hex,Num,Address=0,Type,AddressExt,Comand,SFR,NewSFR,NumPage;
202 
203     char Fichero[20];
204     char *ptr=Fichero;
205     unsigned int BloqueReadAux[256];
206     unsigned int BloqueWrite[256][256];
207     char BloqueRead[256];
208     //unsigned int *PTR_Write=BloqueWrite;
209     char *PTR_Read=BloqueRead;
210     for(i=0;i<(256);i++){
211                         BloqueReadAux[i]=0x00;
212                         BloqueRead[i]=0x00;
213                         for(y=0;y<(256);y++) {
214                                   BloqueWrite[i][y]=0x00;
215                         }
216     }
217         unsigned char a,*dn=DEV,fn[256];
218         const int baudtable[][2]={
219             50,B50,
220             75,B75,
221             110,B110,
222             150,B150,
223             200,B200,
224             300,B300,
225             600,B600,
226             1200,B1200,
227             2400,B2400,
228             4800,B2400,
```

```
229             9600,B9600,
230             19200,B19200,
231             38400,B38400,
232                             57600,B57600,
233             115200,B115200,
234                             921600,B921600,
235                             1500000,B1500000,
236             0,0
237         };
238
239
240
241 /******************************** Program menu ************************************/
242     for (y=1;y<argc;y++){
243             if (*argv[y]=='-'){
244                 switch(argv[y][1]){
245                 case 'h':   printf("Uso: %s opciones\n",argv[0]);
246                             printf("Opciones:\n");
247                             printf("-h : Help\n");
248                             printf("-w <archivo.hex> : copiar archivo.hex en Flash\n");
249                             printf("-b <archivo.bin> : copiar archivo.bin en Flash\n");
250                             printf("-r <Direccion dec> <Numero de Bytes>: Leer Flash \n");
251                                     printf("-d <Numero de pagina> : Borra dicha pagina de
252                             printf("-e : Borra dispositivo \n");
253                             printf("-s <Direccion SFR hex> <Nuevo SFR hex>: Cambia el SFR\n");
254                             printf("-l <Direccion SFR hex> : Lee el SFR\n");
255                                     exit(0);
256                             case 'w':   Comand=0; ptr=argv[++y]; break;
257                             case 'b':   Comand=1; ptr=argv[++y]; break;
258                             case 'r':   Comand=2; Address=atoi(argv[++y]);NumPage=atoi(argv[++
259                             case 'd':   Comand=3; NumPage=atoi(argv[++y]); break;
260                             case 'e':   Comand=4; break;
261                             case 's':   Comand=5; SFR=((TransStrHex(argv[2][0]))<<4)|TransStrH
262                                                     NewSFR=((TransStrHex(argv[3][0])<<4)
263                             case 'l':   Comand=6; SFR=((TransStrHex(argv[2][0]))<<4)|TransStrH
264                 }
265                 }
266         }
267                                     // Debug program menu
268     printf("Su Eleccion:\n");
269     printf("Comando:%d\n",Comand);
270     printf("Fichero:[%s]\n",ptr);
271     printf("Numero pagina:%d\n",NumPage);
272     printf("SFR:%02x\n",SFR);
273     printf("Nuevo SFR:%02x\n",NewSFR);
274
275 /************************* Connection configuration ******************************/
276
277         for (i=0;baudtable[i][0];i++) {
```

```
278             if (baud==baudtable[i][0]) break;
279         }
280         if (!baudtable[i][0]) {printf("Velocidad no soportada\n"); exit(0);}
281 
282         printf("Puerto serie: %s\n",dn);//DEBUG
283         printf("Velocidad: %d baudios\n",baud);//DEBUG
284 
285 
286         // Open port
287         if ((fd=open(dn,O_RDWR/*|O_NONBLOCK*/))==-1) {perror("open"); exit(0);}
288         bzero(&term,sizeof(term));
289         term.c_cflag = baudtable[i][1] | CS8 | CLOCAL | CREAD;
290         term.c_cc[VMIN]=1;          // blocking mode
291         if (tcsetattr(fd,TCSANOW,&term)) { fprintf(stderr,"Fallo en tcsetattr\n"); exit(0); }
292         if (tcflush(fd,TCIFLUSH)) { fprintf(stderr,"Fallo en tcflush\n"); exit(0); }
293 
294         //  STDIN
295         i=fcntl(0,F_GETFL); fcntl(0,F_SETFL,i|O_NONBLOCK);
296         tcgetattr(0,&term);
297 
298         term.c_lflag&=~ICANON;
299         term.c_lflag&=~ECHO;
300 
301         if (tcsetattr(0,TCSANOW,&term)) {
302                 fprintf(stderr,"Fallo en tcsetattr\n");
303                 exit(0);
304         }
305 
306         // Reset
307     if(ioctl(fd, TIOCMGET, &i)) {fprintf(stderr,"Fallo en ioctl TIOCMGET\n"); exit(0);}
308     i|=TIOCM_DTR|TIOCM_RTS; // Reset y Bootloader L
309     if(ioctl(fd, TIOCMSET, &i)) {fprintf(stderr,"Fallo en ioctl TIOCMSET\n"); exit(0);}
310     usleep(300000);
311 
312         // Start
313     i&=~TIOCM_RTS;          // bootloader H
314     if(ioctl(fd, TIOCMSET, &i)) {fprintf(stderr,"Fallo en ioctl TIOCMSET\n"); exit(0);}
315     usleep(300000);
316     i&=~TIOCM_DTR;          // Reset H
317     if(ioctl(fd, TIOCMSET, &i)) {fprintf(stderr,"Fallo en ioctl TIOCMSET\n"); exit(0);}
318 
319 /*************************** IN/OUT DATA ***********************************************/
320 
321         fdin=0;
322         hex=1;
323         crlf=0;
324 //        for (;;) { //for Debug
325                 sl=2;
326                 if (read(fd,&a,1)==1) {
```

```
327                     if (hex) {printf("%02X ",a); fflush(stdout);}
328                     else write(1,&a,1);
329             } else sl--;
330             j=read(fdin,&a,1);
331             if (j==0 && fdin) {
332                     close(fdin);
333                     fdin=0;
334             }
335             if (j==1) {
336                     switch (a) {//File
337                     case         'X'-64:        a=3;//Ctl-X
338                             break;
339                     case        '\n':        if (crlf) a='\r';
340                             break;
341                     case        'F'-64:        printf("\nInclude File: ");//Ctl-F
342                             fflush(stdout);
343                             tcgetattr(0,&term);
344                             term.c_lflag|=ECHO;
345                             tcsetattr(0,TCSANOW,&term);
346                             i=fcntl(0,F_GETFL); fcntl(0,F_SETFL,i&(~O_NONBLOCK));
347                             fgets(fn,255,stdin); fn[strlen(fn)-1]=0;
348                             fcntl(0,F_SETFL,i/*|O_NONBLOCK*/);
349                             term.c_lflag&=~ECHO;
350                             tcsetattr(0,TCSANOW,&term);
351                             if ((fdin=open(fn,O_RDONLY))==-1) fdin=0;
352                             //continue;
353                     }
354                     write(fd,&a,1); usleep(7000);
355                     if (a=='\r') {a='\n'; write(fd,&a,1); usleep(20000);}
356             } else sl--;
357             if (!sl) usleep(40000);
358
359 //        }// for Debug
360
361 /*************************** Selection of the chosen command ****************************
362
363 if (Comand==0){
364          FILE *pFichero;
365      unsigned char byte;
366      pFichero=fopen(ptr, "r");
367      if (pFichero==NULL) {
368       printf("No puedo abrir el archivo %s\n\n",argv[2]);
369       return 0;
370       }
371      for(i=0;i<254;i++){// i= line
372         w=0;// w= position in the linee
373         do {
374
375            byte=fgetc(pFichero);
```

```
376
377             //printf("dato %d,%d:%c\n",i,j,byte);//DEBUG
378
379             BloqueWrite[i][w]=(byte);
380
381
382             w++;
383
384
385
386         }while ((!((BloqueWrite[i][w-1]==10)&&(BloqueWrite[i][w-2]==13)))&&(!(feof(pFichero)))
387
388              *PTR_Read=BloqueRead;
389              BloqueRead[0]=0;
390
391              Num=((TransStrHex(BloqueWrite[i][1]))<<4)|TransStrHex(BloqueWrite[i][2]);
392              printf("Num:%d\n",Num );
393              BloqueRead[1]=Num;
394              Address=((TransStrHex(BloqueWrite[i][3]))<<12)|((TransStrHex(BloqueWrite[i][4]))<<
395                        ((TransStrHex(BloqueWrite[i][5]))<<4)|(TransStrHex(BloqueWrite[i][6]))
396              printf("Address:%04x\n",Address );
397              BloqueRead[2]=((Address>>8)&0x00FF);
398              BloqueRead[3]=(Address&0x00FF);
399              Type=((TransStrHex(BloqueWrite[i][7]))<<4)|(TransStrHex(BloqueWrite[i][8]));
400              printf("Type:%02x\n",Type);
401
402              if (Type==0){
403                h=4;
404                for(z=0;z<(2*Num);z=z+2){
405
406                            Hex=((TransStrHex(BloqueWrite[i][9+z]))<<4)|(TransStrHex(BloqueWrite
407                            BloqueReadAux[z]=Hex;
408                            //printf("Hex:%x\n",BloqueReadAux[z]);//debug
409                            BloqueRead[h++]=BloqueReadAux[z];
410                }
411                    BloqueRead[h++]='\r';
412                BloqueRead[h++]='\n';
413                for(z=0;z<30;z++){
414                        printf("Hex%d:%02x\n",z,BloqueRead[z]);//debug
415                }
416
417                safewrite(fd,PTR_Read,(Num+8));
418
419                        usleep(80000);
420
421                do{
422                      read(fd,&a,1);
423                    }while(a!=0x0C);
424                // printf("\r\n\nCopia de fragmento de archivo correcta\r\n\n");//DEBUG
```

```
425
426             }
427             if (Type==1) {
428                             fclose(pFichero);
429                             break;
430                          }
431             if (Type==4) {
432                             AddressExt=((TransStrHex(BloqueWrite[i][9]))<<12)|((TransStrHex(Bl
433                                         |((TransStrHex(BloqueWrite[i][11]))<<4)|(TransStrHex(B
434                             printf("AddressExt:%x\n",AddressExt);
435             }
436     }//for
437 }// if comand=0
438 /*****************************************************************************************
439 if (Comand==1){
440
441     FILE *pFichero;
442     unsigned char byte;
443     pFichero=fopen(ptr, "rb");
444     if (pFichero==NULL) {
445       printf("No puedo abrir el archivo %s\n\n",argv[2]);
446       return 0;
447       }
448     for(i=0;i<80;i++){// i=line
449         j=0;// j=position in the line
450         do {
451
452            byte=fgetc(pFichero);
453
454            //printf("dato %d,%d:%x\n",i,j,byte);
455
456            BloqueWrite[i][j]=(byte&0x000000FF);
457
458
459            j++;
460
461
462
463         }while ((j<200)&&(!(feof(pFichero))));
464
465         Num=j;
466         *PTR_Read=BloqueRead;
467
468         BloqueRead[0]=1;
469         //printf("Num:%d\n",Num );//debug
470         BloqueRead[1]=Num;
471
472         BloqueRead[2]=((Address>>8)&0x00FF);
473         BloqueRead[3]=(Address&0x00FF);
```

```
474
475         for(z=4;z<(Num+4);z++){
476                         BloqueRead[z]=BloqueWrite[i][z];
477                         //printf("Hex%d:%02x\n",z,BloqueRead[z]);//DEBUG
478               }
479         BloqueRead[z++]='\r';
480         BloqueRead[z++]='\n';
481         for(z=0;z<256;z++){
482
483                         printf("Hex%d:%02x\n",z,BloqueRead[z]);
484               }
485
486
487         safewrite(fd,PTR_Read,(Num+6));
488         usleep(80000);
489
490               do{
491                     read(fd,&a,1);
492                   }while(a!=0x0C);
493         //printf("\r\n\nCopia de fragmento de archivo correcta\r\n\n");//DEBUG
494         Address=Address+Num;
495         //printf("Address:%x\n",Address );//DEBUG
496
497         if (feof(pFichero)) {
498                         fclose(pFichero);
499                         return 0;
500                         } /*feof: check if you get to the end of the file */
501     }
502      fclose(pFichero);
503
504      getchar();
505      return 0;
506
507
508
509 }
510
511 /*****************************************************************************************
512 if (Comand==2){
513
514           PTR_Read=BloqueRead;
515       BloqueRead[0]=2;
516       BloqueRead[1]=NumPage;
517       BloqueRead[2]=Address;
518       BloqueRead[3]='\r';
519       BloqueRead[4]='\n';
520
521       safewrite(fd,PTR_Read,5);
522       usleep(40000);
```

```
523
524                 do{
525                     read(fd,&a,1);
526                   }while(a!=0x0C);
527         //printf("\r\n\nLectura  de fragmento de archivo correcta\r\n\n");//DEBUG
528  }
529
530 /*****************************************************************************************
531 if (Comand==3){
532
533       PTR_Read=BloqueRead;
534       BloqueRead[0]=3;
535       BloqueRead[1]=NumPage;
536       BloqueRead[2]='\r';
537       BloqueRead[3]='\n';
538
539       safewrite(fd,PTR_Read,4);
540       usleep(40000);
541
542                 do{
543                     read(fd,&a,1);
544                   }while(a!=0x0C);
545         //printf("\r\n\nBorrado de página correcto\r\n\n");//DEBUG
546
547 }// if comand=3
548
549 /*****************************************************************************************
550 if (Comand==4){
551
552       PTR_Read=BloqueRead;
553       BloqueRead[0]=4;
554       BloqueRead[1]='\r';
555       BloqueRead[2]='\n';
556
557       safewrite(fd,PTR_Read,3);
558       usleep(40000);
559
560                 do{
561                     read(fd,&a,1);
562                   }while(a!=0x0C);
563         //printf("\r\n\nBorrado de dispositivo correcto\r\n\n");//DEBUG
564
565 }
566
567 /*****************************************************************************************
568 if (Comand==5){
569
570       PTR_Read=BloqueRead;
571       BloqueRead[0]=5;
```

```
572         BloqueRead[1]=SFR;
573         BloqueRead[2]=NewSFR;
574         BloqueRead[3]='\r';
575         BloqueRead[4]='\n';
576 
577         safewrite(fd,PTR_Read,5);
578         usleep(40000);
579 
580                   do{
581                       read(fd,&a,1);
582                     }while(a!=0x0C);
583           //printf("\r\n\nEscritura de SFR correcta\r\n\n");//DEBUG
584 
585 }
586 
587 /*****************************************************************************************
588 if (Comand==6){
589 
590         PTR_Read=BloqueRead;
591         BloqueRead[0]=6;
592         BloqueRead[1]=SFR;
593         BloqueRead[2]='\r';
594         BloqueRead[3]='\n';
595 
596         safewrite(fd,PTR_Read,5);
597         usleep(40000);
598 
599                   do{
600                       read(fd,&a,1);
601                     }while(a!=0x0C);
602           //printf("\r\n\nLectura de SFR correcta\r\n\n");//DEBUG
603 
604 }
605 
606 /*****************************************************************************************
607 
608 
609 while (1);//PCON=1;                  // CPU idle
610 }//last
```

Anexo III

Interfaz del Firmware

```
1  //-----------------------------------------------------------------------------
2  // FILE NAME: InterfazFirmware.c
3  // TARGET DEVICE:          C8051Fxxx
4  // PROGRAMMER DEVICE:         LPC2103
5  // CREATED ON:           30/08/2017
6  // CREATED BY:                  JESUS GARCIA BARAJAS
7  //
8  // Code to program a C8051Fxxx device through its C2 interface.
9  //
10 // This software targets a LPC2103 device, though it could be easily
11 // ported to any device.
12 //
13 // Code assumes the following Port pin connections:
14 // LPC2103: | C8051Fxxx:
15 // P0.4     -> C2CK
16 // P0.5     -> C2D
17 // VDD      -> VDD
18 // GND      -> GND
19 //
20
21 /*************************************************
22                                  Header files
23  *************************************************/
24 //#include <c8051f000.h>    // SFR declarations
25 #include "lpc21xx.h"
26 #include "minilib.h"
27
28 #include <string.h>
29 //#include <asm/termios.h>
30 #include <stdio.h>
31 #include <unistd.h>
32 #include <fcntl.h>
33 #include <sys/signal.h>
34 #include <sys/types.h>
```

```
35
36
37  /*************************************************************
38                          Define SECTION
39  *************************************************************/
40
41  // Times
42  #define TIMERESET 100
43  #define TIMEBIT 2
44
45  // Macros
46  #define C2D (1<<5)
47  #define C2CK (1<<4)
48  #define C2D_INPUT IO0DIR &=~(C2D);
49  #define C2D_OUTPUT IO0DIR |=(C2D);
50  #define C2D_READ ((IO0PIN & (C2D))>>5)
51  #define C2D_1 IO0SET |=C2D;
52  #define C2D_0 IO0CLR |=C2D;
53  #define C2CK_H IO0SET |=C2CK;
54  #define C2CK_L IO0CLR |=C2CK;
55
56  //Register STATUS ADDRESS
57  #define FLBussy    (1<<7)
58  #define EError     (1<<6)
59  #define InBusy     (1<<1)
60  #define OutReady   (1<<0)
61
62  // FLASH information
63  #define FLASH_SIZE 8192          // FLASH size in bytes
64  #define NUM_PAGES FLASH_SIZE/512 // Number of 512-byte FLASH pages
65
66
67  // UART
68  #define MAX_CHAR 16
69
70  /*************************************************************
71                          GLOBAL SECTION
72  *************************************************************/
73
74  /*************************************************************
75                          FUNTION SECTION
76  *************************************************************/
77
78
79  //-------------------------------------------------------------------------------
80  // UART Control Functions
81  //-------------------------------------------------------------------------------
82  // - Allow read and write data in the UART
83  //-------------------------------------------------------------------------------
```

```
84  // ------------------------------------------------------
85  void UART0_putch(char c)
86  {
87   while(!(U0LSR&0x20)); // mientras que U0LSR.THRE == 0
88   U0THR = c;
89  }
90  // ------------------------------------------------------
91  char UART0_getch()
92  {
93   while(!(U0LSR&0x01)); // mientras que U0LSR.DR == 0
94   return U0RBR;
95  }
96  // ------------------------------------------------------
97
98  void UART0_Init()
99  {
100  U0LCR = 0x83;  // 10000011 Line Control Register
101                 //       .. Character length 11 = 8 bits
102                 //      .   Stop bits         0 = 1 bit
103                 //     .    Parity enable     0 = No parity
104                 //    .     Even parity       non sense
105                 //   .      Stick parity      non sense
106                 //  .       Break control     0 = No break
107                 // .        DLAB              1 = Access DLM,DLL
108  U0DLL = 32;
109  U0DLM =  0;    // Baud = 4*14745600/(16* 32) = 115200
110  U0LCR = 0x03;  // Same a above, but DLAB = 0 = Normal access
111  U0FCR = 0x07;  // 00000111 FIFO Control Register
112                 //        . FIFO enable       1 = TX&RX FIFO on
113                 //       .  RX FIFO Reset     1 = Reset RX FIFO
114                 //      .   TX FIFO Reset     1 = Reset TX FIFO
115                 // ..       RX int.t trigger level 00 = 1 byte
116 };
117 // ------------------------------------------------------
118 /*********************************************************
119                       MAIN SECTION
120 *********************************************************/
121
122 void main (void)
123 {
124     unsigned int Comand,NewSFR,Address;
125     unsigned int SFR=0x00;
126     unsigned int i=0,j,z,h,NUM,NumPage;
127     unsigned int FPDAT=0x0000;
128     unsigned int Aux=0x00000000;
129
130
131     unsigned char BufferRead[256];
132     unsigned char *PTR_Read=BufferRead;
```

```
133 /*
134     for(j=0;j<(256);j++){
135 
136                         BufferRead[j]=0x00; //
137 
138     }
139  */
140     //_puts("\r\n\nHello World\r\n\n");//Welcome
141 
142     for(j=0;j<6;j++){// j:Number of UART readings
143 
144             h=0;          //read UART data
145             do{
146                 BufferRead[h]=UART0_getch();
147               // printf("Byte %d:%02x\n",h,BufferRead[h] );//Debug BufferRead
148                 h++;
149             }while (!((BufferRead[h-1]==0x0A)&&(BufferRead[h-2]==0x0D)));
150 
151             //_puts("\r\n\nBuffer de Recepción:\r\n\n");
152  /*           for(i=0;i<(30);i++){
153 
154                         printf("Byte %d:%02x\n",i,BufferRead[i] );//Debug BufferRead
155 
156             }*/
157 
158 
159 
160 /************************ Menu for selection of the chosen command ************************
161             Comand=BufferRead[0];
162             switch (Comand)
163                     {
164                         case 0:{
165                             NUM=BufferRead[1];
166                             Aux=((BufferRead[2]<<8)&0xFF00);
167                             Address=(BufferRead[3]&0x00FF)|Aux;
168                             for (i=0;i<NUM;i++){
169                                 BufferRead[i]=BufferRead[4+i];
170                             }
171                             *PTR_Read=BufferRead;
172                             PTR_Read=&BufferRead[4];
173                             FPDAT=InitDevice();
174                             InitPI();
175                             z=WriteFlashBlock(FPDAT,PTR_Read,Address,NUM);
176                            // if (z==(1)) _puts("\r\nERROR \r\n\n");
177                            // if (z==(0)) _puts("\r\nEscritura correcta \r\n\n");
178                             break;
179                         }
180                         case 1:
181                             NUM=BufferRead[1];
```

```
182                     Aux=((BufferRead[2]<<8)&0xFF00);
183                     Address=(BufferRead[3]&0x00FF)|Aux;
184                     for (i=0;i<NUM;i++){
185                         BufferRead[i]=BufferRead[4+i];
186                     }
187                     *PTR_Read=BufferRead;
188                     FPDAT=InitDevice();
189                     InitPI();
190                     z=WriteFlashBlock(FPDAT,PTR_Read,Address,NUM);
191                   //  if (z==(1)) _puts("\r\nERROR \r\n\n");
192                   //  if (z==(0)) _puts("\r\nEscritura correcta \r\n\n");
193                     break;
194                 case 2:
195                     *PTR_Read=BufferRead;
196                     NUM=BufferRead[1];
197                     Address=BufferRead[2];
198                     FPDAT=InitDevice();
199                     InitPI();
200                     z=ReadFlashBlock(FPDAT,PTR_Read,Address,NUM);
201                  //   if (z==1) _puts("\r\nERROR \r\n\n");
202                  //   if (z==0) _puts("\r\nLectura correcta \r\n\n");
203                     break;
204 
205                 case 3:
206                     NumPage=BufferRead[1];
207                     FPDAT=InitDevice();
208                     InitPI();
209                     z=EraseFlashPage (FPDAT,NumPage);
210                   //  if (z==1) _puts("\r\nERROR \r\n\n");
211                   //  if (z==0) _puts("\r\nBorrado por Página correcto \r\n\n");
212                     break;
213                 case 4:
214                     FPDAT=InitDevice();
215                     InitPI();
216                     z=EraseDevice (FPDAT);
217                  //   if (z==1) _puts("\r\nERROR \r\n\n");
218                  //   if (z==0) _puts("\r\nBorrado del dispositivo correcto \r\n\n");
219                     break;
220                 case 5:
221                     SFR=BufferRead[1];
222                     NewSFR=BufferRead[2];
223                     FPDAT=InitDevice();
224                     InitPI();
225                     z=WriteSFR(FPDAT,SFR,NewSFR);
226                  //   if (z==(1)) _puts("\r\nERROR \r\n\n");
227                  //   if (z==(0)) _puts("\r\nEscritura de SFR correcta \r\n\n");
228                     break;
229                 case 6:
230                     SFR=BufferRead[1];
```

```
231                             FPDAT=InitDevice();
232                             InitPI();
233                             z=ReadSFR(FPDAT,SFR);
234                            // if (z==(1)) _puts("\r\nERROR \r\n\n");
235                            // if (z==(0)) _puts("\r\nLectura de SFR correcta \r\n\n");
236                             break;
237             } //switch
238         if (z==0) UART0_putch(0x0C); //Report to the Pc of completed function successfully
239             //_delay_ms(2000);
240
241     }//for
242
243         while (1);//PCON=1;                 // CPU idle
244
245 }
```

Anexo IV

Manuales de Usuario para el Interfaz C2

AN127: Flash Programming via the C2 Interface

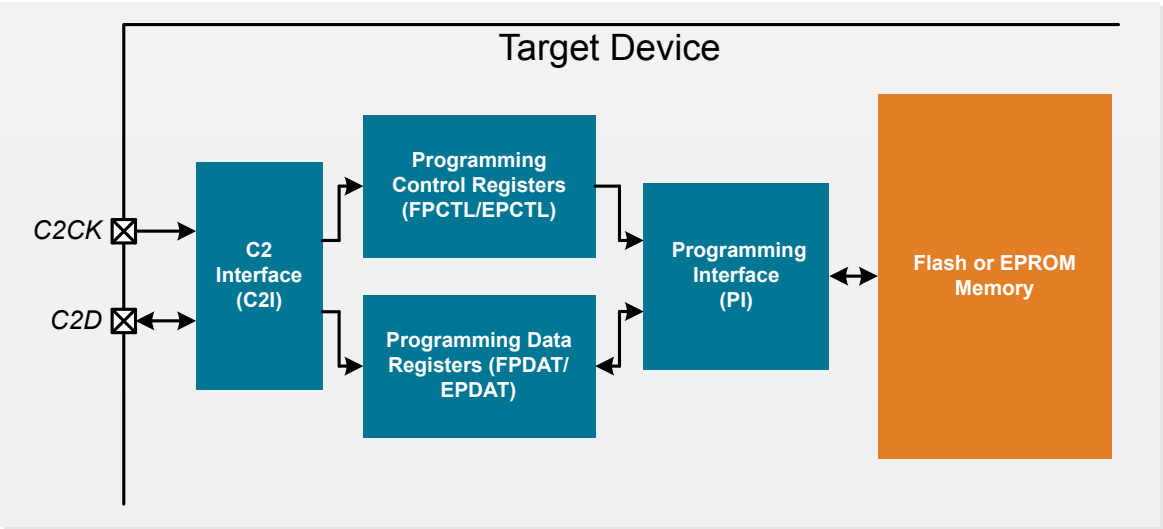
This application note describes how to program the flash or EPROM memory on Silicon Labs microcontroller devices that use the C2 interface (e.g., C8051F30x or C8051T61x). Example software is included.

C2 devices have a Programming Interface (PI) that is accessed via the C2 Interface (C2I) and a set of programming registers. Figure 1 shows the flash access block diagram.

To access the flash or EPROM memory of a C2 device, the programmer must comprehend the C2 interface, the programming registers, and the programming interface.

KEY POINTS

- The C2 interface is a two-wire interface used by most Silicon Labs 8-bit MCUs.
- This interface uses a command-based protocol to modify flash and SFR contents.
- Some devices require additional configuration (i.e. enabling the VDD monitor) to program.



1. C2 Interface

The Silicon Labs 2-Wire Interface (C2) is a two-wire serial communication protocol designed to enable in-system programming and debugging on low pin-count Silicon Labs devices. C2 communication involves an interface master (the programmer/debugger/tester) and an interface target (the device to be programmed/debugged/tested). The two wires used in C2 communication are C2 Data (C2D) and C2 Clock (C2CK).

C2 facilitates a pin-sharing scheme, where the C2 pins on the target device are available for user functions while C2 communication is idle. Each C2 frame is initiated with a START condition on the C2CK pin that signals the target device to configure its C2D pin for C2 communication. Each C2 frame terminates with a STOP condition on the C2CK signal that allows the target device to restore its C2D pin to its user-defined state. The C2CK signal is typically shared with an active-low reset signal (RST) signal on the target device. In this configuration, the width of a low strobe is used to differentiate between a C2 communication strobe and a reset event.

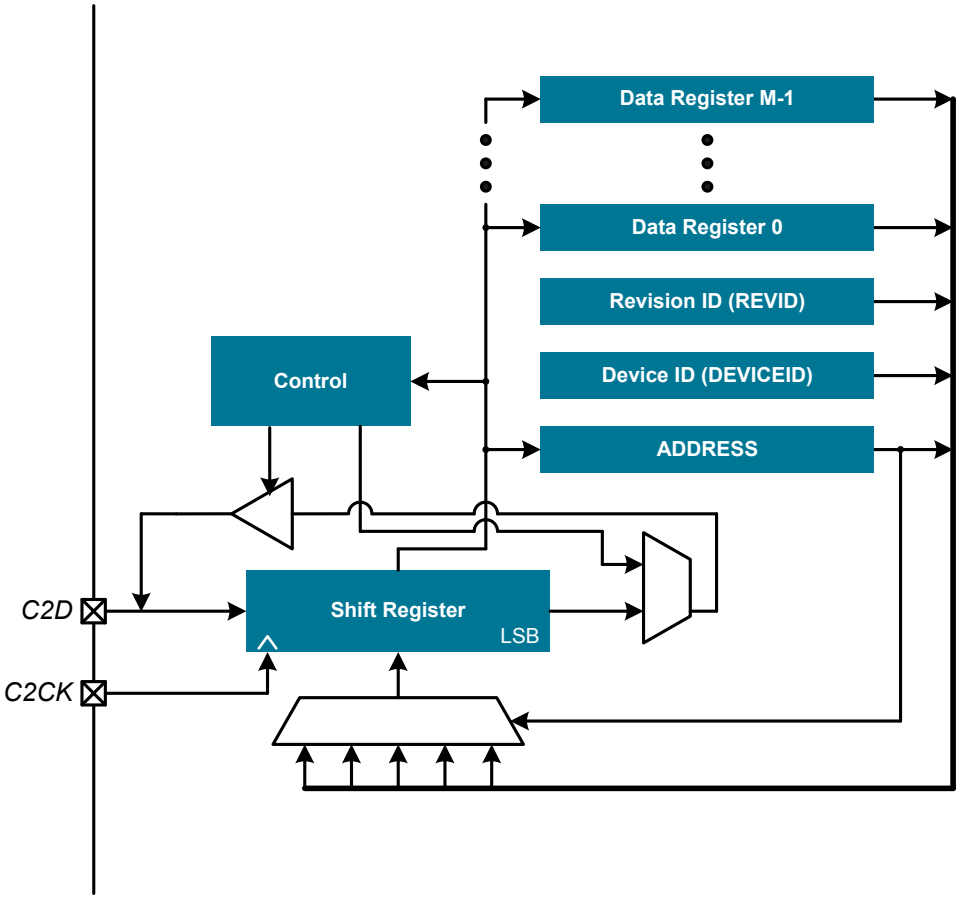


Figure 1.1. C2 Interface Block Diagram

1.1 C2 Basics

The C2 interface operates similar to JTAG with the three JTAG data signals (TDI, TDO, TMS) mapped into one bidirectional C2 data signal (C2D). The signal direction of C2D is strictly specified by the instruction protocol such that contention between the target device and interface master is never allowed. All data is transmitted and received LSB first.

The C2 interface provides access to on-chip programming and debug hardware through a single Address register and a set of Data registers. The Address register defines which Data register will be accessed during Data register read/write instructions (analogous to the JTAG Instruction register). Data registers provide access to various device-specific functions (note: it is not required that all Data registers be both readable and writable). Read and write access to all registers is performed through a common shift register that serves as a serial-to-parallel-to-serial converter.

All C2 devices include an 8-bit Device ID register and an 8-bit Revision ID register. These registers are read-only. Following a device reset, the C2 Address register defaults to 0x00, selecting the 8-bit Device ID register. This allows a C2 master to perform a Device ID register read without knowing the length of the target device's Address register. The length of the target Address register can then be determined using the Device ID register content.

1.2 C2 Registers

1.2.1 ADDRESS: Address Register

The C2 Address register (ADDRESS) serves two purposes in C2 flash or EPROM programming:

1. ADDRESS selects which C2 Data register is accessed during C2 Data Read/Write frames.
2. During Address Read frames, ADDRESS provides PI status information.

Address Reads are used frequently during C2 programming as a handshaking scheme between the programmer and the PI.

The Address Read command returns an 8-bit status code, formatted as shown in the table below.

Table 1.1. C2 Address Register Status Bits

Bit	Name	Description
7	EBusy or FLBusy	This bit indicates when the EPROM or Flash is busy completing an operation.
6	EError	This bit is set to 1 when the EPROM encounters an error.
5:2	—	Unused
1	InBusy	This bit is set to 1 by the C2 Interface following a write to FPDAT. It is cleared to 0 when the PI acknowledges the write to FPDAT.
0	OutReady	This bit is set to 1 by the PI when output data is available in the FPDAT register.

The InBusy bit should be polled following any write to FPDAT, and the OutReady bit should be polled before any reads of FPDAT.

1.2.2 DEVICEID: Device ID Register

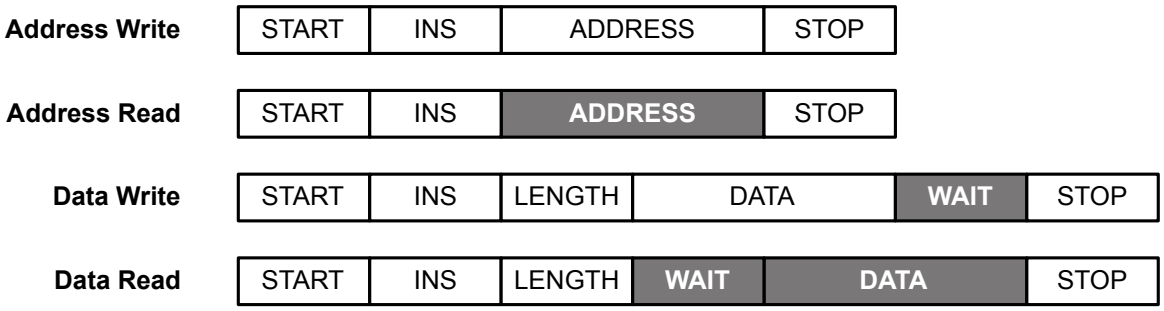
The Device ID register (DEVICEID) is a read-only C2 Data register containing the 8-bit device identifier of the target C2 device. The C2 address for register DEVICEID is 0x00.

1.2.3 REVID: Revision ID Register

The Revision ID register (REVID) is a read-only C2 Data register containing the 8-bit revision identifier of the target C2 device. The C2 address for register REVID is 0x01.

1.3 C2 Instruction Frames

A C2 master accesses the target C2 device via a set of four basic C2 frame formats: Address Write, Address Read, Data Write, and Data Read.



Note: During shaded fields, the C2D signal is driven by the target device.

Figure 1.2. C2 Frame Summary

Note that the master initiates each frame with the START and INS (Instruction) fields. The content of the INS field defines the frame format.

Table 1.2. C2 Instructions

Instruction	INS Code
Data Read	00b
Address Read	10b
Data Write	01b
Address Write	11b

Table 1.3. C2 Bit Field Descriptions

	Field	Description
Master Only	START	A START condition initiates a C2 frame. The master generates this condition by leaving its C2D driver disabled and generating an active-low strobe on C2CK. All C2 frames begin with the START field.
	INS	The INS field is a 2-bit code specifying the current C2 instruction. The four valid C2 instructions are shown in Table 2. All C2 frames include the INS field.
	STOP	A STOP condition ends a C2 frame. The master generates this condition by disabling its C2D driver and generating an active-low C2CK strobe. The slave returns C2D to its user-defined state on the rising edge of this C2CK strobe. All C2 frames are terminated with the STOP field.
	LENGTH	The LENGTH field is a 2-bit code indicating the number of bytes to be read or written during Data register accesses. The number of bytes to transfer is LENGTH + 1 (for example, LENGTH = 01b results in a 2-byte transfer).
Master or Slave	ADDRESS	The ADDRESS field is used to transfer data during Address register accesses. The length of this field must be the same length as the slave device's Address register. The Address register defaults to all zero's following any reset, selecting the Device ID register.
	DATA	The DATA field appears in Data register accesses; the length of this field is determined by the LENGTH field as described above.

	Field	Description
Slave Only	WAIT	A WAIT field appears during Data Read and Data Write frames to allow the slave device to access slower registers or memories. This variable-length field consists of a series of zero or more 0's transmitted by the slave device, terminated by a single 1.
Note: All fields are transmitted LSB first.		

1.3.1 Address Write Frame

An Address Write frame loads the target Address register.

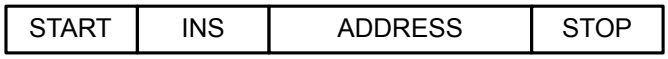


Figure 1.3. Address Write Sequence

The length of the ADDRESS field must always be the length of the slave device's Address register. Following a device reset, the target device's Address register defaults to all zeros, selecting the Device ID register.
Note: All fields are transmitted LSB first.

1.3.2 Address Read Frame

An Address Read frame returns status information or Address register contents from the target device. This instruction is typically used to quickly access status information, though the function of the Address Read instruction is specific to each target device.

Figure 1.4. Address Read Sequence

The length of the ADDRESS field must always be the length of the slave device's Address register.
Note: All fields are transmitted LSB first.

1.3.3 Data Write Frame

A Data Write frame writes a specified value to the target Data register, as selected by the target Address register.

Figure 1.5. Data Write Sequence

LENGTH is a 2-bit field that specifies the length of the DATA field as follows:
DATA length in bytes = LENGTH + 1

The DATA field length must be a multiple of 8 bits. For example, a LENGTH of 01b indicates a DATA length of 2 bytes. The length of the DATA field is not required to be the same length as the target Data register. For example, to write only the eight MSBs of a 10-bit register, LENGTH is set to 00b and DATA specifies only 8 bits of data to be written to the 8 MSBs of the target register. The remaining register bits are undefined. To write all 10 bits of data, LENGTH should be 01b; in this case (shown in Figure 7), the 10 MSBs of the 16-bit DATA field are written to the target register.
Note: All fields are transmitted LSB first.

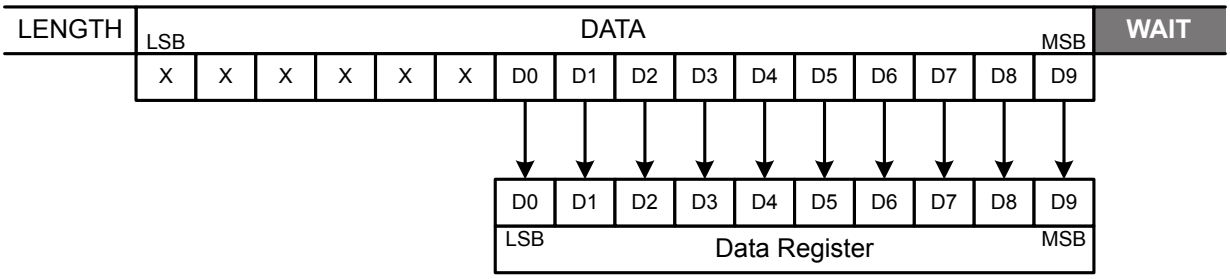


Figure 1.6. DATA Field for a 10-bit Data Register Write

The length of the WAIT field is controlled by the target device. During the WAIT field, the target device transmits 0's on the C2D pin until it has finished writing to the target Data register. To indicate the write complete status, the target device transmits a 1 to terminate the WAIT field.

1.3.4 Data Read Frame

A Data Read frame reads the contents of the target Data register, as selected by the target Address register.

Figure 1.7. Data Read Sequence

LENGTH is a 2-bit field that specifies the length of the DATA field as follows:
DATA length in bytes = LENGTH + 1

The DATA field length must be a multiple of 8 bits. For example, a LENGTH of 01b indicates a DATA length of 2 bytes. As with the Data Write frame, the length of the DATA field is not required to match the length of the target Data register. In this case, the read data is right justified in the DATA field. For example, if LENGTH is 00b (1 byte) and the target register is 12-bits, the 8 LSBs of the target register are read into the DATA field. If LENGTH is 01b (2 bytes) and the target register is 12-bits, the 12-bit Data register makes up the 12 LSBs of the DATA field; the remaining 4 bits are undefined.
Note: All fields are transmitted LSB first.

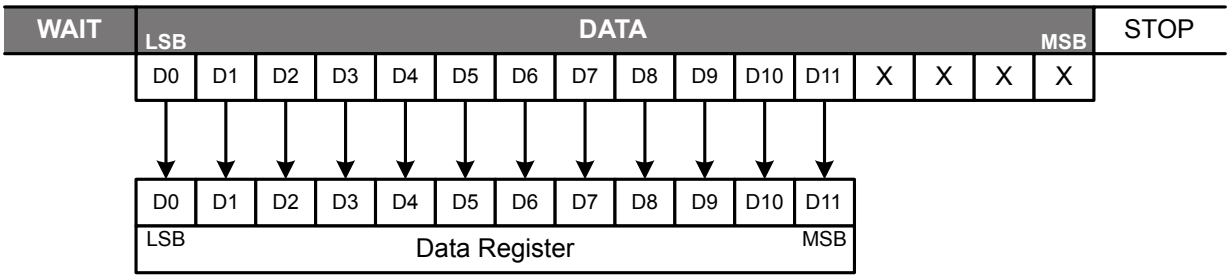


Figure 1.8. DATA Field for a 10-bit Data Register Read

The length of the WAIT field is controlled by the target device. During the WAIT field, the target device transmits 0's on the C2D pin until it has finished reading the target Data register and is ready to shift out data. To indicate the ready status, the target device transmits a 1 to terminate the WAIT field.

1.4 C2 Timing Specifications

This section illustrates the timing sequence for each of the four C2 frame formats and the device reset command.

Table 1.4. C2 Timing Requirements

Parameter	Description	Min	Max
t_{RD}	C2CK low time for a device reset	20 µs	—
t_{SD}	Start bit delay after a device reset	2 µs	—
t_{CL}	C2CK low time for bit transfers	20 ns	5000 ns
t_{CH}	C2CK high time	20 ns	—
t_{DS}	C2D setup time	10 ns	—
t_{DH}	C2D hold time	10 ns	—
t_{ZS}	C2D High-Z setup time	0 ns	—
t_{DV}	C2D valid	—	20 ns
t_{ZV}	C2D High-Z valid	—	20 ns

Because C2CK and RST functions share the same pin, they are distinguished by the length of time the pin is held low. For RST, the pin must be held low for at least 20 µs. For C2CK, the pin cannot be low for longer than 5 µs. If the pin is held low between 5 and 20 µs, the response of the device is undefined.

Data is sampled on C2D on the rising edge of C2CK. C2D changes (as an output) shortly after the rising edge of C2CK. Typical setup and data ready times for C2D are tens of nanoseconds; however some devices may prefer more time (such as the C8051F41x). In general, if C2D is set up before C2CK goes low, C2CK low time is between 80 ns and 5 µs, and C2D is read at least 120 ns after C2CK rising, C2 interface timing will be satisfied.

If the C2CK master is an MCU, care should be taken to disable interrupts while C2CK is low in order to prevent a clock low time extending beyond 5 µs and potentially causing a target device reset.

Shaded C2D bits in these section diagrams indicate times when the master's C2D driver must be disabled.

1.4.1 Device Reset Timing

During C2 instructions, C2CK must not be held low longer than t_{CL}. This requirement allows the device to be reset by holding C2CK low for t_{RD}. The START field of the first C2 instruction must begin at least t_{SD} after C2CK returns high following a device reset.

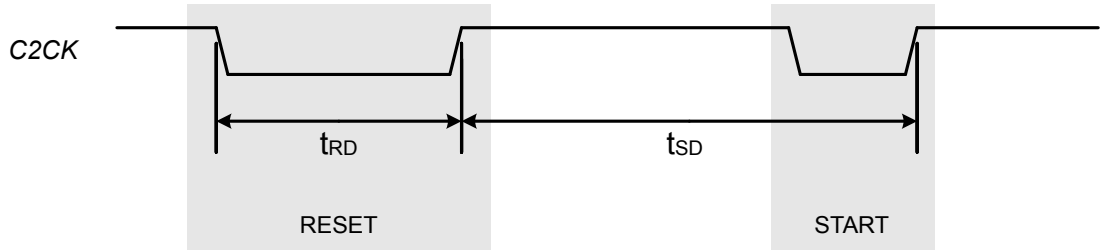


Figure 1.9. Device Reset Timing

1.4.2 Address Write Timing

The 8-bit Address Register Write frame begins with a START (rising edge on C2CK). Note that during a START condition, the master's C2D driver should be disabled. Following the START, the interface master must enable its C2D driver to transmit the INS and ADDRESS bits. Following the last ADDRESS bit, the master disables its C2D driver and strobes C2CK one last time for the STOP field; the slave device returns the C2D pin to its user-defined state following the last rising edge on C2CK.

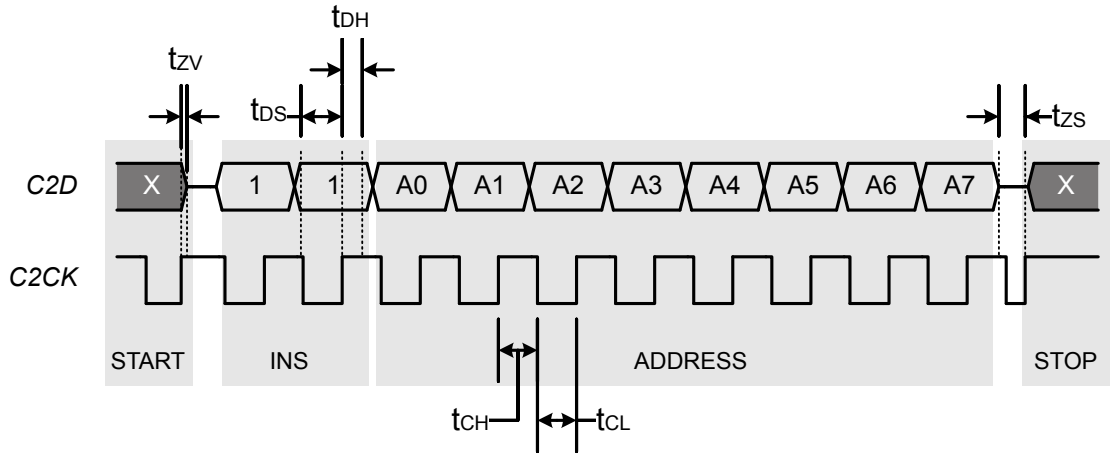


Figure 1.10. Address Write Timing

1.4.3 Address Read Timing

The 8-bit Address Register Read frame begins with a START followed by the 2-bit INS field. Following the INS bits, the interface master disables its C2D driver and strobes C2CK; the slave device outputs the LSB of its Address register on the rising edge of C2CK. Seven more C2CK strobes are required to complete the ADDRESS field. Following the last ADDRESS bit, the master strobes C2CK one last time for the STOP field; the slave device returns the C2D pin to its user-defined state following the last rising edge on C2CK.

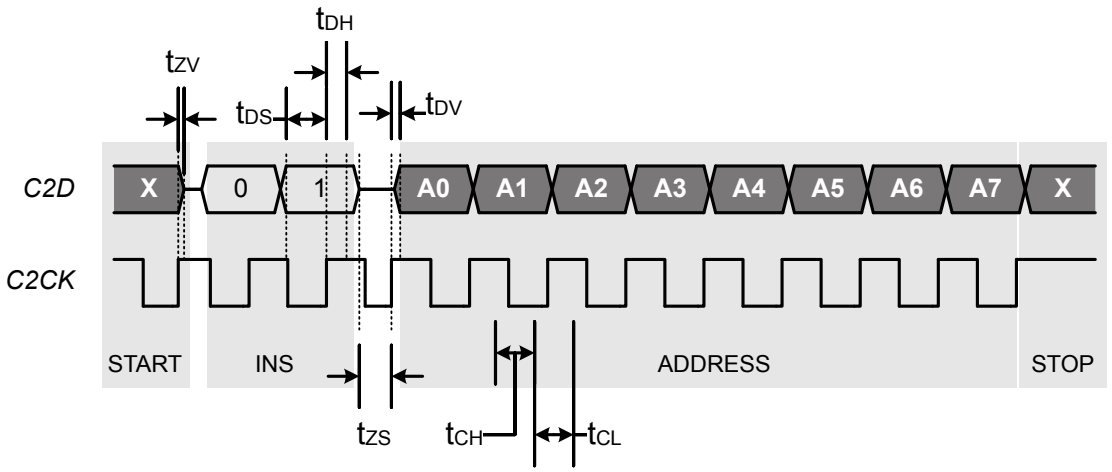


Figure 1.11. Address Read Timing

1.4.4 Data Write Timing

The 1-byte Data Register Write frame begins with a START followed by the 2-bit INS and 2-bit LENGTH fields. In this example, the LENGTH field is 00b indicating a 1-byte transfer. The master transmits the 8-bits of data; following the last DATA bit, the master disables its C2D driver for the WAIT field. In this example the slave transmits only one 0 during the WAIT field. Following the WAIT field, the master strobes C2CK one last time for the STOP field; the slave device returns the C2D pin to its user-defined state following the last rising edge on C2CK.

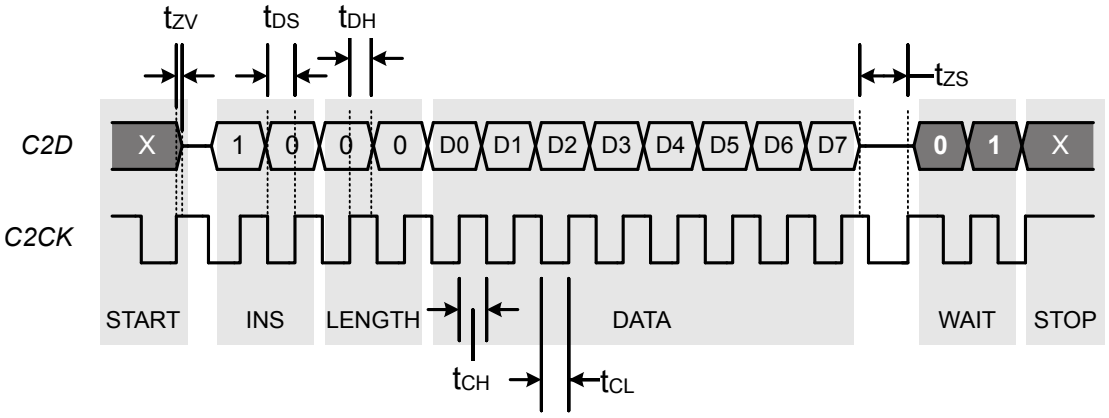


Figure 1.12. Data Write Timing

1.4.5 Data Read Timing

The 1-byte Data Register Read frame begins with a START followed by the 2-bit INS and 2-bit LENGTH fields. In this example, the LENGTH field is 00b indicating a 1-byte transfer. After the last bit of the LENGTH field is transmitted, the master disables its C2D driver for the WAIT field. In this example only one 0 is transmitted during the WAIT field. Following the WAIT field, the master strobes C2CK and the slave shifts out the DATA field. Following the last DATA bit, the master strobes C2CK one last time for the STOP field; the slave device returns the C2D pin to its user-defined state following the last rising edge on C2CK.

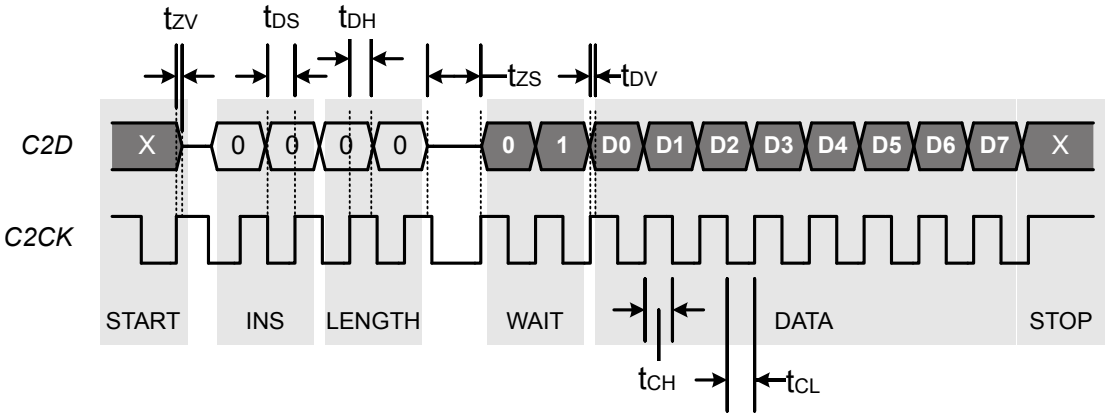


Figure 1.13. Data Read Timing

1.5 C2 Primitives

This section discusses the specific steps for generating various C2 operations.

1.5.1 Device Reset

To generate the reset timing:

1. Turn the C2CK driver on.
2. Force C2CK low.
3. Wait at least 20 µs.
4. Force C2CK high.
5. Wait at least 2 µs.
6. (Optional) Turn the C2CK driver off.

1.5.2 C2CK Clock Strobes

To generate C2CK clock strobes with a microcontroller-based programmer:

1. Turn the C2CK driver on.
2. Wait at least 40 ns. This helps ensure C2D data setup time.
3. Force C2CK low. Ensure interrupts are disabled at this point.
4. Wait between 80 and 5000 ns.
5. Force C2CK high.
6. Wait at least 120 ns. This helps ensure C2D data valid time.

1.5.3 Performing the Address Write Instruction

The C2CK strobes mentioned in this section refer to the steps described in 1.5.2 C2CK Clock Strobes. To write to a target device's ADDRESS register:

1. Disable interrupts.
2. Turn the C2CK driver on.
3. Strobe C2CK. This will generate the START field and forces the C2D pin on the target device to become an input.
4. Turn on the C2D driver.
5. Force C2D high. This sets C2D for the 2-bit INS field, and the Address Write instruction is 11b.
6. Strobe C2CK. This transfers the first bit of the INS field.
7. Strobe C2CK. This transfers the second bit of the INS field.
8. Force C2D to each bit of the address value being written to ADDRESS (starting with bit 0) and strobe C2CK for each bit.
9. Turn off the C2D driver. This prepares C2D to possibly become an output from the target device.
10. Strobe C2CK to generate the STOP field.
11. (Optional) Turn off the C2CK driver.
12. Re-enable interrupts.

1.5.4 Performing the Address Read Instruction

The C2CK strobes mentioned in this section refer to the steps described in 1.5.2 C2CK Clock Strobes. The OutReady and InBusy bits returned by the Address Read instruction are described in 1.2.1 ADDRESS: Address Register.

To read the status code from a target device's ADDRESS register:

1. Disable interrupts.
2. Turn the C2CK driver on.
3. Strobe C2CK. This will generate the START field and forces the C2D pin on the target device to become an input.
4. Turn on the C2D driver.
5. Force C2D low. This sets C2D for the first part of the 2-bit INS field, and the Address Read instruction is 01b.
6. Strobe C2CK. This transfers the first bit of the INS field.
7. Force C2D high. This sets C2D for the second part of the 2-bit INS field.
8. Strobe C2CK. This transfers the second bit of the INS field.
9. Turn off the C2D driver. This prepares C2D to become an output from the target device.
10. Strobe C2CK and read C2D from the target device. The device will return the 8-bit status code returned from an Address Read instruction bit by bit starting with bit 0. C2CK must be strobed after each read of C2D.
11. Strobe C2CK to generate the STOP field.
12. (Optional) Turn off the C2CK driver.
13. Re-enable interrupts.

1.5.5 Performing the Data Write Instruction

The C2CK strobes mentioned in this section refer to the steps described in 1.5.2 C2CK Clock Strobes. To generate a Data Write instruction:

1. Disable interrupts.
2. Turn on the C2CK driver.
3. Strobe C2CK. This will generate the START field and forces the C2D pin on the target device to become an input.
4. Turn on the C2D driver.
5. Force C2D high. This sets C2D for the first part of the 2-bit INS field, and the Data Write instruction is 10b.
6. Strobe C2CK. This transfers the first bit of the INS field.
7. Force C2D low. This sets C2D for the second part of the 2-bit INS field.
8. Strobe C2CK. This transfers the second bit of the INS field.
9. Leave C2D low. This is the start of the LENGTH field, which will be 00b in this example.
10. Strobe C2CK to start the LENGTH field.
11. Strobe C2CK to finish transferring the LENGTH field.
12. Force C2D to each bit of the data value being written to the data register (starting with bit 0) and strobe C2CK for each bit.
13. Turn off the C2D driver. This prepares C2D to become an output from the target device.
14. Srobe C2CK.
15. Check C2D for a value of 1b. If C2D becomes 1, the WAIT field is over, as it can contain zero or more 0's and exactly one 1b. If C2D is not 1b, strobe C2CK and check the status again.
16. At this point, C2D is driving a 1, indicating the end of the WAIT field. Strobe C2CK to generate the STOP field.
17. (Optional) Turn off the C2CK driver.
18. Re-enable interrupts.

1.5.6 Performing the Data Read Instruction

The C2CK strobes mentioned in this section refer to the steps described in 1.5.2 C2CK Clock Strobes. To generate a Data Read instruction:

1. Disable interrupts.
2. Turn on the C2CK driver.
3. Strobe C2CK. This will generate the START field and forces the C2D pin on the target device to become an input.
4. Turn on the C2D driver.
5. Force C2D low. This sets C2D for the first part of the 2-bit INS field, and the Data Read instruction is 00b.
6. Strobe C2CK to transfer the first bit of the INS field.
7. Strobe C2CK to transfer the second bit of the INS field.
8. Leave C2D low. This is the start of the LENGTH field, which will be 00b in this example.
9. Strobe C2CK to start the LENGTH field.
10. Strobe C2CK to finish transferring the LENGTH field.
11. Turn off the C2D driver. This prepares C2D to become an output from the target device.
12. Strobe C2CK.
13. Check C2D for a value of 1b. If C2D becomes 1, the WAIT field is over, as it can contain zero or more 0's and exactly one 1b. If C2D is not 1b, strobe C2CK and check the status again.
14. At this point, C2D is driving a 1, indicating the end of the WAIT field.
15. Strobe C2CK and read C2D from the target device. The device will return the contents of the data register bit by bit starting with bit 0. C2CK must be strobed after each read of C2D.
16. Strobe C2CK to generate the STOP field.
17. (Optional) Turn off the C2CK driver.
18. Re-enable interrupts.

2. Programming Registers

Communication between the PI and C2I is accomplished via two flash programming registers: FPCTL and FPDAT. EPROM devices have additional EPCTL, EPDAT, EPADDRH, EPADDRL, and EPSTAT registers.

2.1 FPCTL: Flash Programming Control Register

The Flash Programming Control register (FPCTL) serves to enable C2 flash or EPROM programming. To enable C2 programming, the following key codes must be written to FPCTL in the following sequence:

1. 0x02
2. 0x04
3. 0x01

The key codes must be written in this sequence following a target device reset. Once the key codes have been written to FPCTL, the target device is halted until the next device reset.

2.2 FPDAT: Flash Programming Data Register

The Flash Programming Data register (FPDAT) is used to pass all data between the C2I and PI on Flash devices. Information passed via FPDAT includes:

- All PI Commands (C2I-to-PI)
- PI Status Information (PI-to-C2I)
- Flash Addresses (C2I-to-PI)
- Flash Data (both directions)

2.3 EPCTL: EPROM Programming Control Register

The EPROM Programming Control register (EPCTL) contains additional controls for EPROM programming over the C2 interface. To enable C2 programming, the following key codes must be written to EPCTL:

1. 0x40
2. 0x58

The key codes must be written in this sequence after the initial writes to FPCTL.

2.4 EPDAT: EPROM Programming Data Register

The EPROM Programming Data register (EPDAT) is used to pass all data between the C2I and PI on EPROM devices. Information passed via EPDAT includes:

- All PI Commands (C2I-to-PI)
- PI Status Information (PI-to-C2I)
- EPROM Addresses (C2I-to-PI)
- EPROM Data (both directions)

2.5 EPADDRH and EPADDRL: EPROM Programming Address

The EPROM Programming Address registers (EPADDRH and EPADDRL) contain the address written to by the PI on the EPROM device.

2.6 EPSTAT: EPROM Programming Status

The EPROM Status register (EPSTAT) provides additional status information when programming the EPROM of a device.

Table 2.1. EPROM Programming Status Bits

Bit	Name	Description
7	WLOCK	This bit indicates the device is locked from EPROM writes.
6	RLOCK	This bit indicates the device is locked from EPROM reads.
5:4	—	Reserved
3:1	—	Unused
0	ERROR	This bit is set to 1 by the PI of an EPROM device detects an error.

3. Programming Interface

The Programming Interface (PI) performs a set of programming commands. Each command is executed using a sequence of reads and writes of the FPDAT register.

Table 3.1. C2 Flash Programming Commands

Command	Code	Supported Devices
Block Write	0x07	all (flash or EPROM)
Block Read	0x06	all (flash or EPROM)
Page Erase	0x08	flash devices only
Device Erase	0x03	flash devices only
Get Version	0x01	all (flash or EPROM)
Get Derivative	0x02	all (flash or EPROM)
Direct Read	0x09	all (flash or EPROM)
Direct Write	0x0A	all (flash or EPROM)
Indirect Read	0x0B	all (flash or EPROM)
Indirect Write	0x0C	all (flash or EPROM)

The most-commonly used commands for flash or EPROM programming are Block Write, Block Read, Page Erase, and Device Erase.

The PI must be initialized with the following sequence:

1. Drive C2CK low.
2. Wait at least 20 µs.
3. Drive C2CK high.
4. Wait at least 2 µs.
5. Perform an Address Write instruction targeting the FPCTL register (0x02).
6. Perform a Data Write instruction sending a value of 0x02.
7. Perform a Data Write instruction sending a value of 0x04 to halt the core.
8. Perform a Data Write instruction sending a value of 0x01.
9. Wait at least 20 ms.

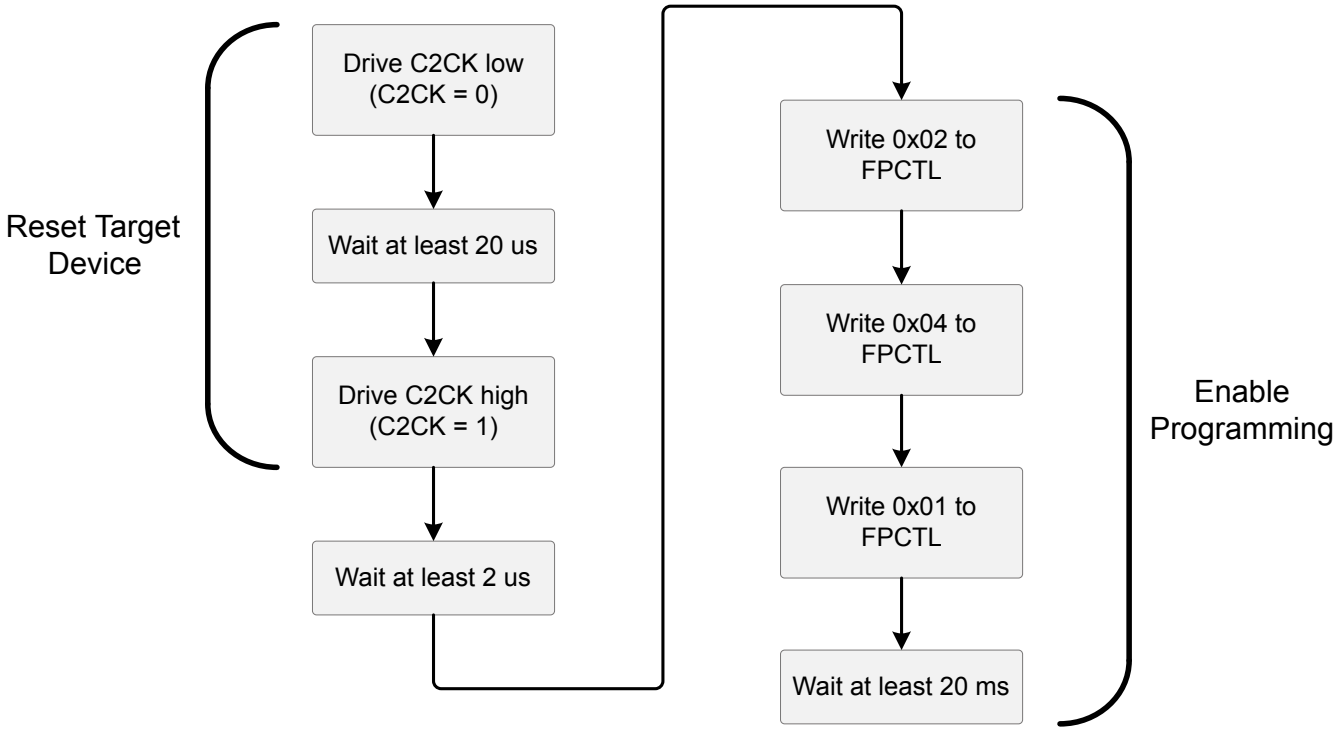


Figure 3.1. PI Initialization Sequence

Some devices need additional configuration before executing write or erase operations. See 3.8 Device-Specific Configurations for more information.

3.1 Writing a Flash or EPROM Block

All flash writes are performed with the Block Write command. The size of the block can be 1-to-256 bytes and is user-defined during the Block Write sequence. The 8-bit data length code is formatted as shown below.

Table 3.2. Block Write Length Formatting

8-bit Data Length Code	Number of Bytes
1 - 255	block size equal to length code
0	256

Figure 3.2 Basic Flash Block Write Sequence on page 16 shows the basic Block Write sequence and assumes the PI has already been initialized by the sequence in Figure 3.1 PI Initialization Sequence on page 15.

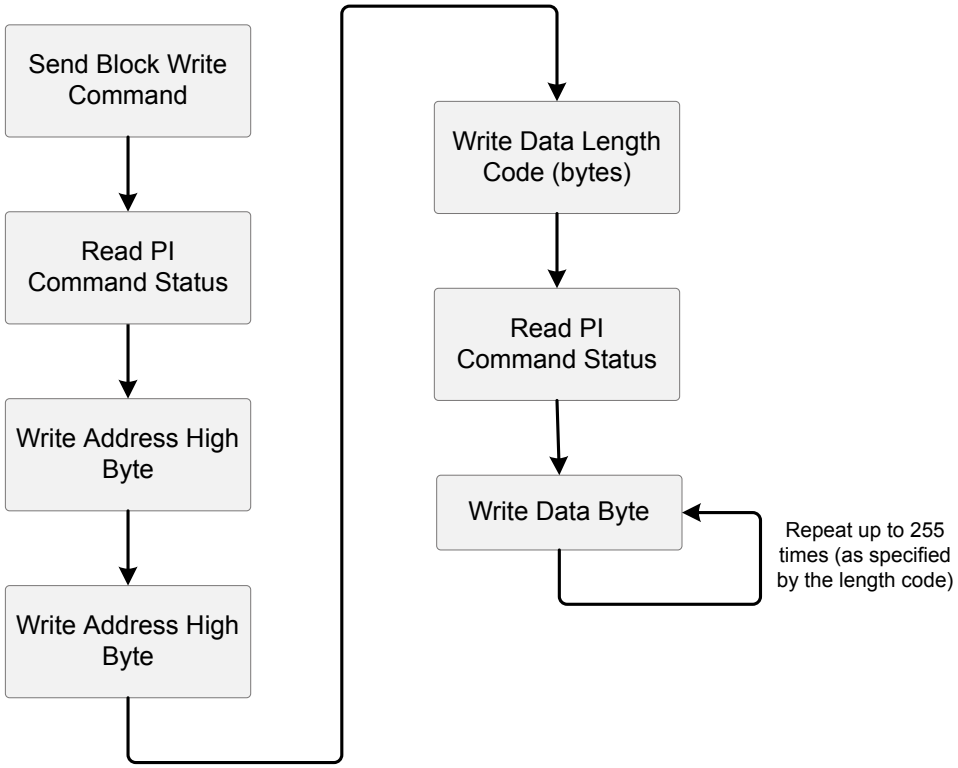


Figure 3.2. Basic Flash Block Write Sequence

To program a flash block:

1. Perform an Address Write with a value of FPDAT.
2. Perform a Data Write with the Block Write command.
3. Poll on InBusy using Address Read until the bit clears.
4. Poll on OutReady using Address Read until the bit set.
5. Perform a Data Read instruction. A value of 0x0D is okay.
6. Perform a Data Write with the high byte of the address.
7. Poll on InBusy using Address Read until the bit clears.
8. Perform a Data Write with the low byte of the address.
9. Poll on InBusy using Address Read until the bit clears.
10. Perform a Data Write with the length.
11. Poll on InBusy using Address Read until the bit clears.
12. Perform a Data Write with the data. This will write the data to the flash. Repeat steps 11 and 12 for each byte specified by the length field.

13. Poll on OutReady using Address Read until the bit set.
14. Perform a Data Read instruction. A value of 0x0D is okay.

To write to an EPROM block:

1. Write 0x04 to the FPCTL register.
2. Write 0x40 to EPCTL.
3. Write 0x58 to EPCTL.
4. Write the high byte of the address to EPADDRH.
5. Write the low byte of the address to address EPADDRL.
6. Perform an Address Write with a value of EPDAT.
7. Turn on VPP.
8. Wait for the VPP settling time.
9. Write the data to the device using a Data Write.
10. Perform Address Read instructions until the value returned is not 0x80 and the EPROM is no longer busy.
11. Repeat steps 9 and 10 until all bytes are written.
12. Turn off VPP. Note that VPP can only be applied for a maximum lifetime amount, and this value is specified in the device data sheet.
13. Write 0x40 to EPCTL.
14. Write 0x00 to EPCTL.
15. Write 0x02 to FPCTL.
16. Write 0x04 to FPCTL.
17. Write 0x01 to FPCTL.

3.2 Reading a Flash or EPROM Block

All flash reads are performed with the Block Read command. The size of the block can be 1-to-256 bytes and is user-defined in the Block Read sequence. The 8-bit data length code is formatted as shown in the table below.

Table 3.3. Block Read Length Formatting

8-bit Data Length Code	Number of Bytes
1 - 255	block size equal to length code
0	256

The Block Read sequence is shown in Figure 3.3 Basic Flash Block Read Sequence on page 18. The flow diagram and sequences described assume the PI has already been initialized by the sequence in Figure 3.1 PI Initialization Sequence on page 15.

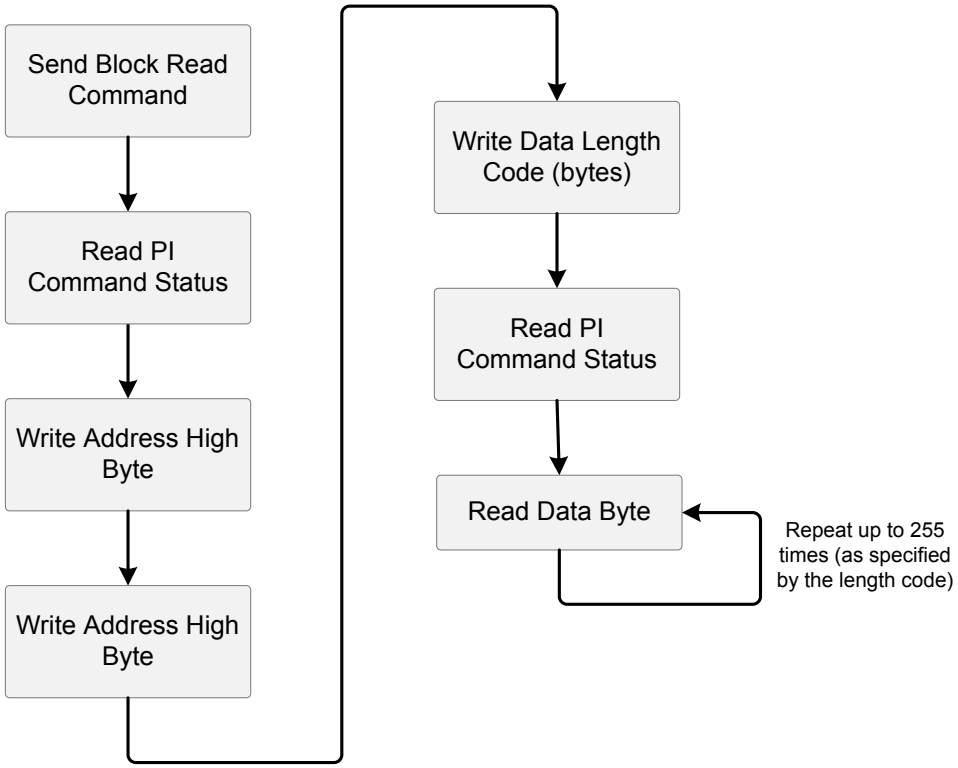


Figure 3.3. Basic Flash Block Read Sequence

To read a flash block:

1. Perform an Address Write with a value of FPDAT.
2. Perform a Data Write with the Block Read command.
3. Poll on InBusy using Address Read until the bit clears.
4. Poll on OutReady using Address Read until the bit set.
5. Perform a Data Read instruction. A value of 0x0D is okay.
6. Perform a Data Write with the high byte of the address.
7. Poll on InBusy using Address Read until the bit clears.
8. Perform a Data Write with the low byte of the address.
9. Poll on InBusy using Address Read until the bit clears.
10. Perform a Data Write with the length.
11. Poll on InBusy using Address Read until the bit clears.
12. Poll on OutReady using Address Read until the bit set.

13. Perform a Data Read instruction. This will read the data from the flash. Repeat step 12 and 13 for each byte specified by the length field.

To read an EPROM block:

1. Write 0x04 to the FPCTL register.
2. Write 0x00 to EPCTL.
3. Write 0x58 to EPCTL.
4. Write the high byte of the address to EPADDRH.
5. Write the low byte of the address to address EPADDRL.
6. Perform an Address Write with a value of EPDAT.
7. Perform Address Read instructions until the value returned is not 0x80 and the EPROM is no longer busy.
8. Read the byte using the Data Read instruction.
9. Repeat steps 7 and 8 until all bytes are read.
10. Write 0x40 to EPCTL.
11. Write 0x00 to EPCTL.
12. Write 0x02 to FPCTL.
13. Write 0x04 to FPCTL.
14. Write 0x01 to FPCTL.

To perform additional status checks during the EPROM read operation, insert these steps in front of step 7 in the previous sequence:

1. Perform an Address Write operation with a value of EPSTAT.
2. Perform a Data Read operation and check the bits of the EPSTAT register.
3. Perform an Address Write operation with a value of EPDAT.

In addition, insert these steps in front of step 8 in the EPROM read sequence:

1. Perform an Address Write operation with a value of EPSTAT.
2. Perform a Data Read operation and check the ERROR bit in the EPSTAT register.
3. Perform an Address Write operation with a value of EPDAT.

3.3 Erasing a Flash Page

Flash memory is erased in pages using the Page Erase command. The size of the page varies depending on the device family. The page to be erased is selected in the Page Erase procedure shown in the figure below.

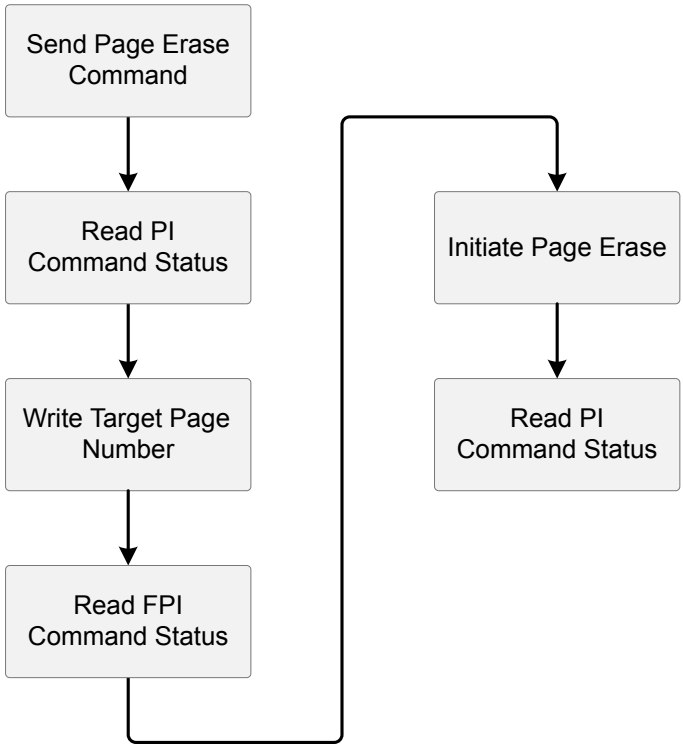


Figure 3.4. Basic Page Erase Sequence

This flow diagram assumes the PI has already been initialized by the sequence in Figure 3.1 PI Initialization Sequence on page 15.

To erase a page of flash:

1. Perform an Address Write with a value of FPDAT.
2. Perform a Data Write with the Page Erase command.
3. Poll on InBusy using Address Read until the bit clears.
4. Poll on OutReady using Address Read until the bit set.
5. Perform a Data Read instruction. A value of 0x0D is okay.
6. Perform a Data Write with the page number.
7. Poll on InBusy using Address Read until the bit clears.
8. Poll on OutReady using Address Read until the bit clears.
9. Perform a Data Read instruction. A value of 0x0D is okay.
10. Perform a Data Write with the a value of 0x00.
11. Poll on InBusy using Address Read until the bit clears.
12. Poll on OutReady using Address Read until the bit set.
13. Perform a Data Read instruction. A value of 0x0D is okay.

3.4 Erasing the Device

The Device Erase command performs a mass erase of the flash memory and erases all pages in the device. A three-byte arming sequence must be written to the PI to enable this procedure:

1. 0xDE
2. 0xAD
3. 0xA5

The erase operation executes following the last byte of the arming sequence. The Device Erase programming sequence is shown in the figure below. This flow diagram assumes the PI has already been initialized by the sequence in Figure 3.1 PI Initialization Sequence on page 15.

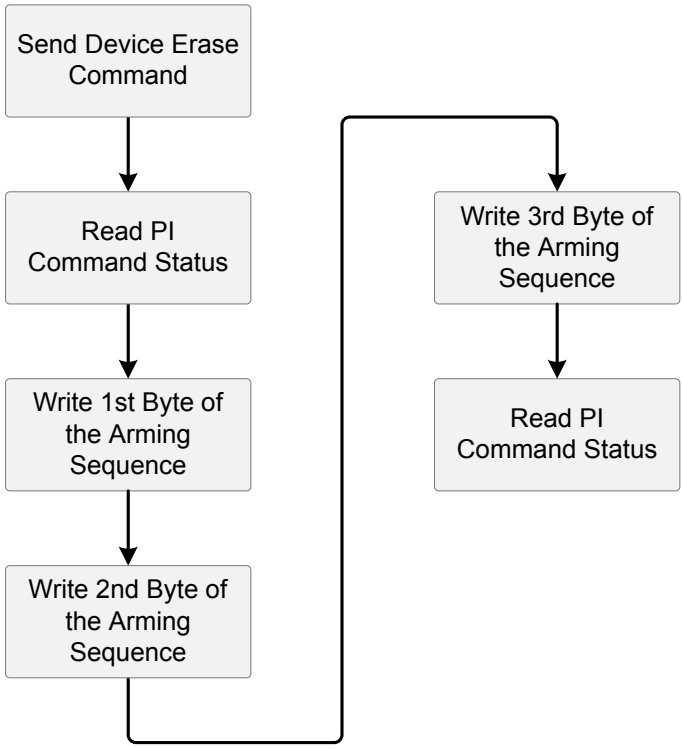


Figure 3.5. Basic Device Erase Sequence

To erase a flash device:

1. Perform an Address Write with a value of FPDAT.
2. Perform a Data Write with the Device Erase command.
3. Poll on InBusy using Address Read until the bit clears.
4. Poll on OutReady using Address Read until the bit set.
5. Perform a Data Read instruction. A value of 0x0D is okay.
6. Perform a Data Write with a value of 0xDE.
7. Poll on InBusy using Address Read until the bit clears.
8. Perform a Data Write with a value of 0xAD.
9. Poll on InBusy using Address Read until the bit clears.
10. Perform a Data Write with a value of 0xA5.
11. Poll on InBusy using Address Read until the bit clears.
12. Poll on OutReady using Address Read until the bit set.
13. Perform a Data Read instruction. A value of 0x0D is okay.

3.5 Writing to an SFR

To write to an SFR on a device that does not have SFR paging:

1. Write the SFR address to the device using the Address Write instruction.
2. Write the SFR value to the device using the Data Write instruction.

For devices with SFR paging, direct writes through the PI using the Direct Write command are recommended to ensure the SFR Page is managed properly.

To write to an SFR from a device with SFR paging:

1. Perform an Address Write with a value of FPDAT.
2. Write the Direct Write command (0x0A) using a Data Write.
3. Poll InBusy until the data is processed by the PI.
4. Poll OutReady it sets to 1.
5. Perform a Data Read to ensure a return value of 0x0D (no errors).
6. Perform a Data Write with a value of the SFR address.
7. Poll InBusy until the data is processed by the PI.
8. Perform a Data Write with a value of 0x01.
9. Poll InBusy until the data is processed by the PI.
10. Perform a Data Write with the new SFR value.
11. Poll InBusy until the data is processed by the PI.

3.6 Reading from an SFR

To read from an SFR on a device that does not have SFR paging:

1. Write the SFR address to the device using the Address Write instruction.
2. Read the SFR value from the device using the Data Read instruction.

For devices with SFR paging, direct reads through the PI using the Direct Read command are recommended to ensure the SFR Page is managed properly.

To read an SFR from a device with SFR paging:

1. Perform an Address Write with a value of FPDAT.
2. Write the Direct Read command (0x09) using a Data Write.
3. Poll InBusy until the data is processed by the PI.
4. Poll OutReady until it sets to 1.
5. Perform a Data Read to ensure a return value of 0x0D (no errors).
6. Perform a Data Write with a value of the SFR address.
7. Poll InBusy until the data is processed by the PI.
8. Perform a Data Write with a value of 0x01.
9. Poll InBusy until the data is processed by the PI.
10. Poll OutReady until it sets to 0.
11. Read the SFR value from the device using the Data Read instruction.

3.7 Reading and Writing Flash on devices with More than 64 KB of Memory

Reading and writing flash memory on devices with more than 64 KB of code space requires translating the linear target address into a 16-bit banked address. Note that erasing a page does not require such a translation because the page erase operation accepts page numbers, not byte addresses.

Prior to starting the actual flash read or write, the PSBANK and COBANK bits must be set appropriately for the bank to be targeted. Ideally this operation should occur prior to writing the address bytes. The mapping of linear address to banked address, along with the correct value of PSBANK, are shown in the table below.

Table 3.4. Translating Linear Addresses to Banked Addresses

Linear Address Range (17 bits)	Banked Address Range (16 bits)	PSBANK Register Value
0x00000–0x0FFFF	0x0000–0xFFFF	0x11 (default)
0x10000–0x17FFF	0x8000–0xFFFF[1]	0x22
0x18000–0x1FFFF	0x8000–0xFFFF	0x33

Note:
1. Set b15 to translate linear address into banked address for linear range 0x10000 – 0x17FF.

PSBANK should be restored to its default value after the flash operation has completed successfully. The PSBANK and COBANK registers are located at different addresses on each device with banked addressing.

On ‘F58x devices, PSBANK is located at SFR address 0xF5. It appears on all SFR pages and can be accessed with the WriteSFR() operation. On ‘F96x devices, PSBANK is located at SFR address 0x84 and appears on all SFR pages.

3.8 Device-Specific Configurations

Some devices require configuration prior to the start of write or erase operations. The required operations include:

1. Setting up the flash timing register.
2. Setting up the voltage regulator (programming it to a high setting).
3. Enabling the VDD monitor and enabling the VDD monitor as a reset source.

Optionally, the system clock of the device can be increased. This will slightly reduce flash write times but will have a very favorable impact on flash read times.

The following table outlines the device-specific programming information, including the device ID values for each family and the location of the FPDAT register.

The procedures for configuring the various options for specific device families are given in Table 11 using the common functions described in 4.1 Common Functions. If a table entry is blank, that means that particular device does not require any special configuration. In general, the sequences should be executed in the order that they are listed in the table from left to right.
Note: The WriteDirect(...) function calls are used on devices with SFR Paging to ensure the SFR Page is managed correctly.

Table 3.5. Device-Specific Programming Information

Family Name	Family (DEVID)	FPDAT Address	Page Size
‘F30x	0x04	0xB4	512
‘F31x	0x08	0xB4	512
‘F32x	0x09	0xB4	512
‘F326/7	0x0D	0xB4	512
‘F33x	0x0A	0xB4	512
‘F336/7	0x14	0xB4	512
‘F34x	0x0F	0xAD	512
‘F35x	0x0B	0xB4	512
‘F36x	0x12	0xB4	1024
‘F38x	0x28	0xAD	512
‘F39x/’F37x	0x2B	0xB4	512
‘F41x	0x0C	0xB4	512
‘F50x/’F51x	0x1C	0xB4	512
‘F52x/’F53x	0x11	0xB4	512
‘F54x	0x22	0xB4	512
’F55x/’F56x/’F57x	0x22	0xB4	512
‘F58x/’F59x	0x20	0xB4	512
‘F70x/’F71x	0x1E	0xB4	512
‘F80x/’F81x/’F82x/’F83x	0x23	0xB4	512
'F85x/'F86x	0x30	0xB4	512
‘F90x/’F91x	0x1F	0xB4	512
‘F92x/’F93x	0x16	0xB4	1024
‘F96x	0x2A	0xB4	1024
‘F99x	0x25	0xB4	512
‘T60x	0x10	0xB4	512

Family Name	Family (DEVID)	FPDAT Address	Page Size
‘T606	0x1B	0xB4	512
‘T61x	0x13	0xB4	512
‘T62x/’T32x	0x18	0xAD	512
‘T622/’T623/‘T326/’T327	0x19	0xAD	512
‘T63x	0x17	0xB4	512
EFM8BB1	0x30	0xB4	512
EFM8BB2	0x32	0xB4	512
EFM8BB3	0x34	0xB4	512
EFM8LB1	0x34	0xB4	512
EFM8SB1	0x25	0xB4	512
EFM8SB2	0x16	0xB4	1024
EFM8UB1	0x32	0xB4	512
EFM8UB2	0x28	0xAD	512

Table 3.6. Device-Specific Programming Sequences

Family Name	Flash Timing	Voltage Regulator Initialization	VDD Monitor Initialization	Oscillator Initialization
‘F30x				WriteSFR(0xB2, 0x07)
‘F31x				WriteDirect(0xEF, 0x00) WriteDirect(0xB2, 0x83)
‘F32x				WriteSFR(0xB2, 0x83)
‘F326/7				WriteSFR(0xB2, 0x83)
‘F33x				WriteSFR(0xB2, 0x83)
‘F336/7				WriteSFR(0xB2, 0x83)
‘F34x	WriteSFR(0xB6, 0x90)		WriteSFR(0xFF, 0x80) WriteSFR(0xEF, 0x02)	WriteSFR(0xB2, 0x83)
‘F35x	WriteSFR(0xB6, 0x10)			WriteSFR(0xB2, 0x83)
‘F36x	WriteDirect (0xA7, 0x0F) WriteDirect (0x84, 0x00) WriteDirect (0xA7, 0x00) WriteDirect (0xB6, 0x00)			WriteDirect(0xA7,0x0F) WriteDirect(0xB7, 0x83) WriteDirect(0xA7, 0x00)
‘F38x	WriteSFR(0xB6, 0x90)		WriteSFR(0xFF, 0x80) WriteSFR(0xEF, 0x02)	WriteSFR(0xA9, 0x03)
‘F39x/’F37x			WriteSFR(0xFF, 0x80) WriteSFR(0xEF, 0x02)	WriteSFR(0xB2, 0x83)
‘F41x	WriteSFR(0xB6, 0x10)	WriteSFR(0xC9, 0x10)	WriteSFR(0xFF, 0xA0) WriteSFR(0xEF, 0x02)	WriteSFR(0xB2, 0x87)

Family Name	Flash Timing	Voltage Regulator Initialization	VDD Monitor Initialization	Oscillator Initialization
‘F50x/’F51x			WriteDirect(0xFF, 0xA0) Delay 100 µs WriteDirect(0xEF, 0x02)	WriteDirect(0xA7, 0x0F) WriteDirect(0xA1, 0xC7) WriteDirect(0x8F, 0x00) WriteDirect(0xA7, 0x00)
‘F52x/’F53x			WriteSFR(0xFF, 0xA0)	WriteSFR(0xB2, 0x87)
‘F54x			WriteDirect(0xFF, 0xA0) Delay 100 µs WriteDirect(0xEF, 0x02)	WriteDirect(0xA7, 0x0F) WriteDirect(0xA1, 0xC7) WriteDirect(0x8F, 0x00) WriteDirect(0xA7, 0x00)
’F55x/’F56x/’F57x			WriteDirect(0xFF, 0xA0) Delay 100 µs WriteDirect(0xEF, 0x02)	WriteDirect(0xA7, 0x0F) WriteDirect(0xA1, 0xC7) WriteDirect(0x8F, 0x00) WriteDirect(0xA7, 0x00)
‘F58x/’F59x	WriteDirect(0xB6, 0x02)		WriteDirect(0xFF, 0xA0) Delay 100 µs WriteDirect(0xEF, 0x02)	WriteDirect(0xA7, 0x0F) WriteDirect(0xA1, 0xC7) WriteDirect(0xA7, 0x00)
‘F70x/’F71x				WriteDirect(0xA7, 0x0F) WriteDirect(0xA9, 0x83) WriteDirect(0xBD, 0x00) WriteDirect(0xA7, 0x00)
‘F80x/’F81x/’F82x/’F83x				WriteSFR(0xB2, 0x83)
'F85x/'F86x			WriteSFR(0xFF, 0x80) Delay 5 µs WriteSFR(0xEF, 0x02)	WriteSFR(0xA9, 0x00)
‘F90x/’F91x				WriteDirect(0xA7, 0x00) WriteDirect(0xB2, 0x8F) WriteDirect(0xA9, 0x00)
‘F92x/’F93x				WriteDirect(0xA7, 0x00) WriteDirect(0xB2, 0x8F) WriteDirect(0xA9, 0x00)
‘F96x	WriteDirect(0xA7, 0x0F) WriteDirect(0xB6, 0x00) WriteDirect(0xA7, 0x00)		WriteDirect(0xFF, 0x88) WriteDirect(0xEF, 0x02)	WriteDirect(0xA7, 0x00) WriteDirect(0xA9, 0x04)
‘F99x	WriteDirect(0xB6, 0x40)		WriteDirect(0xFF, 0x80) WriteDirect(0xEF, 0x02)	WriteDirect(0xA9, 0x04)
‘T60x				WriteSFR(0xB2, 0x07)

Family Name	Flash Timing	Voltage Regulator Initialization	VDD Monitor Initialization	Oscillator Initialization
‘T606				WriteSFR(0xB2, 0x07)
‘T61x				WriteSFR(0xB2, 0x83)
‘T62x/’T32x				WriteSFR(0xB2, 0x83)
‘T622/’T623/‘T326/’T327				WriteSFR(0xB2, 0x83)
‘T63x				WriteDirect(0xB2, 0x83)
EFM8BB1			WriteSFR(0xFF, 0x80) Delay 5 µs WriteSFR(0xEF, 0x02)	WriteSFR(0xA9, 0x00)
EFM8BB2			WriteSFR(0xFF, 0x80) Delay 5 µs WriteSFR(0xEF, 0x02)	WriteSFR(0xA9, 0x00)
EFM8BB3			WriteSFR(0xFF, 0x80) Delay 5 µs WriteSFR(0xEF, 0x02)	WriteSFR(0xA9, 0x00)
EFM8LB1			WriteSFR(0xFF, 0x80) Delay 5 µs WriteSFR(0xEF, 0x02)	WriteSFR(0xA9, 0x00)
EFM8SB1	WriteDirect(0xB6, 0x40)		WriteDirect(0xFF, 0x80) WriteDirect(0xEF, 0x02)	WriteDirect(0xA9, 0x04)
EFM8SB2				WriteDirect(0xA7, 0x00) WriteDirect(0xB2, 0x8F) WriteDirect(0xA9, 0x00)
EFM8UB1			WriteSFR(0xFF, 0x80) Delay 5 µs WriteSFR(0xEF, 0x02)	WriteSFR(0xA9, 0x00)
EFM8UB2	WriteSFR(0xB6, 0x90)		WriteSFR(0xFF, 0x80) WriteSFR(0xEF, 0x02)	WriteSFR(0xA9, 0x03)

4. Software Example

The software example included with this application note is written for a C8051F38x device acting as the programmer. The software can be ported to any other Silicon Labs C8051Fxxx device with some modification. The interconnection diagram used with the example software is shown in the figure below. This connection diagram assumes that the C2CK and/or C2D pins on the target device are not used by the target application.

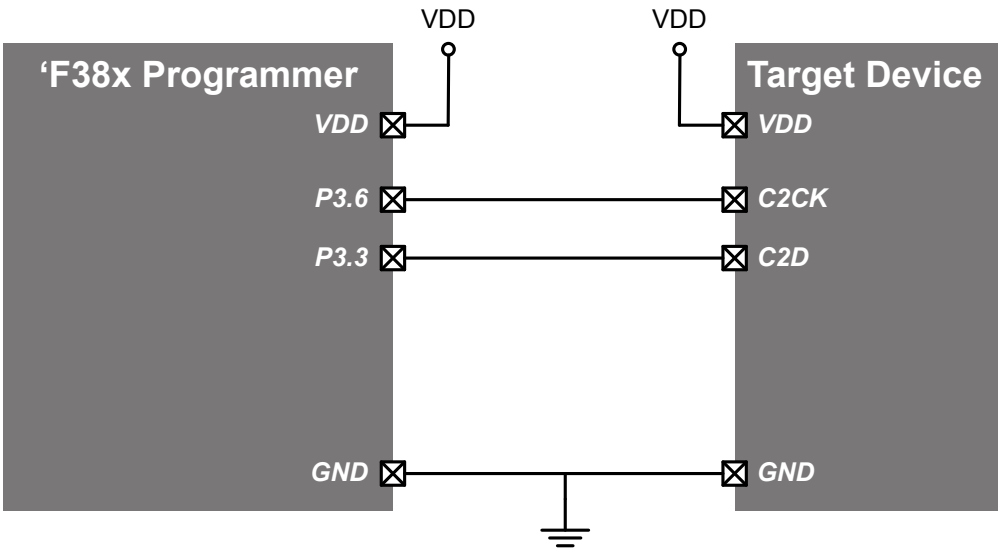


Figure 4.1. C2 Programmer Connection Diagram

If the C2 programming is to be performed on a target device installed in the user application, C2 isolation circuitry may be necessary. See application note "AN124: Pin Sharing Techniques for the C2 Interface" on the Silicon Labs website: http://www.silabs.com.

Table 4.1. C8051F38x Programming Pins

USB Debug Adapter Connector Pin	Signal	C8051F38x (Target Board) Pin
1	+3 V	P3.0 (set high)
2	GND	P3.1 (set low)
3	GND	P3.2 (set low)
4	C2D	P3.3
5	RST pin sharing	P3.4
6	C2D pin sharing	P3.5
7	C2CK	P3.6
8	VPP	P3.7
9	GND	N/A (must cut the TB stake header)
10	VBUS	no connect

4.1 Common Functions

This section outlines common steps grouped into functions in the firmware device programmer example.

4.1.1 WriteSFR

The `WriteSFR(addr,data)` function directly writes an SFR and consists of the following steps:

1. `AddressWrite(addr)`
2. `DataWrite(data)`

4.1.2 ReadSFR

The `ReadSFR(addr)` function directly reads an SFR and consists of the following steps:

1. `AddressWrite(addr)`
2. return `DataRead()`

4.1.3 WriteCommand

The `WriteCommand(command)` function writes a command to the PI. It consists of the following steps:

1. `DataWrite(command)`
2. `Poll_InBusy()`

4.1.4 ReadData

The `ReadData()` function reads data from the PI and has the following steps:

1. `Poll_OutReady()`
2. return `DataRead()`

4.1.5 WriteDirect

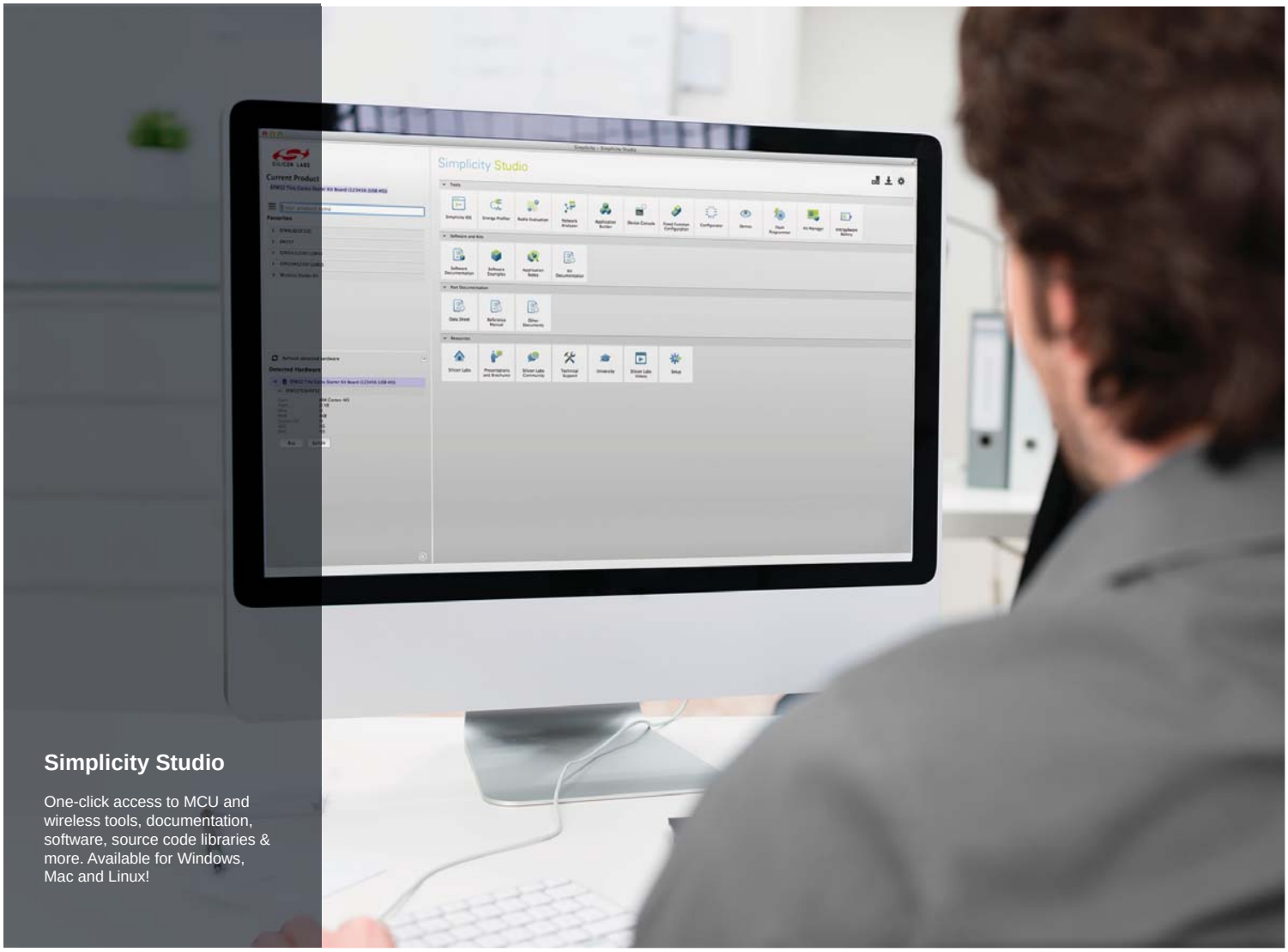
The `WriteDirect(addr,data)` function allows writes to SFRs on devices that have SFR Paging and ensures that the SFR page is managed correctly.

1. `AddressWrite(FPDAT)`
2. `WriteCommand(0x0A) // Direct write`
3. `ReadData() // 0x0D indicates success, all other return values are errors`
4. `WriteCommand(addr)`
5. `WriteCommand(0x01)`
6. `WriteCommand(data)`

4.1.6 ReadDirect

The `ReadDirect(addr)` function allows reads from SFRs on devices that have SFR Paging and ensures that the SFR page is managed correctly.

1. `AddressWrite(FPDAT)`
2. `WriteCommand(0x09) // Direct read`
3. `ReadData() // 0x0D indicates success, all other return values are errors`
4. `WriteCommand(addr)`
5. `WriteCommand(0x01)`
6. return `ReadData()`



Simplicity Studio

One-click access to MCU and wireless tools, documentation, software, source code libraries & more. Available for Windows, Mac and Linux!

IoT Portfolio
www.silabs.com/IoT

SW/HW
www.silabs.com/simplicity

Quality
www.silabs.com/quality

Support and Community
community.silabs.com

SILICON LABS

Silicon Laboratories Inc.
400 West Cesar Chavez
Austin, TX 78701
USA

http://www.silabs.com

Anexo V

DataSheetLPC2103

LPC2101/02/03

Single-chip 16-bit/32-bit microcontrollers; 8 kB/16 kB/32 kB flash with ISP/IAP, fast ports and 10-bit ADC

Rev. 04 — 2 June 2009 Product data sheet

1. General description

The LPC2101/02/03 microcontrollers are based on a 16-bit/32-bit ARM7TDMI-S CPU with real-time emulation that combines the microcontroller with 8 kB, 16 kB or 32 kB of embedded high-speed flash memory. A 128-bit wide memory interface and a unique accelerator architecture enable 32-bit code execution at the maximum clock rate. For critical performance in interrupt service routines and DSP algorithms, this increases performance up to 30 % over Thumb mode. For critical code size applications, the alternative 16-bit Thumb mode reduces code by more than 30 % with minimal performance penalty.

Due to their tiny size and low power consumption, the LPC2101/02/03 are ideal for applications where miniaturization is a key requirement. A blend of serial communications interfaces ranging from multiple UARTs, SPI to SSP and two I^2C-buses, combined with on-chip SRAM of 2 kB/4 kB/8 kB, make these devices very well suited for communication gateways and protocol converters. The superior performance also makes these devices suitable for use as math coprocessors. Various 32-bit and 16-bit timers, an improved 10-bit ADC, PWM features through output match on all timers, and 32 fast GPIO lines with up to nine edge or level sensitive external interrupt pins make these microcontrollers particularly suitable for industrial control and medical systems.

2. Features

2.1 Enhanced features

Enhanced features are available in parts LPC2101/02/03 labelled Revision A and higher:

- Deep power-down mode with option to retain SRAM memory and/or RTC.
- Three levels of flash Code Read Protection (CRP) implemented.

2.2 Key features

- 16-bit/32-bit ARM7TDMI-S microcontroller in tiny LQFP48 and HVQFN48 packages.
- 2 kB/4 kB/8 kB of on-chip static RAM and 8 kB/16 kB/32 kB of on-chip flash program memory. 128-bit wide interface/accelerator enables high-speed 70 MHz operation.
- ISP/IAP via on-chip bootloader software. Single flash sector or full chip erase in 100 ms and programming of 256 bytes in 1 ms.
- EmbeddedICE-RT offers real-time debugging with the on-chip RealMonitor software.
- The 10-bit ADC provides eight analog inputs, with conversion times as low as 2.44 µs per channel and dedicated result registers to minimize interrupt overhead.
- Two 32-bit timers/external event counters with combined seven capture and seven compare channels.

- Two 16-bit timers/external event counters with combined three capture and seven compare channels.
- Low power Real-Time Clock (RTC) with independent power and dedicated 32 kHz clock input.
- Multiple serial interfaces including two UARTs (16C550), two Fast I^2C-buses (400 kbit/s), SPI and SSP with buffering and variable data length capabilities.
- Vectored interrupt controller with configurable priorities and vector addresses.
- Up to thirty-two, 5 V tolerant fast general purpose I/O pins.
- Up to 13 edge or level sensitive external interrupt pins available.
- 70 MHz maximum CPU clock available from programmable on-chip PLL with a possible input frequency of 10 MHz to 25 MHz and a settling time of 100 µs.
- On-chip integrated oscillator operates with an external crystal in the range from 1 MHz to 25 MHz.
- Power saving modes include Idle mode, Power-down mode with RTC active, and Power-down mode.
- Individual enable/disable of peripheral functions as well as peripheral clock scaling for additional power optimization.
- Processor wake-up from Power-down and Deep power-down (Revision A and higher) mode via external interrupt or RTC.

3. Ordering information

Table 1. Ordering information

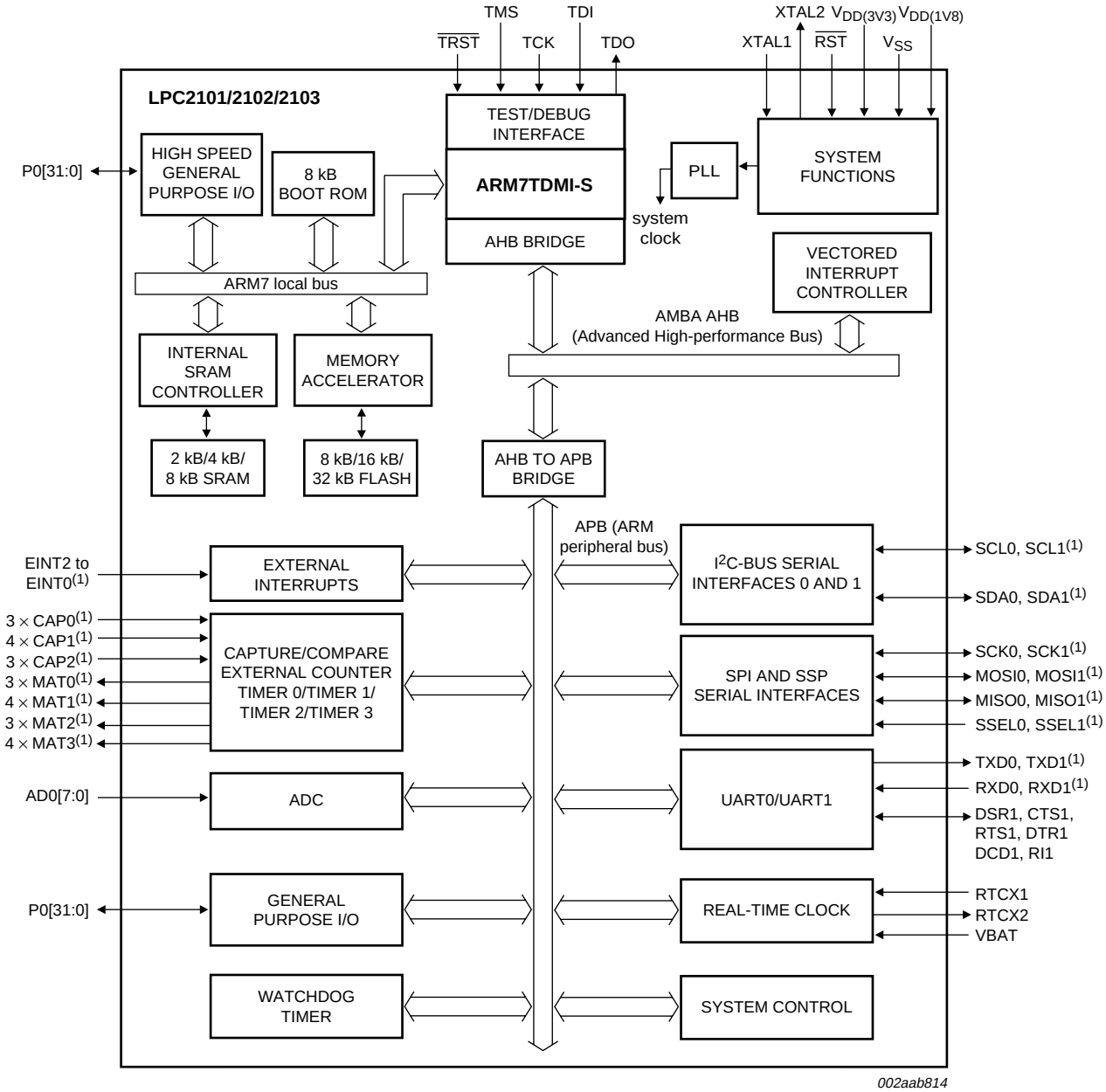
Type number	Package		
	Name	Description	Version
LPC2101FBD48	LQFP48	plastic low profile quad flat package; 48 leads; body 7 × 7 × 1.4 mm	SOT313-2
LPC2102FBD48	LQFP48	plastic low profile quad flat package; 48 leads; body 7 × 7 × 1.4 mm	SOT313-2
LPC2103FBD48	LQFP48	plastic low profile quad flat package; 48 leads; body 7 × 7 × 1.4 mm	SOT313-2
LPC2102FHN48	HVQFN48	plastic thermal enhanced very thin quad flat package; no leads; 48 terminals; body 7 × 7 × 0.85 mm	SOT619-7
LPC2103FHN48	HVQFN48	plastic thermal enhanced very thin quad flat package; no leads; 48 terminals; body 7 × 7 × 0.85 mm	SOT619-7
LPC2103FHN48H	HVQFN48	plastic thermal enhanced very thin quad flat package; no leads; 48 terminals; body 6 × 6 × 0.85 mm	SOT778-3

3.1 Ordering options

Table 2. Ordering options

Type number	Flash memory	RAM	ADC	Temperature range (°C)
LPC2101FBD48	8 kB	2 kB	8 inputs	–40 to +85
LPC2102FBD48	16 kB	4 kB	8 inputs	–40 to +85
LPC2103FBD48	32 kB	8 kB	8 inputs	–40 to +85
LPC2102FHN48	16 kB	4 kB	8 inputs	–40 to +85
LPC2103FHN48	32 kB	8 kB	8 inputs	–40 to +85
LPC2103FHN48H	32 kB	8 kB	8 inputs	–40 to +85

4. Block diagram



(1) Pins shared with GPIO.

Fig 1. Block diagram

5. Pinning information

5.1 Pinning

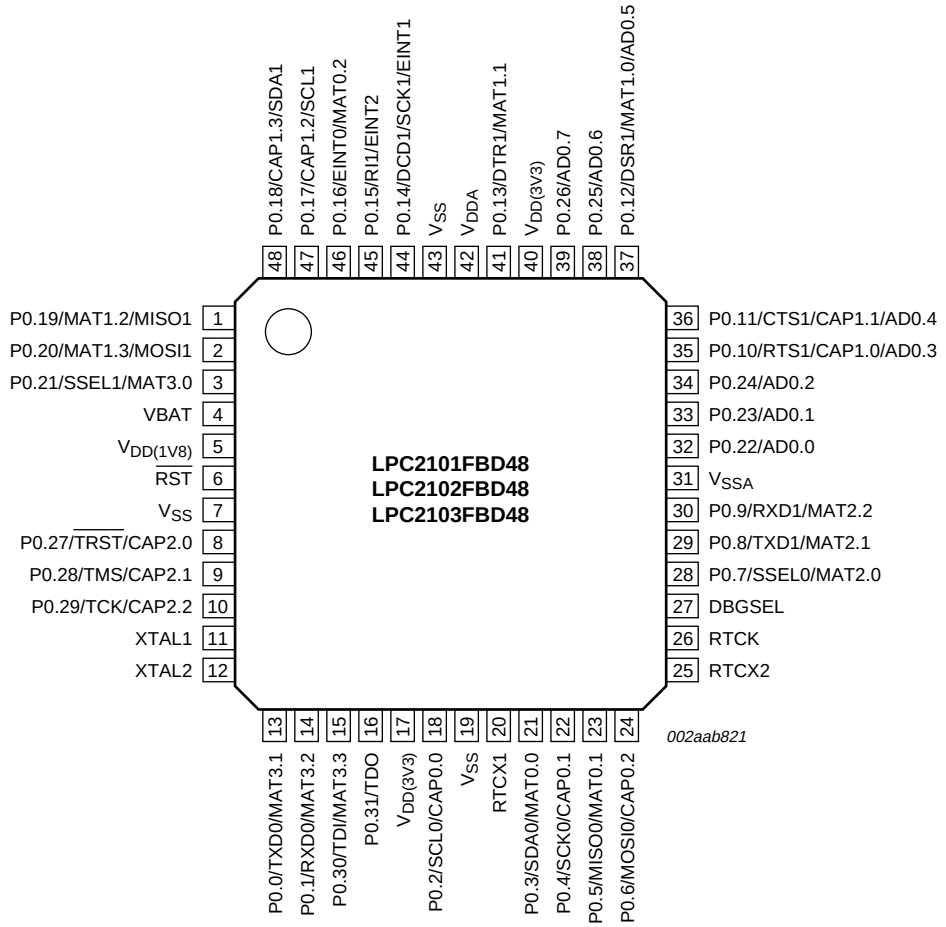


Fig 2. Pin configuration (LQFP48)

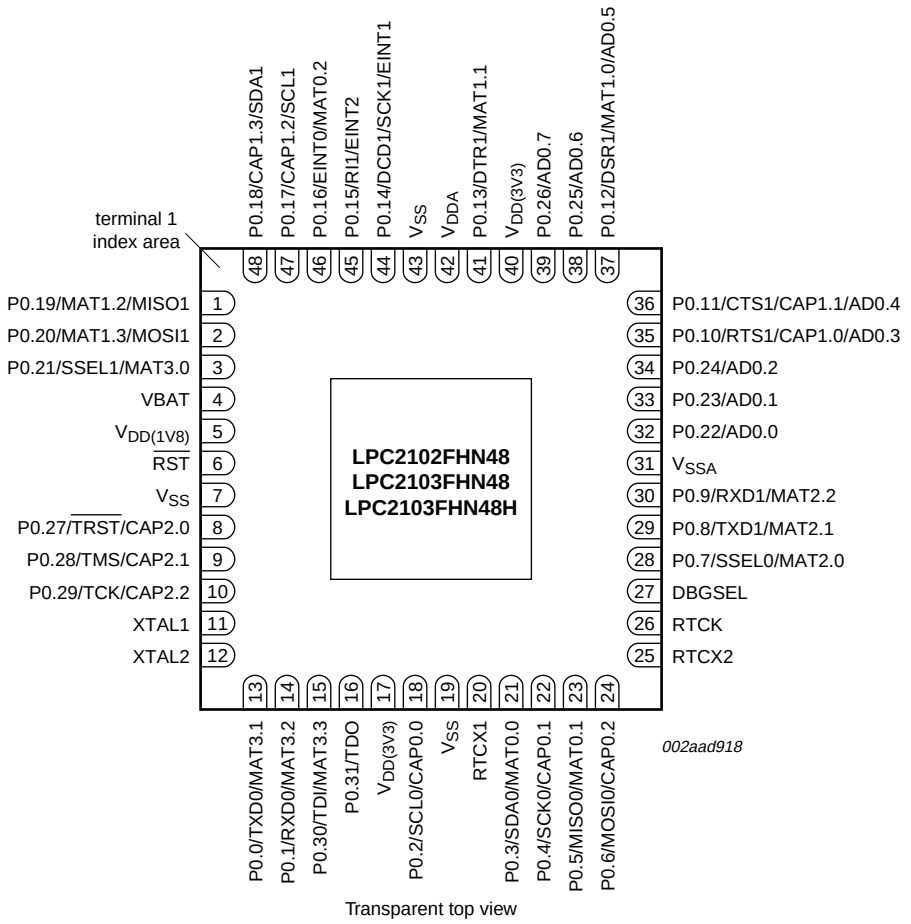


Fig 3. Pin configuration (HVQFN48)

5.2 Pin description

Table 3. Pin description

Symbol	Pin	Type	Description
P0.0 to P0.31		I/O	**Port 0:** Port 0 is a 32-bit I/O port with individual direction controls for each bit. A total of 31 pins of the Port 0 can be used as general purpose bidirectional digital I/Os while P0.31 is an output only pin. The operation of port 0 pins depends upon the pin function selected via the pin connect block.
P0.0/TXD0/ MAT3.1	13[1]	I/O	**P0.0 —** General purpose input/output digital pin.
		O	**TXD0 —** Transmitter output for UART0.
		O	**MAT3.1 —** PWM output 1 for Timer 3.
P0.1/RXD0/ MAT3.2	14[1]	I/O	**P0.1 —** General purpose input/output digital pin.
		I	**RXD0 —** Receiver input for UART0.
		O	**MAT3.2 —** PWM output 2 for Timer 3.
P0.2/SCL0/ CAP0.0	18[2]	I/O	**P0.2 —** General purpose input/output digital pin. Output is open-drain.
		I/O	**SCL0 —** I^2C0 clock Input/output. Open-drain output (for I^2C-bus compliance).
		I	**CAP0.0 —** Capture input for Timer 0, channel 0.
P0.3/SDA0/ MAT0.0	21[2]	I/O	**P0.3 —** General purpose input/output digital pin. Output is open-drain.
		I/O	**SDA0 —** I^2C0 data input/output. Open-drain output (for I^2C-bus compliance).
		O	**MAT0.0 —** PWM output for Timer 0, channel 0. Output is open-drain.
P0.4/SCK0/ CAP0.1	22[1]	I/O	**P0.4 —** General purpose input/output digital pin.
		I/O	**SCK0 —** Serial clock for SPI0. SPI clock output from master or input to slave.
		I	**CAP0.1 —** Capture input for Timer 0, channel 1.
P0.5/MISO0/ MAT0.1	23[1]	I/O	**P0.5 —** General purpose input/output digital pin.
		I/O	**MISO0 —** Master In Slave Out for SPI0. Data input to SPI master or data output from SPI slave.
		O	**MAT0.1 —** PWM output for Timer 0, channel 1.
P0.6/MOSI0/ CAP0.2	24[1]	I/O	**P0.6 —** General purpose input/output digital pin.
		I/O	**MOSI0 —** Master Out Slave In for SPI0. Data output from SPI master or data input to SPI slave.
		I	**CAP0.2 —** Capture input for Timer 0, channel 2.
P0.7/SSEL0/ MAT2.0	28[1]	I/O	**P0.7 —** General purpose input/output digital pin.
		I	**SSEL0 —** Slave Select for SPI0. Selects the SPI interface as a slave.
		O	**MAT2.0 —** PWM output for Timer 2, channel 0.
P0.8/TXD1/ MAT2.1	29[1]	I/O	**P0.8 —** General purpose input/output digital pin.
		O	**TXD1 —** Transmitter output for UART1.
		O	**MAT2.1 —** PWM output for Timer 2, channel 1.
P0.9/RXD1/ MAT2.2	30[1]	I/O	**P0.9 —** General purpose input/output digital pin.
		I	**RXD1 —** Receiver input for UART1.
		O	**MAT2.2 —** PWM output for Timer 2, channel 2.
P0.10/RTS1/ CAP1.0/AD0.3	35[3]	I/O	**P0.10 —** General purpose input/output digital pin.
		O	**RTS1 —** Request to Send output for UART1.
		I	**CAP1.0 —** Capture input for Timer 1, channel 0.
		I	**AD0.3 —** ADC 0, input 3.

Table 3. Pin description *...continued*

Symbol	Pin	Type	Description
P0.11/CTS1/ CAP1.1/AD0.4	36[3]	I/O	**P0.11 —** General purpose input/output digital pin.
		I	**CTS1 —** Clear to Send input for UART1.
		I	**CAP1.1 —** Capture input for Timer 1, channel 1.
		I	**AD0.4 —** ADC 0, input 4.
P0.12/DSR1/ MAT1.0/AD0.5	37[3]	I/O	**P0.12 —** General purpose input/output digital pin.
		I	**DSR1 —** Data Set Ready input for UART1.
		O	**MAT1.0 —** PWM output for Timer 1, channel 0.
		I	**AD0.5 —** ADC 0, input 5.
P0.13/DTR1/ MAT1.1	41[1]	I/O	**P0.13 —** General purpose input/output digital pin.
		O	**DTR1 —** Data Terminal Ready output for UART1.
		O	**MAT1.1 —** PWM output for Timer 1, channel 1.
P0.14/DCD1/ SCK1/EINT1	44[4][5]	I/O	**P0.14 —** General purpose input/output digital pin.
		I	**DCD1 —** Data Carrier Detect input for UART1.
		I/O	**SCK1 —** Serial Clock for SPI1. SPI clock output from master or input to slave.
		I	**EINT1 —** External interrupt 1 input.
P0.15/RI1/ EINT2	45[4]	I/O	**P0.15 —** General purpose input/output digital pin.
		I	**RI1 —** Ring Indicator input for UART1.
		I	**EINT2 —** External interrupt 2 input.
P0.16/EINT0/ MAT0.2	46[4]	I/O	**P0.16 —** General purpose input/output digital pin.
		I	**EINT0 —** External interrupt 0 input.
		O	**MAT0.2 —** PWM output for Timer 0, channel 2.
P0.17/CAP1.2/ SCL1	47[6]	I/O	**P0.17 —** General purpose input/output digital pin. The output is not open-drain.
		I	**CAP1.2 —** Capture input for Timer 1, channel 2.
		I/O	**SCL1 —** I^2C1 clock Input/output. This pin is an open-drain output if I^2C1 function is selected in the pin connect block.
P0.18/CAP1.3/ SDA1	48[6]	I/O	**P0.18 —** General purpose input/output digital pin. The output is not open-drain.
		I	**CAP1.3 —** Capture input for Timer 1, channel 3.
		I/O	**SDA1 —** I^2C1 data Input/output. This pin is an open-drain output if I^2C1 function is selected in the pin connect block.
P0.19/MAT1.2/ MISO1	1[1]	I/O	**P0.19 —** General purpose input/output digital pin.
		O	**MAT1.2 —** PWM output for Timer 1, channel 2.
		I/O	**MISO1 —** Master In Slave Out for SSP. Data input to SSP master or data output from SSP slave.
P0.20/MAT1.3/ MOSI1	2[1]	I/O	**P0.20 —** General purpose input/output digital pin.
		O	**MAT1.3 —** PWM output for Timer 1, channel 3.
		I/O	**MOSI1 —** Master Out Slave for SSP. Data output from SSP master or data input to SSP slave.
P0.21/SSEL1/ MAT3.0	3[1]	I/O	**P0.21 —** General purpose input/output digital pin.
		I	**SSEL1 —** Slave Select for SPI1. Selects the SPI interface as a slave.
		O	**MAT3.0 —** PWM output for Timer 3, channel 0.

Table 3. Pin description *...continued*

Symbol	Pin	Type	Description
P0.22/AD0.0	32[3]	I/O	**P0.22 —** General purpose input/output digital pin.
		I	**AD0.0 —** ADC 0, input 0.
P0.23/AD0.1	33[3]	I/O	**P0.23 —** General purpose input/output digital pin.
		I	**AD0.1 —** ADC 0, input 1.
P0.24/AD0.2	34[3]	I/O	**P0.24 —** General purpose input/output digital pin.
		I	**AD0.2 —** ADC 0, input 2.
P0.25/AD0.6	38[3]	I/O	**P0.25 —** General purpose input/output digital pin.
		I	**AD0.6 —** ADC 0, input 6.
P0.26/AD0.7	39[3]	I/O	**P0.26 —** General purpose input/output digital pin.
		I	**AD0.7 —** ADC 0, input 7.
P0.27/$\overline{\text{TRST}}$/ CAP2.0	8[1]	I/O	**P0.27 —** General purpose input/output digital pin.
		I	**$\overline{\text{TRST}}$ —** Test Reset for JTAG interface. If DBGSEL is HIGH, this pin is automatically configured for use with EmbeddedICE (Debug mode).
		I	**CAP2.0 —** Capture input for Timer 2, channel 0.
P0.28/TMS/ CAP2.1	9[1]	I/O	**P0.28 —** General purpose input/output digital pin.
		I	**TMS —** Test Mode Select for JTAG interface. If DBGSEL is HIGH, this pin is automatically configured for use with EmbeddedICE (Debug mode).
		I	**CAP2.1 —** Capture input for Timer 2, channel 1.
P0.29/TCK/ CAP2.2	10[1]	I/O	**P0.29 —** General purpose input/output digital pin.
		I	**TCK —** Test Clock for JTAG interface. This clock must be slower than $\frac{1}{6}$ of the CPU clock (CCLK) for the JTAG interface to operate. If DBGSEL is HIGH, this pin is automatically configured for use with EmbeddedICE (Debug mode).
		I	**CAP2.2 —** Capture input for Timer 2, channel 2.
P0.30/TDI/ MAT3.3	15[1]	I/O	**P0.30 —** General purpose input/output digital pin.
		I	**TDI —** Test Data In for JTAG interface. If DBGSEL is HIGH, this pin is automatically configured for use with EmbeddedICE (Debug mode).
		O	**MAT3.3 —** PWM output 3 for Timer 3.
P0.31/TDO	16[1]	O	**P0.31 —** General purpose output only digital pin.
		O	**TDO —** Test Data Out for JTAG interface. If DBGSEL is HIGH, this pin is automatically configured for use with EmbeddedICE (Debug mode).
RTCX1	20[7][8]	I	Input to the RTC oscillator circuit. Input voltage must not exceed 1.8 V.
RTCX2	25[7][8]	O	Output from the RTC oscillator circuit.
RTCK	26[7]	I/O	**Returned test clock output:** Extra signal added to the JTAG port. Assists debugger synchronization when processor frequency varies. Bidirectional pin with internal pull-up.
XTAL1	11	I	Input to the oscillator circuit and internal clock generator circuits. Input voltage must not exceed 1.8 V.
XTAL2	12	O	Output from the oscillator amplifier.
DBGSEL	27	I	**Debug select:** When LOW, the part operates normally. When externally pulled HIGH at reset, P0.27 to P0.31 are configured as JTAG port, and the part is in Debug mode[9]. Input with internal pull-down.
$\overline{\text{RST}}$	6	I	**External reset input:** A LOW on this pin resets the device, causing I/O ports and peripherals to take on their default states and processor execution to begin at address 0. TTL with hysteresis, 5 V tolerant.

Table 3. Pin description *…continued*

Symbol	Pin	Type	Description
V_{SS}	7, 19, 43	I	**Ground:** 0 V reference.
V_{SSA}	31	I	**Analog ground:** 0 V reference. This should be nominally the same voltage as V_{SS} but should be isolated to minimize noise and error.
V_{DDA}	42	I	**Analog 3.3 V power supply:** This should be nominally the same voltage as $V_{DD(3V3)}$ but should be isolated to minimize noise and error. The level on this pin also provides a voltage reference level for the ADC.
$V_{DD(1V8)}$	5	I	**1.8 V core power supply:** This is the power supply voltage for internal circuitry and the on-chip PLL.
$V_{DD(3V3)}$	17, 40	I	**3.3 V pad power supply:** This is the power supply voltage for the I/O ports.
VBAT	4	I	**RTC power supply:** 3.3 V on this pin supplies the power to the RTC.

[1] 5 V tolerant (if $V_{DD(3V3)}$ and $V_{DDA} \geq 3.0$ V) pad providing digital I/O functions with TTL levels and hysteresis and 10 ns slew rate control.

[2] Open-drain 5 V tolerant (if $V_{DD(3V3)}$ and $V_{DDA} \geq 3.0$ V) digital I/O I^2C-bus 400 kHz specification compatible pad. It requires external pull-up to provide an output functionality. Open-drain configuration applies to ALL functions on that pin.

[3] 5 V tolerant (if $V_{DD(3V3)}$ and $V_{DDA} \geq 3.0$ V) pad providing digital I/O (with TTL levels and hysteresis and 10 ns slew rate control) and analog input function. If configured for an input function, this pad utilizes built-in glitch filter that blocks pulses shorter than 3 ns. When configured as an ADC input, digital section of the pad is disabled.

[4] 5 V tolerant (if $V_{DD(3V3)}$ and $V_{DDA} \geq 3.0$ V) pad providing digital I/O functions with TTL levels and hysteresis and 10 ns slew rate control. If configured for an input function, this pad utilizes built-in glitch filter that blocks pulses shorter than 3 ns.

[5] A LOW level during reset on pin P0.14 is considered as an external hardware request to start the ISP command handler.

[6] Open-drain 5 V tolerant (if $V_{DD(3V3)}$ and $V_{DDA} \geq 3.0$ V) digital I/O I^2C-bus 400 kHz specification compatible pad. It requires external pull-up to provide an output functionality. Open-drain configuration applies only to I^2C function on that pin.

[7] Pad provides special analog functionality.

[8] For lowest power consumption, pin should be left floating when the RTC is not used.

[9] See *LPC2101/02/03 User manual UM10161* for details.

6. Functional description

6.1 Architectural overview

The ARM7TDMI-S is a general purpose 32-bit microprocessor, which offers high performance and very low power consumption. The ARM architecture is based on Reduced Instruction Set Computer (RISC) principles, and the instruction set and related decode mechanism are much simpler than those of microprogrammed Complex Instruction Set Computers (CISC). This simplicity results in a high instruction throughput and impressive real-time interrupt response from a small and cost-effective processor core.

Pipeline techniques are employed so that all parts of the processing and memory systems can operate continuously. Typically, while one instruction is being executed, its successor is being decoded, and a third instruction is being fetched from memory.

The ARM7TDMI-S processor also employs a unique architectural strategy known as Thumb, which makes it ideally suited to high-volume applications with memory restrictions, or applications where code density is an issue.

The key idea behind Thumb is that of a super-reduced instruction set. Essentially, the ARM7TDMI-S processor has two instruction sets:

- The standard 32-bit ARM set.
- A 16-bit Thumb set.

The Thumb set's 16-bit instruction length allows it to approach twice the density of standard ARM code while retaining most of the ARM's performance advantage over a traditional 16-bit processor using 16-bit registers. This is possible because Thumb code operates on the same 32-bit register set as ARM code.

Thumb code is able to provide up to 65 % of the code size of ARM, and 160 % of the performance of an equivalent ARM processor connected to a 16-bit memory system.

The particular flash implementation in the LPC2101/02/03 allows for full speed execution also in ARM mode. It is recommended to program performance critical and short code sections in ARM mode. The impact on the overall code size will be minimal but the speed can be increased by 30 % over Thumb mode.

6.2 On-chip flash program memory

The LPC2101/02/03 incorporate a 8 kB, 16 kB or 32 kB flash memory system respectively. This memory may be used for both code and data storage. Programming of the flash memory may be accomplished in several ways. It may be programmed in system via the serial port. The application program may also erase and/or program the flash while the application is running, allowing a great degree of flexibility for data storage field firmware upgrades, etc. The entire flash memory is available for user code as the bootloader resides in a separate memory.

The LPC2101/02/03 flash memory provides a minimum of 100,000 erase/write cycles and 20 years of data-retention memory.

6.3 On-chip static RAM

On-chip static RAM may be used for code and/or data storage. The SRAM may be accessed as 8-bits, 16-bits, and 32-bits. The LPC2101/02/03 provide 2 kB, 4 kB or 8 kB of static RAM.

6.4 Memory map

The LPC2101/02/03 memory map incorporates several distinct regions, as shown in Figure 4.

In addition, the CPU interrupt vectors may be re-mapped to allow them to reside in either flash memory (the default) or on-chip static RAM. This is described in Section 6.17 "System control".

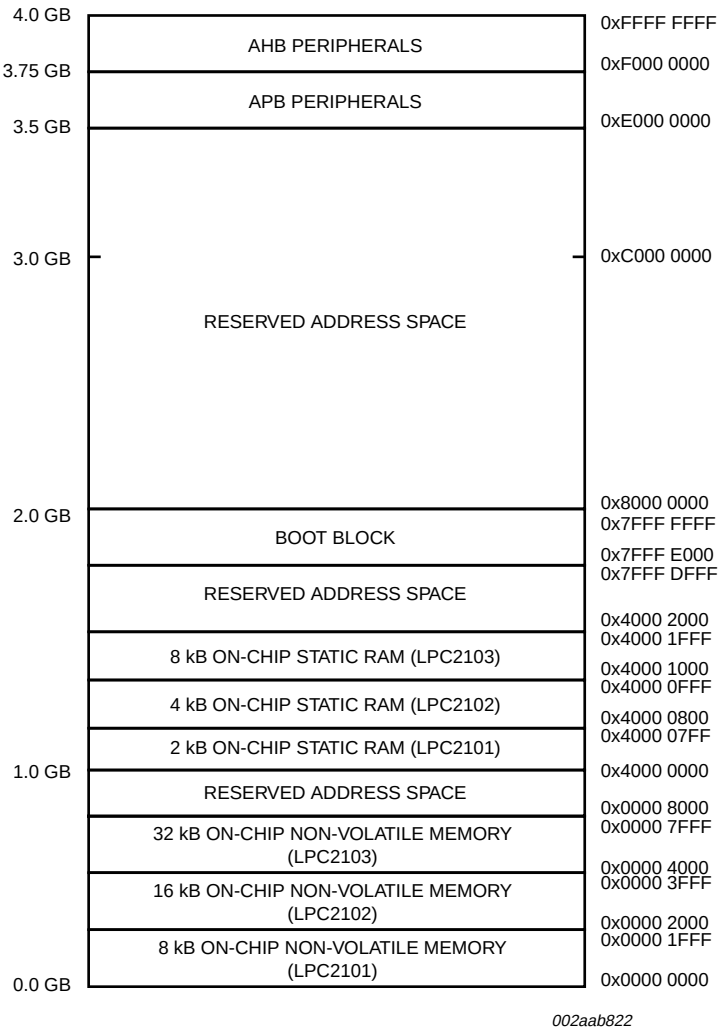


Fig 4. LPC2101/02/03 memory map

6.5 Interrupt controller

The VIC accepts all of the interrupt request inputs and categorizes them as FIQ, vectored IRQ, and non-vectored IRQ as defined by programmable settings. The programmable assignment scheme means that priorities of interrupts from the various peripherals can be dynamically assigned and adjusted.

FIQ has the highest priority. If more than one request is assigned to FIQ, the VIC combines the requests to produce the FIQ signal to the ARM processor. The fastest possible FIQ latency is achieved when only one request is classified as FIQ, because then the FIQ service routine does not need to branch into the interrupt service routine but can run from the interrupt vector location. If more than one request is assigned to the FIQ class, the FIQ service routine will read a word from the VIC that identifies which FIQ source(s) is (are) requesting an interrupt.

Vectored IRQs have the middle priority. Sixteen of the interrupt requests can be assigned to this category. Any of the interrupt requests can be assigned to any of the 16 vectored IRQ slots, among which slot 0 has the highest priority and slot 15 has the lowest.

Non-vectored IRQs have the lowest priority.

The VIC combines the requests from all the vectored and non-vectored IRQs to produce the IRQ signal to the ARM processor. The IRQ service routine can start by reading a register from the VIC and jumping there. If any of the vectored IRQs are pending, the VIC provides the address of the highest-priority requesting IRQs service routine, otherwise it provides the address of a default routine that is shared by all the non-vectored IRQs. The default routine can read another VIC register to see what IRQs are active.

6.5.1 Interrupt sources

Each peripheral device has one interrupt line connected to the Vectored Interrupt Controller, but may have several internal interrupt flags. Individual interrupt flags may also represent more than one interrupt source.

6.6 Pin connect block

The pin connect block allows selected pins of the microcontroller to have more than one function. Configuration registers control the multiplexers to allow connection between the pin and the on chip peripherals. Peripherals should be connected to the appropriate pins prior to being activated, and prior to any related interrupt(s) being enabled. Activity of any enabled peripheral function that is not mapped to a related pin should be considered undefined.

The pin control module with its pin select registers defines the functionality of the microcontroller in a given hardware environment.

After reset all pins of Port 0 are configured as input with the following exceptions: If the DBGSEL pin is HIGH (Debug mode enabled), the JTAG pins will assume their JTAG functionality for use with EmbeddedICE and cannot be configured via the pin connect block.

6.7 Fast general purpose parallel I/O

Device pins that are not connected to a specific peripheral function are controlled by the GPIO registers. Pins may be dynamically configured as inputs or outputs. Separate registers allow setting or clearing any number of outputs simultaneously. The value of the output register may be read back, as well as the current state of the port pins.

LPC2101/02/03 introduce accelerated GPIO functions over prior LPC2000 devices:

- GPIO registers are relocated to the ARM local bus for the fastest possible I/O timing.
- Mask registers allow treating sets of port bits as a group, leaving other bits unchanged.
- All GPIO registers are byte addressable.
- Entire port value can be written in one instruction.

6.7.1 Features

- Bit-level set and clear registers allow a single instruction set or clear of any number of bits in one port.
- Direction control of individual bits.
- Separate control of output set and clear.
- All I/O default to inputs after reset.

6.8 10-bit ADC

The LPC2101/02/03 contain one ADC. It is a single 10-bit successive approximation ADC with eight channels.

6.8.1 Features

- Measurement range of 0 V to 3.3 V.
- Each converter capable of performing more than 400,000 10-bit samples per second.
- Burst conversion mode for single or multiple inputs.
- Optional conversion on transition on input pin or Timer Match signal.
- Every analog input has a dedicated result register to reduce interrupt overhead.

6.9 UARTs

The LPC2101/02/03 each contain two UARTs. In addition to standard transmit and receive data lines, UART1 also provides a full modem control handshake interface.

Compared to previous LPC2000 microcontrollers, UARTs in LPC2101/02/03 include a fractional baud rate generator for both UARTs. Standard baud rates such as 115200 can be achieved with any crystal frequency above 2 MHz.

6.9.1 Features

- 16 byte Receive and Transmit FIFOs.
- Register locations conform to 16C550 industry standard.
- Receiver FIFO trigger points at 1, 4, 8, and 14 bytes

Anexo VI

SFRs C8051F4xxx

11.6. Special Function Registers

The direct-access data memory locations from 0x80 to 0xFF constitute the special function registers (SFRs). The SFRs provide control and data exchange with the CIP-51's resources and peripherals. The CIP-51 duplicates the SFRs found in a typical 8051 implementation as well as implementing additional SFRs used to configure and access the sub-systems unique to the MCU. This allows the addition of new functionality while retaining compatibility with the MCS-51™ instruction set. Table 11.1 lists the SFRs implemented in the CIP-51 System Controller.

The SFR registers are accessed anytime the direct addressing mode is used to access memory locations from 0x80 to 0xFF. SFRs with addresses ending in 0x0 or 0x8 (e.g. P0, TCON, IE, etc.) are bit-addressable as well as byte-addressable. All other SFRs are byte-addressable only. Unoccupied addresses in the SFR space are reserved for future use. Accessing these areas will have an indeterminate effect and should be avoided. Refer to the corresponding pages of the data sheet, as indicated in Table 11.2, for a detailed description of each register.

Table 11.1. Special Function Register (SFR) Memory Map

F8	SPI0CN	PCA0L	PCA0H	PCA0CPL0	PCA0CPH0	PCA0CPL4	PCA0CPH4	VDM0CN
F0	B	P0MDIN	P1MDIN	P2MDIN	IDA1L	IDA1H	EIP1	EIP2
E8	ADC0CN	PCA0CPL1	PCA0CPH1	PCA0CPL2	PCA0CPH2	PCA0CPL3	PCA0CPH3	RSTSRC
E0	ACC	XBR0	XBR1	PFE0CN	IT01CF		EIE1	EIE2
D8	PCA0CN	PCA0MD	PCA0CPM0	PCA0CPM1	PCA0CPM2	PCA0CPM3	PCA0CPM4	CRC0FLIP
D0	PSW	REF0CN	PCA0CPL5	PCA0CPH5	P0SKIP	P1SKIP	P2SKIP	P0MAT
C8	TMR2CN	REG0CN	TMR2RLL	TMR2RLH	TMR2L	TMR2H	PCA0CPM5	P1MAT
C0	SMB0CN	SMB0CF	SMB0DAT	ADC0GTL	ADC0GTH	ADC0LTL	ADC0LTH	P0MASK
B8	IP	IDA0CN	ADC0TK	ADC0MX	ADC0CF	ADC0L	ADC0H	P1MASK
B0	P0ODEN	OSCXCN	OSCICN	OSCICL		IDA1CN	FLSCL	FLKEY
A8	IE	CLKSEL	EMI0CN	CLKMUL	RTC0ADR	RTC0DAT	RTC0KEY	ONESHOT
A0	P2	SPI0CFG	SPI0CKR	SPI0DAT	P0MDOUT	P1MDOUT	P2MDOUT	
98	SCON0	SBUF0	CPT1CN	CPT0CN	CPT1MD	CPT0MD	CPT1MX	CPT0MX
90	P1	TMR3CN	TMR3RLL	TMR3RLH	TMR3L	TMR3H	IDA0L	IDA0H
88	TCON	TMOD	TL0	TL1	TH0	TH1	CKCON	PSCTL
80	P0	SP	DPL	DPH	CRC0CN	CRC0IN	CRC0DAT	PCON
	0(8) (bit addressable)	1(9)	2(A)	3(B)	4(C)	5(D)	6(E)	7(F)

Table 11.2. Special Function Registers

SFRs are listed in alphabetical order. All undefined SFR locations are reserved.

Register	Address	Description	Page
ACC	0xE0	Accumulator	101
ADC0CF	0xBC	ADC0 Configuration	60
ADC0CN	0xE8	ADC0 Control	62
ADC0H	0xBE	ADC0	61
ADC0L	0xBD	ADC0	61
ADC0GTH	0xC4	ADC0 Greater-Than Data High Byte	64
ADC0GTL	0xC3	ADC0 Greater-Than Data Low Byte	64
ADC0LTH	0xC6	ADC0 Less-Than Data High Byte	65
ADC0LTL	0xC5	ADC0 Less-Than Data Low Byte	65
ADC0MX	0xC6	ADC0 Channel Select	59
ADC0TK	0xBA	ADC0 Tracking Mode Select	63
B	0xF0	B Register	101
CKCON	0x8E	Clock Control	237
CLKMUL	0xAB	Clock Multiplier	173
CLKSEL	0xA9	Clock Select	174
CPT0CN	0x9B	Comparator0 Control	86
CPT0MD	0x9D	Comparator0 Mode Selection	88
CPT0MX	0x9F	Comparator0 MUX Selection	87
CPT1CN	0x9A	Comparator1 Control	91
CPT1MD	0x9C	Comparator1 Mode Selection	90
CPT1MX	0x9E	Comparator1 MUX Selection	89
CRC0CN	0x84	CRC0 Control	124
CRC0IN	0x85	CRC0 Data Input	124
CRC0DAT	0x86	CRC0 Data Output	125
CRC0FLIP	0xDF	CRC0 Bit Flip	125
DPH	0x83	Data Pointer High	99
DPL	0x82	Data Pointer Low	99
EIE1	0xE6	Extended Interrupt Enable 1	114
EIE2	0xE7	Extended Interrupt Enable 2	116
EIP1	0xF6	Extended Interrupt Priority 1	115
EIP2	0xF7	Extended Interrupt Priority 2	116
EMI0CN	0xAA	External Memory Interface Control	145
FLKEY	0xB7	Flash Lock and Key	141
FLSCL	0xB6	Flash Scale	142
IDA0H	0x97	Current Mode DAC0 High Byte	71
IDA0L	0x96	Current Mode DAC0 Low Byte	72
IDA0CN	0xB9	Current Mode DAC0 Control	71
IDA1H	0xF5	Current Mode DAC1 High Byte	73

Table 11.2. Special Function Registers (Continued)

SFRs are listed in alphabetical order. All undefined SFR locations are reserved.

Register	Address	Description	Page
IDA1L	0xF4	Current Mode DAC1 Low Byte	73
IDA1CN	0xB5	Current Mode DAC1 Control	72
IE	0xA8	Interrupt Enable	112
IP	0xB8	Interrupt Priority	113
IT01CF	0xE4	INT0/INT1 Configuration	118
ONESHOT	0xAF	Flash Oneshot Period	143
OSCICL	0xB3	Internal Oscillator Calibration	167
OSCICN	0xB2	Internal Oscillator Control	167
OSCXCN	0xB1	External Oscillator Control	171
P0	0x80	Port 0 Latch	155
P0MASK	0xC7	Port 0 Mask	157
P0MAT	0xD7	Port 0 Match	157
P0MDIN	0xF1	Port 0 Input Mode Configuration	155
P0MDOUT	0xA4	Port 0 Output Mode Configuration	156
P0ODEN	0xB0	Port 0 Overdrive	157
P0SKIP	0xD4	Port 0 Skip	156
P1	0x90	Port 1 Latch	158
P1MASK	0xBF	Port 1 Mask	160
P1MAT	0xCF	Port 1 Match	160
P1MDIN	0xF2	Port 1 Input Mode Configuration	158
P1MDOUT	0xA5	Port 1 Output Mode Configuration	159
P1SKIP	0xD5	Port 1 Skip	159
P2	0xA0	Port 2 Latch	161
P2MDIN	0xF3	Port 2 Input Mode Configuration	161
P2MDOUT	0xA6	Port 2 Output Mode Configuration	162
P2SKIP	0xD6	Port 2 Skip	162
PCA0CN	0xD8	PCA Control	261
PCA0CPH0	0xFC	PCA Capture 0 High	264
PCA0CPH1	0xEA	PCA Capture 1 High	264
PCA0CPH2	0xEC	PCA Capture 2 High	264
PCA0CPH3	0xEE	PCA Capture 3 High	264
PCA0CPH4	0xFE	PCA Capture 4 High	264
PCA0CPH5	0xD3	PCA Capture 5 High	264
PCA0CPL0	0xFB	PCA Capture 0 Low	264
PCA0CPL1	0xE9	PCA Capture 1 Low	264
PCA0CPL2	0xEB	PCA Capture 2 Low	264
PCA0CPL3	0xED	PCA Capture 3 Low	264
PCA0CPL4	0xFD	PCA Capture 4 Low	264

Table 11.2. Special Function Registers (Continued)

SFRs are listed in alphabetical order. All undefined SFR locations are reserved.

Register	Address	Description	Page
PCA0CPL5	0xD2	PCA Capture 5 Low	264
PCA0CPM0	0xDA	PCA Module 0 Mode	263
PCA0CPM1	0xDB	PCA Module 1 Mode	263
PCA0CPM2	0xDC	PCA Module 2 Mode	263
PCA0CPM3	0xDD	PCA Module 3 Mode	263
PCA0CPM4	0xDE	PCA Module 4 Mode	263
PCA0CPM5	0xCE	PCA Module 5 Mode	263
PCA0H	0xFA	PCA Counter High	264
PCA0L	0xF9	PCA Counter Low	264
PCA0MD	0xD9	PCA Mode	262
PCON	0x87	Power Control	102
PFE0CN	0xE3	Prefetch Engine Control	119
PSCTL	0x8F	Program Store R/W Control	141
PSW	0xD0	Program Status Word	100
REF0CN	0xD1	Voltage Reference Control	78
REG0CN	0xC9	Voltage Regulator Control	82
RTC0ADR	0xAC	smaRTClock Address	181
RTC0DAT	0xAD	smaRTClock Data	182
RTC0KEY	0xAE	smaRTClock Lock and Key	180
RSTSRC	0xEF	Reset Source Configuration/Status	133
SBUF0	0x99	UART0 Data Buffer	213
SCON0	0x98	UART0 Control	212
SMB0CF	0xC1	SMBus Configuration	197
SMB0CN	0xC0	SMBus Control	199
SMB0DAT	0xC2	SMBus Data	201
SP	0x81	Stack Pointer	98
SPI0CFG	0xA1	SPI Configuration	223
SPI0CKR	0xA2	SPI Clock Rate Control	225
SPI0CN	0xF8	SPI Control	224
SPI0DAT	0xA3	SPI Data	226
TCON	0x88	Timer/Counter Control	235
TH0	0x8C	Timer/Counter 0 High	238
TH1	0x8D	Timer/Counter 1 High	238
TL0	0x8A	Timer/Counter 0 Low	238
TL1	0x8B	Timer/Counter 1 Low	238
TMOD	0x89	Timer/Counter Mode	236
TMR2CN	0xC8	Timer/Counter 2 Control	242
TMR2H	0xCD	Timer/Counter 2 High	243

Table 11.2. Special Function Registers (Continued)

SFRs are listed in alphabetical order. All undefined SFR locations are reserved.

Register	Address	Description	Page
TMR2L	0xCC	Timer/Counter 2 Low	243
TMR2RLH	0xCB	Timer/Counter 2 Reload High	243
TMR2RLL	0xCA	Timer/Counter 2 Reload Low	243
TMR3CN	0x91	Timer/Counter 3Control	247
TMR3H	0x95	Timer/Counter 3 High	248
TMR3L	0x94	Timer/Counter 3 Low	248
TMR3RLH	0x93	Timer/Counter 3 Reload High	248
TMR3RLL	0x92	Timer/Counter 3 Reload Low	248
VDM0CN	0xFF	V_{DD} Monitor Control	130
XBR0	0xE1	Port I/O Crossbar Control 0	153
XBR1	0xE2	Port I/O Crossbar Control 1	154

Anexo VII

Compilación en Linux

Cómo compilar y ejecutar programas en línea de comandos

Facultad de Informática
Universidad Complutense de Madrid

I.E.S. Antonio de Nebrija (Móstoles)

http://www.programa-me.com

Cómo compilar y ejecutar programas en línea de comandos

En las modalidades presenciales de ProgramaMe, el entorno software que se pone a disposición de los participantes está basado en alguna distribución de GNU/Linux. Lo normal será que estén disponibles algunos entornos integrados de desarrollo, típicamente Eclipse y Netbeans, junto con múltiples editores de texto (vi, emacs, gedit, ...)[1]. En las ediciones on-line, son los propios participantes los que deciden qué software utilizan, pudiendo usar incluso sistemas operativos y entornos propietarios distintos. En ese caso, debe recordarse que el juez automático que evaluará los envíos se ejecutará sobre GNU/Linux, por lo que se deberían tener en cuenta las posibles diferencias entre los "dialectos" de los lenguajes, particularmente en C y C++.

En este documento se detalla cómo compilar programas escritos en C, C++ y Java en GNU/Linux utilizando la línea de comandos. Está especialmente orientado a los participantes de los concursos presenciales que decidan no hacer uso de ninguno de los entornos de desarrollo que se les proporcionen.

En la parte final, también se detalla cómo aprovechar los ficheros `sample.*` para probar, mínimamente, los programas realizados. Esta información puede resultar útil para todos los participantes, independientemente de la modalidad del concurso en la que participen.

Las capturas de pantalla mostradas asumen una distribución estándar basada en Gnome. No obstante, el procedimiento es independiente del escritorio que se esté usando.

1 ¿Por dónde empezar?

El primer paso es, naturalmente, escribir el código fuente que se quiere compilar. Si ya tienes destreza utilizando algún editor de texto (como emacs, vim, o cualquier otro), entonces puedes saltar a la siguiente sección.

Si no tienes preferencia por ningún editor, entonces consideramos que lo más sencillo es que te decantes por `gedit` (Gnome) o `Kwrite` o `Kate` (KDE)[2]. Búscalo en el menú de aplicaciones de tu distribución, normalmente en el apartado de Accesorios (quizá con el nombre genérico de "Editor de textos").

Tras abrir el editor, el siguiente paso es escribir el programa. Utilizaremos como ejemplo el icónico *Hola mundo*, que se popularizara a raíz del libro de Dennis Ritchie:

[1] *Consulta* la información particular de la sede del concurso en el vayas a participar para estar seguro del software que tendrás disponible.

[2] Recuerda, esto es únicamente si habéis decidido no utilizar ninguno de los entornos de desarrollo (IDEs) disponibles.

Tras escribir el código, graba el fichero en algún lugar. En este caso, que es un programa en C, usa la extensión `.c`. Usa `.cpp` y `.java` para cada uno de los otros dos lenguajes.

2 Ya tengo mi programa escrito ¿Y ahora qué?

Una vez tengamos escrito nuestro programa, es hora de compilarlo. Para eso, necesitamos una *consola* o *terminal*, que nos proporcione el interfaz en línea de comandos. De nuevo, busca la aplicación en el menú de aplicaciones de tu distribución. Normalmente aparecerá también en la sección de Accesorios. Al final, deberías conseguir una ventana como la siguiente:

Una vez abierta, utiliza el comando `cd` para moverte hasta el directorio donde guardaste tu archivo de código fuente. Cuando estés en él, escribe el comando necesario para compilarlo, que dependerá del lenguaje de programación usado:

- Si el programa lo hemos escrito en C:

  ```
  gcc <archivo.c>
  ```

  Ejemplo 1: compilar el programa `hola.c`, y generar el fichero ejecutable `a.out`:

  ```
  gcc hola.c
  ```

  Ejemplo 2: compilar el programa `hola.c`, y generar el fichero ejecutable `hola`:

  ```
  gcc hola.c -o hola
  ```

- Si el programa lo hemos escrito en C++:

  ```
  g++ <archivo.cpp>
  ```

  Ejemplo 3: compilar el programa `hola.cpp` y generar el ejecutable `hola`:

  ```
  g++  hola.cpp -o hola
  ```

- Si el programa lo hemos escrito en Java:

  ```
  javac <archivo.java>
  ```

  Ejemplo 4: compilar el programa (la clase) `Hola.java` y generar el fichero (ejecutable a través de la máquina virtual de Java) `Hola.class`:

  ```
  javac Hola.java
  ```

Si el código fuente era correcto, se habrá generado el ejecutable y podremos pasar al siguiente punto. En otro caso, analiza los mensajes de error proporcionados por el compilador y solucionalos modificando el código fuente.

3 Mi programa ya compila, ¿qué es lo siguiente?

Ha llegado la hora de ejecutar el programa. Si escribiste el código fuente en C o C++, bastará con escribir `./a.out` (o el nombre que hayas usado con la opción `-o`, como `./hola` en el ejemplo).

Si el programa esta realizado en Java, habrá que escribir `java programa`, donde `programa` es el nombre que hemos dado a nuestro fichero de código, pero sin la extensión (`java Hola` en el ejemplo).

Con el programa en ejecución, toca probarlo introduciendo por teclado la entrada usando el formato definido en el enunciado del problema, y comprobando si la salida es la esperada.

4 ¿Cómo depuro mi programa?

En las modalidades presenciales del concurso, habitualmente tendrás disponible `gdb`, el depurador de GNU/Linux de consola[3]. No obstante, la depuración está fuera del alcance de este documento. Si no se conoce su uso, es preferible que los participantes se acostumbren a buscar los errores del código mediante la inclusión de código que vuelve en pantalla el estado de las variables importantes, con el fin de ir comprobando que es lo que está haciendo el programa.

5 ¿Para qué sirven los archivos `sample.in` y `sample.out`?

En los enunciados de los problemas de los concursos se proporciona siempre un ejemplo de entrada, con la salida esperada, para completar la explicación de lo que se está pidiendo. *Como mínimo*, los participantes deberían probar sus soluciones con esos casos de ejemplo.

Para eso, se puede lanzar la ejecución como se describió antes, y teclear cada una de las líneas del ejemplo mostrado en el enunciado. Sin embargo, para evitar tener que hacerlo contínuamente, en los concursos suelen proporcinarse ficheros de texto con el contenido de los ejemplos que aparecen en los enunciados. En concreto, el archivo `sample.in` de un problema contiene el ejemplo de *entrada* y el fichero `sample.out` contiene la salida esperada para dicho ejemplo del enunciado. Como ejemplos, en la página web hay publicados varios `sample.*` de la sesión de prueba on-line del concurso regional de la Comunidad de Madrid del año 2011.

Para utilizar el fichero `sample.in` nos apoyamos en la *redirección de la entrada estándar* que proporcionan los interfaces en línea de comandos. Al usarla, cuando el programa al que se le está aplicando la redirección lee del teclado, lo que estará realmente haciendo es leer del fichero redirigido. Por tanto, en lugar de escribir manualmente el ejemplo, podremos utilizar:

```
./prog < sample.in
```

donde `prog` es el nombre del programa que hayamos. Si el programa está escrito en Java:

```
java prog < sample.in
```

En la pantalla veremos ahora únicamente la salida generada por el programa, que podrá comprobarse con el ejemplo de salida del enunciado.

Si la salida de ejemplo es muy larga, puede resultar tedioso comprobar si es exactamente igual que la del enunciado. Además, los espacios *no son visibles*. Si, por ejemplo, la solución escribe espacios adicionales al final de las líneas, pasará desapercibido en una comparación manual, pero el juez automático lo considerará un error.

[3] De nuevo, consulta la información específica del entorno software del concurso en el que vayas a participar.

Es por tanto interesante hacer también una comparación *automática* con el `sample.out`. Para eso, redirigimos también la *salida estándar* a un fichero, de modo que todo lo que escriba el programa no saldrá por pantalla, sino que terminará en disco:

```
./prog < sample.in > result.out
```

Si queremos comprobar si la salida del programa (`result.out`), es igual a la salida correcta del ejercicio `sample.out` no tendremos más que utilizar la orden `diff` para saber si hay diferencias y, de haberlas, dónde se encuentran. Si no se obtiene salida, significará que ambos ficheros son iguales, y nuestro programa ha funcionado bien con los casos de ejemplo.

Si estás probando los `sample.*` de la página web, asegúrate de utilizar los apropiados para el sistema operativo que uses. Los saltos de línea no son iguales en GNU/Linux y Windows, y si utilizas el fichero que no corresponda a tu sistema obtendrás falsos errores.

Por último, ten en cuenta que la comprobación exitosa con el ejemplo del enunciado *no* significa que la solución funcione correctamente para todas las posibles entradas. Es decir, la coincidencia entre la salida del programa y el fichero `sample.out` *no* es en medida alguna garantía de que la solución esté bien (pero sí al contrario; si el `sample.out` es distinto, entonces la solución está mal).

Los jueces automáticos utilizarán *otros casos de prueba distintos* para comprobar que el programa funciona bien. Es por eso por lo que los participantes deberían probar las soluciones que desarrollen con sus propios casos de prueba antes de enviarlos al juez automático.

Anexo VIII

Ficheros .HEX

El fichero HEX explicado

Veremos cómo está compuesto el fichero .hex que se genera al compilar nuestro programa. Esta información será muy útil sobre todo para los nuevos diseñadores de grabadores de PIC's o para los que estén entrando en el tema de los Bootloadores. Ambos deben empezar por conocer esto de los .hex

Por: Diego "RedPic" Márquez - http://picmania.garcia-cuervo.net

Introducción

Cada vez que compiláis u ensambláis un programa fuente vuestro compilador u ensamblador genera un fichero .hex cuyo contenido se corresponde exactamente con lo que ha de ser grabado en la memoria de programa (o EEPROM) del microcontrolador.

Fijaos que he dicho "se corresponde" y no que sea exactamente lo que ha de ser grabado en el PIC, no es una "imagen" de la memoria de programa del PIC, sino una serie de instrucciones que el grabador de PIC's o el Bootloader que utilicemos sabe interpretar y por lo tanto grabar lo que corresponde exactamente en su sitio. Más adelante veremos cómo están codificadas las direcciones y su contenido.

Un poco de Historia

El formato .hex es del fabricante de micros INTEL que lo inventó allá por los años 70 del siglo pasado para usarlo exactamente para lo mismo que nosotros lo estamos usando ahora pero para sus microprocesadores 8085 y otros cacharros antediluvianos por el estilo y desde entonces está en uso. Muchos otros fabricantes lo han adoptado como propio y otros lo han copiado, cambiando esto y aquello, para al final hacer lo mismo (por ejemplo el SRecord de Motorola y otros).

En el fondo un fichero .hex no es más que una lista de direcciones de memoria y lo que contiene cada una de estas posiciones.

Aunque hay tres tipos de ficheros HEX: de 8 bits, de 16 bits y de 32 bits también llamados I8HEX, I16HEX e I32HEX respectivamente en esta nota vamos a ver solo el de 8 bits que es el que más utilizamos para nuestros PIC's 16F y 18F, el resto son muy parecidos pero no iguales.

Descripción

Un fichero .hex es un fichero de texto. Por lo tanto puede ser editado con un notepad o similar. Consiste en una serie de líneas consecutivas que empiezan siempre por el carácter ":" (dos puntos) salvo los comentarios que usan ";" (punto y coma) y terminadas en [0x0D][0x0A] (Fín de línea, Retorno de Carro)

Todos números (longitudes, direcciones y datos) están expresados en **HEXADECIMAL** mediante sus caracteres **ASCII** correspondientes.

Una muestra de su apariencia es:

```
:020000040000FA
:1000000043EF00F0EA6A070EE96EEF500DE0060ECE
:10001000016E006A002EFED7012EFBD77B0E006E0C
:10002000002EFED7EF2EF3D7000C0990099208524C
:1000300002E0098001D0098207C003F00650D8B45D
:10004000070606060310 1EE0000E0AB2010E0B6E34
:10005000000E09B0010E0B24016EE8B002D0819CA5
:1000600001D0818C000E0AB2010E0B6E000E09B297
:10007000010E0B24016EE8B002D0819C01D0818C6E
:100080000A2ADAD7000CF86AD09EEA6AE96AC150F7
:10009000C00B0F09C16E070EB46E040E066EFA0E89
:1000A000076EB0DF062EFBD7076A190E066E086AC8
:1000B000BCDF040E066EFA0E076EA4DF062EFBD719
:1000C000076A190E066E010E086EAFDFE6D7030051
:020000040030CA
:0E000000000C1E1F008381000FC00FE00F4098
:00000001FF
;PIC18F4550
```

La estructura de una línea es:

- ***Start code*** Un caracter, ":" para líneas con contenido, ";" para comentarios.
- ***Byte count*** Dos caracteres HEX que indican el número de datos en ***Data***.
- ***Address*** Cuatro caracteres HEX para la dirección del primer dato en ***Data***.
- ***Record type*** Dos caracteres HEX que indican el tipo de línea, de 00 a 05. (ver mas abajo)
- ***Data*** Secuencia de 2 * n caracteres HEX correspondientes a los ***Byte count*** datos definidos antes.
- ***Checksum*** Dos caracteres HEX de Checksum calculado según el contenido anterior de la línea en la forma: El byte menos significativo del complemento a dos de la suma de los valores anteriores expresados como enteros los caracteres hexadecimales (sin incluir ni el ***Start code*** ni al él mismo)

Cada línea puede expresar según su ***Record type***:

- **00** ***data record***: Línea de datos, contiene la dirección del primer dato y la secuencia de datos a partir de ésa.
- **01** ***End Of File record***: Línea de Fin del Fichero HEX. Indica que se han acabado las líneas de datos. Usualmente es "***:00000001FF***"
- **02** ***Extended Segment Address Record***: *Usado para procesadores 80x86 (No nos interesa aquí)*
- **03** ***Start Segment Address Record***: *Usado para procesadores 80x86 (No nos interesa aquí)*
- **04** ***Extended Linear Address Record***: Si las líneas de datos que sigan a ésta necesitan una dirección mayor que la de 16 bits ésta línea aporta los otros 16 bits para completar una dirección completa de 32 bits. Todas las líneas que sigan a esta deben completar su dirección con hasta los 32 bits con el contenido de la última línea de tipo **04**
- **05** ***Start Linear Address Record****: Usado para procesadores 80386 o superiores (No nos interesa aquí)*

Nuestro Ejemplo Podemos así entonces interpretar nuestro ejemplo anterior de la siguiente forma:

: 02 0000 04 0000 FA
-> Línea relevante, con dos bytes de información, de tipo 04 : luego las direcciones siguientes se complementan a 32 bits con 0x0000, 0xFA es el checksum.
: 10 0000 00 43 EF 00 F0 EA 6A 07 0E E9 6E EF 50 0D E0 06 0E CE
-> Línea relevante, con 16 bytes de información, de tipo 00 : así que hay que escribir 0x43 (en 0x00000000), 0xEF (en 0x00000001), ... , 0x0E (en 0x0000000F). 0xCE es el checksum.
: 10 0010 00 01 6E 00 6A 00 2E FE D7 01 2E FB D7 7B 0E 00 6E 0C
-> Línea relevante, con 16 bytes de información, de tipo 00 : así que hay que escribir 0x01 (en 0x00000010), 0x6E (en 0x00000011), ... , 0x6E (en 0x0000001F). 0x0C es el checksum.
...
...
...
: 02 0000 04 0030 CA
-> Línea relevante, con dos bytes de información, de tipo 04 : luego las direcciones siguientes se complementan a 32 bits con 0x0030, 0xCA es el checksum.
: 0E 0000 00 00 0C 1E 1F 00 83 81 00 0F C0 0F E0 0F 40 98
-> Línea relevante, con 14 bytes de información, de tipo 00 : así que hay que escribir 0x00 (en 0x00300000), 0x0C (en 0x00300001), ... , 0x40 (en 0x0030000D). 0x98 es el checksum.
:00000001FF
-> Línea relevante. Fin de Fichero HEX
;PIC18F4550
-> Línea irrelevante. Comentario.

Eso es todo por este documento. Espero que os sea de ayuda.

Anexo IX

Configuración de la conexión serie

LnxComm

Linux Communication

Fernando Pujaico Rivera

LnxComm : Linux Communication

por Fernando Pujaico Rivera

<fernando.pujaico.rivera (en) gmail.com>

Tabla de contenidos

1. Introducción **1**

2. Descripción de funciones **2**

2.1. Previamente 2
2.2. Open_Port 2
2.3. Get_Configure_Port 2
2.4. Configure_Port 2
2.5. Set_Configure_Port 3
2.6. Write_Port 3
2.7. Read_Port 4
2.8. Gets_Port 4
2.9. Getc_Port 4
2.10. Kbhit_Port 5
2.11. Close_Port 5
2.12. Set_Hands_Haking 5
2.13. Set_BaudRate 6
2.14. Set_Time 6
2.15. IO_Blocking 6
2.16. Clean_Buffer 7
2.17. Create_Thread_Port 7

3. Ejemplos **9**

3.1. Previamente 9
3.2. Bloqueante 9
3.3. No-Bloqueante 10
3.4. Timeout 11
3.5. Evento 12

4. Referencias **14**

Capítulo 1. Introducción

La biblioteca "Linux Communication" (LnxComm) está diseñada para brindar un apoyo a los programadores que estén relacionados con el diseño y construcción de hardware. LnxComm nos permite crear una conexión con el puerto serie mediante unas pocas lineas de código.

Otra de las ventajas de esta biblioteca es que nos permite crear programas que podrán ser compilados en sistemas operativos GNU-LINUX y WINDOWS brindando así mayor portabilidad a nuestros programas.

La biblioteca está completamente desarrollada en Lenguaje C.

Capítulo 2. Descripción de funciones

2.1. Previamente

Las funciones cumplen las mismas características tanto como para sistemas operativos GNU-LINUX y WINDOWS.

2.2. Open_Port

Función de lectura del puerto.

```
HANDLE Open_Port(char COMx[]);
```

Abre el puerto serie, recibe como parámetro una cadena con el nombre del puerto y devuelve una variable de tipo *HANDLE* que es el manejador del puerto.

```
COMx[]:  Es una cadena que contiene el nombre del puerto a abrir, ejemplo.
```

En Windows:

```
"COM1"        , "COM2"        , "COM3"        , "COM4"        , ...
```

En Gnu-Linux:

```
"/dev/ttyS0"  , "/dev/ttyS1"  , "/dev/ttyS2"  , "/dev/ttyS3"  , ...
"/dev/ttyUSB0", "/dev/ttyUSB1", "/dev/ttyUSB2", "/dev/ttyUSB3", ...
"/dev/ttyACM0", "/dev/ttyACM1", "/dev/ttyACM2", "/dev/ttyACM3", ...
```

Retorna: El manejador de Puerto Abierto (variable tipo *HANDLE*). En caso de error devuelve *INVALID_HANDLE_VALUE*.

2.3. Get_Configure_Port

Devuelve configuración actual del puerto serie.

```
DCB Get_Configure_Port(HANDLE fd);
```

Esta función devuelve un variable de tipo *DCB* con la configuración actual del puerto serie ,la función recibe un parámetro de tipo *HANDLE* que es el manejador devuelto por la función Open_port.

```
fd     :  Es el manejador del puerto.
```

Retorna: Una estructura *DCB* con una copia de la configuración actual del puerto serie y carga la variable *ERROR_CONFIGURE_PORT* con *FALSE*, en caso de error carga la variable *ERROR_CONFIGURE_PORT* con *TRUE*.

2.4. Configure_Port

Establece la configuración del puerto serie.

```
DCB Configure_Port(    HANDLE          fd,
                       unsigned int    BaudRate,
                       char            CharParity[]);
```

Esta función configura el puerto serie con los parámetros *fd*,*BaudRate* y *CharParity*.

```
fd        : Es el manejador del puerto serie devuelto por Open_port.
BaudRate  : Es la velocidad del puerto serie. (B115200, B19200, B9600, ...)
CharParity: Indica el número de bits de la transmisión. ("8N1","7E1","7O1","7S1")
```

Retorna: Una estructura *DCB* con una copia de la configuración actual del puerto serie y carga la variable *ERROR_CONFIGURE_PORT* con *FALSE*, en caso de error carga la variable *ERROR_CONFIGURE_PORT* con *TRUE*.

2.5. Set_Configure_Port

Establece la configuración del puerto serie.

```
int Set_Configure_Port(        HANDLE  fd,
                               DCB     PortDCB);
```

Restituye/establece la configuración del puerto serie, los parámetros serán pasados mediante una variable tipo *DCB*.

```
fd     : Es el manejador del puerto devuelto por Open_port.
newtio : Es una variable DCB con la configuración del puerto, generalmente
         se usa la devuelta por la función Get_Configure_Port
```

Retorna: *TRUE* si todo fue bien o *FALSE* si hubo algún error.

2.6. Write_Port

Escribe un bloque de datos tipo *char* en el puerto serie.

```
long Write_Port(       HANDLE  fd,
                       char    Data[],
                       int     SizeData);
```

Escribe los *SizeData* primeros caracteres de *Data*.


Se debe escoger un *SizeData* menor o igual que la longitud de *Data*.

```
fd  : Es el manejador del puerto devuelto por Open_port.
Data  : Es el dato a mandar.
SizeData: Es el número de bytes que se quieren escribir.
```

Retorna : En caso de éxito devuelve el número de bytes escritos (cero indica que no se ha escrito nada). En GNU-LINUX en caso de error devuelve -1.

2.7. Read_Port

Recibe un bloque de datos en el puerto serie.

```
long Read_Port(         HANDLE  fd,
                        char    *Data,
                        int     SizeData);
```

Lee los *SizeData* primeros caracteres del puerto y lo carga en *Data*.

 Se debe escoger un *SizeData* menor o igual que la longitud de *Data*.

```
fd  : Es el manejador del puerto devuelto por Open_port.
Data  : Es la variable en donde se reciben los datos.
SizeData: Es el número de bytes que se desea recibir.
```

Retorna : En caso de éxito devuelve el número de bytes leídos (cero indica que no se ha leído nada). En GNU-LINUX en caso de error devuelve -1.

2.8. Gets_Port

Recibe una cadena de caracteres tipo *char* por el puerto serie.

```
long Gets_Port(         HANDLE  fd,
                        char    *Data,
                        int     SizeData);
```

Recibe datos por el puerto, lee hasta encontrar un 0x0A,0x0D o hasta completar *SizeData* caracteres. Los datos son guardados en la variable *Data*

 Se debe escoger un *SizeData* menor o igual que la longitud de *Data*.

```
fd      : Es el manejador del puerto devuelto por Open_port.
Data    : Es la variable en donde se reciben los datos.
SizeData: Es el máximo número de bytes que se desea recibir.
```

Retorna : El número de caracteres recibidos, estos números serán siempre mayores o iguales a cero.

2.9. Getc_Port

Recibe un caracter por el puerto serie.

```
long Getc_Port(        HANDLE  fd,
                       char    *Data);
```

Recibe un único caracter por el puerto y es cargado en la variable *Data* de tamaño 1 byte.

```
fd     : Es el manejador del puerto devuelto por Open_port.
Data   : Es la variable en donde se reciben los datos(1 Byte).
```

Retorna : En caso de éxito devuelve el número de bytes leídos. En GNU-LINUX en caso de error devuelve -1.

2.10. Kbhit_Port

Indica el número de bytes en el buffer de entrada del puerto serie.

```
int Kbhit_Port( HANDLE fd);
```

Recibe como parámetro el manejador del puerto.

```
fd     : Es el manejador del puerto devuelto por Open_port.
```

Retorna: El número de caracteres en el buffer de entrada.

2.11. Close_Port

Cierra el puerto serie.

```
int Close_Port( HANDLE fd);
```

Recibe la variable *fd* y cierra el puerto serie.

```
fd     : Es el manejador del puerto devuelto por Open_port.
```

Retorna: *TRUE* si se ha cerrado el puerto y *FALSE* en el caso contrario.

2.12. Set_Hands_Haking

Configura el control de flujo en el puerto serie.

```
int Set_Hands_Haking(  HANDLE  fd,
                       int     FlowControl);
```

Recibe como variables el manejador del puerto serie y el tipo de control de flujo.

```
fd         : Es el manejador del puerto devuelto por Open_port.
FlowControl: Es un número entero que indica el tipo de control de flujo.
               0    Ninguno
               1    RTSCTS
               2    XonXoff
               3    DTRDSR
```

Retorna : *TRUE* si todo fue bien y *FALSE* si no lo fue.

2.13. Set_BaudRate

Configura la velocidad en baudios del puerto serie.

```
int Set_BaudRate(      HANDLE fd,
                       unsigned int BaudRate);
```

Recibe como datos el manejador del puerto y la velocidad en baudios del mismo.

```
fd      : Es el manejador del puerto devuelto por Open_port.
BaudRate: Es la velocidad del puerto, los valores pueden ser.
              B2400
              B9600
              B19200
              B115200
       Para mas datos vea el archivo baudios.h .
```

Retorna: *TRUE* si todo fue bien y *FALSE* si no lo fue.

2.14. Set_Time

Configura temporizador para las funciones de lectura y escritura en el puerto serie.

```
int Set_Time(  HANDLE          fd,
               unsigned int    Time);
```

Recibe como variables, el manejador del puerto y el máximo tiempo entre bytes en milisegundos (ms)

```
fd     : Es el manejador del puerto devuelto por Open_port.
Time   : Multiplicador, para el tamaño total del TimeOut en la
         lectura y escritura de datos.

       TimeOut = (100 * Time * numero_de_bytes_en_la_lectura) ms
```

Retorna: *TRUE* si todo fue bien y *FALSE* si no lo fue.

2.15. IO_Blocking

Escoge entre el modo bloqueante y no bloqueante en lectura de datos en el puerto serie.

```
int IO_Blocking(       HANDLE fd,
                       int Modo);
```

La función recibe como parámetro el manejador del puerto y *TRUE* si se quiere una lectura de datos bloqueante o *FALSE* si no.

```
fd     : Es el manejador del puerto devuelto por Open_port.
Modo   : TRUE : Modo bloqueante.
         FALSE: Modo no bloqueante.
```

Retorna: *TRUE* si todo fue bien y *FALSE* si no lo fue.

2.16. Clean_Buffer

Termina las operaciones de lectura y escritura pendientes y limpia las colas de recepción y de transmisión en el puerto serie.

```
int Clean_Buffer(      HANDLE fd);
```

La función recibe como parámetro el manejador del puerto.

```
fd     : Es el manejador del puerto devuelto por Open_port.
```

Retorna: *TRUE* si todo fue bien y *FALSE* si no lo fue.

2.17. Create_Thread_Port

Crea una función (hilo) que se ejecuta cuando existan caracteres en el buffer de entrada del puerto serie.

```
pthread_t Create_Thread_Port(  HANDLE *fd);
```

Recibe como parametro el manejador del puerto. y devuelve una variable de tipo *pthread_t*

```
fd     : Es el manejador del puerto devuelto por Open_port.
```

Retorna: El manejador del hilo creado.

Para poder usar la función Create_Thread_Port primero se debe de habilitar escribiendo lo siguiente: *#define ENABLE_SERIAL_PORT_EVENT*, luego se debe de escribir el código de la función *SERIAL_PORT_EVENT(HANDLE *hPort)*

```
#define ENABLE_SERIAL_PORT_EVENT

#include "com/serial.h"

void SERIAL_PORT_EVENT(        HANDLE *hPort)
```

```
{
        // Código de ejemplo aquí
        // char   Data[16];
        // Read_Port(*hPort,Data,15);
        // Data[15]=0;
        // printf("%s",Data);
}
```

La función *SERIAL_PORT_EVENT* recibe como parámetro un puntero de tipo HANDLE que es el manejador del puerto devuelto por Open_port.

Capítulo 3. Ejemplos

Para escribir tus programas puedes escoger cuatro métodos bloqueante, no-bloqueante, time-out y evento.

3.1. Previamente

La cabecera cambia según el sistema operativo.

```
Linux:

#define __LINUX_COM__
#include "com/serial.h"

Windows:

#define __WINDOWS_COM__
#include "com/serial.h"
```

El uso de las comillas dobles (*"com/serial.h"*) indica que la carpeta *com* se encuentra en la misma carpeta del archivo de código fuente que la invoca, osea si se tiene un archivo *ejemplo.c* que usa *"com/serial.h"*, la carpeta *com* se debe de encontrar en la misma carpeta de *ejemplo.c*.

La función Open_Port también cambiara de argumento según el sistema operativo.

```
Linux:

"/dev/ttyS0"  , "/dev/ttyS1"  , ...
"/dev/ttyUSB0", "/dev/ttyUSB1", ...
"/dev/ttyACM0", "/dev/ttyACM1", ...

Windows:

"COM1", "COM2, "COM3", ...
```

Si se esta usando linux como sistema operativo la compilación de los programas que se realicen se hará de la siguiente manera:

```
gcc -o archivo archivo.c
```

En el caso de que se esté creando un hilo se deberá compilar de la siguiente manera.

```
gcc -o archivo archivo.c -lpthread
```

3.2. Bloqueante

Aquí (../../bloqueante.c) se tiene un ejemplo de un programa bloqueante.

```
#define __LINUX_COM__                    // #define __WINDOWS_COM__
```

```
#include "com/serial.h"

int main()
{
    HANDLE fd;
    DCB OldConf;
    char cad[16]="Enviando Texto";
    int n;

    fd=Open_Port("/dev/ttyS0");         // Abre el puerto serie.
                                        // fd=Open_Port("COM1");

    OldConf=Get_Configure_Port(fd);     // Guardo la configuración del puerto.

    Configure_Port(fd,B115200,"8N1");   // Configuro el puerto serie.

                                        // Bloqueante por defecto, pero también
                                        // se puede usar:
                                        // IO_Blocking(fd,TRUE);

    n=Write_Port(fd,cad,16);            // Escribo en el puerto serie.

    while(Kbhit_Port(fd)<16);           // Espero a leer hasta que se tengan
                                        // 16 bytes en el buffer de entrada.

    n=Read_Port(fd,cad,16);             // Leo el puerto serie.
    printf("%s",cad);                   // Muestro los datos.

    Set_Configure_Port(fd,OldConf);     // Restituyo la antigua configuración
                                        // del puerto.

    Close_Port(fd);                     // Cierro el puerto serie.

    printf("\nPresione ENTER para terminar\n");
    getchar();

    return 0;
}
```

3.3. No-Bloqueante

Aquí (../../nobloqueante.c) se tiene un ejemplo de un programa no-bloqueante.

```
#define __WINDOWS_COM__                 // #define __LINUX_COM__

#include "com/serial.h"

int main()
{
    HANDLE fd;
    DCB OldConf;
```

```
    char cad[16]="Enviando Texto";
    int n;

    fd=Open_Port("COM1");               // Abre el puerto serie.
                                        // fd=Open_Port("/dev/ttyS0");

    OldConf=Get_Configure_Port(fd);     // Guardo la configuración del puerto.

    Configure_Port(fd,B115200,"8N1");   // Configuro el puerto serie.

    IO_Blocking(fd,FALSE);              // Seleccionamos lectura no bloqueante.

    n=Write_Port(fd,cad,16);            // Escribo en el puerto serie.

    while(Kbhit_Port(fd)<16);           // Espero a leer hasta que se tengan
                                        // 16 bytes en el buffer de entrada.

    n=Read_Port(fd,cad,16);             // Leo el puerto serie.
    printf("%s",cad);                   // Muestro loas datos.


    Set_Configure_Port(fd,OldConf);     // Restituyo la antigua configuración
                                        // del puerto.

    Close_Port(fd);                     // Cierro el puerto serie.

    printf("\nPresione ENTER para terminar\n");
    getchar();

    return 0;
}
```

3.4. Timeout

Aquí (../../timeout.c) se tiene un ejemplo de un programa con Time-Out.

```
#define __WINDOWS_COM__                 // #define __LINUX_COM__

#include "com/serial.h"

int main()
{
    HANDLE fd;
    DCB OldConf;
    char cad[16]="X";
    int n,TIME=2,i;

    fd=Open_Port("COM1");               // Abre el puerto serie.
                                        // fd=Open_Port("/dev/ttyS0");

    OldConf=Get_Configure_Port(fd);     // Guardo la configuración del puerto.
```

```
    Configure_Port(fd,B19200,"8N1");    // Configuro el puerto serie.

    Set_Time(fd,TIME);                  // Time-Out entre caracteres es TIME*0.1.

    n=Write_Port(fd,cad,1);             // Escribo en el puerto serie.

    n=Gets_Port(fd,cad,16);             // Leo el puerto serie.
    printf("%s",cad);                   // Muestro la cadena.

    Set_Configure_Port(fd,OldConf);     // Restituyo la antigua configuración
                                        // del puerto.
    Close_Port(fd);                     // Cierro el puerto serie.

    printf("\nPresione ENTER para terminar\n");
    getchar();

    return 0;
}
```

3.5. Evento

Aquí (../../evento.c) se tiene un ejemplo de un programa con evento.

```
#define __WINDOWS_COM__                 // #define __LINUX_COM__
#define ENABLE_SERIAL_PORT_EVENT

#include "com/serial.h"

int numero=0;
void SERIAL_PORT_EVENT(HANDLE * hPort)
{
    char c;
    Getc_Port(*hPort,& c);
    printf("[%d]=%c\n",numero,c);
    numero++;
}

int main()
{
    HANDLE fd;
    DCB OldConf;
    char cad[16]="Enviando Texto\n";
    int n;

    fd=Open_Port("COM1");               // Abre el puerto serie.
                                        // fd=Open_Port("/dev/ttyS0");

    OldConf=Get_Configure_Port(fd);     // Guardo la configuración del puerto.

    Configure_Port(fd,B115200,"8N1");   // Configuro el puerto serie.
```

```
    IO_Blocking(fd,TRUE);               // Bloqueante por defecto, pero también
                                        // se puede usar:
                                        // IO_Blocking(fd,TRUE);

    n=Write_Port(fd,cad,16);            // Escribo en el puerto serie.

    Create_Thread_Port(& fd);        // Creo un hilo y le paso el manejador.

    while(TRUE);

    Set_Configure_Port(fd,OldConf);     // Restituyo la antigua configuración
                                        // del puerto.

    Close_Port(fd);                     // Cierro el puerto serie.

    printf("\nPresione ENTER para terminar\n");
    getchar();

    return 0;
}
```

Capítulo 4. Referencias

- http://lnxcomm.sourceforge.net (http://lnxcomm.sf.net)
- http://zsoluciones.com
- http://winapi.conclase.net
- http://google.com

CÓMO Programar el puerto serie en Linux

por Peter H. Baumann, Peter.Baumann@dlr.de
traducción de Pedro Pablo Fábrega pfabrega@arrakis.es
v0.3, 14 Junio 1997

Este documento describe cómo programar comunicaciones con dispositivos sobre puerto serie en una máquina Linux.

Índice General

1 Introducción **1**

1.1 Copyright 2
1.2 Versiones futuras de este Documento 2
1.3 Revisión 2

2 Comenzando **3**

2.1 Depuración 3
2.2 Configuración del Puerto 3
2.3 Conceptos de entrada para dispositivos serie 4
2.3.1 Proceso de Entrada Canónico 4
2.3.2 Proceso de Entrada No Canónico 4
2.3.3 Entrada Asíncrona 4
2.3.4 Espera de Entradas Origen Múltiple 4

3 Programas Ejemplo **5**

3.1 Proceso de Entrada Canónico 5
3.2 Proceso de Entrada NO Canónico 7
3.3 Entrada Asíncrona 9
3.4 Espera de Entradas de Origen Múltiple. 10

4 Otras fuentes de Información **11**

5 Contribuciones **12**

5.1 Traducción 12

6 Anexo: El INSFLUG **12**

1 Introducción

Este es el COMO Programar el puerto serie en Linux. Todo sobre cómo programar comunicaciones con otros dispositivos/ordenadores sobre una línea serie, bajo Linux. Explicaremos diferentes técnicas: E/S Canónica (sólo se transmiten/reciben líneas completas), E/S asíncrona, y espera para una entrada de origen múltiple.

Este documento no describe cómo configurar un puerto serie, ya que esto ha sido descrito por Greg Hankins en el Serial-HOWTO[1]

Tengo que hacer notar encarecidamente que no soy un experto en este campo, pero he tenido problemas con un proyecto que necesitaba tales comunicaciones. Los ejemplos de código añadidos aquí se derivaron del código de *miniterm* disponible en la guía de programadores del *Linux Documentation Project*:

(`ftp://sunsite.unc.edu/pub/Linux/docs/LDP/programmers-guide/` y espejos) en el directorio de ejemplos. Si alguien tiene algún comentario, con gusto lo incorporaré a este documento (ver sección 1.3).

Todos los ejemplos fueron comprobados usando un núcleo Linux 2.0.29 sobre un i386.

1.1 Copyright

1.2 Versiones futuras de este Documento

Las nuevas versiones de COMO Programar el puerto serie en Linux estarán disponibles en: `ftp://sunsite.unc.edu:/pub/Linux/docs/HOWTO/Serial-Programming-HOWTO` y sus espejos. Hay otros formatos, como versiones PostScript y DVI en el directorio `other-formats`.

CÓMO Programar el puerto serie en Linux también está disponible en `http://sunsite.unc.edu/LDP/HOWTO/Serial-Programming-HOWTO.html` y será enviado a `comp.os.linux.answers` mensualmente.

1.3 Revisión

Por favor, mándeme cualesquiera corrección, pregunta, comentario, sugerencia o material adicional. Me gustaría mejorar este HOWTO-COMO. Dígame exactamente qué es lo que no entiende, o qué debería estar más claro. Me puede encontrar en `Peter.Baumann@dlr.de` vía email. Por favor, incluya el número de versión del CÓMO Programar el puerto serie en Linux cuando escriba. Esta es la versión 0.3.

[1] Disponible en castellano como *Serie-COMO*.

2 Comenzando

2.1 Depuración

La mejor forma de depurar su código es configurar otra máquina Linux y conectar los dos ordenadores mediante un cable null-módem.

Use `miniterm`, disponible en el LDP *Programmers Guide*:
(`ftp://sunsite.unc.edu/pub/Linux/docs/LDP/programmers-guide/`
en el directorio de ejemplos) para transmitir caracteres a su máquina Linux. Miniterm se puede compilar con mucha facilidad y transmitirá todas las entradas en bruto del teclado por el puerto serie.

Sólo las sentencias *define* (`#define MODEMDEVICE "/dev/ttyS0"`) tienen que ser comprobadas. Ponga `ttyS0` para COM1, `ttyS1` para COM2, etc.. Es esencial para comprobar que *todos* los caracteres se transmiten en bruto (sin un procesamiento de salida) por la línea. Para comprobar su conexión, inicie `miniterm` en ambos ordemadores y teclee algo. Los caracteres introducidos en un ordenador deberían aparecer en el otro y viceversa. La entrada no tendrá eco en la pantalla del ordenador en el que escribamos.

Para hacer un cable null-modem tiene que cruzar las líneas TxD (transmit) y RxD (receive). Para una descripción del cable vea el Serie-COMO.

También es posible ejecutar estas comprobaciones con un sólo ordenador, si tiene un puerto serie no utilizado. Puede ejecutar dos miniterm en sendas consolas virtuales. Si libera un puerto serie desconectando el ratón, recuerde redirigir `/dev/mouse`, si existe. Si usa una tarjeta multipuerto serie, esté seguro de configurarla correctamente. Yo tenía la mía mal configurada, y todo funcionaba bien mientras hacía las comprobaciones en un sólo ordenador. Cuando lo conecté a otro, el puerto empezó a perder caracteres. La ejecución de dos programas en un ordenador nunca es completamente asíncrona.

2.2 Configuración del Puerto

Los dispositivos `/dev/ttyS*` tienen como misión conectar terminales a su linux, y están configurados para este uso al arrancar. Hay que tener esto presente cuando se programen comunicaciones con un dispositivo. Por ejemplo, los puertos están configurados para escribir en pantalla cada carácter enviado desde el dispositivo, que normalmente tiene que ser cambiado para la transmisión de datos.

Todos los parámetros se pueden configurar fácilmente con un programa. La configuración se guarda en una estructura `struct termios`, que está definida en <`asm/termbits.h`>:

```
#define NCCS 19
struct termios {
      tcflag_t c_iflag;         /* parametros de modo entrada */
      tcflag_t c_oflag;         /* parametros de modo salida */
      tcflag_t c_cflag;         /* parametros de modo control */
      tcflag_t c_lflag;         /* parametros de modo local */
      cc_t c_line;              /* disciplina de la linea */
      cc_t c_cc[NCCS];          /* caracteres de control */
};
```

Este archivo también incluye todas las definiciones de parámetros. Los parámetros de modo entrada de `c_iflag` manejan todos los procesos de entrada, lo cual significa que los caracteres enviados desde el dispositivo pueden ser procesados antes de ser leídos con `read`.

De forma similar `c_oflag` maneja los procesos de salida. `c_cflag` contiene la configuración del puerto, como la velocidad en baudios, bits por carácter, bits de parada, etc... Los parámetros de modo local se guardan en `c_lflag`. Determinan si el carácter tiene eco, señales enviadas al programa, etc...

Finalmente la tabla `c_cc` define el carácter de control para el fin de fichero, parada, etc... Los valores por defecto de los caracteres de control están definidos en <`asm/termios.h`>. Los parámetros están descritos en la página del manual `termios(3)`.

La estructura `termios` contiene los elementos `c_line`. Estos elementos no se mencionan ni las páginas del manual para *termios* de Linux ni en las páginas de manual de Solaris 2.5. ¿Podría alguien arrojar alguna luz sobre esto? ¿No debería estar incluido en la estructura `termio`?

2.3 Conceptos de entrada para dispositivos serie

Hay tres diferentes conceptos de entrada que queremos presentar. El concepto apropiado se tiene que escoger para la aplicación a la que lo queremos destinar. Siempre que sea posible no haga un bucle para leer un sólo carácter a fin de obtener una cadena completa. Cuando he hecho esto, he perdido caracteres, mientras que un `read` para toda la cadena no mostró errores.

2.3.1 Proceso de Entrada Canónico

Es el modo de proceso normal para terminales, pero puede ser útil también para comunicaciones con otros dispositivos. Toda la entrada es procesada en unidades de líneas, lo que significa que un `read` sólo devolverá una línea completa de entrada. Una línea está, por defecto, finalizada con un `NL`(ASCII `LF`), y fin de fichero, o un carácter fin de línea. Un `CR` (el fin de línea por defecto de DOS/Windows) no terminará una línea con la configuración por defecto.

El proceso de entrada canónica puede, también, manejar los caracteres borrado, borrado de palabra, reimprimir carácter, traducir `CR` a `NL`, etc..

2.3.2 Proceso de Entrada No Canónico

El *Proceso de Entrada No Canónico* manejará un conjunto fijo de caracteres por lectura, y permite un carácter temporizador. Este modo se debería usar si su aplicación siempre lee un número fijo de caracteres, o si el dispositivo conectado envía ráfagas de caracteres.

2.3.3 Entrada Asíncrona

Los dos modos descritos anteriomente se pueden usar en modos síncrono y asíncrono. El modo síncrono viene por defecto, donde la sentencia `read` se bloquará hasta que la lectura esté completa. En modo asíncrono la sentencia `read` devolverá inmediatamente y enviará una señal al programa llamador cuando esté completa. Esta señal puede ser recibida por un manejador de señales.

2.3.4 Espera de Entradas Origen Múltiple

No es un modo diferente de entrada, pero puede ser útil si está manejando dispositivos múltiples. En mi aplicación manejaba entradas sobre un *socket* TCP/IP y entradas sobre una conexión serie de otro ordenador de forma casi simultánea. El programa ejemplo dado abajo esperará una entrada de dos orígenes distintos. Si la entrada de una fuente está disponible, entonces será procesada, y el programa esperará otra entrada nueva.

La aproximación presentada abajo parece más bien compleja, pero es importante tener en cuenta que Linux es un sistema operativo multiproceso. La llamada al sistema `select` no carga la CPU mientras espera una entrada, mientras que un bucle hasta que hay una una entrada disponible ralentizaría demasiado el resto de procesos que se ejecuten a la misma vez.

3 Programas Ejemplo

Todos los ejemplos provienen de `miniterm.c`. El buffer está limitado a 255 caracteres, como la longitud máxima de cadena para el proceso de entrada canónica. (<`linux/limits.h`> o <`posix1_lim.h`>).

Vea los comentarios que hay en el código para una explicación del uso de los diferentes modos de entrada. Espero que el código sea comprensible. El ejemplo de entrada canónica está mejor comentado, el resto de los ejemplos están comentados sólo donde difieren del ejemplo de entrada canónica para remarcar las diferencias.

Las descripciones no son completas, por eso le invito a experimentar con los ejemplos para obtener mejores soluciones para su aplicación.

¡No olvide dar los permisos apropiados a los puertos serie:

```
chmod a+rw /dev/ttyS1
```

3.1 Proceso de Entrada Canónico

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>
#include <stdio.h>

/* la tasa de baudios esta definida en  <asm/termbits.h>, que esta
   incluida <termios.h> */

#define BAUDRATE B38400

/* cambie esta definicion por el puerto correcto  */
#define MODEMDEVICE "/dev/ttyS1"

#define _POSIX_SOURCE 1 /* fuentes cumple POSIX  */

#define FALSE 0
#define TRUE 1

volatile int STOP=FALSE;

main()
{
   int fd,c, res;
   struct termios oldtio,newtio;
   char buf[255];

/*
      Abre el dispositivo modem para lectura y escritura y no como controlador
      tty porque no queremos que nos mate si el ruido de la linea envia
      un CTRL-C.
*/

   fd = open(MODEMDEVICE, O_RDWR | O_NOCTTY );
   if (fd <0) {  perror(MODEMDEVICE); exit(-1);  }

   tcgetattr(fd,&oldtio); /* almacenamos la configuracion actual del puerto */
```

```
   bzero(newtio, sizeof(newtio)); /* limpiamos struct para recibir los
                                     nuevos parametros del puerto */

/*
      BAUDRATE: Fija la tasa bps. Podria tambien usar cfsetispeed y cfsetospeed.
      CRTSCTS : control de flujo de salida por hardware (usado solo si el cable
      tiene todas las lineas necesarias Vea sect. 7 de Serial-HOWTO)
      CS8     : 8n1 (8bit,no paridad,1 bit de parada)
      CLOCAL  : conexion local, sin control de modem
      CREAD   : activa recepcion de caracteres
*/

   newtio.c_cflag = BAUDRATE | CRTSCTS | CS8 | CLOCAL | CREAD;

/*
      IGNPAR  : ignora los bytes con error de paridad
      ICRNL   : mapea CR a NL (en otro caso una entrada CR del otro ordenador
      no terminaria la entrada) en otro caso hace un dispositivo en bruto
      (sin otro proceso de entrada)
*/

   newtio.c_iflag = IGNPAR | ICRNL;

/*
      Salida en bruto.
*/
   newtio.c_oflag = 0;

/*
      ICANON  : activa entrada canonica
      desactiva todas las funcionalidades del eco, y no envia segnales al
      programa
      llamador
*/

   newtio.c_lflag = ICANON;

/*
      inicializa todos los caracteres de control
      los valores por defecto se pueden encontrar en /usr/include/termios.h,
      y vienen dadas en los comentarios, pero no los necesitamos aqui
*/

   newtio.c_cc[VINTR]    = 0;     /* Ctrl-c */
   newtio.c_cc[VQUIT]    = 0;     /* Ctrl-\ */
   newtio.c_cc[VERASE]   = 0;     /* del */
   newtio.c_cc[VKILL]    = 0;     /* @ */
   newtio.c_cc[VEOF]     = 4;     /* Ctrl-d */
   newtio.c_cc[VTIME]    = 0;     /* temporizador entre caracter, no usado */
   newtio.c_cc[VMIN]     = 1;     /* bloqu.lectura hasta llegada de caracter. 1 */
   newtio.c_cc[VSWTC]    = 0;     /* '\0' */
   newtio.c_cc[VSTART]   = 0;     /* Ctrl-q */
   newtio.c_cc[VSTOP]    = 0;     /* Ctrl-s */
   newtio.c_cc[VSUSP]    = 0;     /* Ctrl-z */
   newtio.c_cc[VEOL]     = 0;     /* '\0' */
   newtio.c_cc[VREPRINT] = 0;     /* Ctrl-r */
   newtio.c_cc[VDISCARD] = 0;     /* Ctrl-u */
```

```
   newtio.c_cc[VWERASE]   = 0;     /* Ctrl-w */
   newtio.c_cc[VLNEXT]    = 0;     /* Ctrl-v */
   newtio.c_cc[VEOL2]     = 0;     /* '\0' */

/*
      ahora limpiamos la linea del modem y activamos la configuracion del
      puerto
*/

   tcflush(fd, TCIFLUSH);
   tcsetattr(fd,TCSANOW,&newtio);

/*
      configuracion del terminal realizada, ahora manejamos las entradas.
      En este ejemplo, al introducir una  'z' al inicio de linea terminara el
      programa.
*/

   while (STOP==FALSE) {     /* bucle hasta condicion de terminar */

/*
   bloque de ejecucion de programa hasta que llega un caracter de fin de
   linea, incluso si llegan mas de 255 caracteres.
   Si el numero de caracteres leidos es menor que el numero de caracteres
   disponibles, las siguientes lecturas devolveran los caracteres restantes.
   'res' tomara el valor del numero actual de caracteres leidos.
*/

                        res = read(fd,buf,255);
                        buf[res]=0;             /* envio de fin de cadena, a fin de poder usar printf */
                        printf(":%s:%d\n", buf, res);
                        if (buf[0]=='z') STOP=TRUE;
                      }

/* restaura la anterior configuracion del puerto  */

   tcsetattr(fd,TCSANOW,&oldtio);
}
```

3.2 Proceso de Entrada NO Canónico

En el modo de proceso de entrada no canónico, la entrada no está ensamblada en líneas y el procesamiento de la entrada (`erase`, `kill`, `delete`, etc.) no ocurre. Dos parámetros controlan el comportamiento de este modo: `c_cc[VTIME]` fija el temporizador de carácter, y fija el número mínimo de caracteres a recibir antes de satisfacer la lectura.

Si MIN > 0 y TIME = 0, MIN fija el número de caracteres a recibir antes de que la lectura esté realizada. Como TIME es cero, el temporizador no se usa.

Si MIN = 0 y TIME > 0, TIME indica un tiempo de espera. La lectura se realizará si es leído un sólo carácter, o si se excede TIME (t =TIME *0.1 s). Si TIME se excede, no se devuelve ningún carácter.

Si MIN > 0 y TIME > 0, TIME indica un temporizador entre caracteres. La lectura se realizará si se reciben MIN caracteres o el tiempo entre dos caracteres excede TIME. El temporizador se reinicia cada vez que se recibe un carácter y sólo se hace activo una vez que se ha recibido el primer carácter.

Si MIN = 0 y TIME = 0, la lectura se realizará inmediatamente. Devolverá el número de caracteres

disponibles en el momento, o el número de caracteres solicitados. De acuerdo con Antonino (ver contribuciones), podría poner un `fcntl(fd, F_SETFL, FNDELAY);` antes de leer para obtener el mismo resultado.

Modificando `newtio.c_cc[VTIME]` y `newtio.c_cc[VMIN]` se pueden comprobar todos los modos descritos arriba.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>
#include <stdio.h>

#define BAUDRATE B38400
#define MODEMDEVICE "/dev/ttyS1"
#define _POSIX_SOURCE 1 /* fuentes cumple POSIX */
#define FALSE 0
#define TRUE 1

volatile int STOP=FALSE;

main()
{
   int fd,c, res;
   struct termios oldtio,newtio;
   char buf[255];

   fd = open(MODEMDEVICE, O_RDWR | O_NOCTTY );
   if (fd <0) {  perror(MODEMDEVICE); exit(-1);  }

   tcgetattr(fd,&oldtio); /* salva configuracion actual del puerto  */

   bzero(newtio, sizeof(newtio));
   newtio.c_cflag = BAUDRATE | CRTSCTS | CS8 | CLOCAL | CREAD;
   newtio.c_iflag = IGNPAR;
   newtio.c_oflag = 0;

/* pone el modo entrada (no-canonico, sin eco,...) */

   newtio.c_lflag = 0;

   newtio.c_cc[VTIME]    = 0;   /* temporizador entre caracter, no usado */
   newtio.c_cc[VMIN]     = 5;   /* bloquea lectura hasta recibir 5 chars  */

   tcflush(fd, TCIFLUSH);
   tcsetattr(fd,TCSANOW,&newtio);


   while (STOP==FALSE) {                          /* bucle para entrada */
                         res = read(fd,buf,255);   /* devuelve tras introducir 5 */
                         buf[res]=0;               /* asi podemos printf... */
                         printf(":%s:%d\n", buf, res);
                         if (buf[0]=='z') STOP=TRUE;
                       }
   tcsetattr(fd,TCSANOW,&oldtio);
}
```

3.3 Entrada Asíncrona

```
#include <termios.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/signal.h>
#include <sys/types.h>

#define BAUDRATE B38400
#define MODEMDEVICE "/dev/ttyS1"
#define _POSIX_SOURCE 1 /* fuentes cumple POSIX  */
#define FALSE 0
#define TRUE 1

volatile int STOP=FALSE;

void signal_handler_IO (int status);   /* definicion del manejador de segnal */
int wait_flag=TRUE;                    /* TRUE mientras no segnal recibida */

main()
{
   int fd,c, res;
   struct termios oldtio,newtio;
   struct sigaction saio;           /* definicion de accion de segnal  */
   char buf[255];

/* abre el dispositivo en modo no bloqueo (read volvera inmediatamente) */

      fd = open(MODEMDEVICE, O_RDWR | O_NOCTTY | O_NONBLOCK);
   if (fd <0) {  perror(MODEMDEVICE); exit(-1);  }

/* instala el manejador de segnal antes de hacer asincrono el dispositivo */

   saio.sa_handler = signal_handler_IO;
   saio.sa_mask = 0;
   saio.sa_flags = 0;
   saio.sa_restorer = NULL;
   sigaction(SIGIO,&saio,NULL);

/* permite al proceso recibir SIGIO */

      fcntl(fd, F_SETOWN, getpid());

/* Hace el descriptor de archivo asincrono (la pagina del manual dice solo
      O_APPEND y O_NONBLOCK, funcionara con  F_SETFL...) */

   fcntl(fd, F_SETFL, FASYNC);
   tcgetattr(fd,&oldtio); /* salvamos conf. actual del puerto */

/*
      fija la nueva configuracion del puerto para procesos de entrada canonica
*/

   newtio.c_cflag = BAUDRATE | CRTSCTS | CS8 | CLOCAL | CREAD;
   newtio.c_iflag = IGNPAR | ICRNL;
   newtio.c_oflag = 0;
```

```
   newtio.c_lflag = ICANON;
   newtio.c_cc[VMIN]=1;
   newtio.c_cc[VTIME]=0;
   tcflush(fd, TCIFLUSH);
   tcsetattr(fd,TCSANOW,&newtio);

/* bucle de espera para entrada. Normalmente se haria algo util aqui  */

   while (STOP==FALSE) {
                          printf(".\n");usleep(100000);

/*
tras recibir SIGIO, wait_flag = FALSE, la entrada esta disponible y puede ser leida
 */

                if (wait_flag==FALSE) {
                                    res = read(fd,buf,255);
                                    buf[res]=0;
                                    printf(":%s:%d\n", buf, res);
                                    if (res==1) STOP=TRUE; /* para el bucle si solo entra un CR */
                                    wait_flag = TRUE;      /* espera una nueva entrada */
                                  }
                        }
/* restaura la configuracion original del puerto  */
      tcsetattr(fd,TCSANOW,&oldtio);
}


/***************************************************************************
* manipulacion de segnales. pone wait_flag a FALSE, para indicar al bucle  *
* anterior que los caracteres han sido recibidos                           *
***************************************************************************/

void signal_handler_IO (int status)
{
   printf("recibida segnal SIGIO.\n");
   wait_flag = FALSE;
}
```

3.4 Espera de Entradas de Origen Múltiple.

Esta sección está al mínimo. Sólo intenta ser un indicación, y por tanto el ejemplo de código es pequeño. Esto no sólo funcionará con puertos serie, sino que también lo hará con cualquier conjunto de descriptores de archivo.

La llamada `select` y las macros asociadas usan un `fd_set`. Esto es una tabla de bits, que tiene una entrada de bit para cada número de descriptor de archivo válido. `select` aceptará un `fd_set` con los bits fijados para los descriptores de archivos relevantes y devuelve un `fd_set`, en el cual los bits para el descriptor del archivo están fijados donde ocurre una entrada, salida o excepción. Todas la manipulaciones de `fd_set` se llevan a cabo mediante las macros proporcionadas. Ver también la página del manual `select(2)`.

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

main()
```

```
{
   int    fd1, fd2;  /* origenes de entrada  1 y 2 */
   fd_set readfs;    /* descriptor de archivo */
   int    maxfd;     /* mixmum file desciptor used */
   int    loop=1;    /* bucle mientras TRUE */

/*
   open_input_source abre un dispositivo, fija el puerto correctamente
   y devuelve un descriptor de archivo
*/

   fd1 = open_input_source("/dev/ttyS1");   /* COM2 */
   if (fd1<0) exit(0);
   fd2 = open_input_source("/dev/ttyS2");   /* COM3 */
   if (fd2<0) exit(0);
   maxfd = MAX (fd1, fd2)+1;  /* entrada maxima de bits (fd) a probar */

/* bucle para entrada */
      while (loop) {
                     FD_SET(fd1, &readfs);  /* comprobacion origen 1  */
                     FD_SET(fd2, &readfs);  /* comprobacion origen 2 */

/* bloqueo hasta que la entrada esta disponible  */
                        select(maxfd, &readfs, NULL, NULL, NULL);
                     if (FD_ISSET(fd1))         /* entrada de origen 1 esta disponible */
                        handle_input_from_source1();
                     if (FD_ISSET(fd2))         /* entrada de origen 2 esta disponible */
                        handle_input_from_source2();
                  }

}
```

El ejemplo dado bloquea indefinidamente hasta que una entrada de una de las fuentes está disponible. Si necesita un temporizador para la entrada, sólo sustituya la llamada `select` por:

```
int res;
struct timeval Timeout;

/* fija el valor del temporizador en el bucle de entrada  */
Timeout.tv_usec = 0;  /* milisegundos */
Timeout.tv_sec  = 1;  /* segundos */
res = select(maxfd, &readfs, NULL, NULL, &Timeout);
if (res==0)
/* numero de descriptores de archivo con input = 0, temporizador sobrepasado */
```

Este ejemplo concluye el tiempo de espera tras un segundo. Si este tiempo transcurre, `select` devolverá 0, pero tenga cuidado porque `Timeout` se decrementa por el tiempo actualmente esperado para la entrada por `select`. Si el valor de retardo es cero, `select` volverá inmediatamente.

4 Otras fuentes de Información

- El *Linux Serie-COMO* describe cómo configurar un puerto serie y contiene información sobre hardware.
- *Serial Programming Guide for POSIX Compliant Operating Systems*, por Michael Sweet.

- La página del manual `termios(3)` describe todos los parámetros de la estructura `termios`.

5 Contribuciones

Como se mencionó en la introducción, no soy un experto en este campo, pero he tenido mis propios problemas, y encontré la solución con la ayuda de otras personas. Gracias por la ayuda de Mr. Strudthoff de *European Transonic Windtunnel*, Cologne, Michael Carter, `mcarter@rocke.electro.swri.edu`, y Peter Waltenberg, `p.waltenberg@karaka.chch.cri.nz`

Antonino Ianella, `antonino@usa.net` escribió el *Serial-Port-Programming Mini HOWTO*, a la misma vez que yo preparaba este documento. Greg Hankins me pidió que incorporara el Mini-Howto de Antonino en este documento.

La estructura de este documento y el formateo SGML provienen del Serial-HOWTO de Greg Hankins.

5.1 Traducción

Este documento ha sido traducido por

Pedro Pablo Fábrega Martínez, `pfabrega@arrakis.es`

Si encontráis mejoras, añadidos o fallos, de cualquier tipo, indicádmelo para mejorar el documento.

Insultos > /dev/null

6 Anexo: El INSFLUG

El *INSFLUG* forma parte del grupo internacional *Linux Documentation Project*, encargándose de las traducciones al castellano de los Howtos (Comos), así como la producción de documentos originales en aquellos casos en los que no existe análogo en inglés.

En el **INSFLUG** se orienta preferentemente a la traducción de documentos breves, como los *COMOs* y *PUFs* (**P**reguntas de **U**so **F**recuente, las *FAQs*. `:)`), etc.

Diríjase a la sede del INSFLUG para más información al respecto.

En la sede del INSFLUG encontrará siempre las **últimas** versiones de las traducciones: `www.insflug.org`. Asegúrese de comprobar cuál es la última versión disponible en el Insflug antes de bajar un documento de un servidor réplica.

Se proporciona también una lista de los servidores réplica (*mirror*) del Insflug más cercanos a Vd., e información relativa a otros recursos en castellano.

Francisco José Montilla, `pacopepe@insflug.org`.