Proyecto de Grado

Interfaz USB genérica para comunicación con dispositivos electrónicos

Manual de Usuario

A/C Andrés Aguirre, A/C Pablo Fernández y A/C Carlos Grossy

Tutores: MSc Ing. Gonzalo Tejera y MSc Ing. Alexander Sklar

Instituto de Computación - Facultad de Ingeniería

Universidad de la República Oriental del Uruguay

14 de diciembre de 2007

Índice general

Índice de figuras

Capítulo 1

Introducción al Baseboard

El *baseboard* contiene los elementos de hardware necesarios para comenzar a desarrollar una solución modular que permita lograr la interacción entre un PC y dispositivos electrónicos por medio de USB. Como componente central del *baseboard* se puede destacar el microcontrolador PIC18F4550 el cual implementa soporte a USB 2.0 full speed (12 Mb/s), así como diversos módulos y puertos de entrada/salida. Para la conexión con el resto del hardware la placa cuenta con un conector, el U4APort, el cual se utiliza mediante un cable plano multihilo, simular a un cable IDE. Finalmente para la programación/debugging del firmware en el micrcontrolador se cuenta con un conector RJ11.

1.1. Elementos del Baseboard

A continuación se detallan cada uno de los componentes y su ubicacion dentro del hardware en la figura 1.1.

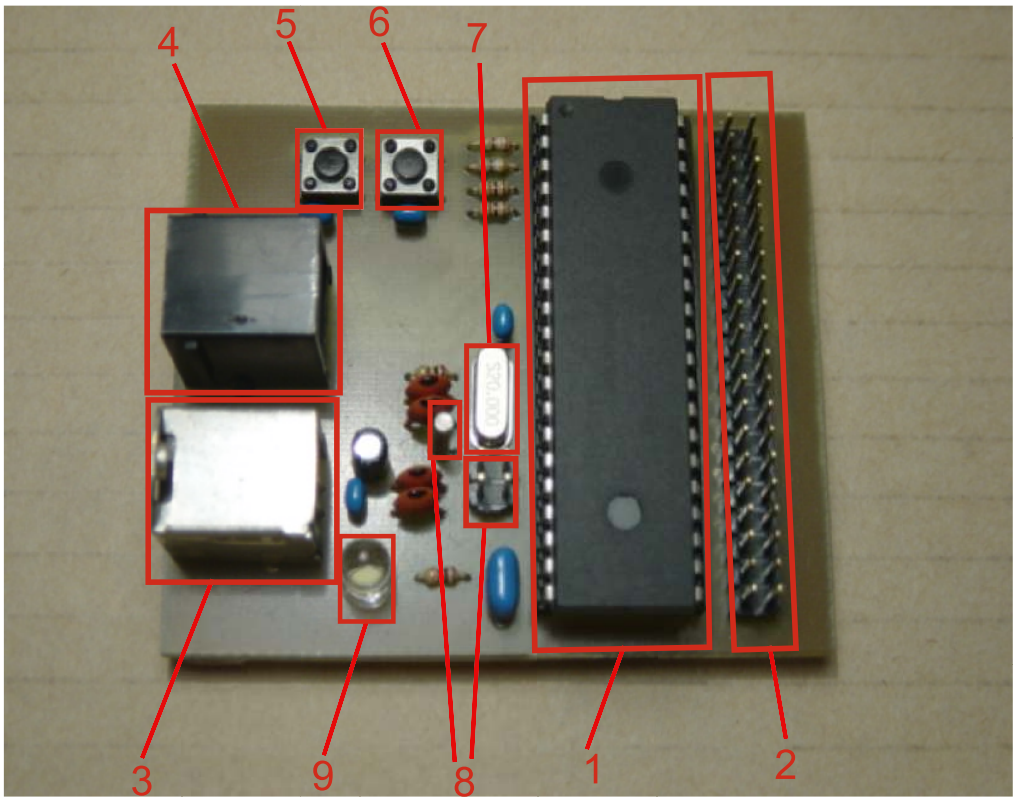

Figura 1.1: *Baseboard.*

1. **Microcontrolador PIC18F4550**: Este microcontrolador es el responsable del manejo de toda la interacción entre el USB y los dispositivos finales o *adapterboards*. Por más detalles del microcontrolador ver el documento [1].

2. **U4APort**: Este conector se utiliza para la conexión con *adapterboards* que contienen o permiten interactuar con el hardware del dispositivo específico que se desea utilizar. Sus pines incluyen a casi todos los pines del microcontrolador, además de pines de alimentación de 5 voltios derivada del conector USB. Dentro de este conjunto de *adapterboards*, se encuentran los que simplemente unen los pines del conector del *baseboard* directamente con dispositivos de baja potencia (leds, otros chips, etc.) y los *adapterboard* que proveen cualquier tipo de acondicionamiento de señales, o controladores de mediana potencia para el funcionamiento del dispositivo (motores, relays, etc.).

3. **Conector USB tipo B**: Permite conectar el *baseboard* con el PC mediante un cable USB A-B estándar, igual al utilizado en impresoras, scanners, etc.

4. **Conector RJ11 para programador/debugger**: Permite conectar el *baseboard* a un programador/debugger, por ejemplo el MPLAB IDC2. Como programador se utiliza para la grabación del bootloader por única vez. También se utiliza si desea debuggear cualquier componente que integra el firmware del *baseboard*.

5. **Botón de Reset**: Pulsar este botón causa el reseteo del microcontrolador y por tanto el cierre de todas las comunicaciones de los programas de aplicación con el *baseboard*. Luego de ello y dependiendo del estado del botón de bootloader del *baseboard*, éste reinicia en modo normal donde interactúa mediante la USB4all API con aplicaciones en el PC, o en modo bootloader para una actualización del firmware.

6. **Botón de Bootloader**: Si se mantiene apretado el botón de bootloader y se pulsa el botón de reset, el *baseboard* inicia en modo bootloader. En este modo se puede actualizar el firmware del *baseboard* mediante una aplicación dedicada a tal sentido en el PC vía USB. Si el botón de bootloader no es oprimido mientras se resetea, entonces el *baseboard* entra en modo normal.

7. **Cristal de 20 MHz**: Cristal utilizado para la generación de una señal de reloj principal para el microcontrolador.

8. **Cristal de reloj 32.768 kHz y jumper**: Al cerrar el jumper se establece el circuito que habilita el uso del oscilador de 32.768 kHz para su uso en aplicaciones relacionadas con reloj de tiempo real (Real Time Clock) (RTC).

9. **LED**: El led se enciende cuando el *baseboard* esta conectado al PC mediante un cable USB.

Capítulo 2

Programación y Grabación en el Baseboard

En esta sección se presentan detalles referentes a la implementación del *USB4all firmware* para explicar como se realiza la grabación de éste dentro microcontrolador del *baseboard* y con que mecanismos se cuenta para configurar el *baseboard* como dispositivo USB.

2.1. Carga de firmware base por primera vez

Lo primero a considerar, es que se utiliza un bootloader para la grabación del *USB4all firmware* dentro del microcontrolador. El bootloader es una pequeña aplicación ubicada al principio de la memoria de programa, cuya finalidad es la de grabar un firmware dentro del microcontrolador, el cual es transferido desde el PC mediante USB. Este bootloader es grabado por única vez en el microcontrolador utilizando un programador (por ejemplo MPLAB IDC2) mediante el conector RJ11 presente en el *baseboard*, esto se muestra en la figura 2.1.

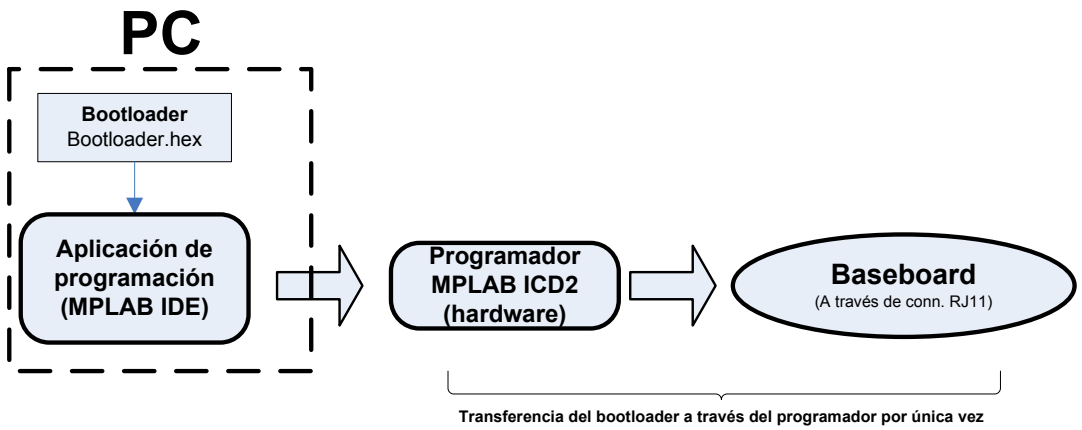

Figura 2.1: Programación del bootloader en el microcontrolador del *baseboard*

El *baseboard* de aquí en más podrá ser iniciado en dos modos, el **modo bootloader** y el **modo normal**. Para iniciarlo en modo bootloader, se deberá mantener apretado el pulsador bootloader luego de soltar el pulsador de reset. De otra forma el *baseboard*, iniciará en modo normal, adquiriendo el control usb4all firmware.

El *USB4all firmware*, está implementado en C, específicamente para el compilador MPLAB C18 de Microchip. Para generar éste firmware es necesario utilizar un proyecto que se brinda a manera de template. Dentro del proyecto se encuentran todos los fuentes del *base firmware* y se deben agregar o seleccionar los *proxies* y los *user modules* que sean necesarios para la solución a construir. El siguiente paso es compilar y linkeditar todos los fuentes, lo que da como resultado

la generación de un archivo .hex, que describe una imagen de memoria de programa que debe ser grabada en el microcontrolador. Finalmente esta imagen es transferida al microcontrolador, utilizando el bootloader. El proceso se muestra en la figura 2.2

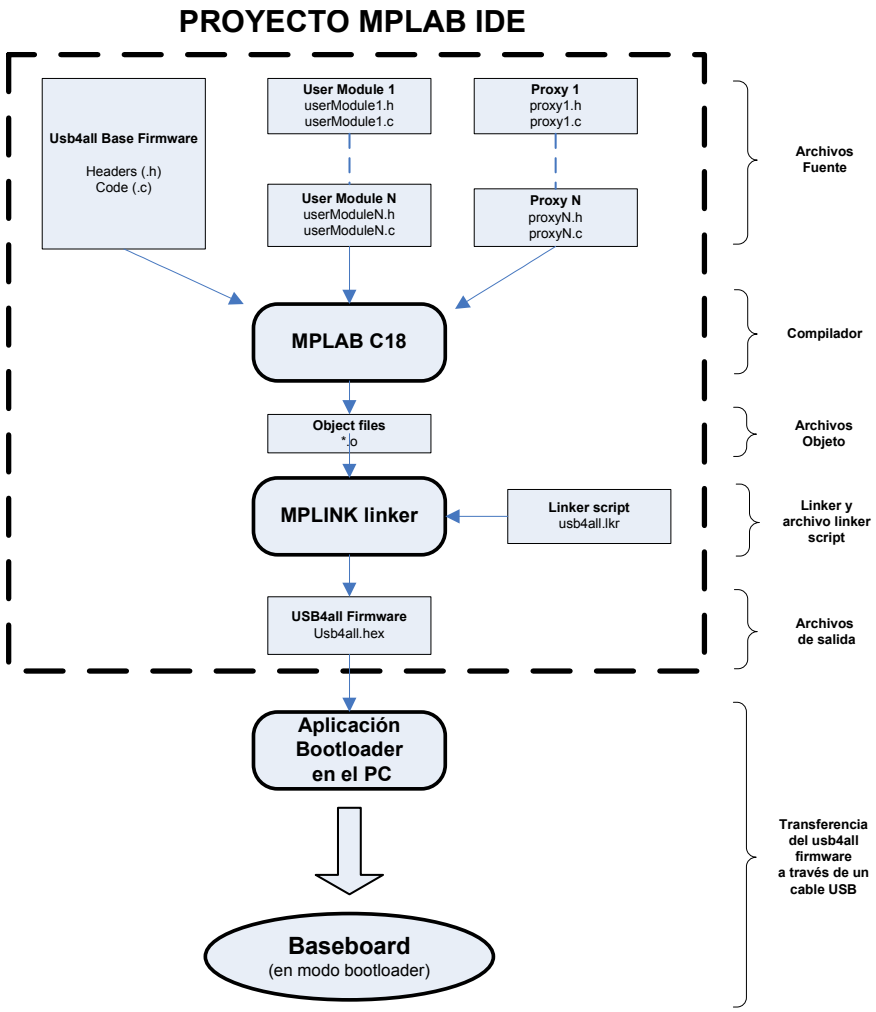

Figura 2.2: Proceso de desarrollo y grabado del *USB4all firmware*.

2.1.1. Organización de la memoria de programa

Para la implementación de mecanismos de eliminación de dependencias del *base firmware* de los componentes y para la configuración de los descriptores USB se dividió la memoria de programa en regiones fijas. En la figura 2.3 se muestra el mapa de memoria de programa del microcontrolador.

La primer área de memoria de programa es ocupada, como ya dijimos, por el código del bootloader. Esta es la única sección del mapa de memoria impuesta debido a las facilidades que otorga el hardware para este cometido, para las demás áreas fueron fijadas tratando de maximizar el espacio para *user modules* y *proxies*. A continuación se describen las demás áreas del *USB4all firmware*:

- **Interrupt vector remapping**: Es una pequeña área de memoria de programa en donde están mapeados los vectores de reset, e interrupciones de alta y baja prioridad. Aquí es donde las aplicaciones subidas por el bootloader ubican los saltos a sus funciones de main y

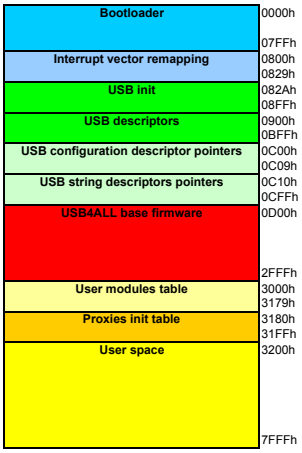

Figura 2.3: Mapa de memoria de programa del *baseboard*.

manejadores de interrupciones, al igual que como lo harían si no estuviera el booloader, pero sumando un offset de 0x0800. A modo de ejemplo, luego de un reset, el microcontrolador setea el contador de instrucciones en 0x0000 y comienza a ejecutar el código ubicado en dicha posición. Aquí el bootloader puede dependiendo del estado del boton de bootloader del *baseboard* iniciar en modo de ejecucion Normal o Bootloader.En el modo Normal realiza un salto a la posición de memoria 0x0800, donde se encuentra un nuevo salto al main loop del *base firmware*. Eso mismo sucede con los vectores de interrupciones. Las direcciones de los nuevos vectores se hallan sumando un offset de 0x0800 a la direcciones predefinidas en el hardware del microcontrolador. En el modo Bootloader el *baseboard* permanecerá esperando que se envíe el *USB4all firmware*, que será grabado dentro de la misma EEPROM del microcontrolador a partir de la dirección 0x0800.

- **USB Init**: Aquí se encuentra el código encargado de la inicialización de los recursos del microcontrolador destinados a la comunicación USB. Se inicializan los registros de la memoria compartida del microcontrolador, indicando tamaño y posición de los buffers de los endpoints, tipos de transferencia y dirección.
- **USB descriptors**: Aquí se encuentran almacenados todos los descriptores USB utilizados por el *baseboard*. Entre ellos se encuentran los descriptores del dispositivo, configuración, interfaces, endpoints y string. Están codificados según las especificaciones del capítulo 9 del estándar USB [5].
- **USB configuration descriptor pointers**: Son punteros hacia posiciones específicas del área USB descriptors. Son utilizados por el *base firmware* en el momento de la enumeración del dispositivo. Incluyen cantidad de interfaces, consumo de energía, y dentro de las interfaces: cantidad de endpoints, código de clase, de subclase, etc.
- **USB String descriptor pointers**: Son punteros hacia posiciones específicas del área USB descriptors. Son utilizados por el *base firmware* en el momento de la enumeración del dispositivo. Incluyen strings, que son descripciones, normalmente en ingles del tipo de producto USB. Además también son utilizados para contener un número de serie del producto.
- **USB4all base firmware**: En esta área se almacena todo el código correspondiente al *base firmware*.
- **User module table**: En esta área fija de memoria se ubica la tabla utilizada por el *base firmware* para encontrar las funciones de los *user modules*.
- **Proxies init table**: Tabla con punteros al código de las operaciones `init` de los **proxies**. Esta tabla es utilizada por el *base firmware* para inicializar todos los *proxies*.
- **User Space**: En esta área de memoria de programa se almacena el código de los *user modules* y *proxies*.

2.2. Carga de configuración y descriptores USB

Si bien el proceso descrito en la sección 2.1 es adecuado para generar el firmware completo para luego grabarlo dentro del microcontrolador, no se recomienda utilizarlo para realizar cambios de configuración en el *baseboard*, cambio de descriptores, o carga de *user modules* o *proxies*. Esto es posible realizarlo de manera automática utilizando la aplicación *USB4all baseboard utility*. En esta sección nos centraremos en como se realiza la carga de la configuración del dispositivo y descriptores USB. Si bien en una primera instancia se pensó realizar una interfaz gráfica para facilitar aún más la configuración del *baseboard*, esto no se llegó a alcanzar. De todas formas se logró realizar toda la lógica necesaria para brindar las funcionalidades, encapsulandolas adecuadamente, y modificando un archivo fuente puede realizarse el proceso de forma satisfactoria. La aplicación *USB4all baseboard utility* fue implementada en Java y realiza de manera intuitiva el armado de los descriptores para la configuración USB del dispositivo. Para el armado de estos descriptores se sigue la misma lógica utilizada en el estándar USB y explicada en detalle en el documento Estado del Arte [1]. Esta aplicación genera el código fuente necesario para la configuración y generación de desciptores, así como tambien la compilación, linkeado y grabación en el microcontrolador por medio del bootloader. Lo primero que se va a explicar es el proceso de generación de archivos fuentes.

2.2.1. Generación de archivos fuente

En la figura 2.4 se muestra como se relacionan entre sí los descriptores USB.

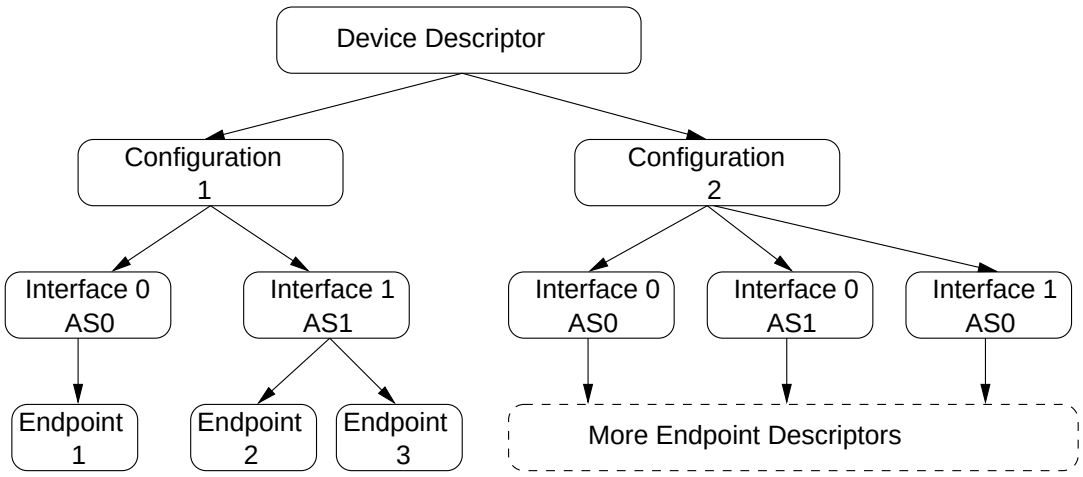

Figura 2.4: Relación entre descriptores USB

Existe una clase que encapsula a cada uno de los tipos de descriptor que se pueden instanciar según las necesidades. Cabe señalar que a pesar que algunos parámetros deberían tener siempre algunos valores determinados dentro del *baseboard*, de todas formas se pensó que este utilitario podía servir para fines otros fines, ademas de los del *USB4all*, por lo que se decidió dejarlos como parámetros que se pudieran utilizar en otras circunstancias. A continuación se pasará a detallar con fragmentos de código el funcionamiento de la aplicación.

Lo primero que debe ser generado es el descriptor de dispositivo (Device Descriptor). Esto se realiza mediante la instanciación de un objeto de esta clase con el siguiente instruccion:

```
DeviceDescriptor(2,0,0,0,8,0x04D8,0x000C,0)
```

El constructor tiene estos parámetros:

```
DeviceDescriptor(float usbSpecReleaseNumber, int classCode, int subClassCode,
int protocolCode, int ep0BuffSize, int vendorId, int productId, int deviceReleaseNumber)
```

Los parametros en orden de aparición son:

- `usbSpecReleaseNumber:` Revision de USB, en general debe utilizarse 2.
- `classCode`: Código de clase, como se usa clase custom debe utilizarse 0.
- `subClassCode`: Código de subclase, como se usa clase custom debe utilizarse 0.
- `protocolCode`: Código del protocolo, como se usa clase custom debe utilizarse 0.
- `ep0BuffSize`: Tamaño del buffer para el endpoint 0, por defecto se utiliza 8.
- `vendorIds`: Identificador del vendedor (VID) para el uso de los driver genéricos proporcionados debe utilizarse el valor 0x04D8
- `productIds`: Identificador del producto (PID) para el uso de los driver genéricos proporcionados debe utilizarse el valor 0x000C
- `deviceReleaseNumbers`: Numero de liberación. Es solo información descriptiva, en este caso se utilizó 0 pero pudo haberse utilizado cualquier otro número.

Luego de instanciado el DeviceDescriptor, se procede a la creación de los descriptores de jerarquías más bajas. Se empieza por los descriptores de String:

```
StringDescriptor sd0 = new StringDescriptor(0x0409)
StringDescriptor sd1 = new StringDescriptor("Aguirre Fernandez Grossy")
StringDescriptor sd2 = new StringDescriptor("USB4all (C)2007")
StringDescriptor sd3 = new StringDescriptor("12345678")
```

Aquí se se crean 4 descriptores de String, el primero es particular y sirve para codificar el lenguaje a ser utilizado, por defecto es inglés y está representado por el valor 0x0409. Luego se instancian un conjunto de StringsDescriptors para ser utilizados como información en lenguaje natural del baseboard.

Para asociarlos se utilizan los siguientes métodos:

```
dd.addStringDescriptor(sd0)
dd.addStringDescriptor(sd1)
dd.addStringDescriptor(sd2)
dd.addStringDescriptor(sd3)
dd.setManufacturerString(sd1)
dd.setProductString(sd2)
dd.setDeviceSerialNumberString(sd3)
```

Los addStringDescriptor sirven para reservar memoria para ellos, y luego se les asigna una semántica especial mediante las instrucciones:

- `setManufacturerString`: Asocia el string al nombre del fabricante, al StringDescriptor pasado como parámetro
- `setProductString`: Asocia el string al nombre del productor, al StringDescriptor pasado como parámetro
- `setDeviceSerialNumberString`: Asocia el string al nombre del productor, al StringDescriptor pasado como parámetro, este parámetro es muy importante ya que junto al VID y PID va a identificar de forma única a una instancia del *baseboard*. No debe conectarse dos o más *baseboards* con el mismo número de serie, ya que puede tener resultados inesperados al ejecutar operaciones en la *USB4all API*.

Luego, cada Device Descriptor, tiene uno o Configuration Descriptors asociados. En el caso del *baseboard* no tiene sentido definir más de una configuración, por lo que sólo se crea una única configuracion:

```
Configuration cfg1 = new Configuration(dd,50)
```

Se utiliza el constructor con los siguientes parametros:

```
Configuration(DeviceDescriptor myDeviceDescriptor, int maxPowerConsuption)
```

Con los siguientes parámetros:

- `myDeviceDescriptor`: Referencia al DeviceDescriptor al cual está asociada la configuración. Hay un único device descriptor por dispositivo USB.
- `maxPowerConsuption`: Corriente máxima en miliamperes que se reserva en el Host USB para este dispositivo, en este caso se utilizan 50 mA, se debe calcular una cota máxima para el dispositivo a conectar. El límite admitido en el estándar son 500 mA, por lo que si se desea consumir más que esta cantidad (por ejemplo, si se desea conectar un motor) se deberá proveer al dispositivo a conectar de una fuente externa.

Luego, se crean también los endpoints, que se asocian a una interfaz:

```
Endpoint ep1out = new Endpoint(1, Endpoint.OUT,Endpoint.INTERRUPT, 64, 32);
Endpoint ep1in = new Endpoint(1, Endpoint.IN,Endpoint.INTERRUPT, 64, 32);
Endpoint ep2out = new Endpoint(2, Endpoint.OUT,Endpoint.BULK, 64, 32);
Endpoint ep2in = new Endpoint(2, Endpoint.IN,Endpoint.BULK, 64, 32);
Endpoint ep3out = new Endpoint(3, Endpoint.OUT,Endpoint.ISOCHRONOUS, 64, 32);
Endpoint ep3in = new Endpoint(3, Endpoint.IN,Endpoint.ISOCHRONOUS, 64, 32);
```

Aqui los endpoints se crean utilizando el constructor:

```
Endpoint(int endpointNumber, int direction, int endpointType, int endpointSize,
int pollingTime)
```

Con los siguientes parámetros:

- `endpointNumber`: Número de endpoint, puede estar entre 1 y 15, y por razones de optimización conviene que se numeren a partir del 1 y de forma consecutiva, sin dejar saltear ninguno.
- `direction`: Dirección del endpoint, puede ser IN, para que el Host reciba datos del *baseboard* o OUT para que el Host envíe datos al *baseboard*.
- `endpointType`: Tipo de endpoint: Puede ser de CONTROL, INTERRUPT, BULK o ISOCHRONOUS.
- `endpointSize`: Tamaño del buffer, puede elegirse entre 8 y 64 bytes, y debe ser potencia de 2.
- `pollingTime`: Tiempo de polling en milisegundos. Es el tiempo que el host escrutiña al dispositivo, útil en los endpoints del tipo INTERRUPT e ISOCHRONOUS.

Debido a que los endpoints podrían llegar a estar vinculados con varias interfaces, se realiza la asociación de los endpoints a las interfases de la siguiente manera:

```
i1.addEndpoint(ep1out)
i1.addEndpoint(ep1in)
i1.addEndpoint(ep2out)
i1.addEndpoint(ep2in)
i1.addEndpoint(ep3out)
i1.addEndpoint(ep3in)
```

Aquí el parámetro de la operación addEndpoint es una referencia a una instancia de Endpoint.

Por último se debe asociar una interface a una configuración y como también una interfaz puede estar asociada a varias configuraciones se realiza mediante la operacion:

```
cfg1.addInterface(i1)
```

Luego de generado este conjunto de descriptores, se procede a la generación de código fuente.

Para ello, se instancia un generador de código al que se le pasa el descriptor de dispositivo y una ruta en donde generar los archivos fuente necesarios para la configuración de estos descriptores en el microcontrolador.

Esto se realiza mediante la operación

```
Usb4allConfigGenerator cg = new Usb4allConfigGenerator(dd, "C:/usb4all/fw/Demo/usb4all/usb
```

Por último se instancia la operación generateFiles del generador de configuraciones, y éste deja los archivos fuente en la carpeta seleccionada.

```
cg.generateFiles();
```

Finalmente se muestra en el algoritmo 1, todos los pasos juntos para la generación de los descritpores para el *baseboard.*

2.2.2. Compilacion y linkedición y grabación en el baseboard

Luego de que la aplicación *USB4all baseboard utility* genera el código fuente de las aplicaciones, se realiza el proceso de compilación y linkedición de los mismos con el resto del firmware, que está dentro de un conjunto de carpetas del proyecto MPLAB. Para esto el *USB4all baseboard utility* realiza un parseo del archivo de proyecto (.mcp), para luego generar un archivo .hex compilando solo los archivos necesarios. Luego, la aplicación también realiza un parseo de linker script del proyecto y del archivo .hex generado. Eso se realiza para grabar solamente algunas secciones del *USB4all firmware*, y no sobreescribir todo lo que se encontraba en el microcontrolador. Si se encapsula todo el proceso de generación de código visto en la sección anterior en la clase USBDescConf solo son necesarias las siguientes tres lineas para todo el proceso:

```
USBDescConf ucfg = new USBDescConf ();
ucfg.usbDescGenerate("C:/usb4all/fw/Demo/usb4all/usbconf/");
ucfg.compileProject("C:/usb4all/fw/Demo/USB4all.mcp");
ucfg.sendBootLoader("C:/usb4all/fw/Demo/USB4all.hex");
```

La primer linea instancia un objeto de la clase USBDescConf, la segunda setea donde se generan los archivos fuente para la configuración de descriptores, la tercera compila el proyecto y genera un .hex y finalmente la última linea graba las secciones necesarias en el micrcontrolador a través del bootloader. Vale la pena destacar que solo se realiza un ciclo de erase-write en la memoria FLASH del microcontrolador, cuando se detectan cambios en las secciones necesarias.

2.3. Carga de user modules y proxies en el baseboard

Debido a que el archivo del proyecto es parseado en busca de todos los fuentes, agregar user module o proxy es tan sencillo como agregarlos en el proyecto del MPLAB IDE. También se pensó el agregado a través de la aplicación *USB4all baseboard utility*, pero finalmente no se implementó, ya que no se consideró prioritario y esta funcionalidad puede ser alcanzada utilizando el proyecto de MPLAB IDE.

2.4. Programación de firmware vía MPLAB IDE

El proyecto que se brinda a modo de template es 100 % funcional y es posible compilar a través del MPLAB IDE. Desde este entorno se pueden realizar en todos los módulos del microcontrolador, incluyendo el *base firmware, user modules, y proxies.* Este es el ambiente

Algoritmo 1 Generación de descriptores con *USB4all baseboard utility*

```
DeviceDescriptor dd = new DeviceDescriptor(2,0,0,0,8,0x04D8,0x000C,0);
//genero string descriptors
StringDescriptor sd0 = new StringDescriptor(0x0409);
StringDescriptor sd1 = new StringDescriptor("Aguirre Fernandez Grossy");
StringDescriptor sd2 = new StringDescriptor("USB4all (C)2007");
StringDescriptor sd3 = new StringDescriptor("12345678");
//los asigno al device descriptor
dd.addStringDescriptor(sd0);
dd.addStringDescriptor(sd1);
dd.addStringDescriptor(sd2);
dd.addStringDescriptor(sd3);
//asigno strings de manofacturer y product
dd.setManufacturerString(sd1);
dd.setProductString(sd2);
dd.setDeviceSerialNumberString(sd3);
//creo configuracion 1
Configuration cfg1 = new Configuration(dd,50);
//creo endpoints
Endpoint ep1out = new Endpoint(1, Endpoint.OUT,Endpoint.INTERRUPT, 64, 32);
Endpoint ep1in = new Endpoint(1, Endpoint.IN,Endpoint.INTERRUPT, 64, 32);
Endpoint ep2out = new Endpoint(2, Endpoint.OUT,Endpoint.BULK, 64, 32);
Endpoint ep2in = new Endpoint(2, Endpoint.IN,Endpoint.BULK, 64, 32);
Endpoint ep3out = new Endpoint(3, Endpoint.OUT,Endpoint.ISOCHRONOUS, 64, 32);
Endpoint ep3in = new Endpoint(3, Endpoint.IN,Endpoint.ISOCHRONOUS, 64, 32);
//creo interface
Interface i1 = new Interface(0,0,0,0,0);
//le asigno los endpoints a la interface i1
i1.addEndpoint(ep1out);
i1.addEndpoint(ep1in);
i1.addEndpoint(ep2out);
i1.addEndpoint(ep2in);
i1.addEndpoint(ep3out);
i1.addEndpoint(ep3in);
//agrego interfaz i1 a la configuracion cfg1
cfg1.addInterface(i1);
//genero los archivos en la carpeta asignada
Usb4allConfigGenerator cg = new Usb4allConfigGenerator(dd,"C:/usb4all/fw/Demo/usb4all/usbconf/");
cg.generateFiles();
```

recomendado para el desarrollo o modificación de *user modules* y *proxies*. Luego es posible pasar el .hex luego de compilar a través del *USB4all baseboard utility* o a través de una aplicación brindada por Microchip (PICDEM FS USB demo tool) para pasar el firmware, previo seteo del *baseboard* en modo bootloader.

Capítulo 3

Programación en el USB4all Firmware

3.1. User Module Skeleton

A manera de ejemplo se brinda un esqueleto de la estructura general de un *user module,* el cual puede encontrarse en la carpeta de fuentes del firmware con el nombre usr_skeleton.c. En general va a ser necesario cambiar la palabra "`SKEL`" por un nombre que identifique al *user module* .

3.1.1. Variables globales

La variable `usrSkelHandler` va a ser la que guarde el handler asignado al *user module* . La variable `sendBufferUsrSkel` apunta al buffer donde el user module puede colocar los datos que desea sean enviados al PC.

3.1.2. Tabla de referencia a funciones del *user module*

Como fue explicado en la sección correspondiente al *user module* en el documento Informe Final [2], se debe especificar una estructura que almacene las referencias a las operaciones de *user module* que el *base firmware* necesita para poder ejecutar dinámicamente inicialización, configuración y cierre. Esto es codificado como muestra el algoritmo 2, donde se debe sustituir `UserSkelInit` por el nombre de la función que implementa la inicialización del *user module*, `UserSkelRelease` por el nombre de la función que implementa la liberación de los recursos, `UserSkelConfigure` por el nombre de la función que implementa la configuración propia de del *user module*, así como los recursos hardware que va a utilizar; por ejemplo si se tiene un *user module* que controla un motor, se puede especificar que bits y que puerto el motor va a utilizar. Esto puede ser cambiado dinámicamente durante la ejecución del *user module*. También es recomendable cambiar `userSkelModuleTable` por un nombre que identifique a esta estructura en el *user module*. Otro cambio muy importante es cambiar `modName` por el nombre que el *user module* va a utilizar para identificarse. Este nombre no puede excederse en los 7 caracteres.

Algoritmo 2 Estructura con referencias a operaciones de *user modules*

```
#pragma romdata user
   uTab userSkelModuleTable =
{&UserSkelInit,&UserSkelRelease,&UserSkelConfigure,"modName"};
#pragma code
```

3.1.3. Función Init

En el algoritmo 3, puede verse la función `UserSkelInit`, la cual es un bosquejo de lo que la función `init` debe hacer.

Sus principales responsabilidades son:

- Almacenar el número de *handler* que el *user module* va a usar para comunicarse.
- Registrar la función encargada de antender los datos recibidos (`UserSkelReceived`) mediante la invocacion de la función `setHandlerReceiveFunction` del *handler manager*.
- Registrar la función que se desea utilizar para realizar la entrada/salida o el procesamiento periodico (`UserSkelProcessIO`) en alguno de los mecanismos de entrada salida disponibles o en los *proxies*.
- Obtener el buffer para envío de información hacia el PC mediante la función `getSharedBuffer`.

Algoritmo 3 Función `UserSkelInit`

```
void UserSkelInit(byte i){
    BOOL res;
    usrSkelHandler = i;
    setHandlerReceiveFunction(usrSkelHandler,&UserSkelReceived);
    res = addPollingFunction(&UserSkelProcessIO);
    sendBufferUsrSkel = getSharedBuffer(usrSkelHandler);
}//end UserSkelInit
```

3.1.4. Función Release

Las responsabilidades básicas de esta función son:

- Liberar el buffer que se utiliza para enviar datos a el PC mediante la operación `unsetHandlerReceiveBuffer`
- Desregistrar la función que se encarga de atender los datos que llegan desde el PC al *user module* mediante la operación `unsetHandlerReceiveFunction`
- Desregistrar la función que se encarga de realizar la entrada salida mediante la operación `removePoolingFunction` si se utiliza pooling o `removeISRFunction` si se utilizan interrupciones o la operacion adecuada de un `proxy`.

Puede verse un ejemplo de esta función en el algoritmo 4.

Algoritmo 4 Función `UserSkelRelease`

```
void UserSkelRelease(byte i){
    unsetHandlerReceiveBuffer(i);
    unsetHandlerReceiveFunction(i);
    removePoolingFunction(&UserSkelProcessIO);
}
```

3.1.5. Función Received

Las responsabilidades de esta función son la de implementar el protocolo que se va a mantener entre el *user module* y la aplicación en el PC, el cual es llamado *user protocol* [2]. Su comportamiento consiste en tomar datos del buffer de entrada e interpretados según el *user protocol*. En el algoritomo 5, puede verse un caso representativo, es interesante notar que debe setearse un contador con la cantidad de bytes que van a escribirse en el buffer de salida. Luego se invoca a `USBWrite` para enviar lo almacenado en el buffer de salida.

Algoritmo 5 Función `UserSkelReceived`

```
void UserSkelReceived(byte* recBuffPtr, byte len){
byte index; char mens[9] = "User Skeleton is alive";
byte userSkelCounter = 0;
switch(((SKEL_DATA_PACKET*)recBuffPtr)->CMD) {
    case READ_VERSION:
        ((SKEL_DATA_PACKET*)sendBufferUsrSkel)->_byte[0] =
((SKEL_DATA_PACKET*)recBuffPtr)->_byte[0];
        ((SKEL_DATA_PACKET*)sendBufferUsrSkel)->_byte[1] =
(SKEL_DATA_PACKET*)recBuffPtr)->_byte[1];
((SKEL_DATA_PACKET*)sendBufferUsrSkel)->_byte[2] = SKEL_MINOR_VERSION;
((SKEL_DATA_PACKET*)sendBufferUsrSkel)->_byte[3] = SKEL_MAJOR_VERSION;
userSkelCounter=0x04
    break;
    case RESET: Reset();
    break;
    case MESS:
        sendMes(mens, sizeof(mens));
    break;
    default:
    break;
}//end switch(s)
if(userSkelCounter != 0) {
    if(!mUSBGenTxIsBusy())
        USBGenWrite2(usrSkelHandler, userSkelCounter);
}//end if
}//end UserSkelReceived
```

3.1.6. Función ProcessIO

Las tareas de esta función, son fundamentalmente la de realizar entrada y salida, también puede usarse para que el *user module* pueda obtener tiempo de CPU realizando tareas que involucren algun procesamiento.

3.2. Proxies

3.2.1. Interrupt proxy

Los *interrupt proxies* son particularmente útiles cuando deben notificarse eventos disparados mediante interrupciones, como suele suceder con módulos de timer, puertos seriales, etc. De esta manera puede implementarse un proxy para un Timer del microcontrolador, y permitir a varios *user modules* sean notificados del evento overflow del registro contador del Timer. Para la interacción con el resto del firmware, el *interrupt proxy* debe implementar la interfaz mostrada en el algoritmo 6.

Algoritmo 6 Interfaz del *interrupt proxy*

```
BOOL addFunction(volatile void (*ISRFun) (void));
BOOL removeFunction(volatile void (*ISRFun) (void));
void initFunctions(void);
void config(void);
void Interrupt(void);
```

A modo de ejemplo se muestra la implementación de un *interrupt proxy* para el uso del Timer:

Lo primero que se realizar es declarar las variables en la sección adecuada (udata) con el siguiente código:

```
#pragma udata
```

```
volatile void ( *t0Function[MAX_T0_FUNCTIONS]) (void);
volatile byte t0Listeners;
volatile byte t0LoadHigh;
volatile byte t0LoadLow;
```

Lo primero que se declara es un arreglo de puntero a funciones, que va a servir para llamar a todos los *user modules* registrados en el proxy. Luego una variable de un byte para contar la cantidad de listeners registrados, y finalmente 2 variables de un byte para guardar la constante de tiempo a cargar en el timer, en parte alta y baja respectivamente

Observación: todas las variables son definidas con el modificador *volatile* y eso le indica al compilador que este tipo de variables puede modificarse durante una interrupción. En las siguientes subsecciones se muestra la implementación de cada una de las funciones que debe implementar la interfaz.

3.2.1.1. Init

Esta es la función que inicializa las variables de arreglo de punteros a funciones y cantidad de user modules registrados.

```
void initT0Functions(void){
    byte i;
    for (i=0; i<MAX_T0_FUNCTIONS;i++){
        t0Function[i] = 0; //supongo 0 = null
    }
    t0Listeners=0;
}
```

3.2.1.2. addFunction

Aquí se se implementa la función que permite registrarse a los *user modules*.

```
BOOL addT0Function(volatile void (*ISRFun) (void)){
    byte i = 0;
    BOOL termine = FALSE;
    while (i<MAX_T0_FUNCTIONS && !termine){
        if ( t0Function[i] == 0) {
            termine = TRUE;
            t0Function[i] = ISRFun;
        }
    i++;
    }
    if (!termine) return FALSE;
    if ((t0Listeners++)==0){
        addISRFunction(&t0Interrupt);
        INTCONbits.TMR0IF = 0; //apago bandera de interrupcion de timer0
        //cuando se agrega la primer funcion listener prendo ints de Timer0
        INTCONbits.TMR0IE = 1;
        //cargo valores del timer
        TMR0L = t0LoadLow;
        TMR0H = t0LoadHigh;
        T0CONbits.TMR0ON = TRUE; //prendo timer0
    }
    return TRUE;
}
```

Aquí lo que se realiza, es colocar el puntero a función en el arreglo, luego, en el caso que sea el primer *user module* en registrarse, el *proxy* se agrega al *dynamic ISR*. Además también apaga la bandera del evento, y levanta la bandera de la interrupción a esperar (la del Timer0). Por último se enciende el timer, y la función retorna.

3.2.1.3. removeFunction

Aquí se muestra la función para desregistrar un *user module* que ya no desea utilizar el mecanismo de interrupciones.

```
BOOL removeT0Function(volatile void (*ISRFun) (void)){
    byte i=0;
    BOOL termine=FALSE;
    while (i<MAX_T0_FUNCTIONS && !termine){
        if ( t0Function[i] == ISRFun) {
            termine = TRUE;
            t0Function [i] = 0;
            }
        i++;
        }
    if (!termine) return FALSE;
    if ((--t0Listeners)==0) {
        T0CONbits.TMR0ON = FALSE; //apago el timer
        INTCONbits.TMR0IF = 0; //apago la bandera del timer
        INTCONbits.TMR0IE = 0; //apago la la generación de int del timer
        removeISRFunction(&t0Interrupt);
    }
    return TRUE;
}
```

La función tiene una estructura análoga a la de `addFuction` la pero el mecanismo implementado es para que un *user module* se desregistre del *proxy*. De esta forma, al desregistrarse una función se elimina la referencia de la misma y se decrementa el contador de *user modules* registrados. Luego de ello, si el *user module* es el último en desregistrarse, entonces apaga el timer, la bandera del timer, y la generación de interrupciones por parte del timer. Finalmente el proxy es desregistrado del *dynamic ISR*.

3.2.1.4. config

Esta es una función que configura el contador a recargar en el timer, cada vez que haya una interrupción. En este caso es una función sin parámetros y carga unos valores "por defecto". En caso de ser necesario, pueden implementarse tantos como sea necesarios, con los parámetros requeridos y éstos son invocados desde los *user modules* cuando sean necesarios.

```
void configT0(void){
    T0CON = 0x03;
    t0LoadLow = 5;
    t0LoadHigh = 5;
}
```

3.2.1.5. interrupt

Esta función es invocada mediante un notify por parte del dynamic ISR y tiene las siguientes responsabilidades:

1. Verificar que es la interrupcion adecuada para el proxy
2. En caso de serla, llama a todos los user modules interesados previamente registrados
3. Apagar la bandera que causó la interrupción.

Para la realización de estas responsabilidades, en general siguen el siguiente pseudocódigo mostrado en el algoritmo 7.

Para el caso del timer, de forma adicional debe realizar otra operación. Esta es la recarga del contador del timer en el valor configurado. A continuación se muestra el código para el timer.

```
void t0Interrupt(void){
    volatile byte i=0;
    if (INTCONbits.TMR0IF){
```

Algoritmo 7 Pseudocódigo del algoritmo que implementan los *interrupt proxies*

```
interrupt():
    Si es la interrupción esperada:
        Para cada función de callback F registrada en la colección
            Invoca F
        Apaga las banderas para la interrupción procesada.
    Retorna
```

```
            //si hubo interrupcion del timer0 apago la bandera de interrupción
            //y hago notify a todos los observers
            INTCONbits.TMR0IF = 0;
            //vuelvo a setear el timer
            TMR0L = t0LoadLow;
            TMR0H = t0LoadHigh;
            while (i<MAX_T0_FUNCTIONS){
                if(t0Function[i] != 0){
                    t0Function[i]();
                }
            i++;
            }
        }
    }
```

3.2.2. Polling Proxies

Los *polling proxies* son útiles cuando se quiere esperar por un evento realizando polling sobre algún módulo de hardware. Deben implementar una interfaz similar a la de los *interrupt proxies*, pero la gran diferencia es que los *polling proxies* son llamados mediante la operación `polling` desde *dynamic polling*. De esta forma periódicamente es ejecutada la operación `polling`, la que a su vez invoca a todos los *user modules* registrados para ese evento. El resto de las interacciones son análogas a las descriptas en *interrupt proxy*. Para la interacción con el resto del *base firmware*, el *polling proxy* debe implementar la interfaz mostrada la figura 8.

Algoritmo 8 Interfaz que deben implementar los *polling proxies*.

```
BOOL addFunction(void (*DPFun) (void));
BOOL removeFunction(void (*DPFun) (void));
void initFunctions(void);
void config(void);
void polling(void);
```

En las siguientes secciones se muestra la implementación de cada una de las operaciones de la interfaz, pero esta ves implementado utilizando un *polling proxy*.

Lo primero que se realizar es declarar las variables en la sección adecuada (udata) con el siguiente código:

```
#pragma udata
void ( *t0Function[MAX_T0_FUNCTIONS]) (void);
byte t0Listeners;
byte t0LoadHigh;
byte t0LoadLow;
```

Lo primero que se declara es un arreglo de puntero a funciones, que va a servir para llamar a todos los *user modules* registrados en el *proxy*. Luego una variable de un byte para contar la cantidad de listeners registrados, y finalmente 2 variables de un byte para guardar la constante de tiempo a cargar en el timer, en parte alta y baja respectivamente

3.2.2.1. Init

Esta es la función que inicializa las variables de arreglo de punteros a funciones y cantidad de *user modules* registrados.

```
void initT0Functions(void){
    byte i;
    for (i=0; i<MAX_T0_FUNCTIONS;i++){
        t0Function[i] = 0; //supongo 0 = null
    }
    t0Listeners=0;
}
```

3.2.2.2. addFunction

Aquí se se implementa la función que permite registrarse a los *user modules*.

```
BOOL addT0Function(void (*DPFun) (void)){
    byte i = 0;
    BOOL termine = FALSE;
    while (i<MAX_T0_FUNCTIONS && !termine){
        if ( t0Function[i] == 0) {
            termine = TRUE;
            t0Function[i] = DPFun;
        }
    i++;
    }
    if (!termine) return FALSE;
    if ((t0Listeners++)==0){
        addPollingFunction(&t0polling);
        INTCONbits.TMR0IF = 0; //apago bandera del timer0
        //cargo valores del timer
        TMR0L = t0LoadLow;
        TMR0H = t0LoadHigh;
        T0CONbits.TMR0ON = TRUE; //prendo timer0
    }
    return TRUE;
}
```

Aquí lo que se realiza, es colocar el puntero a función en el arreglo, luego, en el caso que sea el primer *user module* en registrarse, el *proxy* se agrega al *dynamicPolling*. Además también enciende la bandera del evento (la del Timer0). Por último se enciende el timer, y la función retorna.

3.2.2.3. removeFunction

Aquí se muestra la función para desregistrar un *user module* que ya no desea utilizar el mecanismo de interrupciones.

```
BOOL removeT0Function(void (*DPFun) (void)){
    byte i=0;
    BOOL termine=FALSE;
    while (i<MAX_T0_FUNCTIONS && !termine){
        if ( t0Function[i] == DPFun) {
            termine = TRUE;
            t0Function [i] = 0;
            }
        i++;
        }
    if (!termine) return FALSE;
    if ((--t0Listeners)==0) {
        T0CONbits.TMR0ON = FALSE; //apago el timer
        INTCONbits.TMR0IF = 0; //apago la bandera del timer
        removePollingFunction(&t0polling);
    }
    return TRUE;
}
```

La función tiene una estructura análoga a la de `addFuction` la pero el mecanismo implementado es para que un *user module* se desregistre del *proxy*. De esta forma, al desregistrarse una función se elimina la referencia de la misma y se decrementa el contador de *user modules* registrados. Luego de ello, si el *user module* es el último en desregistrarse, entonces apaga el timer y la bandera del timer. Finalmente el proxy es desregistrado del *dynamicPolling*.

3.2.2.4. config

Esta es una función que configura el contador a recargar en el timer, cada vez que haya una interrupción. En este caso es una función sin parámetros y carga unos valores "por defecto". En caso de ser necesario, pueden implementarse tantos como sea necesarios, con los parámetros requeridos y éstos son invocados desde los *user modules* cuando sean necesarios.

```
void configT0(void){
    T0CON = 0x03;
    t0LoadLow = 5;
    t0LoadHigh = 5;
}
```

3.2.2.5. polling

Esta función es invocada mediante un notify por parte del *dynamicPolling* y tiene las siguientes responsabilidades:

1. Verificar si se encendió la bandera adecuada para el proxy
2. En caso de encenderse, llama a todos los *user modules* interesados (previamente registrados)
3. Apagar la bandera del evento.

Para la realización de estas responsabilidades, en general siguen el siguiente pseudocódigo mostrado en el algoritmo 9.

Algoritmo 9 Pseudocódigo del algoritmo que implementan los *Polling proxies*

```
polling():
    Si se encendío la bandera del evento esperado:
        Para cada función de callback F registrada en la colección
            Invoca F
        Apaga las bandera del evento procesado.
    Retorna
```

Para el caso del timer, de forma adicional debe realizar otra operación. Esta es la recarga del contador del timer en el valor configurado. A continuación se muestra el código para el timer.

```
void t0polling(void){
    volatile byte i=0;
    if (INTCONbits.TMR0IF){
        //si se prendio la bandera del timer0
        //hago notify a todos los observers
        INTCONbits.TMR0IF = 0;
        //vuelvo a setear el timer
        TMR0L = t0LoadLow;
        TMR0H = t0LoadHigh;
        while (i<MAX_T0_FUNCTIONS){
            if(t0Function[i] != 0){
                t0Function[i]();
            }
        i++;
        }
    }
}
```

Capítulo 4

Programación en el PC

4.1. USB4all API

En las siguientes secciones se mostrara por medio de ejemplos, los distintos elementos que un programador necesitara para utilizar la USB4all API o extender su uso para un driver USB genérico particular o para que la API corra en otra plataforma.

4.1.1. Interfaz Pública

El programa principal del algoritmo 10 es un ejemplo de un programa de aplicación que utiliza la *USB4all API* para comunicarse con un dispositivo. En este caso, el programa realiza 500 tomas del valor de la temperatura y para ello establece un vínculo lógico con un *user module* que maneja un sensor de temperatura.

La estructura del programa es muy simple y comienza por inicializar la *USB4all API* para su uso por medio de la invocación de la operación `initAPI`. Nótese que en la invocación de la función se utiliza el puntero `oTemp`, que no fue instanciado previamente. Esto se debe a que la *USB4all API* implementa el patrón de diseño Singleton el cual define que no es necesario una instanciación implícita del componente. Una vez inicializada la *API*, se establece un vínculo lógico con el *user module* de nombre "temp" existente en el *baseboard* de número de serie: 11111111 por medio de la operación `openDevice`. Si la operación de apertura es exitosa, se obtiene el identificador (moduleID) del vínculo.

Luego, se define un arreglo de bytes con la orden de petición de temperatura que el *user module* entiende y se entra en un ciclo de 500 iteraciones donde se envía al *user module*, por medio de la operación `sendData` la petición y se espera la recepción de la respuesta de la misma (`receiveData`), para imprimir en la salida estándar del programa el valor de la temperatura muestreado. Finalmente, al salir del ciclo de iteraciones se procede a cerrar el vínculo lógico con el *user module* por medio de la operación `closeDevice` y termina el programa.

Este ejemplo es simple y apunta a que el lector pueda entender fácilmente el uso de la *USB4all API*, es por ello que no se incluyeron en este ejemplo el resto de las operaciones que posee la *API*. Para más detalle de las mismas, leer la sección *USB4all API* del documento Informe Final [2].

Algoritmo 10 Ejemplo de uso de la *USB4all API*

```
int main(void) {

    u4aapi *oTemp;
    char *dCom1, *dRes1;
    int h1, rawval, rawtemp;
    float degCtemp;

    oTemp->initAPI();
    printf("Opening thermometer...\n");
    h1 = oTemp->openDevice(11111111, "temp", 1, 6);
    if (h1!=255){
        printf("Assigned Id:%d\r\n", h1);
    } else {
        printf("A error ocurred when the thermometer was opened.\r\n");
        exit(1);
    };
    dCom1 = new char[2];
    dCom1[0] = (char) 0x34;
    dCom1[1] = (char) 0x02;
    for (int cant = 0 ; cant <500 ; cant++) {
        if (oTemp->sendData(h1, dCom1, 2,3000)) {
            dRes1 = new char[4];
            oTemp->receiveData(h1,dRes1,3,3000);
            rawval = dRes1[1] + (dRes1[2]<<8);
            if(rawval & (1 <<15)) rawval -= 1 <<16;
            rawtemp = (rawval)>>3;
            degCtemp = (float) (rawtemp * 0.0625);
            printf("The temperature is%f degC (raw:%i, rawval:%i)\n", degCtemp, rawtemp, rawval);
            delete [] dRes1;
        } else {
            printf("A error ocurred when temperature was readed.\r\n");
        };
    };
    delete [] dCom1;
    printf("Closing thermometer...\n");
    if (oTemp->closeDevice(h1)) {
        printf("Closed thermometer.\r\n");
    } else {
        printf("A error ocurred when the thermometer was closed.\r\n");
        exit(1);
    };
    exit(0);
}
```

4.1.2. Soporte de drivers de la USB4all API

La *USB4all API* fue diseñada para poder ser extendida en cuanto a su interacción con distintas implementaciones de drivers USB genéricos. En esta sección mostraremos como ejemplo, algunas trozos de código de implementación realizada para la utilización del driver USB genérico de Microchip desde la *USB4all API.*

El punto de extensión que posee la API para al interacción con los drivers, es la clase abstracta iDriverLayer. La clase que se implementó para el driver de Microchip se llama *DriverMCHP* y hereda de la clase *iDriverLayer*. Como parte de la interfaz que define esta clase abstracta, se encuentran algunas de las siguientes operaciones: `openIn`, `openOut`, `close`, `sendInt`, `sendBulk`, `sendCtrl`, `sendIso`, `receiveInt`, `receiveBulk`, `receiveIso`, `receiveCtrl`. Este conjunto de operaciones son las más importantes a la hora de extender la *API*, pues encapsulan el manejo del driver en particular. Existen dos operaciones `open` para tener separado la lógica necesaria para abrir un endpoint entrante o saliente, lo mismo sucede con las operaciones `send` y `receive` para las cuales existe una variante por cada tipo de transferencia USB.

Para poder utilizar una implementación de un driver USB genérico especifico desde la *API*,

Algoritmo 11 Código fuente del método `sendInt` de clase *DriverMCHP*.

```
bool DriverMCHP::sendInt(int buffer, char *msg, int len, int timeout) {

    bool ok;
    char temp[MAX_LEN_MSG];
    DWORD tsend;

    sprintf(temp,"DriverLayer ==>SendInt, Endpoint = %d",buffer);
    l->printLog(temp, true, true);
    l->printLog("Message =",false, true);
    for (int i=0; i<len; i++) {
        sprintf(temp," %X ",msg[i]);
        l->printLog(temp, false, false);
    };
    l->printLogLn();
    ok = (bool) MPUSBWrite((HANDLE) buffer, msg, len, &tsend, timeout);
    sprintf(temp,"Se enviaron %d bytes.",(int) tsend);
    l->printLog(temp, true, true);
    l->printLogLn();
    return ok;
}
```

Parámetro	Descripción
`_DRIVER_MICROCHIP`	Para indicar que se debe usar el driver modo kernel de Microchip.
`_DRIVER_LIBUSBWIN32`	Para indicar que se debe usar el driver modo usuario LibUSBWin32.
`_DRIVER_LIBUSB`	Para indicar que se debe usar el driver modo usuario LibUSB.
`_DRIVER_CUSTOM`	Para indicar que se debe usar el driver modo kernel construido en este proyecto.

Cuadro 4.1: Parámetros de compilación de la API para indicar el driver a utilizar.

es necesario generar (.dll o .so) el proyecto de la *API* indicandole como parámetro que driver debe utilizar. Los parámetros que se utilizaron en este proyecto para reconocer las distintas implementaciones de drivers utilizados se pueden observar en el cuadro 4.1.

En el caso de quere utilizar otra implementación de driver distinta a las que se utilizaron en este proyecto, es necesario editar el archivo `Conf.h` que se encuentra en la estructura de códigos fuentes del proyecto del *USB4all API* y agregar una nueva sección donde se redefinan para esta nueva implementación ciertas estructuras que usa la *API*, así como la redefinición de ciertas banderas de precompilación. Por ejemplo, para indicar cual es la clase que implementa al componente *DriverLayer* de la API y por lo tanto que implementación de driver se quiere usar, se escribe en el archivo `Conf.h` la siguiente línea:

```
#define _DRIVER DriverMCHP
```

4.1.3. Soporte de plataformas de la USB4all API

Otra forma de extensión que tiene la *USB4all API* contempla el cambio de plataforma donde corre. Para ello se debe extender la clase abstracta *iPlatformLayer* y reescribir sus métodos `findDevices` y `getDescriptors`, con la lógica específica para la detección de los *baseboard* conectados al PC y la obtención de sus descriptores para cada plataforma en particular. En general, no se dan más lineamientos que estas dos operaciones pues cada plataforma posee particularidades diferentes.

Al igual que para el soporte de dirver, se debe compilar el proyecto de la *API* pasando parámetros para indicar que plataforma debe usarse. Los parámetros usados en este proyecto

Algoritmo 12 Sección de la plataforma Linux en el archivo `Conf.h` de la *API*

```
#ifdef _PLATFORM_LINUX
   // Exportación/Importación de simbolos
   #define _IMPEXP_USB4ALLAPI ""
   // Selección del driver
   #ifdef _DRIVER_LIBUSB
      #define _DRIVER_ "DriverLayerLibUsb.h"
      #define _DRIVER DriverLayerLibUsb
   #endif
   #ifdef _DRIVER_CUSTOM
      #define _DRIVER_ "DriverLayerCustom.h"
      #define _DRIVER DriverLayerCustom
   #endif
   // Representación de los descriptores del Baseboard.
   // Clase que implementa la lógica de la plataforma.
   #define _iPLATFORMLAYER_ "PlatformWin32.h"
   #define _iPLATFORMLAYER PlatformWin32
#endif
```

son: `_PLATFORM_WIN32` y `_PLATFORM_LINUX`. En el caso de querer usar otra plataforma distinta a Windows y Linux, se debe editar el archivo `Conf.h` y agregar una nueva sección donde se define el nombre del nuevo parámetro y donde se almacenan todas las entradas de las distintas implementaciones de drivers que soporta dicha plataforma. Por ejemplo para la plataforma Linux el archivo `Conf.h` posee la sección que se ven en el algoritmo 12.

4.2. Driver para Linux propio

El driver desarrollado es genérico, esto implica que no está diseñado para ninguna de las clases de la jerarquía específica de dispositivos USB [5], sino que tiene como objetivo soportar todos los tipos de transferencias (Control, Bulk, Interrupt e Isochronous) así como el máximo número de endpoints (32) en forma abierta sin implementación de lógica para un dispositivo particular.

Dado que la *USB4all API* encapsula en el PC la implementación de los protocolo de comunicación que define la solución lleva a que el driver sólo sea responsable de la comunicación e intercambio de información de las características del dispositivo (número de serie, descriptores) así como del intercambio de datos. Esta característica de la solución permite reutilizar el código que implementan los protocolos de comunicación y evita tener que implementarlos directamente en el driver, ya que son dificiles de depurar (ejecutan en modo núcleo). Además otro de los beneficios que trae el no incluir la lógica de los protocolos en los drivers, es que se pueden sustituir por distintas implementaciones o incluso cambiar la plataforma que se utiliza con un impacto mínimo en la solución.

La implementación del proyecto comenzó utilizando el driver genérico que ofrece gratuitamente el fabricante del microcontrolador (Microchip [4]) pues se ya se había utilizado en las pruebas del mismo durante el relevamiento del estado del arte. La utilización de este driver como parte de la solución no permite satisfacer totalmente los objetivos planteados pues sólo funciona en la plataforma Windows y no en Linux, por lo tanto se tuvo que buscar alguna otra opción para hacer funcionar el dispositivo bajo este sistema operativo.

Una de las opciones era enumerarse como alguna clase definida para la cual Linux implemente un driver, como ser HID o CDC, pero si se toma esta opción se termina atado a determinado tipo de transferencia y cantidad de endpoints (los definidos por las interfaces) lo cual es algo que se pretende evitar dado que uno de los principales objetivos del proyecto es que la solución debe ser lo más genérica posible. La siguiente opción fue utilizar LibUsb [3] la cual parecía una muy buena opción ya que contaba con implementaciones para diferentes sistemas operativos (Linux, FreeBSD, NetBSD, OpenBSD, Darwin, MacOS X, Windows). Se realizaron pruebas con LibUsb y se lograron buenos resultados pero sólo permitía utilizar tipos de transferencias de control y bulk y no transferencias interrupt e isochronous ni tampoco transferencias asincrónicas. Dado que la última versión estable es de hace varios años, nos hizo suponer que el proyecto había sido

abandonado, por lo tanto se decidió investigar la posibilidad de realizar un driver propio el cual fue desarrollado como un modulo del kernel de Linux.

Para la realización del driver se tomaron las siguientes decisiones:

- **Un file (/dev/usb4all<instancia>) por *baseboard* conectado:** La otra opción manejada fue tener un file por endpoint el cual simplificaba las transferencias a diferentes endpoints pero dificulta el intercambio de descriptores, ya que no se cuenta con un único punto de acceso. Además, para el usuario siempre sería más difícil contar la cantidad de files existentes que pedir a un único punto de acceso toda la información del dispositivo en estructuras de datos claras, así como simplificar la interacción del driver genérico con la *USB4all API*.
- **Encapsular lo elemental en modo Kernel:** Se coloca sólo lo elemental en el módulo de kernel y se deja para programar en modo usuario (en la *API*) las tareas que pueden resolverse a partir de un conjunto de primitivas básicas exportadas por el driver. De esta manera se puede extender con mayor facilidad y permite depurar con todas las ventajas que se cuentan en el modo usuario.

El driver se comunica con las aplicaciones que corren en modo usuario para el intercambio de información del dispositivo mediante llamadas IOCTL [7], las cuales implementan un conjunto de primitivas descriptas en la tabla 4.2.

Operación	Descripción
GET_DEVICE_DESCRIPTOR	Devuelve el descriptor de dispositivo.
GET_ENDPOINT_DESCRIPTOR	Devuelve el descriptor de endpoint.
GET_INTERFACE_DESCRIPTOR	Devuelve el descriptor de interfaz.
GET_CONFIGURATION_DESCRIPTOR	Devuelve el descriptor de configuración.
GET_STRING_DESCRIPTOR	Devuelve el string descriptor.
SET_DESC_INDEX	Setea un índice utilizado para obtener un descriptor determinado de los tipos que poseen mas de uno por tipo.
SET_IN_ENDPOINT	Setea la dirección del endpoint in a utilizar.
SET_OUT_ENDPOINT	Setea la dirección del endpoint out a utilizar.
SET_TRANSFER_TYPE	Setea un tipo de transferencia a utilizar.
SET_TIMEOUT	Setea un tiempo de espera máximo para la llegada de los datos.

Cuadro 4.2: Interfaz para el intercambio de configuración entre el driver y el programa de usuario.

El conjunto de operaciones del tipo *get* permiten obtener los descriptores del dispositivo, de esta manera la *API* puede contar con toda la información necesaria para decidir que tipo de endpoint utilizar o la información de la instancia particular como por ejemplo el número de serie. Por medio de las operaciones de tipo *set* se cuenta con un mecanismo para poder configurar el endpoint a utilizar, tipo de transferencia y tiempo de espera (timeout).

Para utilizar el driver basta con invocar al comando del sistema operativo insmod (insmod usb4all.ko) para la lograr la instalación del mismo y para la desinstalación se procede con el comando rmmod (rmmod usb4all). Luego de ser configurado el driver mediante las IOCTLs, se maneja mediante llamadas del sistema: read y write, sobre el file descriptor obtenido mediante la llamada open al file que corresponde con el *baseboard*. Dada la cualidad de genérico del driver, este puede ser utilizado para cualquier dispositivo USB cambiando el vid y pid necesarios en el código fuente.

Se implementó siguiendo el framework para desarrollo de drivers USB que el kernel implementa [6, 7]. Como podemos ver en la figura 4.1 el primer paso para interactuar con el driver es realizar una llamada open sobre el file definido para el *baseboard*, luego se obtiene el descriptor de interfaz asociado al dispositivo *baseboard* con esa información se determina cuantos endpoints se poseen y se puede pasar a obtener sus descriptores como se muestra en el recuadro de la figura. Una vez obtenidos los descriptores de endpoints se cuenta con la información para poder

setear que endpoint de entrada y salida se va a utilizar. Finalmente el driver se encuentra en un estado donde es posible comenzar a escribir y leer del endpoint seteado mediante las llamadas al sistema read y write. Existen variantes de este caso de uso donde se obtienen otros descriptores como el de device para determinar el número de serie del *baseboard* por ejemplo.

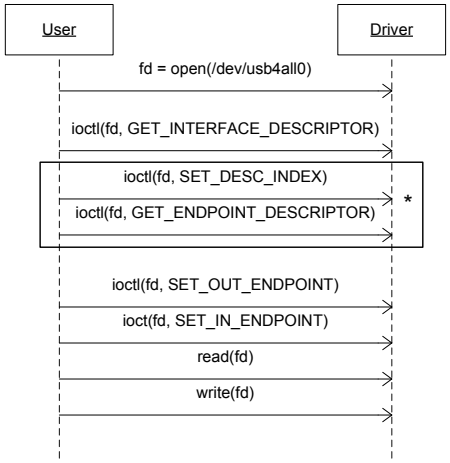

Figura 4.1: Ejemplo de apertura y envió de datos por un endpoint.

Algunas consideraciones que hay que tener en cuenta con el driver son:

- Se debe ser usuario con privilegios de root para instalar el driver
- en algunas versiones del kernel se debe desinstalas el driver ldusb (rmod ldusb) el cual puede generar conflictos con nuestro driver.
- si se desea utilizar para otro dispositivo basta con cambiar las constantes definidas al comienzo del driver VID, PID con los valores del dispositivo a controlar.

Capítulo 5

Esquemáticos y Pin-Outs

5.1. Pin-Out del conector de programación/debugging (RJ11)

En la figura 5.1, se ve la explicación de cada línea del conector RJ11 utilizado en el *baseboard* para la programación y depuración. En el cuadro 5.1, se muestra el significado de cada uno de los pines del conector RJ11.

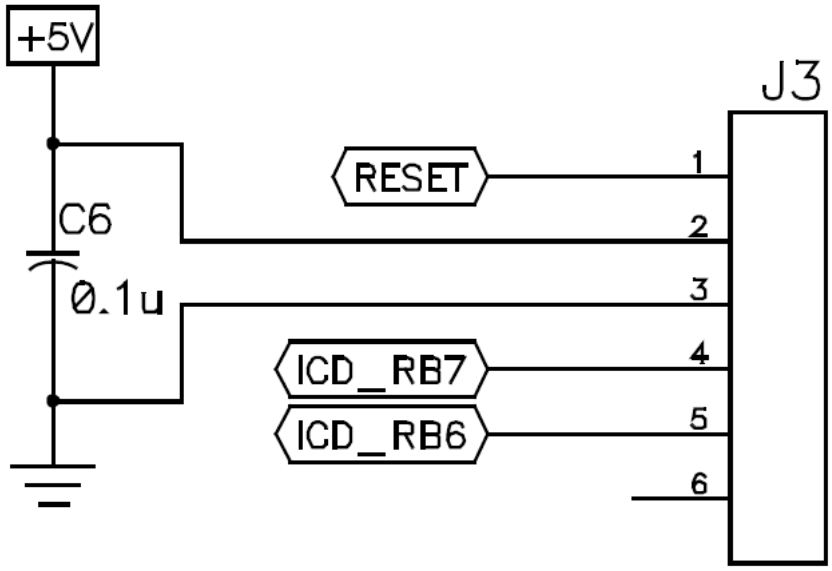

Figura 5.1: Pin-Out del RJ11 del *baseboard*

Pin	Descripción
1	Reset
2	+ 5V
3	GND
4	ICD_RB7
5	ICD_RB8
6	N/A

Cuadro 5.1: Explicación de cada pin del conector RJ11

5.2. Esquemático del Baseboard

En la figura 5.2, se ve el esquemático del circuito eléctrico impreso del *baseboard*.

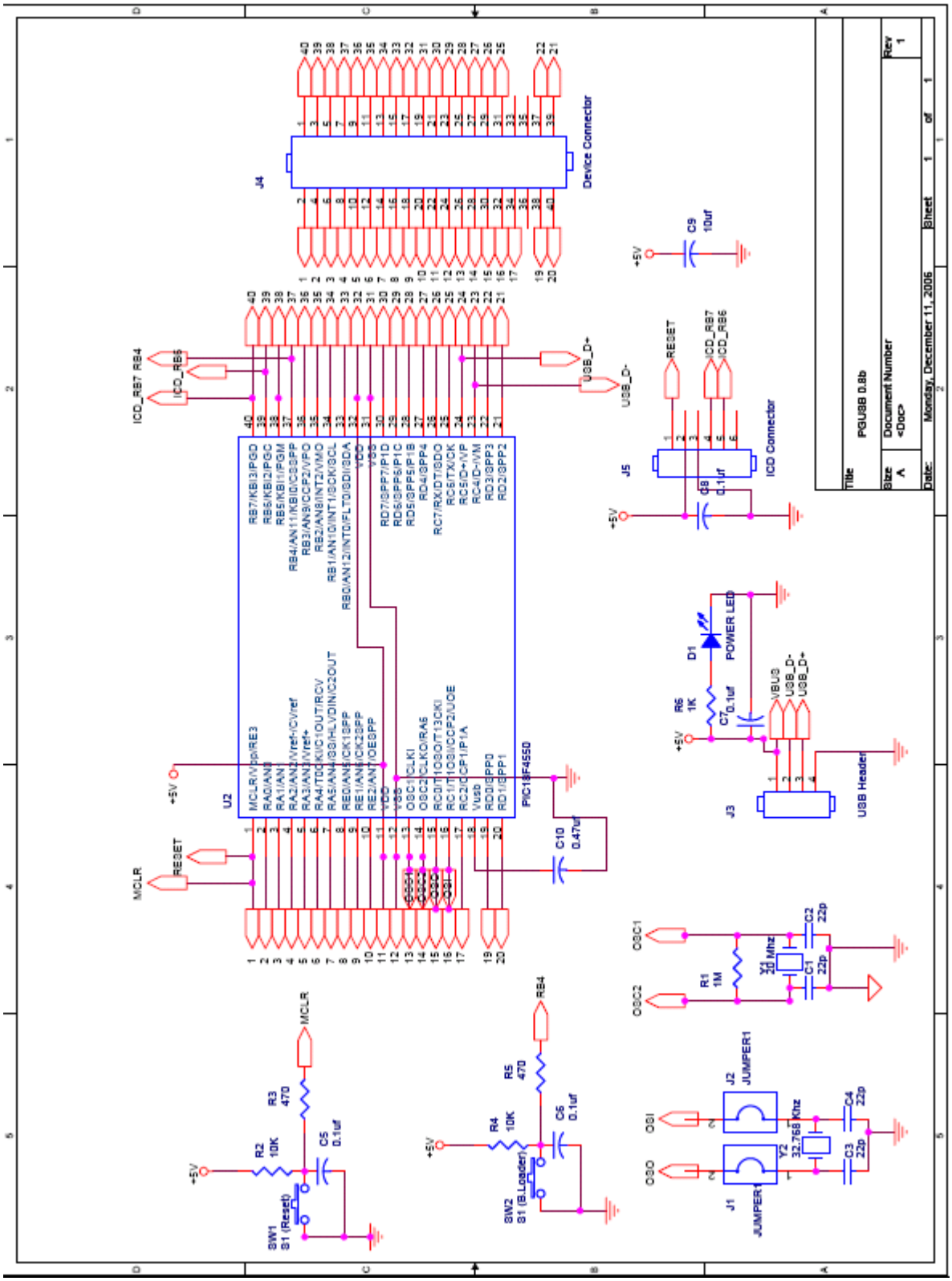

Figura 5.2: Esquemático del *baseboard*.

5.3. Pin-Out del PIC18F4550 de Microchip

En la figura 5.3, se ve la explicación de cada pin del microcontrolador PIC18F4550 que posteriormente se conectan al *U4APort* como muestra la figura 5.2 del esquemático del *baseboard*.

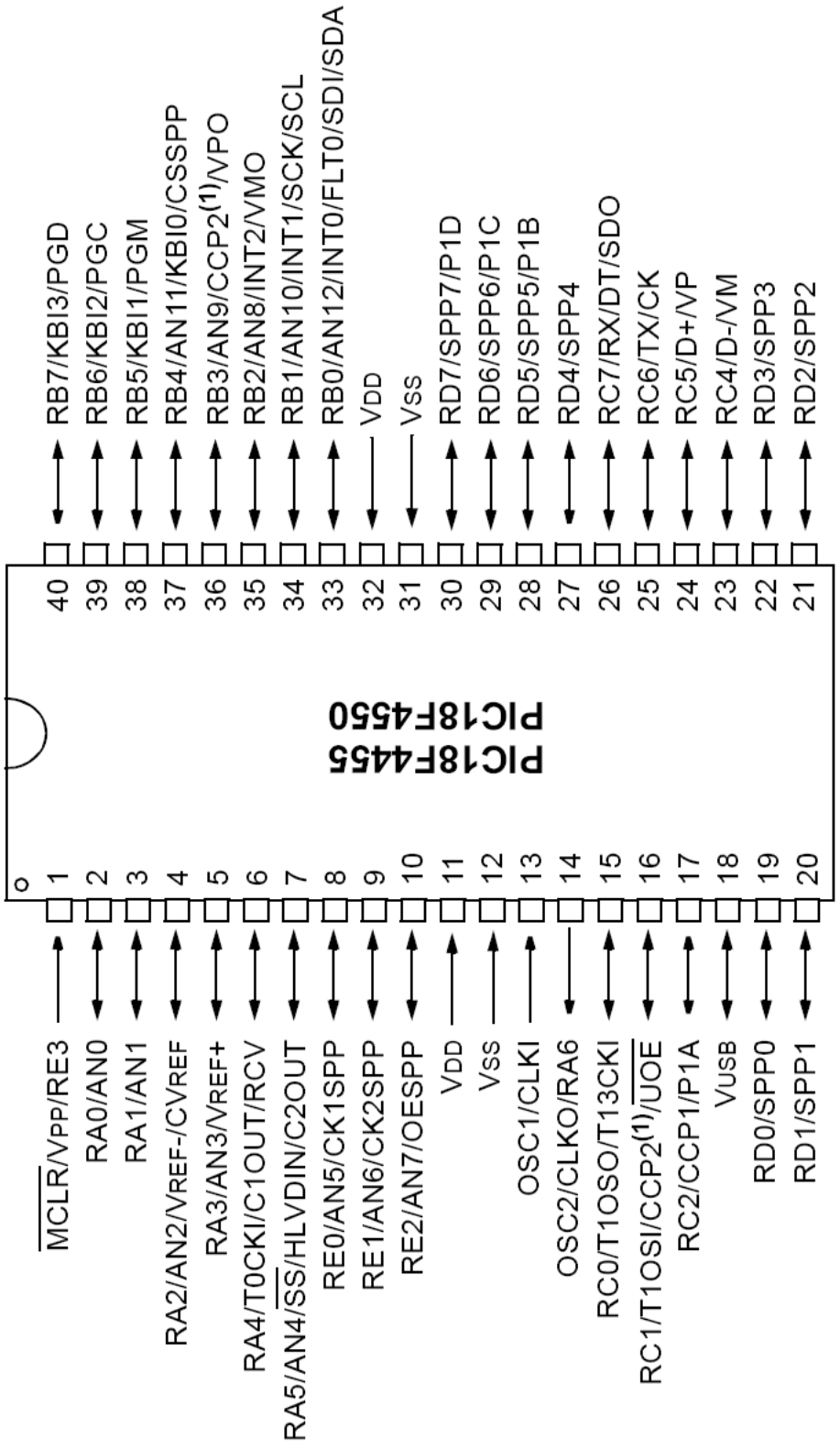

Figura 5.3: Pin-Out del PIC18F4550

Bibliografía

[1] AGUIRRE, Andrés; FERNÁNDEZ, Rafael; GROSSY, Carlos; Estado del Arte; UDELAR - FING, 2007.
(Citado en las páginas 5 y 9.)

[2] AGUIRRE, Andrés; FERNÁNDEZ, Rafael; GROSSY, Carlos; Informe Final; UDELAR - FING, 2007.
(Citado en las páginas 14, 15 y 22.)

[3] LibUSB, http://libusb.sourceforge.net, último acceso octubre 2006.
(Citado en la página 25.)

[4] Microchip Technology Inc., http://www.microchip.com, último acceso octubre 2006.
(Citado en la página 25.)

[5] Estándar USB 2.0, http://www.usb.org/developers, último acceso noviembre 2006.
(Citado en las páginas 8 y 25.)

[6] FLIEGL, Detlef; Programming Guide for Linux USB Device Drivers, Departamento de Informática , Universidad Técnica de Munich, 2000.
(Citado en la página 26.)

[7] CORBET, Jonathan; RUBINI, Alessandro; KROAH-HARTMAN, Greg; Linux Device Drivers, 3rd Edition, O'Reilly Media, 2005.
(Citado en la página 26.)

Apéndice A

Firmas de funciones

A.1. USB4all API

A.1.1. u4aAPI

Algoritmo 13 Firma de los métodos de la clase *u4aAPI*

```
int openDevice(int, char*, int, int);
bool configDevice(int, char*, int);
bool sendData(int, char*, int, int);
bool receiveData(int, char*, int, int);
bool closeDevice(int);
int getBaseBoardSerial(int);
int qtyBaseBoards();
int getUserModuleName(int, int, char*);
int qtyUserModules(int);
void resetBaseBoard(int);
void initAPI();
int apiVersion();
int firmwareVersion(int);
```

A.1.2. iDriverLayer

Algoritmo 14 Firma de los métodos de la clase *iDriverLayer*

```
virtual int* getU4ABoards(int&) =0;
virtual int qtyDsc(int) =0;
virtual dscEndpoint* getEndpointDsc(int, int) =0;
virtual int openIn(int, int, int) =0;
virtual int openOut(int, int, int) =0;
virtual bool close(int) =0;
virtual bool sendInt(int, char *, int, int) =0;
virtual bool sendCtrl(int, char*, int, int) =0;
virtual bool sendIso(int, char*, int, int) = 0;
virtual bool sendBulk(int, char*, int, int) =0;
virtual char* receiveInt(int, int&, bool&, int) =0;
virtual char* receiveCtrl(int, int&, bool&, int) =0;
virtual char* receiveIso(int, int&, bool&, int) =0;
virtual char* receiveBulk(int, int&, bool&, int) =0;
```

A.1.3. iPlatformLayer

Algoritmo 15 Firma de los métodos de la clase *iPlatformLayer*

virtual map<int,itemDsc>`getDescriptors`() =0;
virtual void `findDevices`() =0;

A.2. USB4all Firmware

A.2.1. User Module

Algoritmo 16 Firma de las funciones del *User Module*

void `UserSkelProcessIO`(void);
void `UserSkelInit`(byte);
void `UserSkelReceived`(byte*, byte);
void `UserSkelRelease`(byte);
void `UserSkelConfigure`(void);

A.2.2. Interrupt Proxy

Algoritmo 17 Firma de las funciones del *Interrupt Proxy*

bool `addFunction`(volatile void *);
bool `removeFunction`(volatile void *);
void `initFunctions`(void);
void `config`(void);
void `Interrupt`(void);

A.2.3. Polling Proxy

Algoritmo 18 Firma de las funciones del *Polling Proxy*

bool `addFunction`(void *);
bool `removeFunction`(void *);
void `initFunctions`(void);
void `config`(void);
void `Polling`(void);

A.2.4. DynamicISR

Algoritmo 19 Firma de las funciones del *DynamicISR*

bool `addISRFunction`(volatile void *);
bool `removeISRFunction`(volatile void *);
void `initISRFunctions`(void);
void `interruption`(void);

A.2.5. Dynamic Polling

Algoritmo 20 Firma de las funciones del *Dynamic Polling*

bool `addPollingFunction`(void*);
bool `removePoolingFunction`(void*);
void `initPollingFunctions`(void);
void `polling`(void);

A.2.6. Handler Manager

Algoritmo 21 Firma de las funciones del *Handler Manager*

```
void checkHandlerManagerIO(void);
void USBGenRead2(void);
void USBGenWrite2(byte, byte);
void initHandlerBuffers(void);
void setHandlerReceiveBuffer(byte, byte *);
void setHandlerReceiveFunction(byte, void*);
byte newHandlerTableEntry(byte, rom near char*);
bool existsTableEntry(rom near char*);
void initHandlerTable();
void initHandlerManager(void);
respType removeHandlerTableEntry(byte);
respType configureHandlerTableEntry(byte);
void removeAllOpenModules(void);
void unsetHandlerReceiveBuffer(byte);
void unsetHandlerReceiveFunction(byte);
byte* getSharedBuffer(byte);
byte getEPSizeIN(byte);
byte getEPSizeOUT(byte);
```