UNIVERSIDADE DA CORUÑA

ESCUELA UNIVERSITARIA POLITÉCNICA

Grado en Ingeniería Electrónica Industrial y Automática

TRABAJO FIN DE GRADO

TFG. Nº: **770G01A145**

TÍTULO: **CREACIÓN DE UN PROTOCOLO DE COMUNICACIÓN ENTRE UN PC Y UN PUERTO USB DE UN MICROCONTROLADOR Y SU INTEGRACIÓN EN MATLAB**

AUTOR: CARBALLAS CHAS SERGIO

TUTOR: JOSÉ LUIS CASTELEIRO ROCA

FECHA: **SEPTIEMBRE DE 2018**

Fdo.: EL AUTOR Fdo.: EL TUTOR

TÍTULO: **CREACIÓN DE UN PROTOCOLO DE COMUNICACIÓN ENTRE UN PC Y UN PUERTO USB DE UN MICROCONTROLADOR Y SU INTEGRACIÓN EN MATLAB**

INDICE GENERAL

PETICIONARIO: **ESCUELA UNIVERSITARIA POLITÉCNICA**

AVDA. 19 DE FEBRERO, S/N

15405 - FERROL

FECHA: **SEPTIEMBRE DE 2018**

AUTOR: EL ALUMNO

Fdo.: CARBALLAS CHAS SERGIO

Índice General

Índice de Tablas

Índice de Figuras e imágenes

TÍTULO: **CREACIÓN DE UN PROTOCOLO DE COMUNICACIÓN ENTRE UN PC Y UN PUERTO USB DE UN MICROCONTROLADOR Y SU INTEGRACIÓN EN MATLAB**

MEMORIA

PETICIONARIO: **ESCUELA UNIVERSITARIA POLITÉCNICA**

AVDA. 19 DE FEBRERO, S/N

15405 - FERROL

FECHA: **SEPTIEMBRE DE 2018**

AUTOR: EL AUTOR

Fdo.: CARBALLAS CHAS SERGIO

Índice MEMORIA

2 MEMORIA

2.1 Objeto

Estudio del protocolo USB para implementarlo con un microcontrolador (de la casa Microchip). A continuación se implementará el protocolo para recibir y enviar datos a través del programa Matlab, para poder adquirir y generar señales a modo de tarjeta de adquisición datos.

2.2 Alcance

El presente trabajo abordará los siguientes puntos:

- Estudio del protocolo USB.
- Análisis de las opciones disponibles para comunicar un microcontrolador con un PC.
- Estudio del funcionamiento de los puertos USB en los microcontroladores de la casa Microchip.
- Creación de un protocolo para la comunicación efectiva entre el PC y el microcontrolador a través del USB.
- Creación de los programas de Matlab necesarios para implementar el protocolo y poder adquirir/generar señales desde el programa.
- Comparación de la velocidad de conexión por medio del protocolo creado.

2.3 Antecedentes

El USB es una tecnología muy cercana a nosotros; en nuestro día a día se está usando constantemente: cargar el teléfono móvil, transferir datos a través de los puertos USB del ordenador a un *pendrive*, usar un teclado, un ratón, etc. Esto es

lo que se ve desde el punto de vista de un usuario, un dispositivo que conectamos a un puerto (teclado, ratón, tablet, etc.) y automáticamente el *host* ya lo reconoce, permitiéndole ejecutar su actividad de forma normal (siempre que no se haya producido un error en la conexión); cuando se quiere cesar su actividad simplemente se desconecta del puerto. Pero para poder realizar este Trabajo Final de Grado se deberá profundizar más en lo que es el protocolo USB, sus especificaciones, su arquitectura y cómo es capaz de realizar estas operaciones tan sencillas a ojos del usuario.

En este apartado se establecerán los conceptos teóricos básicos del protocolo USB sobre los que se sustenta el firmware y el programa de Matlab (ver análisis de soluciones).

2.3.1 Inicios del USB

El desarrollo y mantenimiento del USB corre a cargo del USB Implementers Forum (USB-IF) que es una organización sin ánimo de lucro formada por el grupo de compañías que se encargaron de desarrollar la especificación del USB. El objetivo de este foro es facilitar el desarrollo y la extensión de la tecnología USB. Además garantiza la calidad de los productos y que aquellos dispositivos que lleven el logotipo USB hayan pasado las pertinentes pruebas de funcionamiento. Este foro agrupa a un inmenso número de empresas de sectores como la informática, telecomunicaciones, electrónica, etc. Entre las cuales figuran algunas líderes de sus respectivos mercados a nivel mundial: Apple, HP Inc, Intel Corporation, Microsoft Corporation, Texas Instrument, etc. A parte de servir como punto para la investigación y promocionar a nivel de marketing el protocolo USB, guarda y publica todas las especificaciones del USB desde el 1996, año en que se lanzo la especificación 1.0 (aunque no se haría popular hasta dos años mas tarde, 1998, cuando se lanzó la especificación 1.1).

Estas especificaciones permiten definir un estándar a nivel industrial. Los documentos que la describen son los atributos del bus, la definición del protocolo, y la interfaz de programación para diseñar y producir periféricos que cumplan con los estándares USB. El principal objetivo es permitir la compatibilidad de los dispositivos independientemente del fabricante y que el constante desarrollo del

mercado no deje obsoletos los dispositivos que optan por este protocolo de intercambio de datos. A día de hoy existen las siguientes especificaciones del puerto USB:

- USB 1.0. Fue la primera especificación que se implementó, pero no tuvo una buena acogida por parte de los usuarios. La velocidad de transmisión era de1.5Mbps (dispositivos *low-speed*).
- USB 1.1. Año de lanzamiento 1998. Logró solucionar la mayoría de los problemas de la primera versión del USB. La velocidad de transmisión se aumentó hasta los 12Mbps (dispositivos *full-speed*). Admite cableado tipo A y tipo B. Entre las novedades que incorporó en su momento destacó la conexión *plug and play*.
- USB 2.0. Año de lanzamiento 2000. Es La especificación más extendida, aunque actualmente está siendo desplazada por su versión superior, sobre todo en el ámbito de la informática, aunque sigue siendo una gran alternativa. La velocidad de transmisión de 480Mbps (dispositivo*s high-speed*) (ver logo en la imagen 2.3.1.1).
- USB 3.0. Año de lanzamiento 2008. Empieza a ser el más usado imponiéndose al USB 2.0 actualmente. La velocidad de transmisión de 5Gbps (dispositivo *super-speed*). Emplea cableado tipo A, tipo B y tipo C (ver logo en la imagen 2.3.1.1).
- USB 3.1. Año de lanzamiento 2013. Se trata de una actualización de la versión 3.0 que permitió doblar prácticamente todos sus parámetros. Su velocidad de transmisión aumentó hasta los 10Gbps (S*uper Speed Gen2*).

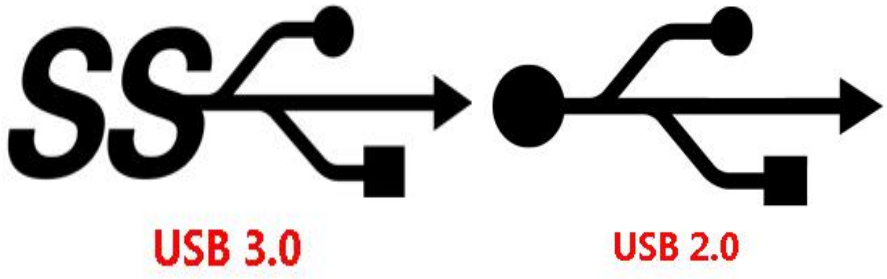


Imagen 2.3.1.1 – Logos USB

2.3.2 USB 2.0

Todo este trabajo estará enfocado a la especificación USB 2.0 (las razones se tratarán en el apartado de análisis de soluciones). En este apartado se especificarán los conceptos teóricos necesarios para poder entender el protocolo desarrollado en base al protocolo USB 2.0 y las soluciones adoptadas para resolver la problemática que se plantea en el alcance de este TFG.

2.3.2.1 Topología

Dentro de la arquitectura del bus USB podemos encontrar tres elementos principales:

- Host: en un sistema USB sólo existe un *host* que contiene todo el *hardware* y *software* necesario para permitir la comunicación del bus con los dispositivos. El *hardware* está formado por dos elementos principales el *host controller* y el *root hub* (el cual contiene los puertos USB), el software serán todos aquellos programas (Sistema Operativo) que permiten la comunicación entre el *host* y el puerto donde se encuentra conectado el dispositivo, es lo que denominamos como *driver*. Generalmente el *host* es un ordenador por lo que a partir de aquí usaré el término *host* y PC como sinónimos.

Normalmente un PC contiene un *host controller* (puede tener más de uno) y dos o más puertos USB. La función del *host* es crítica dentro de la comunicación USB ya que es el que se encarga de saber cuando un dispositivo está conectado (*plug and play)* e inicia la comunicación con él. Además decide en que momento se transfiere los datos y cuando.

- Periféricos: son los dispositivos que se conectan directamente al puerto USB de un host o por medio de un Hub. Se considera un dispositivo USB:
  - Los propios Hubs, que permiten expandir el número de dispositivos que se pueden conectar al host.
  - Cada dispositivo que aporte una o varias funciones al host, como por ejemplo un micrófono, un ratón, un joystick, etc.

Existe una categoría especial de dispositivos que son los *compound devices l*os cuales están formados por un *hub* y un dispositivo con una varias funciones. A diferencia de los dispositivo USB normales que solo tienen una única dirección, los *compound devices* desde el punto de vista del *host* son dos dispositivos diferentes por un lado el *hub* y por otro para la función que realiza, por lo que tienen dos o más direcciones asignadas.

También podemos encontrar dispositivos compuestos, los cuales en un mismo dispositivo físico realizan diversas funciones. Esto es debido a que en su programación presentan múltiples interfaces (ver sección de los descriptores), de tal manera que cada *interface* puede comunicar con un *driver* diferente del *host* .

- Hub: el *hub* consiste en un dispositivo formado por un conector aguas arriba para conectar con el *host* y otros conectores aguas abajo, los cuales se equiparan cada uno a un puerto USB (ver imagen 2.3.2.1.1).

  La principal misión de los *hub* es la de permitir que un puerto USB en el cual sólo se puede conectar un único dispositivo, se convierta en un punto que permita la múltiple conexión de dispositivos con diversas funciones. Además el *hub* realiza otras funciones como la detección de la conexión/desconexión de los dispositivos en los puertos situados aguas abajo, permite el tráfico de datos de forma bidireccional (del *host* a los dispositivos USB y de los dispositivos USB al *host*) y se encarga de alimentarlos cuando están conectados. Adicionalmente soporta diferentes modos de velocidad: *high-speed*, *full-speed*, *low-speed*. Lo que le permite la conexión de un rango de dispositivos mayor.

  De acuerdo a la especificación USB el *hub* está formado por tres componentes: el controlador del *hub* (*Hub Controller*), el repetidor (*Hub Repeater*), y el traductor (*Transaction Translator*).El controlador del *hub* permite la comunicación con el *host* y que éste pueda configurar e intercambiar datos con los dispositivos conectados al *hub*. El *Hub Repeater* es un controlador que se encarga del intercambio de datos entre el puerto aguas arriba y los conectores aguas abajo. Por último, el *Transaction Translator* le permite soportar al *hub* los diversos modos de velocidad: *full-speed*, *low-speed*, *high-speed*.

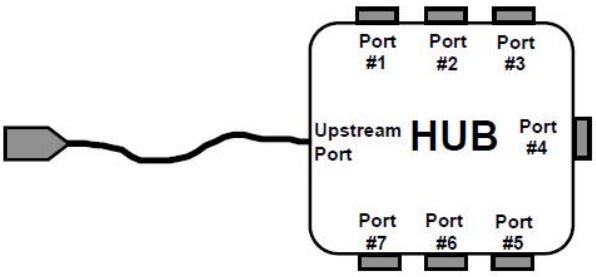


Figura 2.3.2.1.1 – Hub

Estos elementos se combinan entre ellos para permitir la transmisión de datos siguiendo una topología del bus USB en estrella jerarquizada ver imagen (2.3.2.1.2).

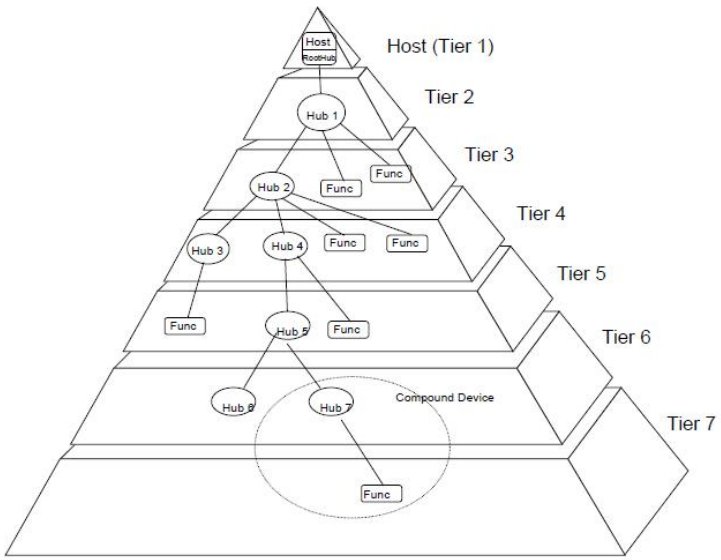


Figura 2.3.2.1.2 – Topología estrella jerarquizada

En esta topología no importa el número de dispositivos que se conecten al bus a efectos de permitir la comunicación entre el *host* y los periféricos debido a que a la hora de programar la comunicación sólo se tiene en cuenta la dirección lógica (ver sección de los descriptores). Sin embargo, el *host* sólo puede comunicar con un único dispositivo de cada vez, por lo que la única de forma de incrementar el ancho de banda del dispositivo USB es aumentar el número de *host controllers*.

2.3.2.2 Topología física.

La transmisión de datos y de alimentación se realiza a través de un cable trenzado de cuatro hilos (ver figura 2.3.1.2).

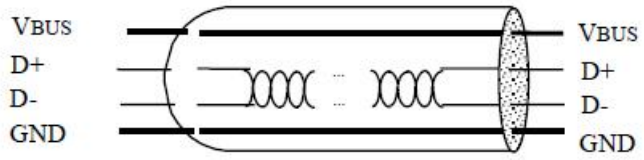


Figura 2.3.2.2.1 – Cable USB

El cable USB está formado por un par trenzado de líneas (D+ y D-) de datos que realizan una transmisión diferencial empleando codificación NRZI y *bit de stuffing,* y dos líneas adicionales para alimentación Vbus y GND.

Tal como están diseñadas las líneas de datos no existe una para la escritura y otra para la lectura sino que ambas pueden enviar en ambos sentidos (*device-host* o *host-device*).

El NRZI (No Retorno al Cero Invertido) es el método de codificación que emplea USB para transmitir los datos. En este tipo de codificación el nivel alto de voltaje representa el estado J mientras que el nivel bajo de voltaje representa el estado K. De tal forma que los cambios entre los estado J y K los va provocar un bit a 0, mientras que los bits a 1 mantendrán el estado actual. Permitiendo el envío de pequeños segmentos de datos en el estado J seguidos de períodos donde no se van a enviar datos en el estado K. Esto facilita enormemente la comunicación porque junto con la técnica del *bit de stuffing* (explicado a continuación), se puede cortar en cualquier momento el paquete de datos para enviarlo y generar la transacción, evitando que se produzcan retardos debido a datos “muy pesados” y agilizando la comunicación.

El *bit de stuffing* se realiza una vez que se ha completado la codificación de los datos. Consiste en insertar un cero adicional cada vez que se detectan más de seis unos consecutivos, de tal manera que este cero adicional va actuar como un

flag que forzará la transacción del paquete de datos, evitando que un paquete de datos supere el tiempo máximo de envío.

2.3.2.2.1 Conectores USB.

Los conectores USB están diseñados para permitir la retrocompatibilidad entre las diferentes especificaciones USB y para soportar los tres modos de velocidad disponibles en esta especificación. Actualmente en el mercado con la llegada de la especificación 3.1 existen nuevos conectores, pero ya que este trabajo se centra en la especificación 2.0, sólo se va abarcar los conectores tipo A y tipo B (ver imagen 2.3.2.2.1.1).

Estos conectores siguen un protocolo por el cual siempre los conectores tipo A se sitúan en el *host* mientras que los conectores tipo B se asignan a los dispositivos. Por lo que el cable USB siempre tiene que ser del tipo A-B

Figura 2.3.2.2.1.1 – Conectores USB

2.3.2.2.2 Modos de velocidad.

En la especificación USB 2.0 se han implementado tres modos de velocidad:

- *Low-speed*: fue el primer modo de velocidad del protocolo USB. Implementado en la especificación 1.0. Permite una tasa de transferencia máxima de 1.5 Mbits/s.
- *Full-speed*: revisión del modo de velocidad anterior, implementado en esta especificación permite una tasa de transferencia máxima de 12 Mbits/s.
- *High-speed*: modo de velocidad más alto en la especificación 2.0, implementado en esta especificación permite una tasa de transferencia máxima de 480 Mbits/s.

2.3.2.3 Elementos lógicos de una comunicación USB.

En este apartado se abordará cómo se envían los datos mediante protocolo USB. Para ello hay que introducir una serie de nuevos conceptos, ya que desde el punto de vista lógico los dispositivos USB no son más que una colección de *endpoints* que permiten implementar interfaces.

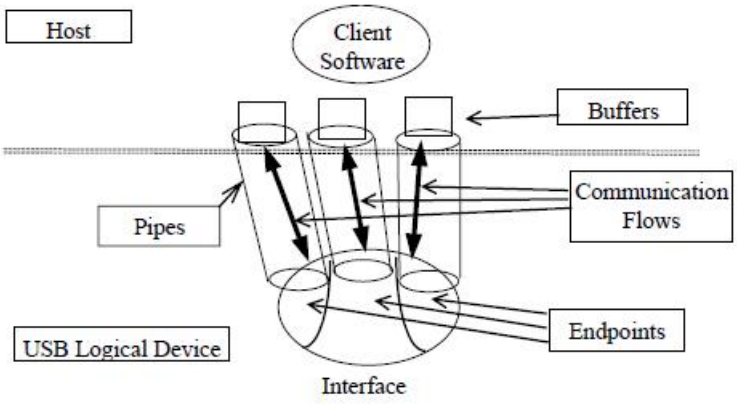


Figura 2.3.2.3.1 – Elementos lógicos

En la figura 2.3.2.3.1 se puede observar los distintos elementos que conforman la comunicación USB desde el punto de vista lógico; es decir, la forma en la que se programa el *host* y el dispositivo para poder conseguir el correcto intercambio de datos.

2.3.2.3.1 Device endpoint.

Podemos considerar los *device endpoint* como un bloque de memoria; es decir, un registro que permite almacenar de forma temporal los datos recibidos del *host*

o los datos que envía el dispositivo, de tal manera que constituye el punto de partida y el punto final para el tráfico de datos. Como su nombre indica, el concepto de *endpoint* sólo existe para los dispositivos USB, mientras que en los PC, pese a existir registro que funcionan de manera parecida, no reciben esta denominación ya que no se consideran como tales.

Cómo se ha explicado anteriormente, cada dispositivo USB tiene una dirección única con la que se identifica ante el *host;* del mismo modo, cada *endpoint* del dispositivo recibe un identificador, número del *endpoint* que le permite al *host* direccionar los datos a un *endpoint* en concreto. El número máximo de *endpoint* que puede albergar un dispositivo depende del modo de velocidad con el que se haya configurado (ver modos de velocidad para la especificación 2.0), de tal manera que para un dispositivo configurado como *low-speed* tendrá un máximo de cinco *endpoints*, el *endpoint 0* y cuatro adicionales; mientras que los dispositivos configurados como *full-speed* o *high-speed* tendrán hasta 30 *endpoints* más el *endpoint* 0. Para los *low-speed* su dirección contendrá un número de enpoint que variará entre 0 y 2, mientras que en el caso de los modos *high-speed* y *full-speed* la dirección variará entre 0 y 15. Se puede observar que el número que se asigna a los *endpoints* es la mitad del número de *endpoints* totales. Esto es debido a que por cada dirección asignada a un *endpoint* en concreto este se desdobla en dos *endpoints* que solo permiten el envío de datos en único sentido, para el cual se toma como referencia siempre la perspectiva del *host* por lo que tendremos dos tipos *endpoints* por cada dirección asociada aun *endpoint*:

- *In endpoints*: son los que envían los datos del dispositivo al *host* (por ejemplo *Endpoint 1 IN*).
- *Out endpoints*: son los que reciben los datos del *host* y los almacenan en el dispositivo (por ejemplo *Enpoint 1 OUT*).

Hay que considerar otros parámetros importantes dentro de los *endpoints* que se exponen en secciones posteriores, como son el ancho de banda requerido, *error handling*, tamaño máximo de paquete que es capaz de recibir/enviar (ver sección de los descriptores) y el tipo de transferencia (ver sección de tipos de transferencia).

Pese a que hemos dicho que los *endpoints* son unidireccionales existe un caso especial *endpoint* que permite el flujo de datos de manera bidireccional ese es el *endpoint 0*, también denominado *endpoint* de control (*control endpoint*). Este *endpoint* es el registro mínimo que todo periférico USB debe poseer, debido a que una vez que se ha conectado al puerto, el *host* le da alimentación y lo emplea para configurar el dispositivo en función de la información que éste le envíe y colocarlo en un estado que le permita funcionar.

2.3.2.3.2 Pipes.

Los *pipes* son una asociación entre el *endpoint* del dispositivo y el software del *host*, cuya función principal es permitir el tránsito de datos. Existen dos tipos de *pipes* en función del método de comunicación que emplean:

- *Stream*: los datos que circulan por el no tienen un formato determinado, únicamente deben contener obligatoriamente campo de datos y la dirección (*IN/OUT*). Soportan tres tipos de transferencias: *interrupt*, *bulk*, *isochronous*. Están asociados a los *endpoints* unidireccionales y por lo que tanto el host como los dispositivos pueden controlarlos para enviar datos en cualquier momento.
- *Message*: los datos siguen el formato definido por la especificación USB. Sólo soportan *Control Transfer.* En este tipo de *pipe* el *host* es el que controla toda la comunicación a través de *request* que el dispositivo debe responder con la correspondiente información.

El *endpoint* de control visto en el apartado anterior lleva asociado el denominado Default *Control Pipe*. Este pipe está siempre disponible una vez se hayan completado los pasos de de alimentación y el bus ha recibido un *reset*. Permite configurar el dispositivo al transmitir las transacciones *Setup* durante el proceso de enumeración (ver sección de la enumeración) permitiendo al *host* establecer los otros pipes conforme lo establecido por los descriptores del dispositivo y soporta las transferencias de control. Del mismo modo que el *host* crea el número de *pipes* que requiera el dispositivo, también se encarga de eliminarlos cuando el dispositivo se desconecta.

2.3.2.4 Transferencias USB:

En el protocolo USB los datos se transmiten en mensajes denominados *frames* o *microframes*. Cada unos de estos *frames* o *microframes* contienen una serie de transacciones que a su vez contienen los bloques de datos. Estas transacciones se transmite siguiendo una serie de protocolos que constituyen cuatro tipos de transferencias básicos, los cuales están pensados para cubrir las necesidades de los desarrolladores a la hora de fabricar los periféricos.

Cada *frame* establece una unidad de tiempo base de 1 milisegundo(para dispositivos que soportan velocidades *full/low-speed* bus) y una base de 125 microsegundos) denominada *microframe* (para dispositivos que soportan velocidades *high speed* bus). Dependiendo del tipo de configuración del dispositivo en un frame o microframe puede haber más de una transferencia por lo tanto también habrá varias transacciones, dependiendo del tipo de transferencia que se haya seleccionado.

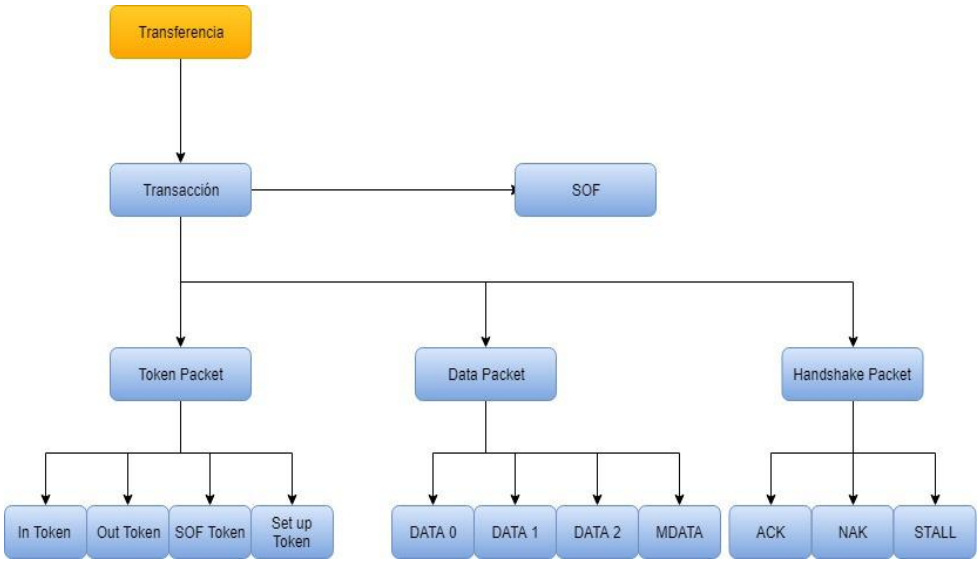


Imagen 2.3.2.4.1 – Esquema transacciones

Existen tres tipos básicos de transacciones, los cuales contienen sus propios paquetes de datos (ver imagen 2.3.2.4.1):

- Token Packet: identifica el *endpoint* destino del paquete de datos y en el caso de las *Control transfer* identifica las transacciones Setup. En general existen cuatro tipos:
  - Out Token: identifica que la transacción se ha enviado del *host* al dispositivo.
  - In Token: identifica que la transacción se ha enviado del dispositivo al *host*.
  - SOF Token: indica que el paquete de datos es de tipo *SOF*.
  - Setup Token: identifica que la transacción es del tipo *SetUp*.
- Data Packet: transmite los request y los datos. Existen cuatro tipos:
  - DATA 0: este tipo de paquete de datos generalmente se emplea en las *Control Transfer*.
  - DATA 1: común a todos los tipos de transferencia. Soporta paquete de datos con un tamaño máximo de 1024 *bytes*.
  - DATA 2: tipo de datos para dispositivos configurados como *high-speed*.
  - MDATA: tipo de datos para dispositivos configurados como *high-speed*.
- Handshake Packet: paquete de datos cuya función es informar tanto al *host* como al dispositivo del estado de la transferencia. Contiene los siguientes paquetes de datos.
  - ACK: indica si la comunicación se ha realizado con éxito.
  - NAK: indica si el bus está ocupado, o sino se va a retornar ningún dato
  - STALL: el dispositivo no soporta el *request* o no se puede acceder al *endpoint*.
  - NYET: no se recoge en la imagen 1.3.2.4.1, ya que es un paquete de datos reservado para ciertos tipos de dispositivos que operan a muy alta velocidad para USB. Indica que los datos se han aceptado pero el *endpoint* no está listo para otra transferencia.
- Start of Frame Packet (SOF): se trata de un paquete datos de 11 *bits* que se envía cada 1ms en el caso de dispositivos *full-speed* y cada 125µs en dispositivos *high-speed* y le permite al *host* iniciar las transacciones con el dispositivo.

Todos este paquetes están formados por 6 campos de datos básicos:

- SYNC: paquete de datos de 8 bits para dispositivos full/low speed y de 32 bits para dispositivos high-speed. Se emplea para sincronizar el reloj.
- PID: identifica el tipo del paquete de datos.
- ADDR: identifica el destinatario del paquete de datos.
- ENDP: está formado por 4bits e indica el número del *endpoint* al que se dirige el paquete de datos.
- CRC: *Cyclic Redundancy Checks* indica que no hay errores dentro del paquete de datos.
- EOP: *End of Packet*, indica el final del paquete de datos.

Teniendo en cuenta lo visto anteriormente, las transacciones se clasifican de una forma más sencilla según la dirección en la que se manden los datos, es decir, en función del Token Packet. Por lo que podemos distinguir tres tipos básicos de transacciones:

- Transacciones *IN*: el dispositivo envía datos al *host*.
- Transacciones *OUT*: el dispositivo recibe datos del *host*.
- Transacciones *Setup*: se trata de una transacción de tipo OUT, pero cuya función es inicializar una transferencia de control. Por lo tanto este tipo de transacción solo se produce una vez que el dispositivo se ha conectado al puerto usb, el *host* ha detectado que se ha conectado y le proporciona alimentación y se ha *reseteado* el bus. Es de vital importancia que el dispositivo pueda recibir las transacciones *Setup*, ya que contiene los *request* (ver sección descriptores) que necesita contestar para que el *host* sepa de que tipo de dispositivo se trata y como realizar su configuración.

2.3.2.5 Tipos de transferencias.

Una de las principales características de la especificación USB es su flexibilidad y capacidad para adaptarse a distintos periféricos, con distintas tasas de transferencia, tamaño de los paquetes de datos, y corrección de errores. Para ello la especificación USB define cuatro tipos de transferencia de datos, los cuales están pensados para transportar los datos entre un registro del *host* y el *endpoint* del dispositivo a través de los *pipe* de la forma más óptima posible. Cada uno de los tipos de transferencia definen la comunicación USB en los siguientes aspectos:

- Formato y secuencia de los datos.
- Dirección del flujo de comunicación.
- Tamaño del paquete de datos.
- Acceso al bus.
- Latencia.
- Tratamiento de los errores.

Estos cuatro tipos de transferencia son los siguientes:

- *Control transfer* (transferencia de control): es el único de los cuatro tipos de transferencia que todo dispositivo USB necesita obligatoriamente implementar. Su función principal es la de una vez que el dispositivo es alimentado por el *host*, le permite al *host* identificarlo y configurar la comunicación USB. Una vez configurado, se puede emplear para intercambiar pequeños paquetes de datos. Este tipo de transferencia se realiza sobre el *Default Pipe* y el *endpoint de control.* Tiene corrección de errores y no garantiza una latencia estable.
- *Bulk transfer*: permite transferir un grandes cantidades de datos con absoluta integridad, sin embargo puede consumir más tiempo del estimado, es decir no se garantiza que la latencia se mantenga constante, esto es debido a que el paquete de datos no se transmitirá hasta que el ancho de banda necesario esté disponible. Incorpora detección y corrección de errores y el ancho de banda puede ser variable. Este tipo de transferencia se utiliza en dispositivos como impresoras, escáner, pendrive, etc.
- *Isochronous transfer*: permite el intercambio de una gran cantidad de datos (paquetes de hasta 1023 *bytes*) garantizando la latencia del bus, se trata de una comunicación periódica entre el *host* y el dispositivo dónde el parámetro que más importa es la entrega a tiempo de los datos. Sin embargo se pueden producir pequeñas pérdidas de cantidades de datos. No incorpora corrección de errores. Se emplea para aplicaciones de streaming audio, video, etc.
- *Interrupt transfer:* permite la transmisión de pequeñas cantidades de datos, en el instante que el periférico necesite comunicarle algo al *host* o a la inversa en el momento que el *host* tenga datos para mandarle al dispositivo. Tiene corrección de errores y se garantiza al igual que las transferencias

isochronous la latencia máxima entre transferencia. Se emplea con dispositivos como teclados, ratones, etc.

Tanto las transferencia de control como las *interrupt* pueden implementar los tres modos de velocidad (*low-/full-/high-speed*) que soporta la especificación 2.0, en cambio los tipos *bulk* e *isochronous* no permiten implementar *low-speed*. En la siguiente tabla se recoge las tasas de transmisión para cada uno de los cuatro tipos de transferencia vistos anteriormente. No se trata de establecer una comparativa entre los tipos de transferencia, debido a que cada tipo de transferencia, como se ha comentado, implementa diferentes parámetros en el bus. El uso de uno u otro depende principalmente de la función del dispositivo y la cantidad de datos que transmite, la comparativa es, en todo caso, dentro de un tipo de transferencia la tasa de transferencia que ofrece en función del modo de velocidad que se pueda implementar en el dispositivo (ver tabla 2.3.2.5.1).

Modos Velocidad	Control	Bulk	Interrupt	Isochronous
High speed(bytes/milisegundo por transferencia)	15,872 (33 transacciones de 64 byte/microframe)	53,248(13 transacciones de 512 byte/microframe)	24,576(3 transaciones de 1024 byte/microframe)	24,576 (3 transacciones de 1024 byte/mciroframe)
Full speed(bytes/milisegundo por transferencia)	832 (13 transacciones de 64 byte/frame)	1216 (19 transacciones de 64 byte/frame)	64(una transacción de 64 byte/frame)	1023 (1 transacción 1023 byte/frame)
Low speed(bytes/milisegundo por transferencia)	24 (3 de 8 byte/microframe)	no permitido	0,8 (8 bytes 10 milisegundos)	no permitido

Tabla 2.3.2.5.1 – Tipos de transferencia y tasas de transferencia

Para la realización de este trabajo unicamente se necesitaron la transferencia de control y la *interrupt transfer* (ver documento Análisis de Soluciones), por lo que las siguientes secciones tratarán más a fondo estos dos tipos de transferencia.

2.3.2.5.1 Control transfer.

Es el único tipo de transferencia que poseen todos los dispositivos USB. Tiene dos funciones, administrar los *request* y permitirle al *host* configurar los dispositivos USB. Una vez configurado el dispositivo también se pueden usar para enviar y recibir datos, aunque no garantiza un ancho de banda constante.

Esta transferencia se produce en el instante que el periférico se conecta al *host*, y emplea el *Endpoint 0* y el *Default Pipe*. Cada *Control Transfer* está formada por tres transferencias:

- *Setup*: en esta primera operación el *host* envía los *request*. Los paquetes de datos contienen un identificador PID que los identifican como una transferencia de control.
- *Data*: contienen transacciones de datos, tanto si se trata de una lectura como de una escritura.
- *Status*: está formado por una serie de transacciones de entrada y de salida que se encargan de informar si se han realizado con éxito las operaciones anteriores.

Las transferencias *Setup* y *Status* son obligatorias, mientras que las de tipo *Data* sólo se emplean en el caso de que el dispositivo no disponga de un tipo de transferencia específico para la comunicación.

El tamaño de los paquetes de datos dependen de la transferencia *Data,* y éste varía con la velocidad del dispositivo. Para dispositivos *low-speed* el tamaño máximo es de 8 *bytes,* para *full/high-speed* el tamaño máximo es de 64 *bytes.*

Si se produce algún error en la transferencia, es decir, el *host* no recibe el correspondiente paquete de *handshake*, vuelve a intentar la transferencia dos veces más, superado ese tiempo el *host* detiene la comunicación y notifica el error.

2.3.2.5.2 Interrupt transfer.

Este tipo de transferencia está diseñado para aquellos dispositivos que necesitan transmitir o recibir datos de forma no periódica pero en una cantidad de tiempo determinada. Aunque el nombre sugiere que el dispositivo puede enviar datos en cualquier momento, provocando una interrupción que el *host* deba atender en ese instante (y viceversa), este tipo transferencia sigue un método *polls* en el cual el ordenador interroga al dispositivo; lo que sí se garantiza es que los datos se van a enviar o recibir con un retardo mínimo.

Este tipo de transferencia no es de uso obligatorio para todos los dispositivos USB como puede ser la *control transfer* pero en cambio sí que ciertas clases de dispositivos, como la clase HID, requieren implementarse obligatoriamente con este método de transferencia (fue creada con ese objetivo). Muchos periféricos emplean este modo de transferencia como teclados, ratones, mandos de video-juegos, etc.

En una *interrupt transfer* los *pipes* son unidireccionales y los datos no siguen una estructura específica.

Por otra parte los *endpoints* son muy importantes, ya que determinan el tamaño de los paquetes de datos que se pueden transmitir e identifican la dirección en la que se transmiten (*IN* o *OUT*). El tamaño del paquete de los datos dependen del modo de velocidad que se haya seleccionado para el dispositivo, la *interrupt transfer* soporta los tres modos de velocidad que define la especificación 2.0:

- *Low-speed*: limitado a 8 *Bytes* por paquete de datos.
- *Full-speed*: limitado a 64 *Bytes* por paquete de datos.
- *High-speed*: limitado a 1024 *Bytes* por paquete de datos, aunque requiere dos o tres transacciones por *microframe*.

De manera homónima los *Host Controllers* deben soportar el tamaño máximo del paquete de datos para que se puede transmitir con éxito. No siempre se tiene que transmitir el tamaño máximo del paquete, el *USB System Software* puede determinar el tamaño de los *interrupt pipe* una vez que se ha configurado correctamente el dispositivo.

El tamaño del paquete se puede configurar modificando el campo *wMaxPacketSize* (ver sección de los descriptores) .

En este tipo de transferencia la frecuencia del bus y el tiempo de cada *frame* o *microframe* limita el número de transacciones que se pueden realizar. Para un dispositivo *high-speed* el número es inferior a 134 mientras que para un dispositivo *full-speed* el máximo es de 108 y para un dispositivo *low-speed* no superan las 10 transacciones por *frame*.

Cada *endpoint* define el periodo deseado de acceso al bus. En función de la velocidad con la que se haya configurado el *endpoint*:

- Para dispositivos con *low-speed endpoints* están limitados a una latencia entre 10ms y 255ms.
- Para dispositivos con *full-speed endpoints* están limitados a una latencia entre 10ms y 255ms.
- Para dispositivos con *high-speed endpoints* pueden establecer un período (2^*bInterval**125us). Donde *bInterval* es uno de los campos de los descriptores (ver sección de los descriptores).

Las *Interrupt transfer* están formadas por una serie de transacciones *IN* y *OUT*, que se transfieren por separado, es decir, las transacciones *IN* tienen unos *endpoint* y *pipes* asignados mientras que las transacciones *OUT* tienen los suyos.

Este tipo de transferencia puede terminar de dos formas: cuando el tamaño de datos que se recibe supera el tamaño esperado o cuando el campo de datos es cero.

También se caracteriza por ser un tipo de transferencia que soporta protocolos específicos.

2.3.2.6 Enumeración.

En el momento en que el dispositivo se conecta al *host*, se inicia el proceso mediante el cual “aprende de él”, denominado enumeración. Mediante este proceso el *host* le asignará una dirección al dispositivo, leerá los descriptores, con

los que podrá configurarlo, y el sistema le asignará un *driver* para comunicarse, permitiendo que el dispositivo esté listo para transmitir y/o recibir datos.

Este proceso es automático para el usuario, y en un PC con sistema operativo Windows, culmina cuando se genera el mensaje de que el dispositivo se ha conectado correctamente. Si a continuación se consulta el Panel de Control se podrá ver el dispositivo en función del *driver* que el Sistema Operativo le haya asignado.

La especificación USB define seis estados para los dispositivos USB: *Attached*, *Powered*, *Default*, *Address*, *Configured*, *Suspend*. Durante el proceso de enumeración los dispositivos USB deben superar cuatro estados para que puedan iniciar la comunicación: *Powered*, *Default*, *Address*, *Configured*, los cuales son comunes entre el *host* y el dispositivo, e inician la comunicación. Mientras que *Attached* y *Suspend* son estados internos propios del dispositivo.

En el diagrama de la imagen 2.3.2.6.1, se definen la relación entre los seis estados.

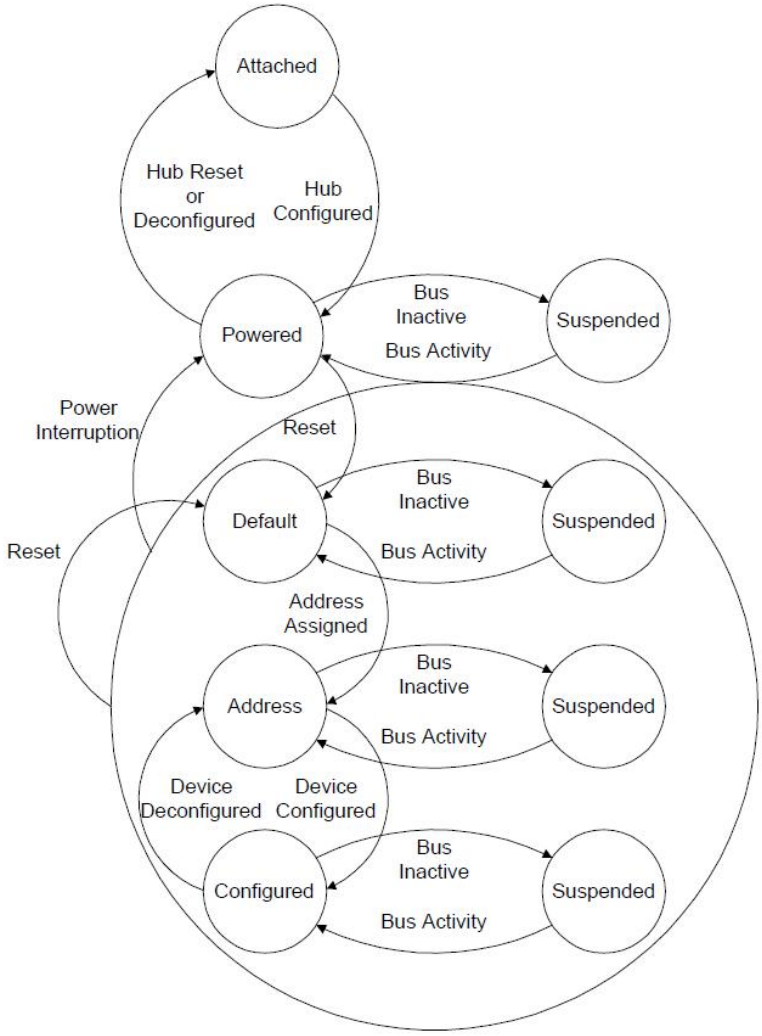


Imagen 2.3.2.6.1 – Diagrama estados USB

En cada uno de los estados el dispositivo realizará una serie de operaciones específicas que le permitirán avanzar al siguiente estado:

- *Attached*: inicia la comunicación USB. Se produce cuando el usuario conecta el dispositivo al puerto USB, pero el *host* aún no lo ha alimentado. Generalmente los dispositivos USB no se encuentran mucho tiempo en este estado porque el *host* comienza a alimentarlo inmediatamente una vez detecta la conexión.
- *Powered*: la especificación reconoce tres modos por los cuales los dispositivos son alimentados:
  - *Self-powered*: en los cuales los dispositivos tienen una fuente de alimentación externa. Aunque ya tienen alimentación no se consideran

que se encuentran en el estado de *Powered* hasta que no estén conectados al *host* y superan el estado *Attached*.

- *Bus-powered*: en este caso es el bus USB el que actúa como fuente de alimentación para el dispositivo. Una vez que se ha superado el estado de *Attached* el *host* comenzará a suministrar corriente a través del puerto USB. El dispositivo establece la cantidad de corriente máxima que necesitará del *host* a través del *Configuration descriptor*.
- *Dual* en el cual el dispositivo presenta alimentación mediante *Self-powered* y *Bus-powered*. El dispositivo debe indicar esto en el *Configuration descriptor*, de tal manera puede cambiar entre un modo y el otro sin que se produzca un reset. La corriente máxima que el dispositivo debe solicitar al *host,* la que circule por Vbus.

Si un dispositivo está configurado según los modos *self/bus-powered* y necesita cambiar su configuración se producirá un *reset*, perdiendo la configuración y la dirección del dispositivo. Una vez que se ha alimentado el dispositivo con éxito (supera el estado de *Powered*), se le asignará una dirección (*Adress*) dentro de un determinado período de tiempo.

- *Default*: es un estado intermedio en que el dispositivo se ha alimentado correctamente y antes de que reciba ninguna transacción debe realizar un *reset*. Una vez que se ha realizado, el dispositivo puede recibir la dirección por defecto.
- *Address*: en este estado se le asigna una dirección por defecto al dispositivo hasta que el sistema operativo termina de configurarlo. A continuación le asigna el *driver* y la dirección única con la que se dirigirá al periférico. Esta dirección se mantiene aunque el dispositivo entre en un estado de *Suspend*. Cuando el dispositivo se desconecta del puerto del *host*, el sistema operativo inmediatamente libera la dirección para que pueda ser empleada por otro periférico.
- *Configured*: este estado engloba todas las operaciones que necesita el *host* para configurar el dispositivo. Todo cambio que afecte a la configuración del dispositivo provoca que la configuración actual de los *endpoints* e *interfaces* del dispositivo sean eliminados y se restauren a sus valores por defecto.

Cesarían todas las transferencias y únicamente se permitiría la comunicación a través de *Control transfer*.

- *Suspended*: se trata de un estado de ahorro de energía, en el cual el dispositivo USB puede entrar en cualquier momento cuando no se detecta tráfico en el bus durante un período de 3ms. En este estado el dispositivo guarda la configuración actual y la dirección que le asignó el *host*. Del mismo modo si el *host* entra en un estado de suspensión el dispositivo deberá cesar inmediatamente cualquier actividad e ingresar simultáneamente en el estado de suspensión.

El proceso de enumeración en un sistema operativo Windows sigue la siguiente secuencia de operaciones, que pese a que están enumeradas de forma secuencial, el *firmware* del dispositivo no debe asumir que van ocurrir en el orden en el que se detallan a continuación, sino que pueden verse interrumpidas en cualquier momento:

1. El usuario conecta el dispositivo al puerto USB. En ese momento se genera un evento al *host* de la presencia de un periférico en el puerto. Esta conexión también puede hacerse por medio de un *hub* colocado entre el periférico y el *host*.
2. Los dispositivos USB cuentan con una resistencia de pull-up de 900 a 1575Ω en la línea D+ para configuraciones *full-speed* y en la línea D- para configuraciones *low-speed* que le permite la detección al host de la conexión del dispositivo. Los dispositivos configurados como *high-speed* se detectan como si fuesen *full-speed*. Una vez detectado el periférico, el *host* espera 100ms para que se complete el proceso de inserción al puerto USB y mueve el el dispositivo al estado de *Powered*.
3. Una vez en el estado de *Powered* el dispositivo realiza el *reset* que requiere el puerto USB del *host*, permitiendo que el USB avance al estado de *Default*. En ese momento el *host* alimenta al periférico con una corriente máxima de 100mA a través del Vbus.
4. El *host* asigna una dirección por defecto al dispositivo, en este momento la comunicación con el periférico está abierta a través del Control Pipe.

5. El *host*, a través del *Device Descriptor* determina el tamaño del paquete de datos que va usar el Control Pipe y le asigna una dirección única al dispositivo permitiendo que éste se mueva al estado de *Address*.
6. El *host* continúa mandando los demás *request* para determinar la configuración del dispositivo. Este proceso puede tardar unos cuantos milisegundos en completarse.
7. En función de la información recibida, a través de los descriptores, el *host* asigna una configuración al periférico. El dispositivo se mueve al estado de *Configuration*, y es cuando todos los *endpoints* que se han especificado en los descriptores se encuentran accesibles para el *host*, así como también se modifica la corriente máxima que el *host* le proporciona al dispositivo por lo que el periférico se encuentra listo para usar.

2.3.2.7 Standard Request.

Como se ha comentado anteriormente, durante el proceso de enumeración toda la comunicación tiene lugar a través del *Default Pipe* mediante el *Control transfer*. El primer objetivo de esto tipo de transferencia es que el host puede obtener toda la configuración del dispositivo para así establecer la comunicación en el bus. Para ello el *Host controller* dispone de una serie de comandos que son enviados al dispositivo a través de paquetes *Setup* de 8 bytes. Estos comandos se denominan *request*.

Los *request* son estructuras de datos divididos en varios campos. Al enviarse en los paquetes de datos *Setup* que están formados por 8 *bytes*, ver tabla 2.3.2.7.1:

- *bmRequestType*: define las características del *request* mediante tres parámetros. El primero de ellos define el sentido de la transferencia, un cero indica que la transferencia se realiza del *host* al dispositivo, un 1 indica que la transferencia se realiza en sentido contrario. Este parámetro se emplea cuando hay datos en el campo *Data* para transmitir si el campo *wLength* es cero, se ignora. El segundo parámetro define el tipo de *request*, debido a que una clase en particular puede tener sus propios *request*. La especificación define cuatro tipos de *request*:
  - 0 = Standar.

- 1 = Class.
- 2 = Vendor.
- 3 = Reserved.

Por último el tercer parámetro establece el destinatario del *request*:

- 0 = Dispositivo.
- 1 = Interface.
- 2 = Endpoint.
- 3 = Otro.
- 4....31 = Reservado.

- wIndex: presenta dos formatos en función si es referido a un *endpoint* o una *interface*. Cuando se refiere a un *endpoint* el bit 7 indica la dirección del endpoint mientras que los bits 3 a 0 indican el número del *endpoint*. En el caso de que se refiera a una *interface* se emplean los 8 primeros bits para referirse al número de *interface*.

Posición	Campo	Tamaño en bytes	Descripción
0	bmRequestType	1	Establece tres parámetros: dirección de la transferencia de Datos, tipo de request, y receptor
1	bRequest	1	Indica el tipo de request
2	wValue	2	Varía en función del request permite pasar parámetros específicos al dispositivo
4	wIndex	2	Se emplea para especificar un determinada endpoint o interface
6	wLength	2	Número de bytes a transferir en el campo de Datos durante el control transfer

Tabla 2.3.2.7.1 – Formato Request

La especificación define 11 *standard request*. Los dispositivos deben responder a cada uno de los *request*. Muchos de ellos funcionan en parejas de tal manera que cada *Set request* tiene asociado un *Get* o *Clear request*, a excepción de los *request Set_Address*, *Synch_Frame* y *Get_Status*. Los *request* serían:

- Clear_Feature request.
- Get_Configuration request.
- Get_Descriptor request.
- Get_Interface request.
- Get_Status request.
- Set_Address request.
- Set_Configuration request.
- Set_Descriptor request.
- Set_Feature request.
- Set_Interface request.
- Synch_Frame request.

2.3.2.8 Descriptores.

Los descriptores son estructuras de datos con un formato definido, en el cual éstos están formados por varios campos; a su vez cada uno de los campos está formado por uno varios *bytes*. Cada campo contiene uno o varios atributos que permiten configurar el periférico, por lo que el tipo y la cantidad de los campos varían de un descriptor a otro. Aunque siempre comienzan con dos campos que indican el número total de *bytes* que contiene el descriptor e identifica el tipo de descriptor del que se trata. A medida que el *host* los va recibiendo obtiene información sobre el dispositivo que se ha conectado:

- Información general sobre el dispositivo.
- Información sobre la configuración del periférico.
- Configuración de las *interfaces*.
- Configuración de los *endpoints*.

Estas estructuras de datos durante el proceso de enumeración se transmiten mediante *Control transfer*. Para ello el *host* emplea los *request* ver sección 2.3.1.6.

En general, los descriptores siguen una estructura jerarquizada dónde los descriptores de alto nivel informan al host de la existencia de descriptores de un nivel inferior, de tal manera que éste pueda seguir mandando *request* al periférico para poder configurarlo.

La especificación USB define los siguientes descriptores estándar:

bDescriptorType	Descriptor Type
01h	device
02h	configuration
03h	string
04h	interface
05h	endpoint
06h	device_qualifier
07h	other_speed_configuration
08h	Interface_power
09h	OTG
0Ah	debug
0Bh	interface_association

Tabla 2.3.2.8.1 – Identificador de tipo de descriptores

Como podemos observar cada uno de los descriptores presentan un campo identificador con el cual se identifican al *host*, y le permiten al periférico seleccionarlos cuando recibe el correspondiente *request*.

Como se ha mencionado anteriormente el primer descriptor que se enviaría sería el *Device Descriptor* el cual contiene información general acerca del dispositivo y el número de configuraciones que admite (indica al *host* la existencia de un descriptor que contenga dicha configuración). A continuación se enviaría el *Configuration Descriptor* que establece el consumo máximo de corriente y el número de interfaces que soporta cada configuración. El siguiente descriptor será el de *interface* el cual define el número de *Endpoints*. Y por último se envía el *Endpoint Descriptor* que contiene la información acerca de la configuración de los

endpoints. Estos tres descriptores (*Configuration*, *Interface*, *Endpoint*) se envían uno tras otros conforme se recibe el *Request_configuration*.

Los descriptores: *Device*, *Configuration* e *Interface* son necesarios y obligatorios para que el *host* pueda establecer con éxito la comunicación en el bus. El *Endpoint Descriptor* pese a ser necesario en la mayoría de aplicaciones, no es obligatorio ya que en algunos dispositivos podría darse el caso que no hacen falta *Endpoints* adicionales al *Endpoint 0*. El resto de descriptores que aparecen en la tabla 2.3.2.8.1 son opcionales:

- *String Descriptor*: almacena y transmite textos en cadena de caracteres.
- *Device_qualifier* y *other_speed_configuration*: son obligatorios para dispositivos que utilizan modos de velocidad *high-speed*.
- OTG: para dispositivos que emplean *On-the-Go* (ver sección de definiciones y abreviaturas).
- *Interface_association*: únicamente para dispositivos compuestos (*composite device*).

Estos constituyen los descriptores estándar, pero determinadas clases de dispositivos pueden emplear otros descriptores.

A continuación se explica cada uno de los descriptores con sus correspondientes campos. Cada campo está precedido de un prefijo que indica el tipo de datos en el que están codificados y/o su función (ver sección de definiciones y abreviaturas).

2.3.2.9 Device Descriptor.

Contiene la información básica del periférico, el cual se envía el *host* cuando el dispositivo recibe el *Get_Descriptor request*. Un periférico solo puede contener un único *Device Descriptor*. Está formado por 14 campos, los cuales contienen información sobre el propio descriptor -el dispositivo, las configuraciones y la clase a la que pertenece el dispositivo- . El tamaño total del descriptor es de 18 bytes (ver tabla 2.3.2.9.1).

Algunos campos presentan ciertas particularidades:

- bcdUSB: indica la especificación USB con la que se implementó el *firmware*. El número se expresa en BCD, el primer byte por la izquierda representa el número entero mientras que los siguientes cuatro son las décimas:
  - 1.0 = 0100h.
  - 1.1 = 0110h.
  - 2.0 = 0200h.

- Los campos idVendor e idProduct funcionan como identificadores del dispositivo. Permiten al *host* identificar el dispositivo y le ayudan a la hora de seleccionar el *driver*.
- Los campos *iProduct*, *iSerialNumber*, *iManufacturer* permanecerán a cero si no hay los correspondientes *String Descriptor*.
- bDeviceClass: normalmente la clase del dispositivo se establece en el *Interface Descriptor*, pero existen clases particulares que se establecen en el *Device Descriptor (*ver tabla 2.3.2.9.1).
- bDeviceSubClass: sirve para definir subclases dentro de las clases que añaden funcionalidades específicas al dispositivo. Si la clase está definida en el *Interface Descriptor* el valor de este campo debe ser 0. Valores entre 1 y EFh, hacen referencia a las subclases recogidas por la especificación USB. Un valor FFh significa que la subclase está definida por el fabricante.
- bDeviceProtocol: si el valor del bDeviceClass está entre 01h y FEh, el protocolo está definido por la especificación USB.

bDeviceClass	Descripción
00h	La clase está determinada por el interface descriptor
02h	Dispositivo de comunicaciones
09h	Dispositivo Hub
DCh	Dispositivo de Diagnóstico
E0h	Controlador Wireless
EFh	Dispositivo Miscellaneous
FFh	Dispositivo especificado por el fabricante

Tabla 2.3.2.9.1 – Clase de dispositivos

Posición	Campo	Tamaño en bytes	Descripción
0	bLengh	1	Tamaño del descriptor en bytes
1	bDescriptorType	1	Identificador del descriptor: 01h
2	bcdUSB	2	Especificación de USB
4	bDeviceClass	1	Código de la clase de dispositivo
5	bDeviceSubClass	1	Código de la subclase
6	bDeviceProtocol	1	Código del protocolo
7	bMaxPacketSize0	1	Tamaño máximo del paquete de datos del Endpoint 0
8	idVendor	2	Vendor ID
10	idProduct	2	Product ID
12	bcdDevice	2	Número de versión del dispositivo en BCD
14	iManufacturer	1	Indice del string descriptor para el fabricante
15	iProduct	1	Indice del string descriptor para el producto
16	iSerialNumber	1	Indice del string descriptor que contiene el número de serie
17	bNumConfiguration	1	Número de configuraciones que soporta el dispositivo

Tabla 2.3.2.9.2 – Campos Device Descriptor

2.3.2.10 Configuration Descriptor.

Establece las características y funcionamiento del dispositivo (consumo de corriente, número de interfaces soportados, etc.). A diferencia del *Device Descriptor*, cada dispositivo puede tener en su *firmware* múltiples configuraciones es decir, varios *Configuration Descriptor*, pero sólo puede tener una sola

configuración activa al mismo tiempo. Cada cambio de configuración provocará un *reset* del dispositivo.

El *Configuration Descriptor* tiene varios descriptores subordinados, de tal manera que cuando el *host* envía el *Get_Configuration request*, el dispositivo envía toda la configuración: *Configuration Descriptor*, *Interface Descriptor*, *Endpoint Descriptor*. Si el *host* ha recibido correctamente el *request* colocará el bit del campo *wValue* del *Setup transaction* igual a 2.

El *Configuration Descriptor* está formado por 8 campos ver tabla 2.3.2.10.1. Las particularidades más importantes de los campos son:

- bmAttributes: informa al *host* sobre el tipo de alimentación que emplea el dispositivo y sobre si soporta *remote wake up*:
  - Bit 6: 1 si el dispositivo es *self-powered* o 0 si es *bus-powered*.
  - Bit 5: 1 si el dispositivo soporta *remote wake-up*.

  El resto de los bits se ponen a 0.

- bMaxpower: define cuanta corriente consume el bus. El valor de este campo debe ser igual a la mitad de mA requeridos. Esto se debe a que la corriente máxima que puede suministrar el bus a un periférico es de 500mA, almacenando la mitad del número permite que con un solo *byte* se pueda escribir el valor del campo. Si la corriente necesaria no está disponible, el *host* no configurará el dispositivo.
- bNumInterfaces: siempre debe haber como mínimo una *interface* por lo que su valor mínimo será 1.
- bConfigurationValue: Permite identificar la configuración cuando se recibe los *Get_Configuration* y los *Set_Configuration request*. Su valor debe ser al menos 1, es decir siempre debe haber una configuración disponible para activarse. En caso contrario el dispositivo entraría en un estado de no configurado.
- iConfiguration: si no hay un *string* que describa la configuración su valor será 0.

Posición	Campo	Tamaño en bytes	Descripción
0	bLengh	1	Tamaño del descriptor en bytes
1	bDescriptorType	1	Identificador del descriptor: 02h
2	wTotalLength	2	Número total de bytes del configuration descriptor y los descriptores subordinados
4	bNumInterfaces	1	Número de interfaces en la configuración
5	bConfigurationValue	1	Identificador para el Set_Configuratio y el Get_Configuracion request
6	iConfiguration	1	Indice del string descriptor
7	bmAttributes	1	Modo de alimentación
8	bMaxPower	1	Corriente requerida por el Bus en mA

Tabla 2.3.2.10.1 – Campos Configuration Descriptor

2.3.2.11 Interface descriptor.

Establece la *interface* que implementa el periférico, es decir, su función. Contiene información sobre la clase, subclase, protocolo y número de *endpoints* que emplea el *firmware* del dispositivo.

Una configuración puede tener múltiples interfaces activas al mismo tiempo. El *host* puede cambiar a una *interface* alternativa mediante el *Set_Interface request* y lee la *interface* actual activa con el *Get_Interface request.* Todas estas operaciones se realizan sin necesidad de cambiar la configuración actual. Cada *interface* tiene sus propios descriptores subordinados (*Endpoint Descriptor*).

El *Interface Descriptor* está formado por nueve campos ver tabla 2.3.2.11.1. Algunos dispositivos no emplean todos los campos:

- bInterfaceNumber: cada *interface* debe tener un descriptor con un valor en este campo. En caso negativo el dispositivo no se configurará.
- bAlternateSetting: cuando una configuración admite múltiples interfaces, cada una de las interfaces tendrá un descriptor con el mismo valor en el campo bInterfaceNumber y un valor único en el campo bAlternateSetting. De tal manera que el *Get_Interface request* recupera la *interface* actual. El *Set_Interface request* selecciona la *interface* a utilizar. Si no hay interfaces alternativas el valor es cero.
- bNumEndpoints: si no hay *endpoinst* adicionales al *Endpoint 0* su valor es cero.
- bInterfaceClass, bInterfaceSubClass, bInterfaceProtocol: funcionan de manera similar a los campos bDeviceClass, bDeviceSubClass y bDeviceProtocol (ver sección *Device Descriptor*). Se emplean en dispositivos que definen su función en la *interface*, que son la mayoría de los periféricos. Para el primero de ellos, bInterfaceClass, se recoge en la tabla 2.3.2.11.2, los códigos para cada clase de dispositivo. En el caso de la bInterfaceSubClass presenta un valor de cero. Si no hay clase, un valor entre 1 y FEh. Si la clase está definido por la especificación y un valor de FFh significa que está definido por el fabricante. bInterfaceProtocol presenta los mismos valores que el campo bInterfaceSubClass.

Posición	Campo	Tamaño en bytes	Descripción
0	bLength	1	Tamaño del descriptor en bytes
1	bDescriptorType	1	Identificador del descriptor: 04h
2	bInterfaceNumber	1	Número de identificación de la interface
3	bAlternateSetting	1	Seleccionar interfaces alternativas
4	bNumEndpoints	1	Número de endpoints soportados sin contar el endpoint 0
5	bInterfaceClass	1	Clase del dispositivo
6	bInterfaceSubclass	1	Subclase del dispositivo
7	bInterfaceProtocol	1	Protocolo
8	iInterface	1	Indice del string descriptor de la interface

Tabla 2.3.2.11.1 – Campos Interface Descriptor

Código de la clase	Descripción
01h	Audio
02h	Communication Device Class (CDC)
03h	Human Interface Device (HID)
05h	Physycal
06h	Image
07h	Impresora
08h	Dispositivo de almacenamiento (Mass storage)
09h	Hub
0Bh	Smart Card
0Eh	Video
DC	Dispositivo de diagnóstico
E0	Wireless controller
FE	Aplicación específica
FF	Definido por el fabricante

Tabla 2.3.2.11.2 - Clases de dispositivo

2.3.2.12 Endpoint Descriptor.

Cada *endpoint* definido por el *firmware* tiene su propio *endpoint* descriptor a excepción del *Endpoint 0*, el cual está definido por el *Device Descriptor*.

El descriptor está formado por cinco campos ver tabla 2.3.2.12.1. Algunas particularidades de los campos:

- bEndpointAddress: los 8 bits que lo forman se distribuyen de la siguiente forma:
  - Bits 0 a 3: número del endpoint, que funciona como su identificador.
  - Bit 7: dirección, 1 si es un In-endpoint y 0 si es un Out-endpoint.
  - Resto de los bits a cero.

- bmAttributes: bits 1 y 0 especifican el tipo de transferencia que soporta el dispositivo:
  - 00 = Control transfer.
  - 01 = Isochronous transfer.
  - 10 = Bulk transfer.
  - 11 = Interrrupt transfer.

  Bits 2 a 5 para indicar el modo de velocidad que soporta el *endpoint* (*full* o *high speed*). Y los bits 3 y 2 indican el tipo de sincronización. Bits 5 y 4 indican el tipo de uso:

  - 00 = data endpoint.
  - 01 = feedback endpoint.
  - 10 = implicit feedback data endpoint.
  - 11 = reserved.

  Bits 6 y 7 a cero.

- wMaxPacketSize: máximo número de *bytes* que el *endpoint* puede transferir en cada transacción. El tamaño lo indica mediante los bits 10 a 0 (le permite un valor máximo de 1023bytes). Los bits 12 y 11 se emplean para indicar el número de transacciones adicionales por *microframe* que puede soportar un *endpoint* configurado como *high-speed*:
  - 00 = sin transacciones adicionales (número de total de transacciones por *microframe* = 1).
  - 01 = 1 transacción adicional (número de total de transacciones por *microframe* = 2)
  - 10 = 2 transacciones adicionales (número de total de transacciones por *microframe* = 3).
  - 11 = *reserved*.

- bInterval: indica tres propiedades del *endpoint*. El valor máximo de la latencia en el caso de que se hayan seleccionado *interrupt endpoints*, el intervalo de *polling* para *isochronous endpoints* y el máximo *NAK ratio* (ver sección de

definiciones y abreviaturas) para *bulk endpoints* configurados como *high-speed* y para el *control endpoint*.

Posición	Campo	Tamaño en bytes	Descripción
0	bLength	1	Tamaño del descriptor en bytes
1	bDescriptorType	1	Identificador del descriptor: 05h
2	bEndpointAddress	1	Número y dirección del endpoint
3	bmAttributes	1	Tipo de transferencia que soporta el endpoint
4	wMaxPacketSize	2	Tamaño máximo del paquete de datos soportado
5	bInterval	1	Máxima latencia/ intervalo de polling/ NAK rate

Tabla 2.3.2.12.1 – Campos Endpoint Descriptor

2.3.2.13 String Descriptor.

El string descriptor es completamente opcional en la mayoría de clases de dispositivos. Únicamente contiene un texto descriptivo transmitido como cadena de caracteres que pueden aportar información directa al usuario.

Un dispositivo puede no tener *string descriptor*, como tener uno o varios. Generalmente aportan información del nombre del dispositivo/producto, fabricante, número de serie, configuración e *interface*.

El *host* recibe el *String Descriptor* mediante el envío de un *Get_Descriptor request*, con el *byte* del campo vwValue de la transacción *Set_up* igual a 3.

Está formado por 3 campos ver tabla 2.3.2.13.1. A continuación se comentan algunas particularidades de los campos:

- wLangid: se emplea en el *String Descriptor 0*. Este *string* contiene códigos de uno o más 16bit-language ID, que indican el idioma para el que los *strings* están disponibles, de tal manera que cuando el sistema operativo del *host* los reciba pueda mostrarlos en el idioma seleccionado. El código para el inglés es 0009h que están disponibles en los anexos adicionales a la especificación. Los dispositivos que no disponen de *String Descriptor* en su *firmware* no devuelven ningún valor de *languageID*.
- bString: los caracteres que se transmiten en los *String Descriptor* están codificados en Unicode (ver sección de definiciones y abreviaturas).

Posición	Campo	Tamaño en bytes	Descripción
0	bLength	1	Tamaño del descriptor en bytes
1	bDescriptorType	1	Identificador del descriptor: 03h
2	bString o wLangid	[0...N]	Indica Language ID en el string descriptor 0 y para el resto de string descriptors establece la codificación Unicode

Tabla 2.3.2.13.1 – Campos String Descriptor

2.3.3 Human Interface Device Class.

Esta clase incluye todos los dispositivos que emplean las personas para controlar los sistemas operativos (ordenador). Dentro de ella podemos encontrar periféricos como: teclados, ratones, joysticks, paneles de control, etc. Aunque también se incluyen otros dispositivos que a pesar de no tener una interacción directa humano máquina, también emplean este tipo de clase para transmitir datos: termómetros, lectores de código de barra, etc. En general se caracterizan por una transmisión muy rápida de los datos de tal manera que una persona no pueda notar el retardo que se produce en el envío entre *host* y dispositivo USB.

En la clase HID los datos se transmiten siguiendo un formato determinado denominado *HID report*. Esto le permite a los dispositivos HID responder rápidamente a los *request*, ahorrar memoria, e interactuar con aplicaciones genéricas de forma sencilla.

Al ser una clase de dispositivo propia, definida por la especificación USB, presenta su propia especificación, por lo que las siguientes secciones están conforme a la versión 1.11 de la especificación para la clase HID.

2.3.3.1 Subclase y protocolos.

Esta clase de dispositivos está pensada para abarcar un gran número de periféricos con características propias y funciones diferentes. Si se estableciesen subclases se estaría obligando a los fabricantes a trabajar con unos periféricos específicos, que fuera de ellos los dispositivos no funcionarían correctamente. Este problema se solventó mediante el empleo de mensajes con un formato, los *reports*, evitando el tener que crear un número inmenso de subclases.

En cambio sí que existen protocolos (ver tabla 2.3.3.1.1). La especificación define dos protocolos para dos dispositivos muy relacionados con los PC: el ratón y el teclado. Esto se debe a que para los *drivers* HID del sistema operativo, son periféricos con permisos especiales, por lo que Windows interrumpe su funcionamiento normal para atenderlos siempre que lo deseen.

Código Protocolo	Dispositivo
0	Genérico
1	Teclado
2	Ratón
3-255	Reservado

Tabla 2.3.3.1.1 – Protocolos HID

2.3.3.2 Tipos de transferencia.

Esta clase de dispositivos sólo soporta dos tipos de transferencia:

- *Control transfer*: la cual es común a todas las clases de dispositivos.
- *Interrupt transfer*: se adapta perfectamente a las necesidades de esta clase de dispositivos los cuales deben compartir datos de forma asíncrona (sin que se produzcan *request*) con el host, y mantener una latencia baja. Como mínimo deben tener un *endpoint* que soporte este tipo de transferencia, aunque dicho *endpoint* no tiene porque poseer un *endpoint Out* (como en el caso de los ratones).

2.3.3.3 Report.

Describe el formato de los datos que se van a transmitir.

Los *report* están formados por pequeños fragmentos de datos denominados *items* que son los que definen el formato del mensaje y que deben aparecer configurados como tal en el report descriptor. Se transmiten por el *Interrupt In pipe* y por el *Interrupt Out pipe* (aunque también se puede realizar la transferencia a través del *Control pipe*). Está dividido en dos campos principales: el campo de datos y el del *Report ID* (ver figura 2.3.3.3.1).

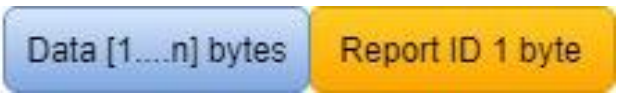


Figura 2.3.3.3.1 – Formato report

Todos los *items* están almacenados en el campo data. El tamaño de este campo es variable debido a la existencia de ciertos *items* que permiten modificar el propio campo (ver sección descriptores).

El *report ID* debe ser declarado en el *report* descriptor. Si se declara, todos los descriptores deben llevar este prefijo. Por el contrario, si no se declara, todos los valores retornan en un único *report* el cual carece de este campo.

En general los *report* se pueden clasificar en función de la dirección de envío que marque su *main item* (ver sección descriptores):

- *Input report*: transferencia de datos de dispositivo al *host*.
- *Output report*: transferencia de datos del *host* al dispositivo
- *Feature report*: transferencia de datos bidireccional para determinados tipo de parámetros de configuración.

Existen algunas restricciones en cuanto al formato del *report*:

- Sólo se permite un *report* por transferencia USB. Aunque un *report* puede obligar a realizar varias transacciones.
- El nivel más alto de una *collection* debe ser una *application*, y sólo debe haber una *application* por cada *collection*.
- Una vez que se ha determinado el tamaño del *report* siempre se debe transmitir el mismo número de *bytes* (hasta que se produzca un cambio de *Collection*), en caso de no completar el número de *bytes* solicitado se rellenan los *report* con 0's.

2.3.3.4 Descriptores.

Se ha comentado anteriormente (ver antecedentes sección descriptores), la posibilidad de que determinadas clases de dispositivos tengan descriptores propios, éste es el caso de la clase HID (ver figura 2.3.3.4.1). La manera que tiene de definir los formatos de los *report* es mediante el denominado *HID descriptor* el cual tiene dos descriptores subordinados: *Report Descriptor* y *Physical Descriptor*.

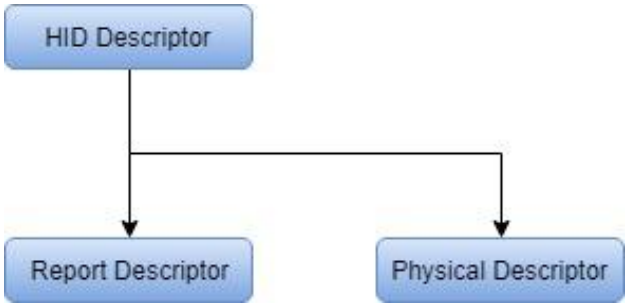


Imagen 2.3.3.4.1 – Diagrama descriptores específicos HID

La especificación dedicada a esta clase de dispositivo también establece que la clase HID debe ser definida en el *Interface descriptor*, lo que le permite ser una clase compatible con otras en los dispositivos compuestos. Además debe contener un *endpoint* (*Endpoint IN* y *Endpoint Out*) de un tamaño de 64 bytes.

En las siguientes secciones se desarrollan los descriptores de forma individual.

2.3.3.4.1 HID descriptor.

El HID descriptor establece las características de los descriptores subordinados y las propiedades básicas de un dispositivo HID. Está formado por 10 campos, los cuales presentan algunas particularidades (ver tabla 2.3.3.4.1):

- bCountryCode: es el código del país donde está localizado el hardware. Normalmente su valor es cero aunque sin embargo algunos dispositivos como los teclados pueden emplear este valor para indicar el lenguaje en el que vienen configuradas las letras .
- bNumDescriptors: como mínimo debe ser igual uno, ya que todo HID descriptor debe contener como mínimo un *report descriptor*.

Posición	Campo	Tamaño en bytes	Descripción
0	bLength	1	Tamaño del descriptor en bytes
1	bDescriptorType	1	Identificador del descriptor
2	bcdHID	2	Número de especificación de la clase HID
4	bCountryCode	1	Identificador del país donde se fabricó el hardware
5	bNumDescriptors	1	Número de descriptores subordinados
6	bDescriptorType	1	Identificador como un descriptor de clase
7	wDescriptorLenght	2	Tamaño total del Report descriptor subordinado
9	[bDescriptorType]	1	Constante del physical descriptor opcional
10	[wDescriptorLength]	2	Tamaño total del physical descriptor opcional

Tabla 2.3.3.4.1 – HID Descriptor

2.3.3.4.2 Physical descriptor.

Es completamente opcional. Provee información sobre las partes del cuerpo humano que se emplean para activar los controles del dispositivo HID.

2.3.3.4.3 Report descriptor.

A diferencia de los descriptores vistos anteriormente el *report* descriptor no se configura como una tabla de valores, es más en todos los descriptores el tamaño y la estructura del descriptor es constante mientras que en el caso del *Report descriptor* varía dependiendo de la aplicación.

Como se ha comentado anteriormente el *report* está constituido por una colección de *items*, por lo que la función del *report descriptor* es configurar estos *items* en función de la aplicación que se esté desarrollando. Los *items* están formados por varios uno o varios *bytes* y presentan el siguiente formato clasificándose en dos grupos:

- *Short item*: tamaño total entre de 1 a 5 *bytes*. Es el formato más común para los *items.* Se divide en los campos que se muestran en la tabla 2.3.3.4.3.1.

Posición	Campo	Tamaño	Descripción
1	data	1 Byte	Datos del item
0	bTag	7-4 bits	Tamaño del item
0	bType	3-2 bits	Tipo de item
0	bSize	1-0 bits	Función del item

Tabla 2.3.3.4.3.1 – Formato *short item*

- *Long item*: tamaño total entre los 3-258 *bytes*. Es un tipo de formato que se emplea para estructuras de datos. Se divide en los campos que se muestran en la tabla 2.3.3.4.3.2.

Posición	Campo	Tamaño	Descripción
1	data	1 Byte	Datos del item
2	bLongItemTag	1Byte	Tag de long item
3	bDataSize	1Byte	Tag de tamaño para long item
0	bTag	7-4 bits	Tamaño del item
0	bType	3-2 bits	Tipo de item
0	bSize	1-0 bits	Función del item

Tabla 2.3.3.4.3.2 - Formato *long item*

Existen tres tipos de *item type*:

- Main item: se emplea para definir un grupo de cierto tipo de datos dentro del *Report Descriptor*. Está dividida en cinco categorías de *tags*:
  - Input item tag: describe la información contenida en los campos de datos, y que se envía al host.
  - Output item tag: sirve para describir el campo de datos de salida en un *report*. En cuanto a su estructura, se construye exactamente igual que el *Input item*.
  - Feature item tag: describen información sobre la configuración del dispositivo. En general este tipo de *reports* no son visibles para el software.
  - Collection item tag: define una agrupación de *Input*, *Output* y *Feaute items* que realizan una misma función.
  - End Collection item tag: cierra la *collection* de *items*.

- Global item: describe los atributos asignados al campo de datos del *Main item* asociado. Pueden modificar el valor de la tabla de estados. Deben colocarse antes de los *Input*, *Output* y *Feature tag* a los que se refieren.
- Local item: definen características de los controles que transmiten información en los campos de datos. A diferencia de los *Global items* no tienen una

colación específica sino que se asignan de manera secuencial, a medida que se declaran se van asignando a cada control que desee transmitir datos.

Aunque se ha comenzado afirmando que la estructura del *Report descriptor* es variable, existe una cantidad mínima de *items* que normalmente están presentes en este tipo de descriptor que son los siguientes:

- Input/Output/Feature.
- Usage: Local item. Define el uso de una determinada colección. Se emplea sobre todo con dispositivos que emplean múltiples controles, por lo que dentro de una misma colección se hace referencia a cada uno de ellos.
- Report ID: Global item. Los *endpoints* pueden soportar múltiples estructuras de *reports*, la forma que tienen de identificar cada una es mediante el *item Report ID*, el cual consta de 1 byte para identificar el tipo de *report*. Si no se recoge el *item Report ID* en los descriptores se asume que sólo hay una única estructura para cada tipo de transferencia *Input, Output* o *Feauture*.
- Usage Page: Global item. Proporcionan información acerca del tipo de control que actualmente están usando, es decir el *report* define el formato de los datos, así como el campo *Usage* define qué hacer con esos datos. Un *report descriptor* puede contener varios *Usage* en función de la aplicación para la que se esté desarrollando. En el descriptor el campo *Usage* ocupa un espacio de 32 bits de los cuales los 16 más altos definen el *Usage* Page y los 16 más bajos el *Usage ID*.
- Logical Minimum: Global item. Es el valor mínimo que una variable puede reportar. Cada unidad de la variable corresponde a una unidad a la hora de escribir el valor mínimo.
- Logical Maximum: Global item. Es el valor máximo que una variable puede reportar. Cada unidad de la variable corresponde a una unidad a la hora de escribir el valor máximo. En el caso de que este *item* y el *item logical minimum* sean positivos se asume que el dato a enviar será un valor sin signo. En caso contrario se transmiten en complemento a 2.
- Report Size: Global item. Establece el tamaño de cada campo del report en bytes. Le permite al analizador del *driver* construir un *item map* del *report handler* para poder usarlo.

- Report Count: Global item. Indica el número de campos que contiene el report.
- Application: Collection item. Agrupa un conjunto de *items* de *Main items* que realizan una función similar dentro del dispositivo. Puede haber más de una *collection* dentro de un periférico. Ejemplos de *collection* sería la que agrupa todas las teclas de un teclado.

El formato de estos *items* se debe completar con una combinación específica de bits para cada uno y en especial en el caso de los *items Usage* y *Usage Page* que son los que marcan la función final de los *reports* y que dependen del tipo de dispositivo que se esté implementando. Algunas combinaciones son variables como puede ser el caso del *Report Count* o el *Report Size* (ya que como se ha visto anteriormente su función es configurar el propio *report*), pero aún así se deben codificar con la misma combinación de bits que definen parámetros como el *item type*, *item tag*, etc. vistos anteriormente. Estos códigos se recogen en una serie de tablas proporcionados por USB-IF, que se pueden localizar en dos documentos:

- Device Class Definition for Human Interface Devices HID.
- HID Usage Tables.

En la tabla 2.3.3.4.3.3 se muestran los valores para los *items* principales vistos anteriormente:

Item	Item tag	Valor
Application	Collection	0x01
Usage Page	Global	0b000001..[n..n]
Logical Minimum	Global	0b000101...[n..n]
Logical Maximum	Global	0b001101...[n..n]
Report Size	Global	0b011101...[n..n]
Report ID	Glogal	0b100001...[n..n]
Report Count	Global	0b100101...[n..n]
Usage	Local	0b000010...[n..n]
End Collection	Main item	0b110000...[n..n]

Tabla 2.3.3.4.3.3 – Principales items y su valor

En el caso particular, de los *items tags Input*, *Output* y *Feature* se han recogido en una tabla a parte, donde se va a tratar cada bit de forma individual. Se hace esto porque a diferencia de los anteriores, los cuales sí que presentan una configuración constante, estos *items* cambian constantemente en función de los datos que se pretenden enviar en el *report*(ver tabla 2.3.3.4.3.4).

Main item	Data	Valor	Descripción
Input, Output, Feature	Bit 0	Data(0)/Constante(1)	Indica si se trata de una constante o de un dato
	Bit 1	Array(0)/Variable(1)	Indica si se crea un array dentro de un campo o una variable
	Bit 2	Absoluto(0)/Relativo(1)	Indica si el dato es absoluto o relativo (especifcar origen)
	Bit 3	No Wrap(0)/Wrap(1)	Indica cuando un dato supera el valor máximo y en ese momento lo vuelve inicializar
	Bit 4	Lineal(0)/No Lineal(1)	Indica si existe una relación entre dos datos que se envían en un report
	Bit 5	Preferred State(0)/No preferred(1)	Indica la existencia de un estado al cual el dispositivo puede retornar si no se está produciendo ninguna acción

	Bit 6	No Null(0)/Null state(1)	Indica si se debe ignorar determinados campos en los cuales no hay estados cargados
	Bit 7	Reserved(0)	-
	Bit 8	Campo de bit(0)/Registro de Bytes(1)	Indica si el contenido del campo debe ser interpretado bit a bit o a tratado como una cadena de Bytes
	Bit 31-9	Reserved (1)	-

Tabla 2.3.3.4.3.4 – Formato item tags

2.3.3.5 Request.

En la sección 1.3.2.7, se comenta que existen *request* a mayores de los *standard request*. La clase HID tiene seis *request* específicos que le permiten al *host* obtener información sobre los estados de los *items* que constituyen los *report*. El formato de cada uno de los *request* es el mismo que el expuesto en la sección de *request* (ver sección 1.3.2.6 standard request):

- *Get_Report Request*: permite al *host* recibir un *report* a través del *Control pipe*; se emplea para *inicializar* los *absolute items* y determinados estados de los *feature items*.
- *Set_Report Request*: permite al *host* enviar un *report* al dispositivo, permitiendo modificar el estado de los *Input*/O*utput*/*Feature reports*.
- *Get_Protocol Request*: lee el protocolo que está activo.
- *Set_Protocol Request*: selecciona el protocolo requerido por el descriptor.
- *Get_Idle Request*: lee el estado *idle* para un *Input report*.
- *Set_Idle Request*: paraliza el envío de un *report* mediante *Interrupt In* hasta que ocurra un evento en un período de tiempo específico. Se emplea para

limitar la frecuencia de envío de *reports* a través del *interrupt endpoint in*. La duración del retardo en el que el dispositivo está en *idle* es variable, siendo de 500ms para los teclados e infinita para ratones o *game controllers*.

2.3.3.6 HID driver.

Se trata de una de las mayores ventajas de la clase HID, y es que el driver ya viene instalado en todos los sistemas operativos por defecto.

Contiene un analizador, que se encarga de leer el *report* descriptor y almacenar el estado de los *item* en una tabla de estados. De esta forma se podrá leer e interpretar los *report*.

Desde el punto de vista del analizador los *item* siguen una estructura jerarquizada como se puede observar en el diagrama (ver imagen 2.3.3.6.1), el cual define la especificación para esta clase.

Cuando el analizador encuentra un *Main item*, selecciona inmediatamente esa configuración del *report* e inicializa de nuevo la tabla de valores. En ese momento elimina todos los *item* de tipo Local pero mantiene los de tipo Global. Para remover los *item* Globales es necesario definir una nueva *collection*. Este es el método general por el cual el *driver* actualiza su tabla de estados en función del *Main item*.

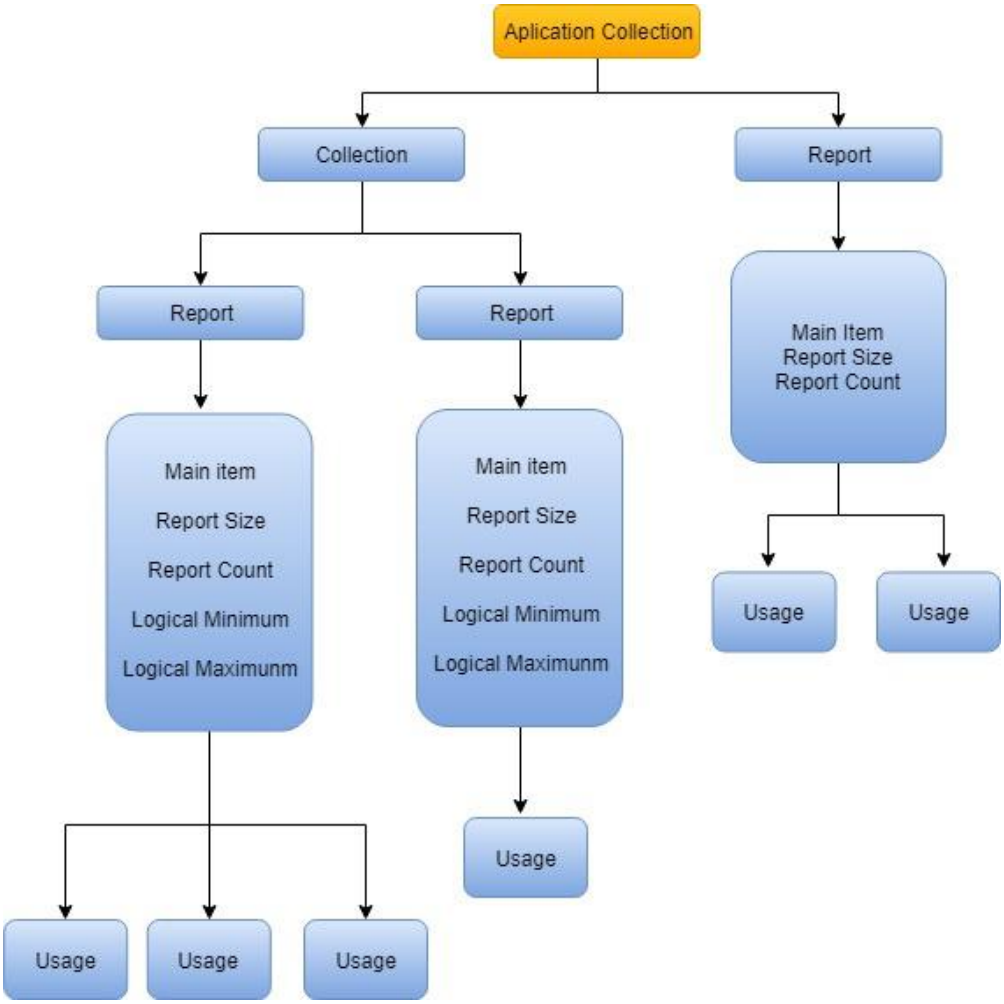


Figura 2.3.3.6.1 – Analizador

2.4 Normas y referencias

2.4.1 Bibliografía

[1] **Jan Axelson**.; "*USB Complete Third Edition" [en línea]. Jan Axelson's Lakeview Research* 2005.

[2] **Varios**.; "*Universal Serial Bus Specification" [en línea]. usb.org* Revisión 2.0, 2000.

[3] **Varios**.; *"Universal Serial Bus Device Class Definition for Human Interface Devices (HID)" [en línea] usb.org* versión 1.11, 2001.

[4] **Varios**.; "*Universal Serial Bus HID Usage Tables" [en línea]. usb.org,* versión 1.12, 2004.

[5] **Shijas Mayan**.; *"AN1546 USB Keypad Reference Design" [en línea]. Microchip AppNotes* 2013 [Fecha de consulta: 24 de Junio del 2018].

[6] **Reston Condit**.; "*TB054 An Introduction to USB Descriptors with a Game Port to USB Game Pad Translator Example*" *[en línea]. Microchip AppNotes 2013* [Fecha de consulta: 2 de Julio del 2018].

[7] **Reston Condit**.; "*TB055 PS/2 to USB Mouse Translator*" *[en línea]. Microchip AppNotes 2013* [Fecha de consulta: 3 de Junio del 2018].

[8] **Reston Condit**.; "*TB056 Demonstrating the Set_Report Request*" *[en línea]. Microchip AppNotes, 2013* [Fecha de consulta: 3 de Junio del 2018].

[9] **Signal11**.; *"HID API for Linux, Mac OS X, and Windows*" *[en línea]. signal11.us, 2010* [Fecha de consulta: 3 de Junio del 2018].

[10] **Beyond Logic**.; "*USB in a Nutshell*" *[en línea]. beyondlogic.org, 2008* [Fecha de consulta: 14 de Julio del 2018].

[12] **The MathWorks, Inc**.; "*External Interfaces Reference.calllib*"*[en línea]. matlab.izmiran.ru, 2005* [Fecha de consulta: 23 de Agosto del 2018].

[12] **The MathWorks, Inc**.; "*External Interfaces Reference.loadlibrary*"*[en línea]. matlab.izmiran.ru, 2005* [Fecha de consulta: 20 de Agosto del 2018].

2.4.2 Programas de cálculo

Matlab R2014a.

Visual Studio 2011.

MPLAB X IDE v4.15 compilador XC8.

MPLAB IPE v4.15

Proteus 8 Professional.

KiCad+ visualizador 3d Wings 3D.

Wireshark+USBCapture.

Microsoft Office Word.

Microsoft Office Excel.

Autocad.

2.5 Definiciones y abreviaturas

- ADC: *analog digital converter* (convertidor analógico digital).
- b-: byte (8 bits).
- bcd-: binary-code decimal.
- bm-: bit map.
- Gbps: *Gigabits* por segundo.
- DAC: digital *analog converter* (convertidor digital analógico).
- DAQ: Data Adquisition. Tarjeta de adquisición de datos.
- Duty Cicle: ciclo de trabajo
- Firmware: programa que establece la lógica que gobierna la electrónica de un dispositivo.
- -h: valor en hexadecimal.
- HID: Human Interface Device.
- I-: index.
- Id-: identifier.
- Kernel: es el núcleo del Sistema Operativo que permite comunicar el *software* del sistema con el *hardware* (funciona a modo de *firmware)*. Lo más importante es que contiene los *drivers* del Sistema Operativo.

- *LSB*: bit más bajo.
- Mbps: *Megabits* por segundo.
- *MSB*: bit más alto.
- Puerto USB: es un término que engloba tanto la parte del conector USB como el *host controller.*
- PWM: Modulación por ancho de pulso (Pulse Width Modulation).
- SIE: serial interface engine
- Sniffer: programa de captura y análisis de paquete de datos.
- Tad: tiempo de conversión del convertidor, es el tiempo que tarda en completar la conversión de un bit, para una conversión de 10 bit, el tiempo total requerido es de 11.5 Tad períodos.
- USB On-the-Go: se trata de una revisión de la especificación 2.0 que le permite a los dispositivos adquirir funciones reservadas al host. Como por ejemplo una cámara que se conecta directamente a una impresora y le envía las fotos.
- Unicode: unicode emplea 16 bits para representar cada carácter de alfabeto. En general se puede establecer un paralelismo entre la codificación normal ANSI de tal manera que los códigos que van del 00h al 7Fh corresponden a valores Unicode de 0000h a 007Fh.
- w-: word (16 bits).

2.6 Requisitos de diseño

Emplear el programa Matlab.

Conexión USB pura sin usar Puerto Serie.

2.7 Análisis de las soluciones

El presente trabajo tiene como objeto implementar un protocolo de comunicación basado en USB sobre un microcontrolador del tipo PIC, para luego comunicarlo con Matlab. Pero no sólo se busca lograr la comunicación, sino que ésta mejore la

tasa de transferencia de datos de las tarjetas de adquisición y hardware comercial, es decir que sea más rápida.

Para poder aproximarse a la idea que se está persiguiendo a lo largo de este proyecto, se va partir del estudio de la comunicación de una tarjeta comercial muy popular basada en el arduino. Exactamente nos vamos a fijar en el módelo de Arduino que usa el microcontrolador ATMega2560.

Este hardware se comunica con el ordenador a través de USB, es más, está basado en la especificación USB 2.0 (ver antecedentes), que es la que se va a implementar en el PIC (es la especificación que soporta). Pero el Arduino Mega emplea el protocolo USB, para emular una comunicación Serie a través de los puertos virtuales COM del ordenador. Lo que le permite beneficiarse de todas las ventajas del USB: *plug and play,* conectores USB más sencillos y más pequeños que los conectores RS232 por lo que se puede integrar más fácilmente en un ordenador, es más barato, alto grado de compatibilidad, etc. y por otro lado presenta las ventajas de la comunicación Serie: protocolo de comunicación sencillo, con un simple driver se puede operar directamente los datos recibidos, al transmitir a bit a bit no hace falta adaptar los datos a un formato específico lo que lo hace especialmente útil para los instrumentos de medida, ahorro de memoria debido a que el protocolo es más sencillo, etc.

Lo más fácil sería implementar un puerto Serie emulado en el PIC mediante el empleo de la clase USB CDC y un driver genérico, que se puede descargar de la página web del fabricante Microchip. Pero se estaría replicando el mismo método de comunicación que el Arduino, con el handicap de que Arduino dispone de un driver dedicado (incluido cuando se descarga el entorno IDE de Arduino), un microcontrolador y un hardware de mayores prestaciones que el que se va emplear para este proyecto (todo esto es extensible para el resto de tarjetas comerciales). En consecuencia, no se podrá mejorar la comunicación si se está haciendo lo mismo.

Entonces teniendo en cuenta lo expuesto en el párrafo anterior, ¿qué se puede hacer para conseguir una comunicación más rápida mediante USB?. Hay un punto clave en la comunicación Serie que es que se transmite bit a bit, lo cual es

muy limitado cuando se requiere transmitir datos de gran tamaño. Esto llevó al desarrollo de métodos de comunicación en paralelo para poder transmitir varios bits al mismo tiempo, lo que mejoraba enormemente las tasas de transferencia, con la contraprestación de que aumenta el cableado (un cable para cada bit), el tamaño de los conectores y por supuesto el coste. Es en este punto donde entra en escena el USB, el cual mediante un único cable (ver antecedentes), que sin necesidad de tarjetas de comunicación diferentes en función de los tipos de protocolo, y conectores universales puede transmitir paquetes de datos de varios *bytes* lo que mejora enormemente las tasas de transmisión.

Volviendo al caso en particular del Arduino Mega, éste sacrifica velocidad (tasa de transferencia), en aras de conseguir una mayor funcionalidad. Para poder implementar una comunicación Serie sobre el protocolo USB emplea un microcontrolador a mayores del ATMega2560 que es el ATMega16U2 (en la imagen 2.7.1 se puede observar en la zona marcada), el cual funciona como un convertidor a puerto serie que soporta la comunicación USB entre el PC y el Arduino, y junto con el *driver* que instala el IDE de Arduino, el cual contiene un terminal que permite la comunicación serie.

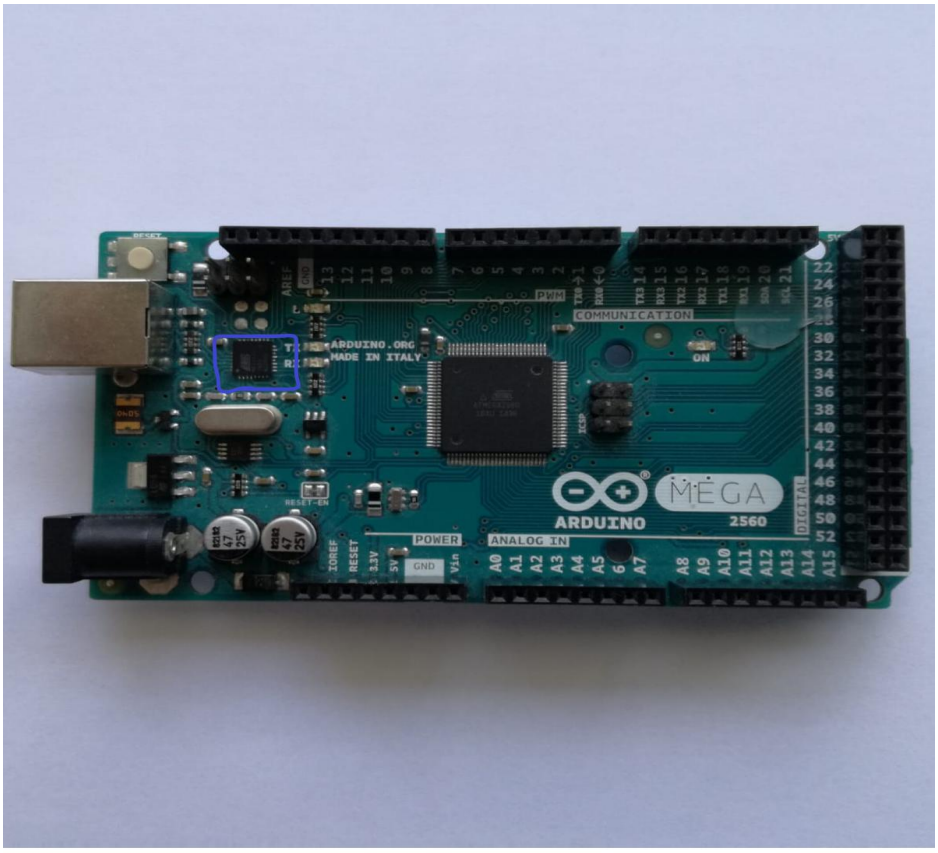


Imagen 2.7.1 – Arduino Mega 2560

Si se busca el Arduino en el administrador de equipos se puede observar que aparece en los puertos COM del ordenador, asignados a la comunicación serie (ver imagen 2.7.2)

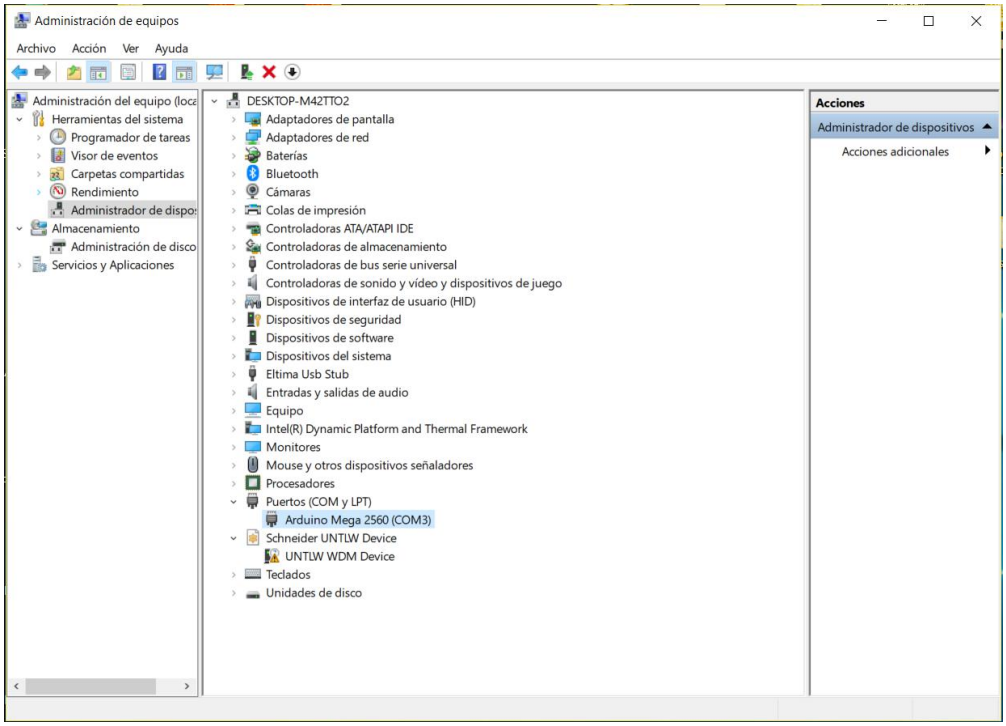


Imagen 2.7.2 – Arduino administrador de dispositivos

Todo esto se traduce en que la velocidad de transferencia máxima normalizada de Arduino es de 115200 *baudios* (aunque existen métodos para aumentarla un poco más) mientras que el USB 2.0 ofrece una velocidad de transferencia máxima teórica de 480Mbps (aunque la tasa de transferencia real es de 280Mbps). Por lo que el convertidor Serie, ateniéndonos sólo a las tasa de transferencia está capando la velocidad máxima que puede desarrollar en este caso el Arduino (lo que es extensible a numerosas tarjetas comerciales). Con esta idea, ahora sí que se puede afrontar la pregunta que se formulaba anteriormente, y la respuesta por la que se ha optado es la de suprimir el convertidor y por ende la comunicación por Puerto Serie.

Esto supondrá desarrollar un protocolo en cuatro etapas (ver figura 2.6.3) basado en el protocolo USB. Motivado por el empleo del microcontrolador PIC y que el

Arduino y demás *hardware* similar no se puede emplear debido a la presencia de convertidores, se creará una tarjeta de adquisición de datos desde cero, que soporte la comunicación USB con sus correspondientes circuitos.

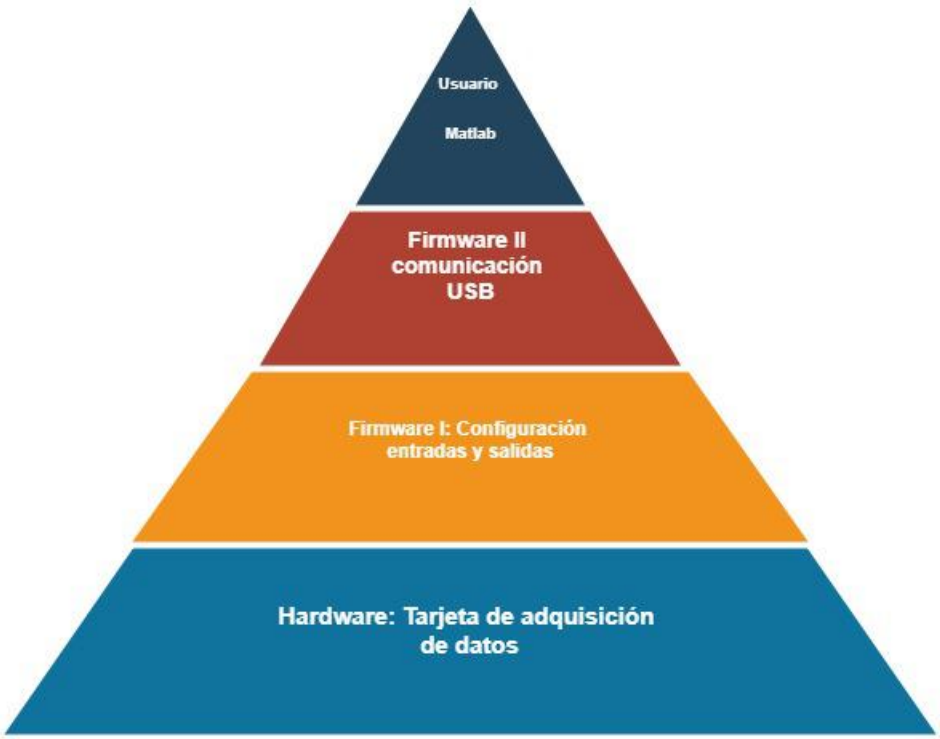


Figura 2.6.3 – Diagrama capas del protocolo

- La capa más alta del protocolo, será con la que el usuario interactuará directamente. Engloba los procesos de presentación de datos y desarrollo de la aplicación. En este caso siguiendo los requisitos de diseño, la aplicación seleccionada es Matlab, a través de ella el usuario podrá mandar comandos a la tarjeta de adquisición de datos y al mismo tiempo leer todos los sensores conectados a sus entradas.
- La capa intermedia *Firmware* II: tanto ésta como la siguiente capa son la misma, aunque se ha decidido dividirlas en dos, ya que contienen operaciones complementarias pero bastante diferenciadas. *Firmware* II contiene toda la comunicación USB y la secuencia de operaciones para poder enviar los datos y actúa como una capa de transporte.
- La capa intermedia *Firmware* I: contiene la configuración del microcontrolador, y los correspondientes periféricos para permitirle funcionar como una tarjeta de adquisición de datos.

- La capa más baja, la capa física (*hardware*), que está formada por todos los circuitos (alimentación, filtros, cable USB, puertos, etc.), que le permiten a la tarjeta de adquisición de datos funcionar y comunicar con el PC.

Ahora bien, la pregunta que se podría formular es, ¿cómo va implementar el protocolo la comunicación USB con el PC si descartamos el método de emular un puerto serie? Para poder responder ahora esta pregunta hay que tener en cuenta una serie de factores:

- No se pueden crear *drivers* dedicados como en el caso de las tarjetas comerciales. Debido a que el protocolo debe funcionar en todos los PC's sin que haya necesidad de realizar instalaciones previas y teniendo en cuenta que el desarrollo de *drivers* es bastante complejo.
- La tarjeta va a transmitir pequeñas cantidades de datos en momentos muy concretos, pero no de manera continúa. Sólo va a transmitir cuando se produzcan lecturas o cuando el PC desee activar algunas de sus salidas.

Por lo tanto la opción más viable es cambiar la clase del dispositivo (ver antecedentes). Por ejemplo el Arduino mencionado anteriormente implementa la clase CDC la cual obliga al empleo del *driver* y del puerto Serie, obviamente no es válida. Mientras que dispositivos como los *pendrives* emplean la clase *MassStorage* pensada para grandes transferencias de datos en el momento que lo solicita, lo que contradice el segundo punto.

En este contexto sólo hay una clase de dispositivo USB que se adapta completamente a las necesidades del protocolo que se está desarrollando, la clase HID (ver antecedentes). Los *drivers* que necesita el PC para operar esta clase vienen instalados por defecto en los sistemas operativos, además de que el método de transferencia que implemente es el *Interrupt* (ver antecedentes) que se adapta perfectamente a la forma de trabajo de una tarjeta de adquisición de datos.

Lo más importante es que la clase HID no se va conectar emulando un puerto Serie, ni necesita de un convertidor, sino que va a transmitir aprovechando todo el ancho de banda del protocolo USB, de tal forma que una tarjeta configurada como un dispositivo de esta clase podría alcanzar esa tasa de transferencia teórica de

480Mbps (hay que tener en cuenta que la tasa de transferencia real es inferior), mejorando de esta forma la velocidad con respecto a las tarjetas comerciales y al hardware como Arduino.

En las siguientes secciones se explican cada una de las capas de protocolo, y su función en el esquema del protocolo creado.

2.7.1 Capa física

La primera capa está formada por la tarjeta de adquisición de datos. Como se ha mencionado anteriormente, en el caso de este protocolo no se pueden emplear tarjetas convencionales o *hardware* como Arduino debido a que no implementan microcontroladores PIC y suelen llevar incorporados convertidores a puerto serie o directamente emplean puerto serie o puerto paralelo. Por lo que se ha optado por diseñar una DAQ nueva, que cumpla los requisitos de diseño.

Para cumplir con estos requisitos la tarjeta de adquisición de datos implementará las siguientes funciones:

- 4 Entradas con un rango de tensión de 0 a 5V.
- 2 salidas PWM con un rango de tensión de 0 a 5V.
- 1 salida analógica con un rango de tensión de 0 a 5V.
- *Leds* para señalizar estado de funcionamiento.
- Botón de *reset*.
- Un modo de ahorro de energía.

Ver el plano del esquemático de la placa en la sección de planos.

El microcontrolador seleccionado es un PIC16F1455, con un encapsulado PDIP (ver imagen 2.7.1.1). En la DAQ va montado sobre zócalo, de tal forma que se puede retirar en cualquier momento por si se necesitase hacer cambios en su *firmware* o se averiase.

Imagen 2.7.1.1 – Encapsulado PIC

La conexión y la alimentación (ver modos de alimentación en la sección módulo USB) de la tarjeta de adquisición de datos se realiza a través de un conector USB hembra tipo B. Debido a que los conectores de los PC son hembra del tipo A, y la especificación obliga a que la conexión se realice siempre tipo A-B o B-A (ver antecedentes). El cable, por supuesto, será del tipo A-B (ver imagen 2.7.1.2).

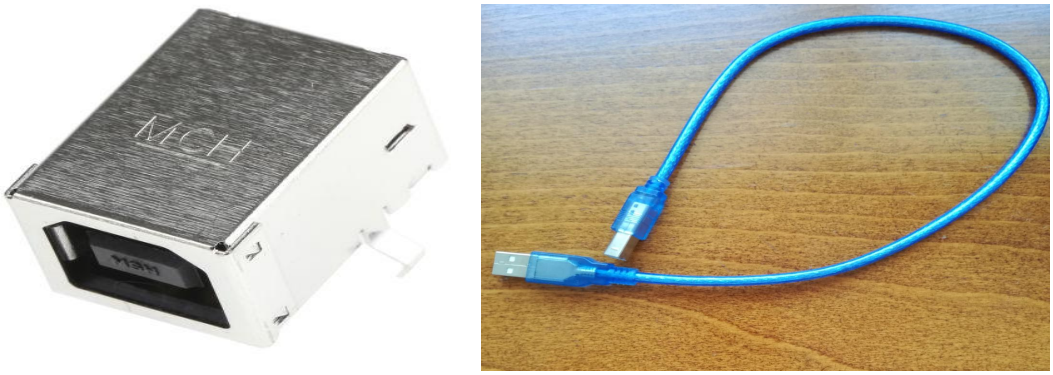


Imagen 2.7.1.2 – Conector tipo B y cable A-B

El conector tiene cuatro pines: Vd, Vs, D+ y D-, siguiendo la especificación USB 2.0. Se han descartado los conectores micro USB debido a su mayor coste y a que con el tiempo tienden aflojarse y pueden aparecer problemas de que el cable no se conecte correctamente. La tensión máxima con la que se alimenta el PIC es de 5V.

Se ha diseñado un circuito de acondicionamiento para las entradas y salidas de la placa formado por amplificador y un filtro paso bajo (ver Anexo IV de cálculos). El objetivo es lograr una impedancia de entrada infinita y una de salida nula.

En el caso de los amplificadores se emplea dos integrados AD8659, los cuales son un cuádruple amplificador. Cada uno de los amplificadores está configurado como seguidor de tensión. En el caso de la salida DAC, el *datasheet* del fabricante recomienda emplear un seguidor de tensión o un elemento pasivo para que haya consumo de corriente.

Se ha seleccionado el integrado AD8659 debido a que es de bajo consumo (22µA de consumo de corriente por amplificador), algo importante teniendo en cuenta que la corriente que puede ofrecer el bus es como máximo de 500mA, y que implementa un modo *Rail to Rail,* lo que permite que con una alimentación de 5V (máxima que puede ofrecer Vbus) se obtenga una salida de 0 a 5V. Finalmente se ha seleccionado un encapsulado de tipo *SOIC_N* para su inserción en la placa como componente de montaje superficial. Para estabilizar la alimentación de este integrado se emplea un condensador de 0.1µF (ver imagen 2.6.1.2).

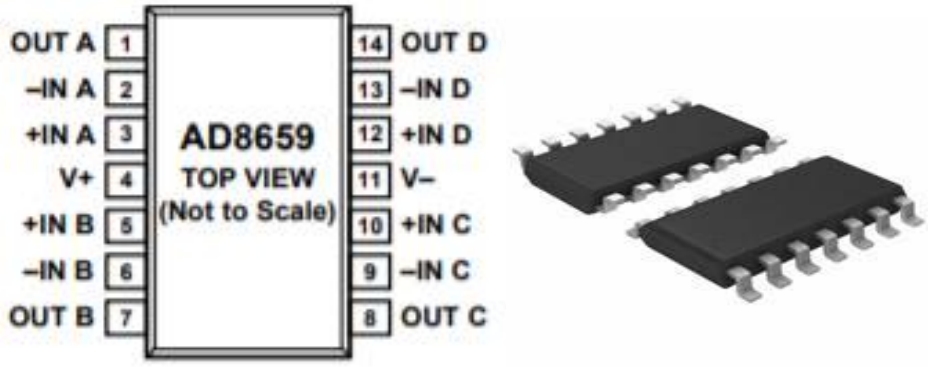


Imagen 2.6.1.2 – AD8659 y encapsulado

Se han introducido 6 filtros RC paso bajo, los cuales se conectan a la entrada de los amplificadores, distribuidos de las siguiente forma:

- 2 filtros para las salida PWM (ver anexo IV de cálculos).
- 4 filtros para las entradas analógicas (ver anexo IV de cálculos).

También se ha incluido un led para indicar que la DAQ se encuentra alimentada, y otro que va estar controlado por el PIC para indicar cuando se está enviando datos. Tanto los leds, como las resistencias y los condensadores son de montaje superficial.

Se añade un condensador a mayores de los necesarios para los filtros, para la alimentación del *transceiver* del módulo USB del PIC, debido a que la alimentación de la tarjeta es de 5V mientras que la del *transceiver* debe ser de 3.3V (como especifica el *datasheet* del fabricante). Y otros dos condensadores de valor 0.1μF para estabilizar la tensión de los integrados.

Los conectores empleados serán de tornillo, distribuidos de tal forma que por cada entrada o salida a su lado tenga una masa, lo que resulta en un conector de seis *pads* para las entradas situadas a la derecha del PIC y un conector de 8 *pads* para las salidas situadas a la izquierda del PIC.

En el caso de las 4 entradas se ha añadido una protección mediante diodos Zener para evitar que sobretensiones que puedan dañar el PIC.

También se ha incluido un botón de *reset* por si el usuario necesita reiniciar su funcionamiento en algún momento.

Para finalizar el *hardware* de la placa, esta se ha diseñado mediante el programa Kicad, fijando un ancho de pista de 0,4 mm.

2.7.2 Firmware I: configuración de entradas y salidas

Es la segunda capa del protocolo y se encarga de la configuración del microcontrolador, es decir, la configuración de las entradas y salidas de la DAQ.

Como ya se ha comentado en la sección anterior, el PIC seleccionado es el PIC16F1455 de la casa Microchip Technology. El motivo principal es que el microcontrolador posee el módulo USB que soporta la especificación USB 2.0, y presenta un número adecuado de entradas, salidas y periféricos para poder desarrollar las funcionas expuestas en la primera capa del protocolo ya que en

esta capa nos vamos a centrar en lo que es el software del PIC (parte fundamental del protocolo).

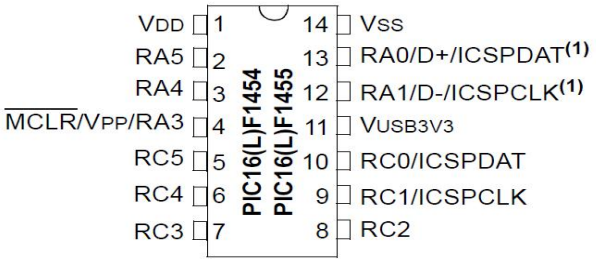


Imagen 2.7.2.1 – Pines PIC16F1455

Como se puede observar en la imagen 2.6.2.1 el PIC cuenta con 14 pines que se emplean en su totalidad para implementar todas las funciones de la placa. En la tabla 2.7.2.1 se recogen la relación de los pines con la función que desempeñan:

Numero	Pines	Función
1	Vdd	Alimentación +5V
2	RA5	PWM2
3	RA4	AD AN3
4	RA3	MCLR reset
5	RC5	PWM1
6	RC4	Led de señalización
7	RC3	DAC
8	RC2	AD AN6
9	RC1	AD AN5
10	RC0	AD AN4
11	VUSB3V3	Alimentación *transceiver*
12	RA1	Linea de datos D+
13	RA0	Linea de datos D-
14	Vss	Masa

Tabla 2.7.2.1 – Relación entradas/salidas - pines PIC

Cada una de las funciones asociadas a los pines está desempeñada por los periféricos que ofrece el microcontrolador, los cuales hay que configurar en función de las especificaciones que tenga la DAQ. Para ello los periféricos que se han empleado son los siguientes:

- Módulo Universal Serial Bus.
- 5-bit Convertidor digital analógico (DAC).
- 16-bit Timer (TMR1).
- 8-bit Timer (TMR2).
- 10 Channel 10-bit Convertidor analógico digital (ADC).
- 2 Módulos PWM.

Cada uno de estos periféricos debe ser configurado de forma independiente, y según las especificaciones de la DAQ. Aprovechando que se configuran de forma independiente se han programado de tal manera que todos los registros y funciones se encuentren dentro de un archivo independiente del fichero del programa principal, de tal forma que cada periférico tendrá asociado dos archivos: *header file*(extensión .h) y *source file* (extensión .c). En general el primero de ellos contendrá las definiciones de las constantes y los prototipos de las funciones; mientras que el segundo de ellos contendrá el algoritmo que deberá ejecutar como tal el periférico.

Al definir los prototipos de las funciones en *header files* podemos emplear estas funciones en diferentes *source files*, únicamente deberá declararse al comienzo de los archivos mediante la sentencia *include*. Ésta forma de trabajo es la que emplean las librerías para usuarios disponibles en la página web de Microchip, y es la forma en la que se han programado los periféricos que se emplean en el *firmware* del dispositivo, resultando como el ejemplo de la figura 2.7.2.2.

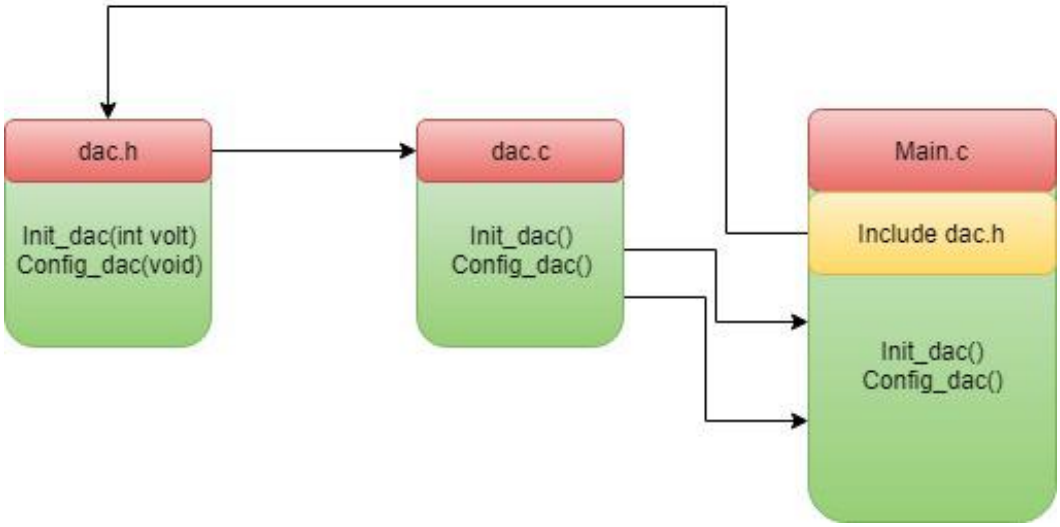


Imagen 2.7.2.2 – Diagrama ficheros

Este método permite agrupar el algoritmo perteneciente a periféricos del microcontrolador en funciones específicas dentro de un *source file* dedicado para él, por lo que evitamos sobrecargar el fichero del programa principal de funciones. Del mismo modo creamos un código altamente reutilizable y fácil de modificar, que puede llevarse de un programa a otro incluyéndolo en la biblioteca como si fuese una librería.

En los siguientes apartados se comentará como se ha configurado y programado cada uno de los periféricos que se necesitaron para la creación de la tarjeta de datos.

2.7.2.1 Bits de configuración.

No constituyen uno de los periféricos mencionados anteriormente, pero tienen una gran importancia ya que configuran al microcontrolador en cuanto a su comportamiento interno: fuentes de *reset*, tipo de oscilador, frecuencia del oscilador, protección del código, etc.

En la tabla 2.7.2.1.1, se recogen cada uno de los bits de configuración y su valor para, para más información consultar el *datasheet* del microcontrolador (ver Anexo V).

Bit	Significado	Configuración
FOSC	Selecciona tipo de oscilador	Oscilador Interno
WDTE	Watch Dog Timer, fuente de reset	Desahabilitado, para que el código se ejecute con todo el tiempo que necesite
PWRTE	Habilita Power-Up Timer, lo que permite un pequeño retardo cuando se sale de un reset	Habilitado, permite estabilizar tensión de alimentación
MCLRE	Master Clear, fuente de reset	Habilitado, en el correspondiente pin
CP	Código de protección	Desahabilitado
BOREN	Habilita Brown Out Reset	Habilitado
CLKOUTEN	Oscilador externo	Desahabilitado, se emplea el oscilador interno
IESO	Cambiar entre oscilador interno/externo	Desahabilitado, se emplea sólo el oscilador interno
FCMEN	Fail-Safe Clock Monitor, permite que el PIC siga trabajando si falla el oscilador externo	Desahabilitado
WRT	Flash Memory Self-Write Protection, proteje el PIC contra intentos de auto-escritura	Desahabilitado
CPUDIV	Divisor del reloj del sistema	Divisor 3
USBLSCLK	Frecuencia de reloj máxima que espera el módulo USB	48MHz
PLLMUT	Multiplicado de la frecuencia seleccionado para el oscilador	Multiplicador x3
PLLEN	Habilita multiplicador	Habilitado multiplicador x3
STVREN	Habilita reset por desbordamiento de la pila	Desahabilitado
BORV	Mantiene el dispositivo en Reset hasta que no alcanza el nivel de tensión requerido	A nivel alto

LPBOR	Resetea el dispositivo cuando se detecta una tensión muy baja en Vdd	Desahabilitado
LVP	Modo de bajo voltaje	Desahabilitado

Tabla 2.7.2.1.1 – Bits de configuración

2.7.2.2 Convertidor analógico digital (ADC).

Todas las entradas de la tarjeta pasarán a través del convertidor analógico digital, que se encuentra asociado a los pines: RA4, RC0, RC1, RC2. Cada uno de los pines está asociado aun canal del multiplexor del convertidor (ver imagen 2.7.2.2.1) , de la siguiente forma:

- RA4: AN3.
- RC0: AN4.
- RC1: AN5.
- RC2: AN6.

El ADC que incorpora el microcontrolador es un convertidor de aproximaciones sucesivas, con una resolución máxima de 10bits, la cual se emplea para nuestra aplicación.

El rango de tensiones de entrada de la tarjeta será de 0 a 5V, por lo que la tensión de referencia positiva será la misma que la de alimentación de la tarjeta Vdd. La referencia de tensión negativa será Vss.

Se empleará el oscilador interno, como reloj de convertidor, funcionando a una decimosexta parte de la frecuencia de éste. Por lo que el tiempo de conversión del convertidor será de: Tad= 1.0μs por cada canal (valor tabulado en el *datasheet* del microcontrolador).

El resultado se almacena en dos registros ADRESH y ADRESL. Se ha seleccionado justificado hacia a la derecha, por lo que los 8bits MSB están en el registro ADRESH mientras que 2LSB están en el registro ADRESL.

El *auto-conversion trigger* permite realizar mediciones periódicas sin que estén programadas por algoritmo (unicamente modificando el correspondiente registro). En este caso no aporta nada por lo que se deshabilitará.

No se empleará el modo *sleep* del PIC y se utilizará el bit GO/DONE como *flag* para saber cuando la conversión ha finalizado, por lo que se deshabilitará la interrupción del ADC.

Los registros asociados empleados para la configuración del ADC, según las premisas expuestas anteriormente, serían:

- CHS: del registro ADCON0, permite seleccionar el canal de entrada del multiplexor. En cada cambio de canal, el *datasheet* recomienda que se produzca un breve *delay*.
- GO/DONE: del registro ADCON0, actúa como *flag* para indicar que la conversión ha terminado. El microcontrolador lo *resetea* automáticamente una vez que el ADC ha terminado, aunque también se puede *resetea* por software.
- ADON: del registro ADCON0, habilita el funcionamiento del convertidor.
- ADFM: del registro ADCON1, modo de justificado.
- ADCS: del registro ADCON1, frecuencia del reloj de conversión.
- ADPREF: del registro ADCON1, voltaje de referencia positivo.
- TRIGSEL: del registro ADCON2, selecciona *auto-conversion Trigger*.

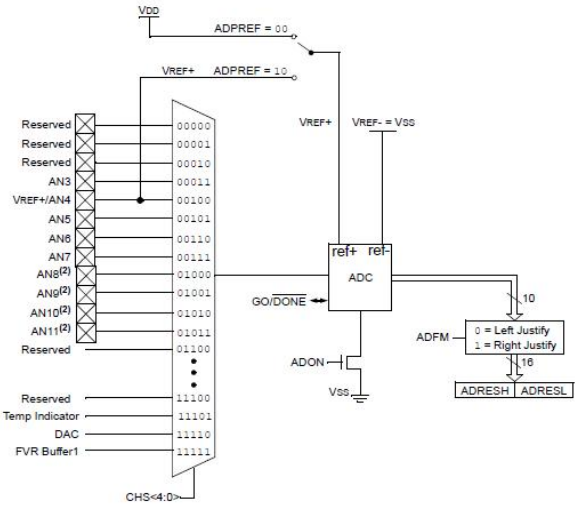


Imagen 2.7.2.2.1 – Esquema convertidor AD

El fabricante define una secuencia típica en la que se debe realizar la conversión, la cual se ha reflejado en el código del periférico:

1. Configuración del puerto
2. Configuración del módulo ADC.
3. Configuración de la interrupción ADC (en este caso esta deshabilitada por lo que se saltará este paso).
4. Espera que requiera el tiempo de adquisición.
5. Inicio de la conversión mediante GO/DONE.
6. Espera a que se termine la conversión reflejándose en el *flag* de GO/DONE.
7. Resultado de la conversión.

2.7.2.2.1 Algoritmo.

Consultar código en el Anexo III códigos de programación, ver flujograma (figura 2.7.2.2.1).

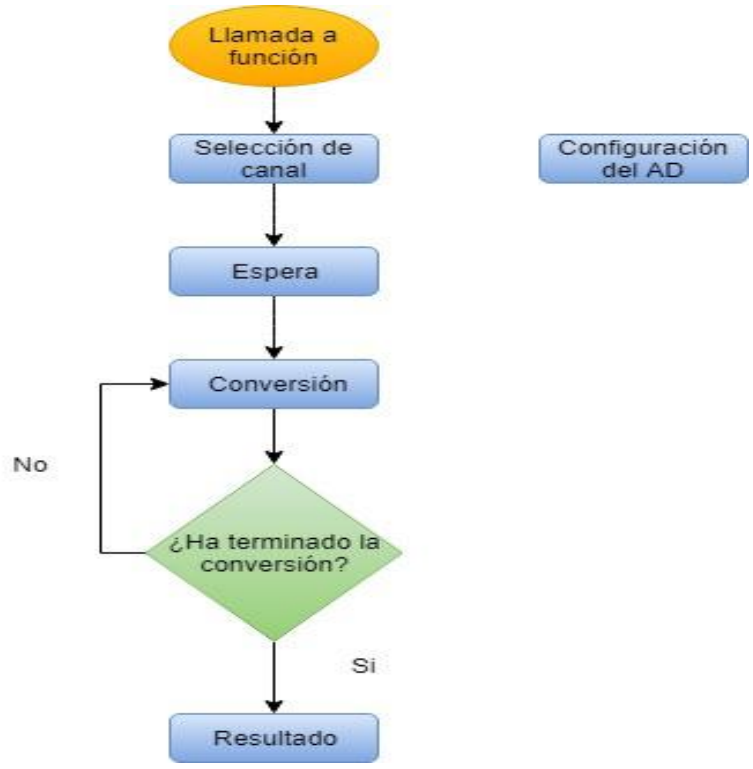


Figura 2.7.2.2.1.1 – Flujograma convertidor AD

Está dividido en dos funciones:

- adc_initialize( ): establece la configuración del convertidor AD y lo inicializa. Contiene los registros que permiten configurarlo:
  - ADPREF: establecemos referencia positiva (Vref+), conectada a la alimentación del tarjeta Vdd.
  - ADCS: fuente de reloj la decimosexta parte del oscilador interno.
  - ADFM: justificado hacia a la derecha.
  - ADCON2: se deshabiltia auto_trigger.
  - Se limpian los registros ADRESL y ADRESH.
  - Se deshabilita la interrupción por periférico del convertidor.
  - ADON: se activa el convertidor.
- adc_get_conversion (): es la función que lleva a cabo la conversión. Está pensada para que en el momento en el que se le pase el canal, realice la conversión del pin asociado a ese canal y devuelva el valor de la conversión en un número de 0 a 1023 (número de 10 bits). Para ello se emplean los siguientes registros
  - CHS: canal seleccionado, en este caso variará de 3 a 6.
  - GO_nDONE: se escribe a nivel alto para dar inicio a la conversión, y del mismo modo se emplea en un bucle *while* como *flag*, ya que cuando la conversión termina, se limpia automáticamente por software.
  - ADRESH y ADRESL: al primero de ellos se le aplica un desplazamiento y se le suma el segundo almacenándolo en la variable *adc_value* que será la que devuelva la función.

Se realiza un pequeño retardo a través de la instrucción ***__delay_us***, para darle tiempo al convertidor de que pueda cambiar canal y para iniciar una nueva conversión.

Finalmente se realiza una simulación del código para comprobar que este funciona correctamente. Para ello se emplea una pantalla lcd para comprobar que funciona correctamente, y un potenciómetro conectado a RC0 para simular una tensión variable a la entrada (ver imagen 2.7.2.2.2.2)

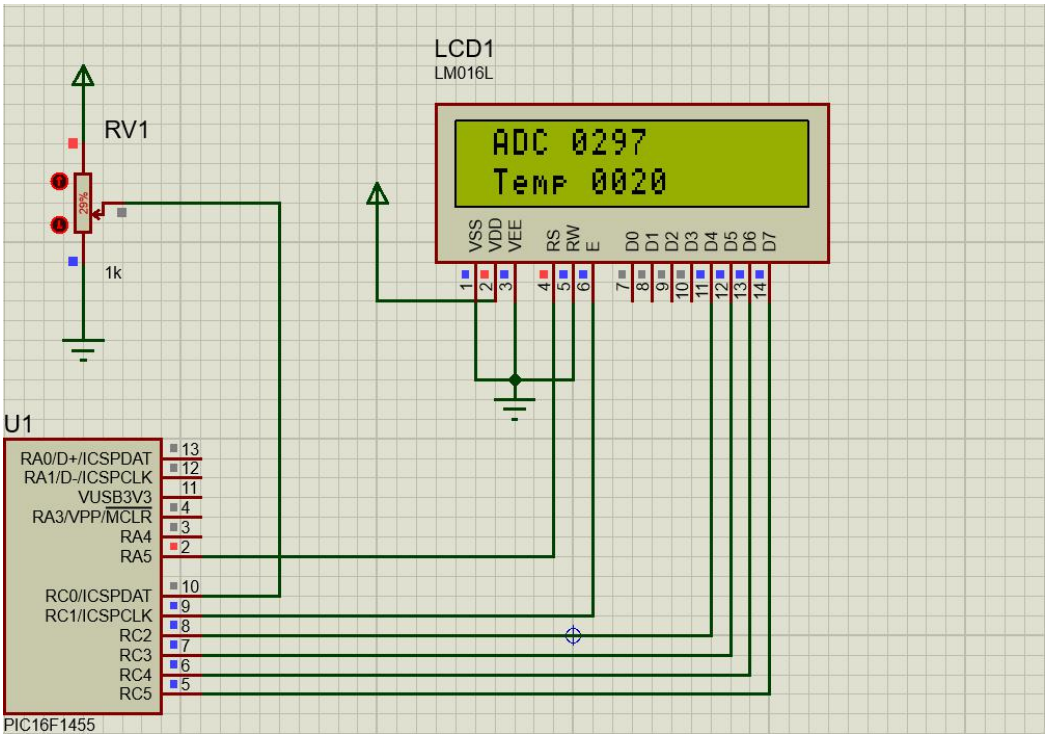


Imagen 2.7.2.2.2 – Simulación convertidor AD

Se puede observar que un potenciómetro al 29% el convertidor muestra un valor de 297 (sobre una escala de 0 a 1023).

2.7.2.3 Módulo PWM.

La PWM se generará en los puertos RC3 y RC5 (ver tabla 2.7.2.1), en ambos el procedimiento y el funcionamiento será exactamente el mismo por lo que se tomará como referencia la generación de la PWM1 asociado al puerto RC5.

El microcontrolador permite programar los tres parámetros fundamentales de la PWM:

- Período, o su inverso, la frecuencia.
- Ciclo de trabajo (*duty cicle*).
- Ancho de pulso (*pulse width*).

El período de la PWM depende de tres valores: los registros PR2 y *prescaler* del Timer 2 (ver sección Timer 2), y frecuencia del oscilador. Los tres se pueden expresar matemáticamente, de acuerdo a la siguiente ecuación:

$$periodo = [(PR2) + 1] * 4 * Tosc * (TMR2\,\mathrm{Pr}\,escaler) \quad \textbf{(2.7.2.3.1)}$$

La frecuencia de la PWM calculada como el inverso de la fórmula anterior, interesa que sea lo más alta posible para luego poder diseñar una etapa de filtrado lo más simple posible, colocada a la salida de los pin de la PWM, y que se pueda obtener 2 salidas analógicas. El registro PR2 y el Tosc están fijados a unos valores específicos, el primero a 255 (ver sección Timer2) y el segundo a la inversa de 16MHz (ver sección bits de configuración). Por lo que la única variable es el valor del *prescaler*, el cual debe ser lo más bajo posible por lo que se seleccionará un *prescaler* de 1. En definitiva, la frecuencia de la salida PWM será de 16KHz (ver Anexo II Cálculos).

El ciclo de trabajo depende del valor que se encuentre en los registros PWMxDCL y PWMxDCH y del valor de PR2. Anteriormente se ha mencionado en el cálculo del período de la PWM el valor de PR2 estaba fijado de antemano, y esto se debe a que se emplea para el cálculo del ciclo de trabajo. Si se observa la figura 1.6.2.3.1, la salida de los registros PWMxDCL y PWMxDCH van un módulo comparador donde se conecta la salida del Timer 2, de tal forma, que cuando se alcanza el valor fijado por estos registros la salida PWM del pin asociado se pone a 0 hasta que el Timer 2 completa la temporización, en ese momento modifica la salida a 1 generando de esta forma la señal cuadrada. El que la señal se mantenga a 0 o a 1 dependerá de el período fijado en PR2, estableciendo la ecuación 2.6.2.3.2:

$$Dutycicle = \frac{(PWMxDCH : PWMxDCL <7:6>)}{4(PR2 + 1)} \quad \textbf{(2.7.2.3.2)}$$

PWMxDCH y PWMxDCL son dos registros de 8 bits en los cuales se almacena el valor del ciclo de trabajo. El módulo PWM ofrece una resolución de 10 bits por que el valor máximo que se puede almacenar es 1024, para que se emplee la resolución al máximo el valor de PR2 debe ser 255 (ver anexo de cálculos). Debido a esto a nivel de código se emplean dos registros que permiten almacenar los 8*MSB* en el registro PWMxDCH y los 2*LSB* en el registro PWMxDCL.

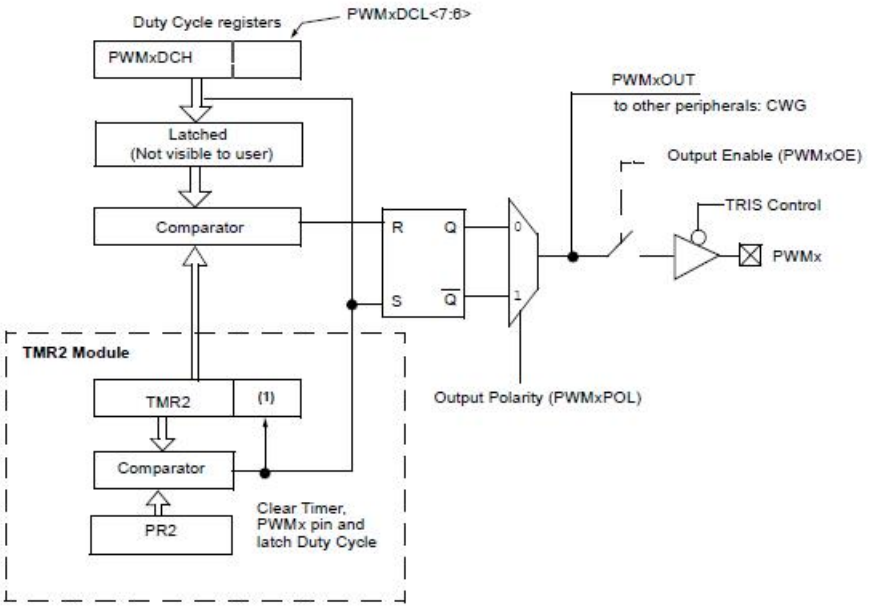


Imagen 2.7.2.3.1 – Esquema módulo PWM

El datasheet del microcontrolador define la siguiente secuencia de operaciones para generar correctamente la PWM, secuencia que se ha reflejado en el algoritmo (figura 2.7.2.3.2):

1. Desactivar los pines de salida, a través del registro TRISbits.
2. Limpiar el registro PWMxCON.
3. Cargar el registro PR2 con el período de la PWM.
4. Limpiar los registros PWMxDDCH y PWMxDCL.
5. Configurar e iniciar el Timer 2.
6. Habilitar el pin de salida PWM, mediante los registros PWMxOE y TRISbits.
7. Configurar el módulo PWM mediante el registro PWMxCON.

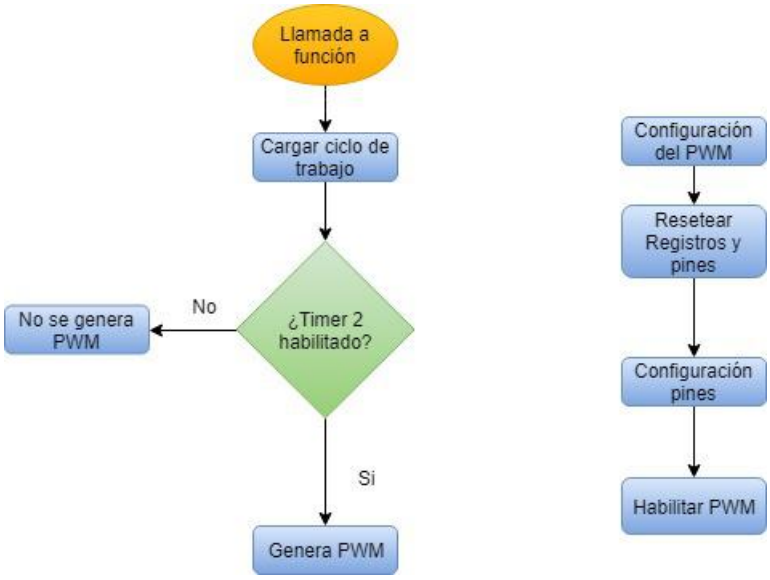


Figura 2.7.2.3.2 – Flujograma PWM

Hay una particularidad en el caso de la PWM2 asociado al puerto RA5. La PWM2 se encuentra "duplicada" entre los puertos RA5 y RC3, si no se selecciona lo contrario por defecto el PIC la ubica en el puerto RA5, que para la aplicación en desarrollo se encuentra ocupado por el AD, por lo que se debe modificar el valor del bit P2SEL del registro APFCONbits, para que la PWM2 se genera a través del puerto RC3.

El registro PWMxCON cuando se vaya activar la PWM se cargará con el valor 0xC0 lo que significa que el modulo está habilitado, se ha habilitado el pin de salida y se ha seleccionado la salida de la PWM a nivel bajo.

Para asegurar el funcionamiento de la PWM antes de probarlo directamente en el microcontrolador y debido a que en no siempre se encontraba disponible un osciloscopio, se empleó el programa de simulación Proteus, que, pese a sus limitaciones, permite comprobar exactamente si está generando correctamente la señal cuadrada:

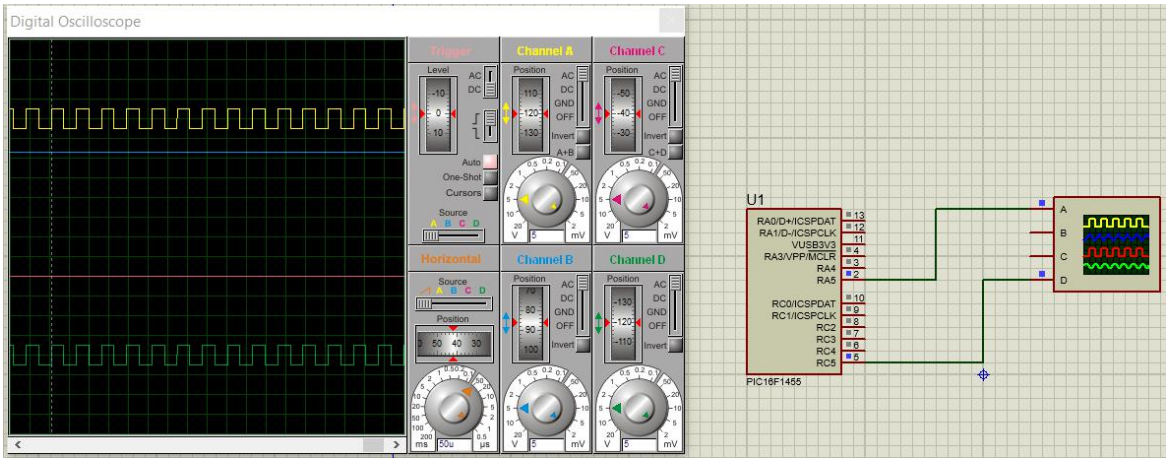


Imagen 2.7.2.3.2– Simulación PWM

2.7.2.4 Timer 2.

Como se ha explicado en el apartado 2.6.2.3, el empleo del Timer 2 está supeditado a su uso como temporizador para la generación de la PWM, por lo que su configuración depende prácticamente del período y el ciclo de trabajo que se quiera obtener a la salida de los pines RA5 y RC5.

El temporizador cuenta con los siguientes registros asociados (ver figura 2.7.2.4.1):

- PR2: registro de 8 bits, que se puede actualizar en cualquier momento, y que establece el período del temporizador. El TMR2 se incrementa desde el valor 00h en cada ciclo de reloj, teniendo en cuenta que la frecuencia de su reloj es la Fosc/4.
- TMR2: registro que almacena la cuenta del temporizador. Se incrementa desde el valor 00h en cada ciclo de reloj, hasta que alcanza el valor establecido en el registro PR2, momento en el que se restablece a cero de nuevo y vuelve a iniciar la cuenta el temporizador. En ausencia de *prescaler* la frecuencia de reloj es Fosc/4.
- TMR2IF: *bit flag* de interrupción que indicia cuando se ha producido un desbordamiento del timer.
- TMR2IE: habilita interrupciones globales.
- *Prescaler* y *postscaler*: el primero de ellos ya se ha establecido en la sección anterior (apartado 2.7.2.3), que debe ser lo más bajo posible, ya que divide la

señal de entrada de entrada del temporizador mientras que el segundo no tiene ningún efecto a la hora de generar la señal PWM por lo que se mantendrá a cero.

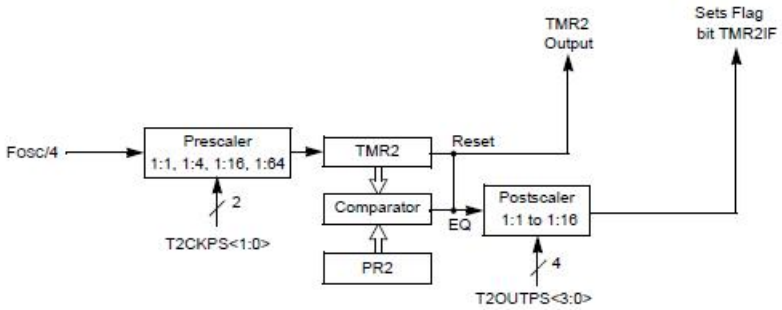


Imagen 2.7.2.4.1 – Esquema Timer 2

Al igual que en el caso del módulo PWM el datasheet recomienda una secuencia de operaciones para el Timer 2:

- Limpiar el registro TMR2.
- Configurar el timer con el registro T2CON.
- Iniciar el Timer 2.

2.7.2.4.1 Algoritmo.

Este apartado se completa con los códigos situados en el anexo III de códigos de programación, referente a la sección del código del Timer 2 y el flujograma (ver figura 2.7.2.4.1.1).

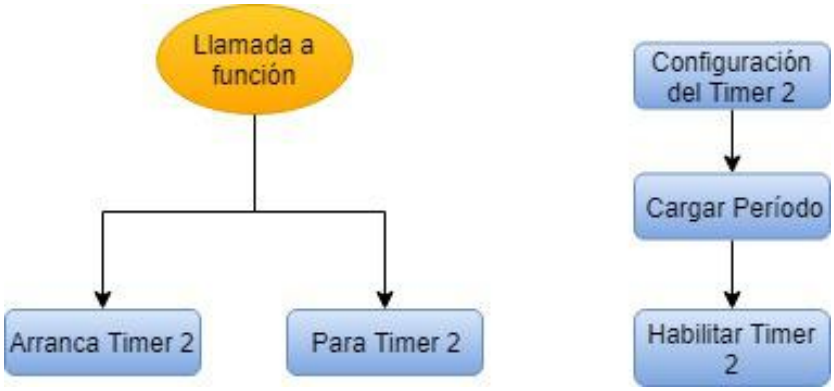


Figura 2.7.2.4.1.1 – Flujograma Timer 2

El algoritmo está compuesto de tres funciones:

- **timer2_initialize()**: se trata de una función de inicialización que contiene los registros que permiten configurar el timer 2 e iniciar la temporización. Registros asociados:
  - PR2: se cargará con el 0xFF (255 en decimal).
  - TMR2: se resetea el registro antes de iniciar la temporización.
  - TMR2IF: se resetea antes de iniciar la temporización.
  - TMR2IE: se habilita la interrupción global del Timer 2, escribiéndolo a 0.
  - T2CON: se configurará con el valor 0x04, de tal manera que se deshabilita, tanto el *prescaler* (ver sección PWM), como el *postscaler* y se inicia el Timer 2.
- **timer2_start()**: inicia la temporización escribiendo el bit del TMR2ON a nivel alto.
- **timer2_stop()**: detiene la temporización escribiendo el bit del TMR2ON a nivel bajo.

2.7.2.5 Timer 1.

La única función de este *timer* es la de provocar un ligero retardo para evitar que el bus USB se sature. Se ha configurado de manera similar al *timer 2.*

Si se modifican los *endpoints* a la frecuencia del oscilador interno del PIC, se ha comprobado experimentalmente que el bus tiende a saturarse cuando se intenta enviar datos al PIC a la vez que este está enviando datos al PC. Esto puede ser debido a los tiempos de acceso al bus USB que son del orden de milisegundos (como ya se ha comentado en la sección de antecedentes). Por ello se emplea este *timer* para corregir este problema.

El *timer 1* emplea el oscilador interno con un *preescaler* de un cuarto. Lo que establece una temporización de aproximadamente 0.25μs. Además se han configurado los registros de TMR1H y TMR1L, para aumentar la temporización, por si se quisiera modificar en un futuro la velocidad del USB, poder modificar este retardo. Para evitar errores se ha fijado el valor de la temporización en 25μs.

2.7.2.5.1 Algoritmo.

Esta sección se complementa con el anexo III de códigos de programación. El código creado se sintetiza en el flujograma de la figura 2.7.2.5.1.1.

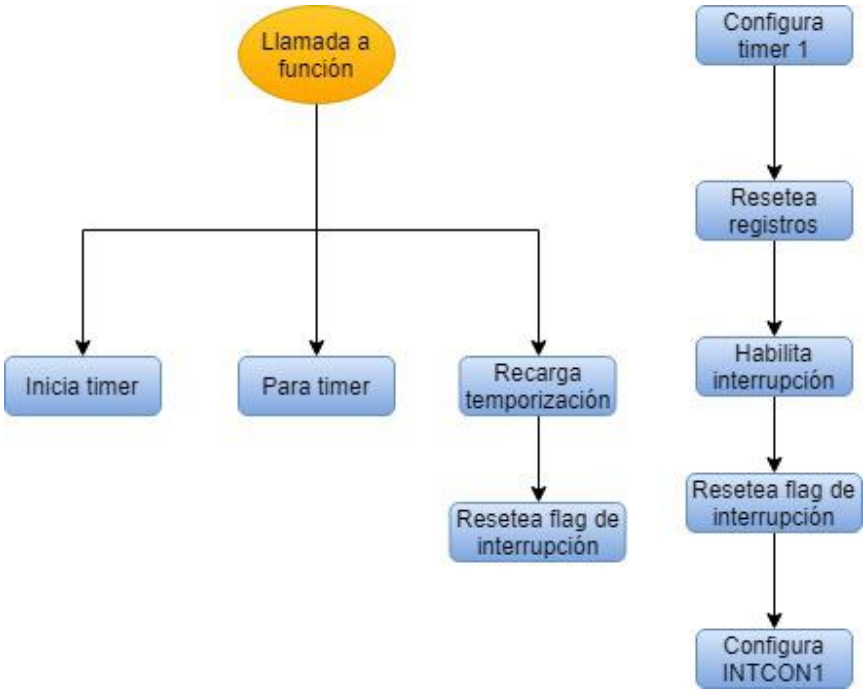


Figura 2.7.2.5.1.1 – Flujograma Timer 1

El algoritmo está compuesto por cuatro funciones:

- **Timer1_initialize()**: Configura el timer 1 como temportizador. Contiene los siguientes registros:
  - TMR1H y TMR1L: los resetea antes de iniciar la temporización.
  - INTCON: se habilitan las interrupciones globales.
  - TMRIF: se resetea el *flag* de interrupción.
  - TMRIE: se habilita interrupción en el *timer 1*.
- **Timer1_start()**: arranca el *timer 1*, bit TMR1ON.
- **Timer1_stop()**: para el *timer 1*, bit TMR1ON.
- **Timer1_reload()**: establece el número de cuentas que debe hacer el timer 1, es decir el valor de la temporización. El valor se pasa como un argumento y se carga en los registros TMR1H y TMR1L mediante desplazamientos de bits. Esta función esta pensada para funcionar dentro de la rutina de interrupción por lo que limpiará el *flag* de interrupción al acabar de cargar los registros.

2.7.2.6 Convertidor digital analógico (DAC).

Permite generar una salida analógica a través del puerto RC2. Produce un voltaje de referencia con 32 niveles seleccionables.

La tensión de salida de la DAC es proporcional a la tensión de alimentación del PIC por lo que el rango de tensiones de salida será de 0 a 5V (se puede modificar seleccionando una fuente de alimentación externa, de la que no se dispone para esta aplicación).

El convertidor digital analógico del que dispone este PIC es una escalera de resistencias, con cada resistencia conectada a una referencia de voltaje positiva (fuente de alimentación) y una negativa. Se configura mediante los siguientes registros asociados (ver imagen 2.7.2.6.1):

- DACR: registro de cinco bits establece 32 niveles de voltaje de salida, que permiten modificar el voltaje de referencia.
- DACEN: habilitación de la salida DAC.
- DACOE2: la salida de voltaje se encuentra en el pin DACOUT2.
- DACPSS: selecciona el fuente de alimentación que emplea el módulo DAC.

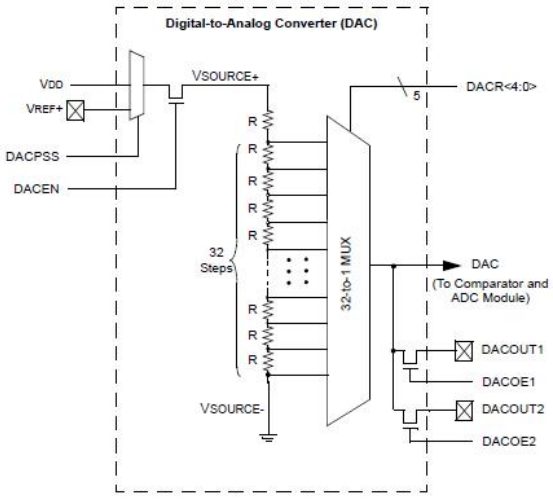


Imagen 2.7.2.6.1 – Esquema DAC

A nivel de montaje *hardware* el convertidor digital analógico entrega muy poca corriente a su salida por lo que el *datasheet* recomienda el empleo de un seguidor de tensión para aumentar el consumo de corriente entregada.

2.7.2.6.1 Algoritmo.

Consultar código en el Anexo III códigos de programación.

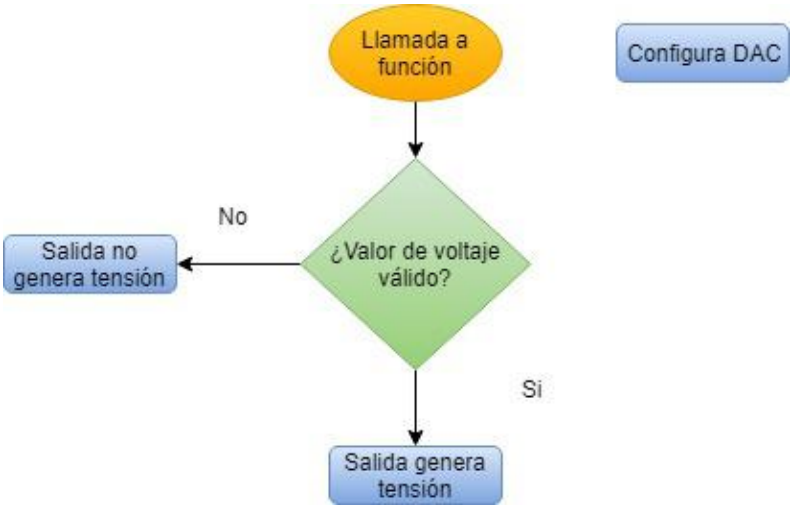


Imagen 2.7.2.6.1.1 – Flujograma DAC

El algoritmo está formado por dos funciones:

- **config_dac()**: contiene el registro DACCON0 con el valor 0x90 lo que habilita el funcionamiento del módulo DAC, y se habilita la salida de DACOE2 a través del pin RC3. Además se establece la fuente de referencia del registro DACPSS como VDD.
- **init_dac()**: se escribe el porcentaje de voltaje que se desea tener en la salida por valor, el cual se almacena en la variable ***volt***. Se realiza una detección de errores para que el valor no sea inferior a 0 o mayor que 32 y se carga en el registro DACCON1, para obtener la tensión de salida deseada.

2.7.2.7 Módulo USB.

Para que el PIC y cualquier dispositivo puedan soportar el protocolo USB deben tener un módulo también conocido *USB Controller ,* que en el caso del PIC se resume en el diagrama (imagen 2.7.2.7.1).

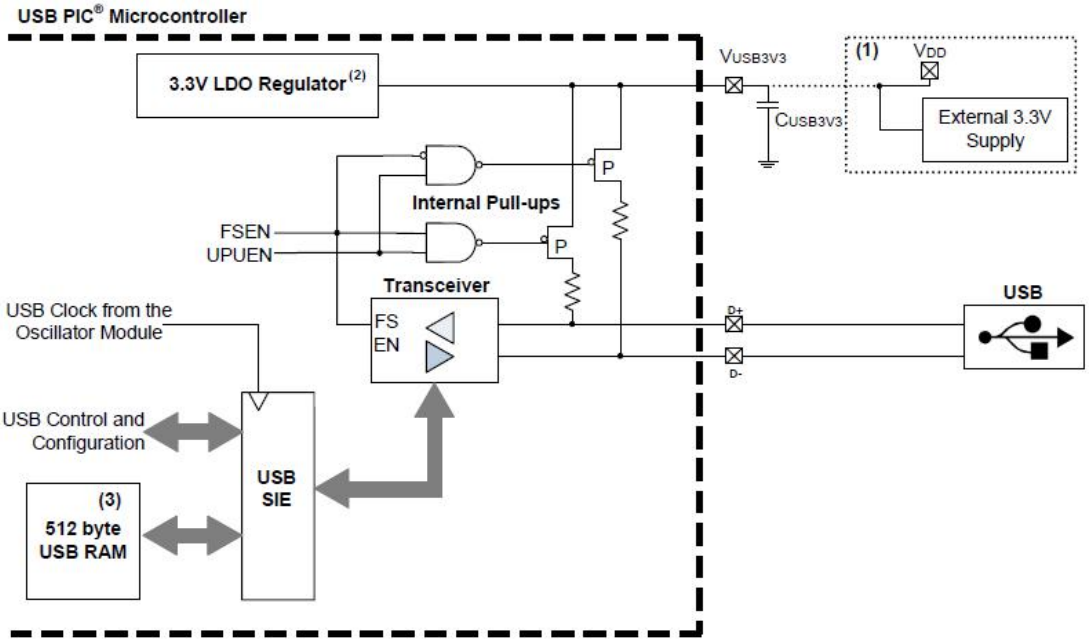


Imagen 2.7.2.7.1 – Esquema módulo USB

A parte del puerto y las líneas de datos (D+ y D-), presenta cuatro elementos principales:

- *Transeceiver*: funciona como una *interface* entre el conector USB y el SIE. Necesita de alimentación externa que la consigue a través del pin 11 (ver sección placa). Soporta hasta la especificación 2.0 USB.
- *USB SIE*: se encarga de controlar el envío y la recepción de datos, nunca de su interpretación. Soporta los modos de velocidad de *low-speed* y *full-speed*. Así como los cuatro tipos de transferencia. También cumple otras funciones.
  - Detección de paquetes de datos de entrada.
  - Envío de paquetes.
  - Detectar y generar paquetes de datos específico: *Start-of-Packet*, *End-of-Packet*, *Reset* y *Resume*.

  - Codificar y decodificar los datos en el formato que requiere el bus: NRZI con *bit de stuffing*.
  - Comprueba y genera los *PacketIDs*.
- *USB RAM*: permite el intercambio de datos entre el procesador del PIC y el SIE. Se trata de una memoria *RAM dual-port*, que se encuentra dentro de la memoria de datos del microcontrolador. Es donde se encuentra el *endpoint* de control.
- Resistencias internas de *pull-up*: están incorporadas para poder soportar dos modos velocidad: *full-speed* y *low-speed*. Permiten garantizar que nunca haya corriente en la línea *Vbus* del cable USB.

La configuración de este módulo recae sobre tres registros de configuración. Aunque también se emplean un total de 14 registros auxiliares para controlar las transacciones USB.

- UCON: *USB Control Register*.
- UCFG: *USB Configuration Register*.
- USTAT: *USB Transfer Status Register*.
- UADDR: *USB Device Address Register*.
- UFRM: *Frame Numbrer Register*.
- UEPn: *Endpoint Enable Register*.
- Registros *BDN*.
- UIR: *USB Interrupt Status*.
- UIE: *USB Interrupt Enable Register*.
- UIER: *Error Interrupt Status Register*.

La programación y el control de estos registros se encuentran en el archivo *device.c*, creado por Microchips.

Este módulo es el responsable de la alimentación de la placa. En secciones anteriores se ha comentado que la placa se alimentaría con una tensión de continua de 5V. Esto es debido a que de entre los tres modos de alimentación que establece la especificación y que soporta el PIC, se ha optado por el *Bus Power only* (ver imagen 2.6.2.6.2).

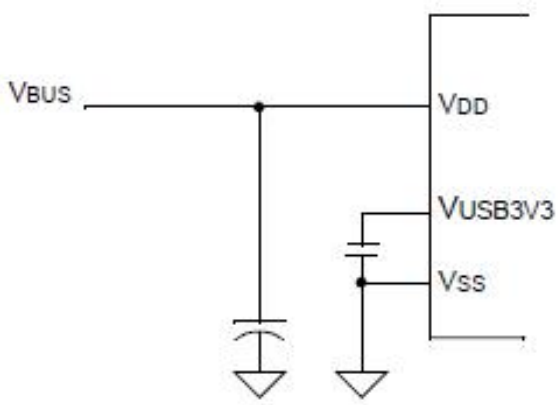


Imagen 2.7.2.7.2 – Esquema Bus Power only

A parte de la alimentación también influye en otros factores como la frecuencia de reloj del PIC. La frecuencia del PIC debe ser de 48MHz, pero el bus USB obliga a ajustar está frecuencia a 16MHz para que se puede producir la comunicación USB.

A nivel de algoritmo, todo el código de esta sección es el grueso del *firmware* del dispositivo conforma la siguiente capa.

2.7.3 Firmware II: comunicación USB

Tercera capa del protocolo, engloba todas las funciones de la capa anterior e incorpora los códigos relacionados con el control del módulo y la comunicación USB, creando un programa único que le permite al PIC procesar todas las entradas y salidas de la tarjeta y responder a las demandas del PC, es decir el software que controla el *hardware*, lo que se conoce como *firmware* del dispositivo.

Para poder crear el *firmware* del dispositivo primero se debe definir la subclase de dispositivo HID que se va a crea. En este caso no interesa asignarlo a las subclases de teclado o ratón sino que se debe asignar a una subclase genérica, debido a que éstas emplean protocolos que tienen sus efectos obvios en *Windows*. Por lo que se va a emplear lo que se denomina una subclase genérica también conocida como *Generic HID*.

Una vez definida la clase del dispositivo, la subclase y partiendo de todos los códigos que configuran el PIC, se procede a realizar el algoritmo propio de la comunicación USB. Debido a la alta complejidad del protocolo USB y motivado por su algoritmo se va utilizar como base el código *Generic HID* que provee Microchip para sus microcontroladores, el cual se encuentra en un archivo denominado MLA.

La MLA contiene todas las librerías y códigos de aplicaciones que desarrolla Microchip, y por supuesto todo lo relacionado con la comunicación USB. Por lo que una vez descargada hay que buscar la carpeta Generic HID Firmware, la cual se trata de un ejemplo de firmware preparada para configurar el microcontrolador como un dispositivo HID al que se le envía un dato desde el ordenador y este devuelve dicho dato por USB. La ruta donde se encuentra será similar a la siguiente: "*C:\Microchip Solutions v2010-10-19\USB Device - HID - Custom Demos\Generic HID - Firmware"* (ver imagen 2.6.3.1).

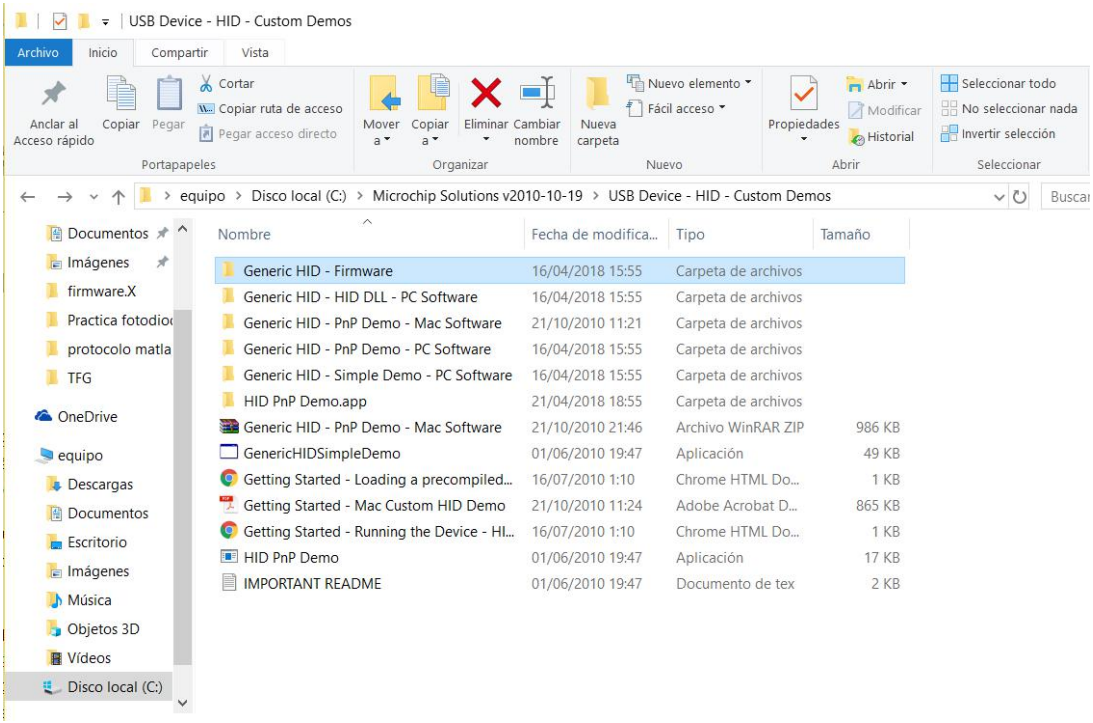


Imagen 2.7.3.1 - MLA

Este código constituye el punto de partida para la realización del firmware de la tarjeta, que se va a dividir principalmente en cuatro programas con sus

correspondientes librerías (incluidos los programas de los periféricos vistos en la sección del microcontrolador):

- Main.c.
- Usb_descriptor.c.
- Usb_function_hid.c.
- Usb_device.c.

Cada uno de estos programas cumple una función ya sea controlar la secuencia de ejecución del programa (main.c), información de configuración del dispositivo USB (usb_descriptor.c), códigos específicos de la clase HID (usb_function_hid.c) o la configuración del módulo USB y control de las transferencias (usb_device.c). No son programas independientes sino que están relacionados entres sí (ver diagrama 1.6.3.1). En las siguientes secciones se irán explicando en detalle cada uno de ellos.

2.7.3.1 Librerías.

Todas las librerías que se han empleado en este proyecto se pueden encontrar en la carpeta MLA, detallada anteriormente. En general las librerías contienen todas las definiciones de las constantes, variables y prototipos de las funciones (al fin y acabo se tratan de *header files*).

Muchas de ellas contienen llamadas entre sí, por lo que se han agrupado todas en la carpeta del proyecto y se han modificado las direcciones de las instrucciones *include* para evitar errores (ver imagen 2.7.3.1.1).

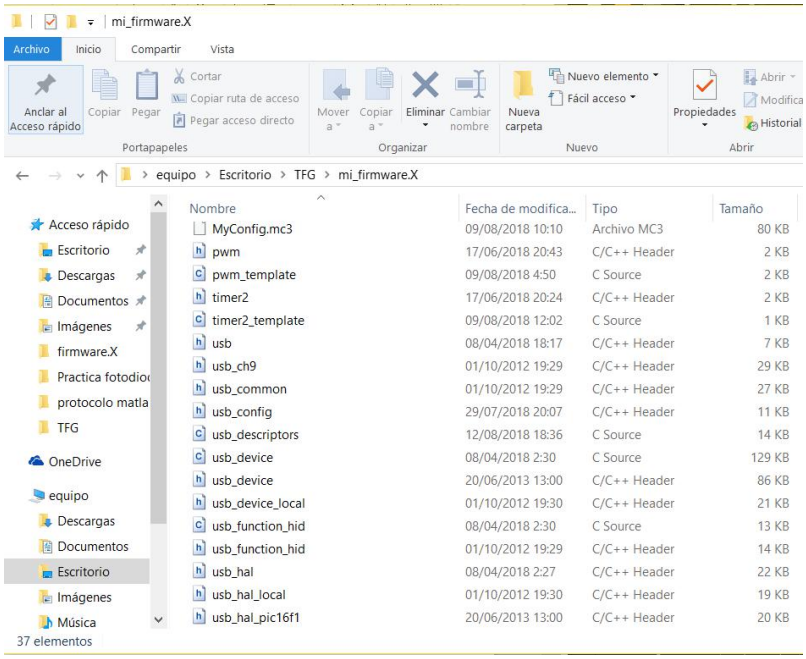


Imagen 2.7.3.1.1 – Librería en la carpeta del proyecto

Las librerías que se tienen que incluir son las siguientes:

- **HardwareProfile.h**: identifica la familia del PIC que se está empleando y en función de ello asigna un perfil de *hardware* u otro. En esta librería hay que hacer una pequeña modificación para que pueda identificar el PIC16F1455, por defecto viene preparado para el PIC16F1459.
- **HardwareProfile-PICDEMFUSB.h**: es el *hardware* asignado al PIC16F1459, pero que se puede emplear para nuestro microcontrolador. Su función es proporcionar al *firmware* información sobre algunas configuraciones.
- **Usb.h**: define y contiene los *include* de todas las librerías necesarias que permiten la comunicación USB.
- **Usb_ch9.h**: implementa las estructuras de datos, constantes y macros descritos en el capítulo nueve de la especificación USB 2.0.
- **Usb_common.h**: define los tipos de datos y constantes comunes que emplea *Firmware Stack* que ha desarrollado Microchip para el USB.
- **Usb_device.h**: esta librería contiene todas las constantes y macros del archivo usb_device.c (ver sección 1.6.3.3).
- **Usb_device_local.h**: contiene los macros de los *callbacks* (sección main.c) y configura los *endpoints* en función de si tienen un registro *ping-pong*.

- **Usb_function_hid.h**: contiene los macros y definiciones específicas para la clase HID.
- **Usb_hal.h**: identifica el tipo de microcontrolador de que se trata y en función de la familia le asigna una librería *usb_hal_pic* u otra. Es útil cuando se está programando con diferentes familias de microcontroladores y en la configuración del proyecto está incluido cada tipo de librería *hal_pic*. En este proyecto para ahorrar memoria del PIC sólo hay una librería de tipo *hal_pic* incluida que se explica en el siguiente punto.
- **Usb_hal_pic16f1.h**: existen varios tipos para esta librería en función de la familia del microcontrolador; en este proyecto el PIC empleado es de la familia 16F por lo que se debe buscar esta librería dentro de la carpeta MLA.

Por lo tanto las librerías se verían en el proyecto como en la imagen 2.6.3.1.2

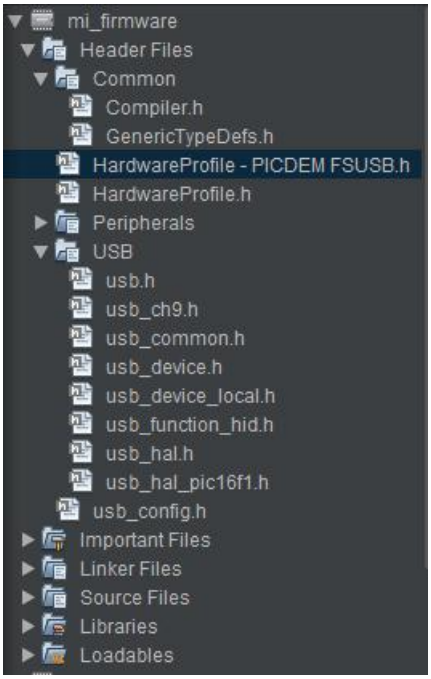


Imagen 2.6.3.1.2 – Librería en el proyecto

2.7.3.2 Usb_config.h.

Este archivo no es una librería, por ello se recoge en un apartado distinto. Se trata de un archivo de configuración generado por el desarrollador y cuya función es configurar la comunicación USB.

Para esta aplicación se va a utilizar el archivo disponible en la librería, el cual contiene todas las posibles configuraciones. Aunque también se puede emplear una aplicación de escritorio, denominada Microchip USB OTG Configuration Tool, que de manera más visual, ayuda a los desarrolladores a elegir la elección adecuada (ver imagen 2.7.3.2.1).

Algunas de las configuraciones que recoge, mediante constantes son las que se cargan en el archivo descriptor.c, además de las macros para poder enviarlas al PC.

Algunas características que se configuran desde este archivo en la tabla 2.7.3.2.1

Parámetro	Valor
Tamaño Endpoint 0	8 bytes
Numero de interface	1
Numero máximo de endpoints	1
Registro ping-pong	Full-ping-pong
Resistencias USB de pull-up	activas
Velocidad USB	Full-speed
Número de string descriptors	3
Activa todos los handlers	habilitado
Constante fabricante MY_VID	0x04D8
Constante producto MY_PID	0x0007
HID_Endpoint	1
Tamaño report	47 bytes
Tamaño input report	4 bytes
Tamaño output report	5 bytes
Tamaño feature report	2 bytes

Tabla 2.7.3.2.1 – Archivo de configuración

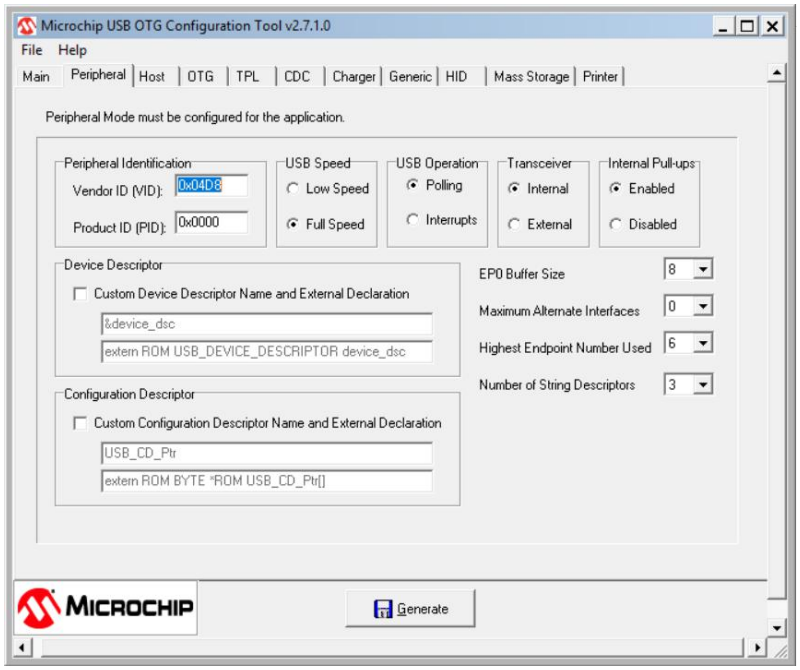


Imagen 2.7.3.2.1 – Microchip USB OTG Configuration Tool

2.7.3.3 Usb_device.c.

Si se estableciera una analogía entre el *firmware* y un coche, este archivo sería el motor del coche. Todas las constantes y macros que contiene se indican en la librería **usb_device.h.**

Contiene dos funciones que son críticas para el *firmware*:

- **USBDeviceInit()**: inicializa el dispositivo *reseteando* todas las variables, registros, *flags*, etc.
- **USBDeviceTask()**: es la función principal, que se debe ejecutar periódicamente al estar definidos los endpoints como registros *ping-pong*. Aunque como el tipo de transferencia definido es *Interrupt* basta con que se ejecute con cada interrupción. Entre sus funciones destaca:
  - Detección de los eventos USB.
  - Controla las *Control transfer*.
  - Proceso de enumeración.

2.7.3.4 Usb_function_hid.c.

Contiene las constantes y funciones específicas de la clase HID para comunicación USB, y es donde a nivel de algoritmo se cargan los descriptores para poder enviarlos. Las notas de aplicación de Microchip recomiendan no modificar este archivo como el anterior.

2.7.3.5 Usb_descriptores.c.

Contiene todos los descriptores para configurar la tarjeta como dispositivo HID, tal como se ha expuesto en la sección de antecedentes 2.3.2.7.

A parte de los descriptores obligatorios que debe tener todo dispositivo USB (ver sección antecedentes) y los descriptores obligatorios de la clase HID, se han incluido tres descriptores de cadena que contienen el nombre del producto y el fabricante con el fin de ayudar a identificar el periférico cuando se conecte al PC, resultando el diagrama de la imagen 2.7.3.5.

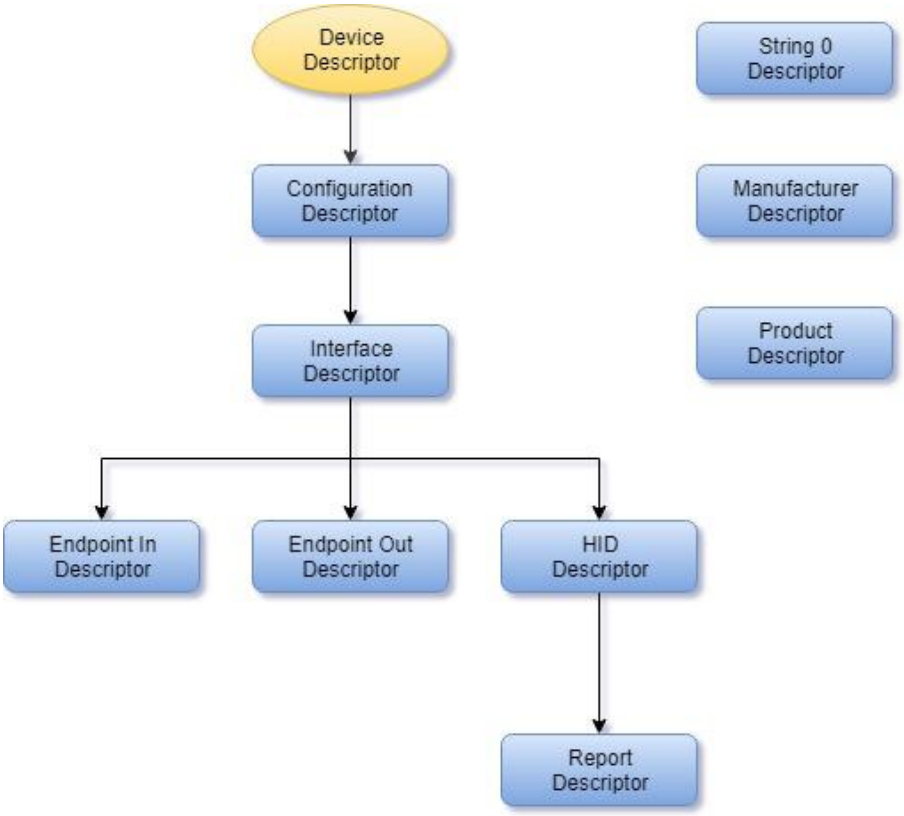


Imagen 2.7.3.5.1 – Diagrama Descriptores

La *interface* que se ha configurado es la de un dispositivo HID, como ya se ha expuesto anteriormente, formado por dos *endpoints* y un *HID descriptor*. Se ha optado por una *interface* donde todas las lecturas de la tarjeta se direccionen al *endpoint In* mientras que los parámetros que le envía el ordenador irán direccionados al *endpoint Out*.

A continuación se detallan los valores que con los que se ha configurado cada descriptor, es decir como está configurado el dispositivo:

- Device Descriptor: está programado como una estructuras de datos. Cada campo contiene su correspondiente configuración según la tabla 2.6.3.5.1 :

Posición	Campo	Configuración	Valor
0	bLengh	-	0x12
1	bDescriptorType	USB_device_descriptor	0x01
2	bcdUSB	Especificación 2.0	0x0200
4	bDeviceClass	Clase definida en la interface	0x00
5	bDeviceSubClass	Subclase definida en la interface	0x00
6	bDeviceProtocol	Protocolo definida en la interface	0x00
7	bMaxPacketSize0	Tamaño mínimo para ahorrar memoria	8 bytes
8	IdVendor	Vendor ID (se emplea el que proporciona microchip)	0x04D8
10	IdProduct	Product ID (definido por mi)	0x0007
12	bcdDevice	Se le asigna un número aleatorio	0x0001
14	iManufacturer	Se declara string	0x01
15	iProduct	Se declara string	0x02
16	iSerialNumber	No declarado	0x00
17	bNumConfiguration	Configuración única	0x01

Tabla 2.7.3.5.1 – Device Descriptor

- Configuration Descriptor:

Posición	Campo	Configuración	Valor
0	bLengh	-	0x09
1	bDescriptorType	USB descriptor configuration	0x02
2	wTotalLength	Tamaño total de los descriptores subordinados para esta aplicación	0x29
4	bNumInterfaces	Solo se ha programado el dispositivo con una interface	1
5	bConfigurationValue	Configuración única	1
6	IConfiguration	No se declara string	0
7	bmAttributes	Alimentación mediante bus_powered y remote wake up soportado	0b1010000
8	bMaxPower	Corriente máxima 500mA.	250

Tabla 2.7.3.5.2 – Configuration Descriptor

- Interface Descriptor:

Posición	Campo	Configuración	Valor
0	bLength	-	0x09
1	bDescriptorType	USB_Decriptor_interface	0x04
2	bInterfaceNumber	Interface número 0	0
3	bAlternateSetting	No hay interface alternativas	0
4	bNumEndpoints	Dos endpoints	2
5	bInterfaceClass	Clase HID	0x03
6	bInterfaceSubclass	No se declara subclase, dipositivo genérico	0
7	bInterfaceProtocol	No se declara protocolo, dispositivo genérico	0
8	iInterface	No se declara sting	0

Tabla 2.7.3.5.3 – Interface descriptor

- Endpoint Descriptor In:

Posición	Campo	Configuración	Valor
0	bLength	-	0x07
1	bDescriptorType	USB_descriptor_endpoint	0x05
2	bEndpointAddress	Endpoint HID, Endpoint IN	0x01,0x80
3	bmAttributes	Soporta Interrupt transfer	0x03
4	wMaxPacketSize	-	0x40
5	bInterval		0x01

Tabla 2.7.3.5.4 – Endpoint Descriptor In

- Endpoint Descriptor Out:

Posición	Campo	Configuración	Valor
0	bLength	-	0x07
1	bDescriptorType	USB_descriptor_endpoint	0x05
2	bEndpointAddress	Endpoint HID, Endpoint Out	0x01,0x00
3	bmAttributes	Soporta Interrupt transfer	0x03
4	wMaxPacketSize	-	0x40
5	bInterval		0x01

Tabla 2.7.3.5.5 – Endpoint Descriptor Out

- String 0 Descriptor:

Posición	Campo	Configuración	Valor
0	bLength	-	1 Byte
1	bDescriptorType	USB_Descriptor_String	03h
2	bString o wLangid	Código de inglés	0x0409

Tabla 2.7.3.5.6 – String 0 Descriptor

- Manufacturer Descriptor:

Posición	Campo	Configuración	Valor
0	bLength	-	1 Byte
1	bDescriptorType	USB_Descriptor_String	03h
2	bString o wLangid	Código de inglés	0x0409

Tabla 2.7.3.5.7 – Manufacturer Descriptor

- Product String Descriptor 0:

Posición	Campo	Configuración	Valor
0	bLength	-	1 Byte
1	bDescriptorType	USB_Descriptor_String	03h
2	bString o wLangid	Código de inglés	0x0409

Tabla 2.7.3.5.8 – Product string Descriptor

- HID Descriptor:

Posición	Campo	Mi configuración	Valor
0	bLength	-	0x09
1	bDescriptorType	HID_Descriptor	0x21
2	bcdHID	La especifcación que se está empleando es la 1.11	0x11,0x01
4	bCountryCode	El código del país se ha fijado en España: 25 en decimal	0x19
5	bNumDescriptors	Se ha optado por prescindir del physical descriptor y solo se un report descriptor por lo que es igual a 1	1
6	bDescriptorType	Solo se ha declarado un Report descriptor subordinado	0x22
7	wDescriptorLenght	El tamaño de Report descriptor es de 47 bytes	47
9	[bDescriptorType]	No declarado	0
10	[wDescriptorLength]	No declarado	0

Tabla 2.7.3.5.9 – HID Descriptor

- Report Descriptor:

Principalmente se han programado dos tipos de *report* para cada uno de los *endpoints* que soportan transferencia de tipo *Interrupt*: *Input report* y *Output report*. Cada uno de ellos presenta una estructura similar compuesta por los *items* que se puede observar en el diagrama de la imagen 2.7.3.5.1.

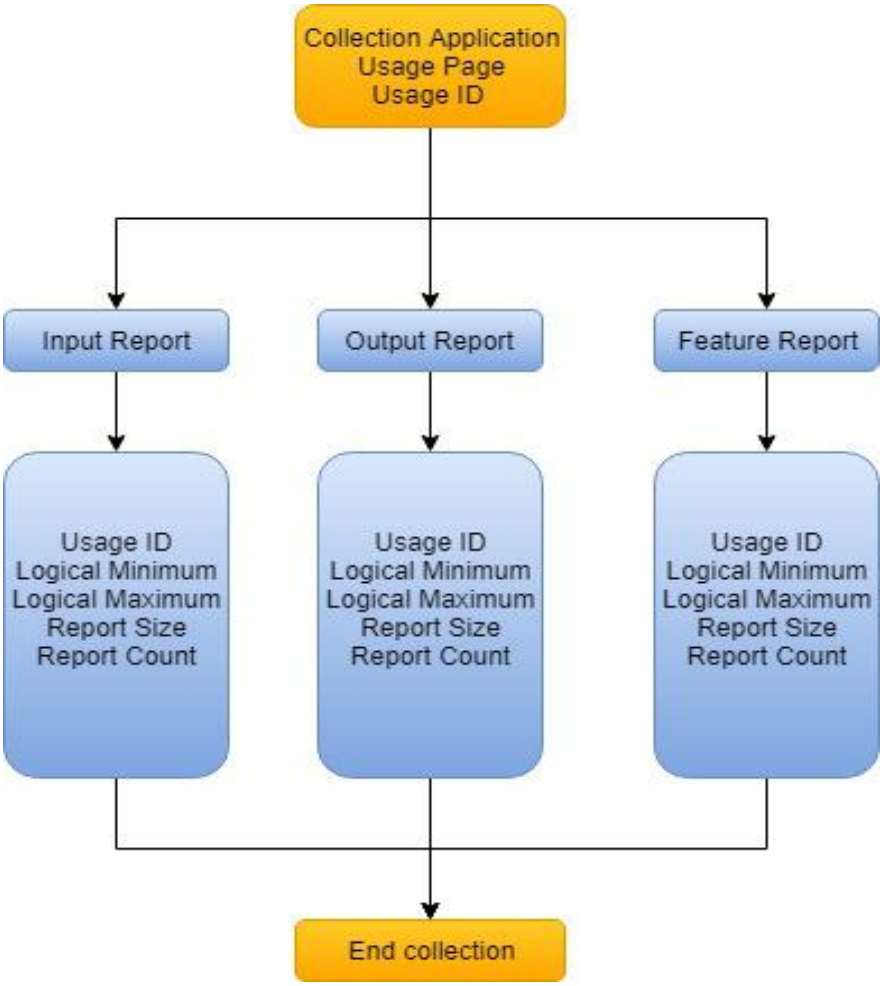


Imagen 2.7.3.5.1 – Diagrama Report Descriptor

La colección se ha definido como tipo aplicación, mientras que para identificar el *report descriptor* se ha definido un *Usage Page* de tipo específico, la cual no pertenece a ningún equipo en particular (botones, teclados, instrumentos médicos, etc.) sino que está definido por el fabricante.

Al tener tres tipos de *report* se ha definido un *Usage ID* para identificar cada uno.

En el caso del *Input report* los *items* se han configurado de la siguiente forma (ver tabla 2.7.3.5.10):

- **Logical Minimum y Logical Maximum**: se han definido para ofrecer un rango de valores de 0 a 255, aprovechando que las transferencias de datos serán de ocho bits (1 byte).

- **Input item**: establece que cada campo va a transmitir un dato, no lineal, de tipo variable, en el que se almacena bit a bit, de valor absoluto por lo que no se debe transmitir ninguna referencia, ni tampoco valores nulos y el *wrap* está deshabilitado.
- **Report Count**: se establecen 4 campos.
- **Report Size**: tamaño de cada campo de 8 bits.

Input Report Field	Posición 0	Posición1	Posición2	Posición3
Valor del convertidor	an3	an4	an5	an6
Tamaño	8 bytes	8 bytes	8 bytes	8bytes
Valor máx	255	255	255	255
Valor mín	0	0	0	0

Tabla 2.7.3.5.10 – Input Report

En el caso del *Output report* los *items* se han configurado de la siguiente forma (ver tabla 2.6.3.5.11):

- **Logical Minimum y Logical Maximum**: se han definido para ofrecer un rango de valores de 0 a 255, aprovechando que las transferencias de datos serán de ocho bits (1 byte).
- **Output item**: establece que cada campo va a transmitir un dato, no lineal, de tipo variable, en el que se almacena bit a bit, de valor absoluto por lo que no se debe transmitir ninguna referencia, ni tampoco valores nulos y el *wrap* está deshabilitado.
- **Report Count**: se establecen 5 campos en el *report*, lo que permite escribir 5 registros de los endpoints.
- **Report Size**: tamaño de cada campo de 8 bits.

Output Report Field	Posición 0	Posición1	Posición2	Posición3	Posición 4
Valor del las salidas	sleep	Pwm1 ciclo	Pwm2 ciclo	Dac ciclo	Output channel
Tamaño	8 bytes	8 bytes	8 bytes	8bytes	8 bytes
Valor máx	1	255	255	255	3
Valor mín	0	0	0	0	0

Tabla 2.7.3.5.11 – Output Report

También se ha incluido un *Feature report* obligado por la programación de la plantilla del firmware, que es necesario para el envío de datos de configuración.

Desde el punto de vista de algoritmo sólo hay que rellenar cada uno de los *items* con el valor de los bits adecuado. Estos valores son los indicados en las tablas de la sección antecedentes HID (sección 2.3.3.4.3). Aunque para este proyecto se ha partido de un descriptor de un *Generic Hid Device* y únicamente se han modificado algunos *items* para adaptarlos a los requisitos de diseño de la tarjeta. Generalmente se emplean en aplicaciones, que ya escriben en hexadecimal los valores de los bits correspondiente a la configuración de los *item*, sin tener que consultar las tablas. Un ejemplo de esta aplicación se puede encontrar en la de USB IF:

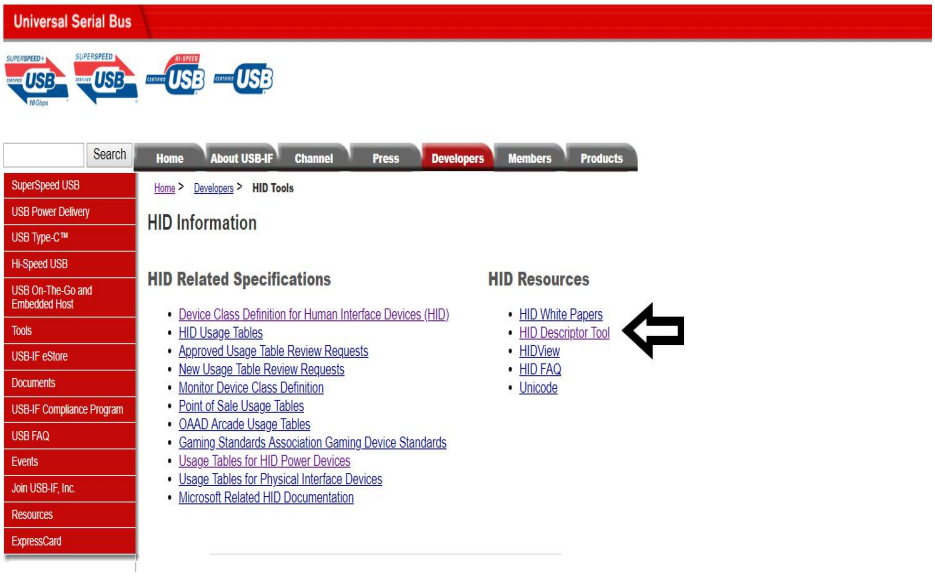


Imagen 2.7.3.5.2 – Web descarga HID Descriptor Tool

En la imagen 2.7.3.5.3 se indica un ejemplo de cómo se verían los descriptores detallados anteriormente en la aplicación:

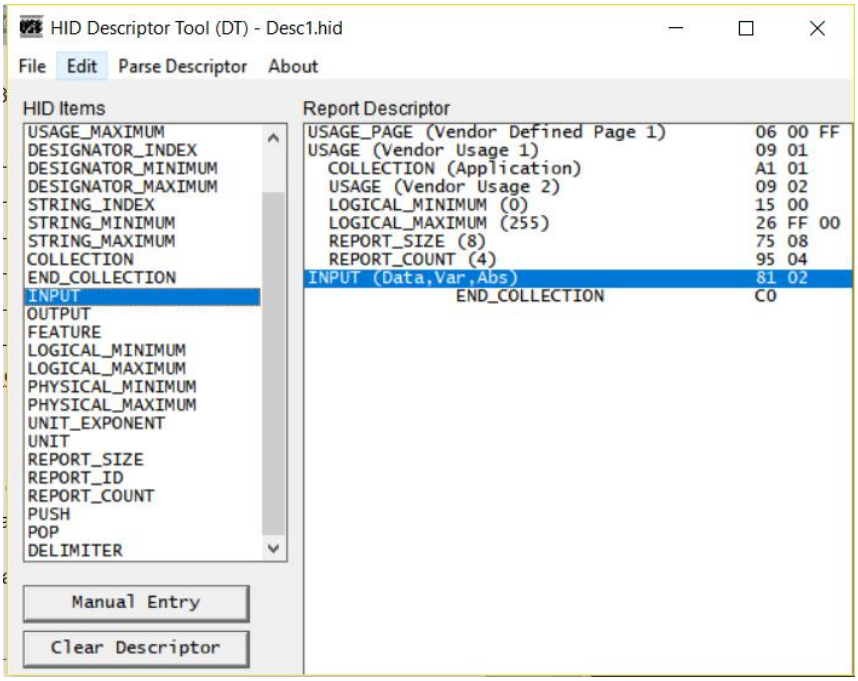


Imagen 2.7.3.5.3 – HID Descriptor Tool

2.7.3.6 Main.c.

Es el algoritmo que contiene todas las funciones para que la tarjeta pueda operar correctamente. Como ya se ha expuesto las principales funcionalidades que implementa son las siguientes:

- Procesamiento de las entradas y de las salidas.
- Comunicación con el PC.
- Función *Suspend*.
- Función *Remote Wake-up*.
- Señalización del estado de operación.

En esta sección se comentarán las soluciones adoptadas en el código que se puede encontrar en el Anexo III códigos de programación (ver flujograma imagen 2.7.3.3.6).

El archivo principal debe incluir las librerías de los periféricos que se han explicado en la sección 2.3.6.1, así como todas las librerías para la comunicación USB.

Cuenta con un doble registro dónde se almacena la dirección que le asigna Windows. De tal forma que el *driver* del sistema operativo puede localizarlo y dirigirse a él. Las variables que almacenan esta dirección son: **RX_DATA_BUFFER_ADDRESS** y **TX_DATA_BUFFER_ADDRESS**.

También se encuentran una serie de variables denominadas *handle.* Estas variables son punteros que pueden acceder a la memoria del USB del PIC para obtener información del estado de las transferencias y del bus.

Está formado por las siguientes funciones:

- **Void interrupt ISRCode()**: se encarga de atender las interrupciones por periféricos, en este proyecto sólo hay dos fuentes de interrupción:
  - *Timer 1*.
  - USB_INTERRUPT.

Las interrupciones USB aunque se atienden en primera instancia en esta función, luego se procesan en la función ***USBDeviceTask()***.

- **Int main()**: es la función principal del algoritmo (ver imagen 1.6.3.6.1). Primera llama a la función de iniciación de los puertos del microcontrolador comenzando el proceso de comunicación llevando al dispositivo a su estado inicial de conectado mediante la función **USBDeviceAttach().** A continuación se ejecuta un bucle while, el cual contiene las funciones de la DAQ (**ProcessUserInput()** y **ProcessUserOutput()**), y la que permite enviar los datos al USB (**ProcessIO()**). Dentro de este bucle se analiza la variable **sleep**, de tal forma que cuando el usuario seleccione el modo de bajo consumo (finaliza la comunicación con la tarjeta), se deja de actualizar el valor de las entradas y salidas de la DAQ, y unicamente se llama a la función **ProcessIO()** para que se puede rearmar la comunicación.
- **InitializeSystem()**: se trata de una función de configuración. La cual se puede dividir en tres partes:
  - Configuración del reloj del sistema: a través del registro OSCON, el cual se configura con el valor 0x7C, seleccionando el oscilador interno a una frecuencia de 16MHz con un multiplicador de 3. El módulo USB obliga el empleo del registro ACTCON, el cual habilita el *active clock tuning*, cuya función es autoajustar el oscilador interno a una frecuencia de 16MHz cada ciclo de reloj.
  - Configuración de los puertos: primeramente se limpian todos los registros LAT y PORT de los puertos para poder inicializarlos. A continuación se configuran los registros ANSEL y TRIS para configurar los pines de los periféricos como entradas o como salidas según corresponda.
  - Se inicializan los periféricos (ADC, PWM, DAC y *timers*), mediante sus correspondientes funciones de inicio.

  Por último se realiza una llamada a la función **USBDeviceInit()** la cual inicia la configuración la comunicación y estableces los *pipes* y los *endpoints*.

- **ProcessUserInput()**: está función se encarga de procesar las 4 entradas a través del convertidor AD. Para esta aplicación se va aprovechar el hecho de que sólo tenemos 4 entradas por lo que está planeada para muestrearlas

continuamente, y almacenar su valor en cuatro variables (**an3**, **an4**, **an5**, **an6**) que están asignadas a su correspondiente canal de entrada:

- **an3** = RA4.
- **an4** = RC0.
- **an5** = RC1.
- **an6** = RC2

Para muestrear estas entradas se emplea un bucle *for* el cual va cambiando el canal del convertidor en cada iteración.

Además se ha añadido un registro de desplazamiento a la derecha de dos bits, de tal forma que se adapta el valor del convertidor de 10 *bits* (valor máximo 1024) a 1 *byte*. De esta forma se puede cumplir que todas las transferencias USB sean de 1 *byte.*

- **ProcessUserOutput()**: se encarga de procesar los valores que se deben aplicar a las salidas (PWM y DAC) de la tarjeta. Al realizarse toda la comunicación USB en 8 *bits* el rango de valores que llegará estará entre 0 y 255, por lo que primero se aplica un control de errores, y a continuación se emplea en el caso de las PWM un registro de desplazamiento a la izquierda de 2 *bits* (se adapta a 10 *bits* y un rango de valores de 0 a 1023) y en el caso de la DAC un registro de desplazamiento a la derecha (se adapta a 5 *bits* y un rango de valores de 0 a 31). Finalmente se emplea un *switch case*, para activarlos en función del número de valores que llega por la entrada (se indica a través de la variable ***output channel***) de la siguiente forma:
  - **Output_channel**=1 : **PWM1**.
  - **Output_channel**=2: **PWM1**, **PWM2**.
  - **Output_channel**=3: **PWM1**, **PWM2**, **DAC**.
- **ProcessIO()**: es la función desde el punto de vista del USB más importante ya que es la encargada de recibir los datos del PC y de mandárselos una vez que la tarjeta los ha procesado.

  Primero se comprueba que el dispositivo se ha configurado correctamente y se encuentra preparado (que ha superado el estado *Configured* y no se

encuentra en un estado de *notConfigured*). A continuación se realizan los intercambios de datos:

- Lectura de datos: se realiza la comprobación de que el bus no está ocupado y se cargan los datos en el *endpoint.* Dentro de esta comprobación se encuentra el *flag* que controla el *timer 1*, para evitar que el bus se sature. Una vez que la comprobación se ha realizado con éxito se almacenan los datos en un vector de 8 *bytes* denominado **ToSendDataBuffer**, el cual se encarga de cargar los datos en el *endpoint OUT*, para luego enviarlos al PC. Los datos se envían según el formato de la tabla 2.7.3.6.1.

ToSendDataBuffer	0	1	2	3	4...7
Datos	an3	an4	an5	an6	--

Tabla 2.7.3.6.1 – Data Report Input

- Escritura de datos: al igual que en la lectura se realiza la comprobación de que el bus no esté ocupado. Si es así se consulta el tamaño de la variable **USBOutHandle,** de tal forma que si es distinto de cero significa que se ha recibido datos. En ese momento se consulta el vector **ReceivedDataBuffer** (de 8 bytes de tamaño) el cual almacena los datos que se han recibido en el *endpoint IN.* Finalmente se almacena cada dato en su correspondiente variable (ver tabla 2.7.3.6.2).

ReceivedDataBuffer	0	1	2	3	4
Datos	sleep	pwm_ciclo ao1	pwm_ciclo ao2	pwm_ciclo ao3	Output channel

Tabla 2.7.3.6.2 – Data Report Output

A mayores de estas funciiones existen otras relacionados intrínsicamente con el USB, que se denominan *callbacks*, las cuales están hechas para responder a eventos particulares. Aunque sólo se han modificado dos funciones *callbacks* (**USBCBSuspend()** y **USBCBWakeFromSuspend()**), es importante comentar en líneas generales para que sirve cada una:

- **USBCBSuspend()**: se trata de una función de reducción de consumo de energía. Si se detecta un período de tiempo de más de 3ms sin que haya actividad en el bus (definido por la especificación ver antecedentes) el periférico entra en un estado de suspensión definido por esta función. Al no poder emplear la función *sleep* se ha optado por reducir la frecuencia de reloj a 62KhHz, y deshabilitar todos los periféricos (en el caso de la PWM con parar el *Timer2* es suficiente). El valor al que se reduce la frecuencia de reloj es el mismo que se emplea en otras aplicaciones que realiza el fabricante como por ejemplos teclados numéricos.

  Esto permite disminuir el consumo de corriente del puerto USB, cuando la tarjeta no está trabajando.

- **USBCBWakeFromSuspend()**: esta función se emplea cuando la tarjeta sale del estado de Suspensión, y vuelve a compartir datos con el ordenador. Permite reactivar todos los periféricos y devuelve el reloj del microcontrolador a su ciclo de operación normal.
- **USBCBInitEp()**: configura e inicializa los *endpoints*. *Resetea* el *endpoint Out*.
- **USBCBSendResume()**: otorga al dispositivo la capacidad de ejecutar el RemoteWakeUp del PC. Para ello primero comprueba que el periférico ha entrado en el estado de suspensión. Una vez que se ha producido la suspensión del dispositivo realizar una temporización de 3 ms (lo obliga la especificación ver antecedentes). A continuación se coloca a 1 el bit *Resume* del registro UCON, y se mantiene a nivel alto durante un período de 2ms. Este bit controla una función que incorpora el módulo USB del PIC y que envía un *Set_Feature request* al PC para salir del modo de suspensión.

Se trata de una función que implementa teclados, ratones, etc. Es más, el algoritmo de la misma, se basa en el código de un teclado numérico por USB, desarrollado por microchip.

- **USER_USB_CALLBACK_EVENT_HANDLER()**: está función se emplea para notificar al usuario los eventos USB que van ocurriendo (*request*, errores, estados, etc.).
- **ReportSupported()**: comprueba si la comunicación USB HID programada soporta *Output* o *Feature report*. Toda la comunicación de la tarjeta se realiza mediante *Output report*.
- **User_Get_Report_Handler()**: si la comunicación HID se realiza mediante Input report o Feaute report.En esta aplicación como ya se ha explicado anteriormente toda la comunicación se realiza mediante *Input report*.
- **User_Set_Report_Handler()**: se emplea para cuando se desea recibir datos mediante *Control transfer*, a través del campo *Data stage*.

Todas estas funciones están relacionadas entre sí como muestra el flujograma de la Imagen 2.7.3.3.1. Este flujograma se centra en las funciones comentadas anteriormente, es decir únicamente en el *main.c*, pero numerosas funciones realizan llamadas que se encuentran en los archivos *usb_function_hid.c* y *usb_device.c*, pero sería complicar en exceso el diagrama, por lo que se suprimirán los procesos específicos de estas funciones:

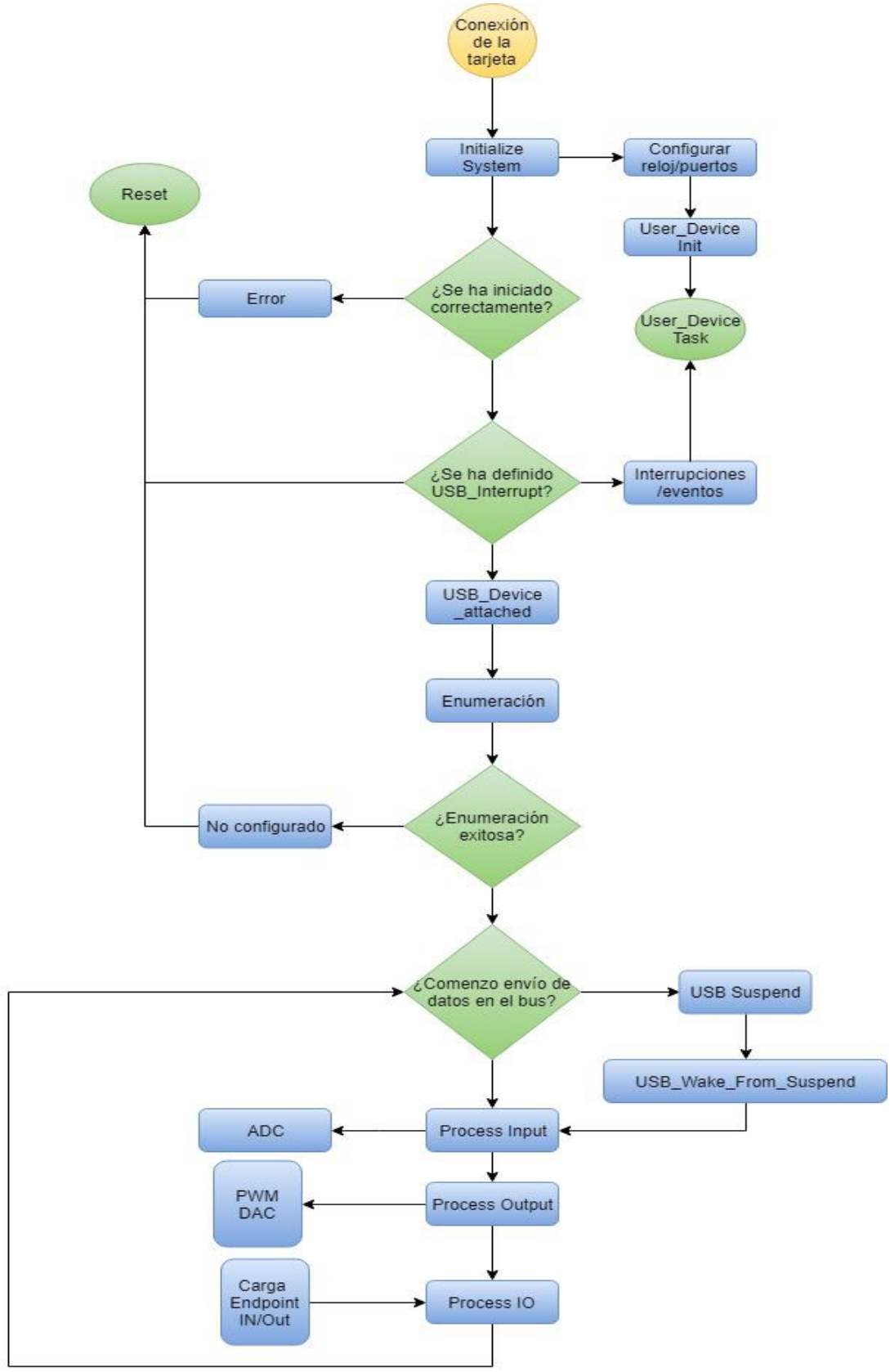


Imagen 2.7.3.3.1 – Flujograma archivo principal

2.7.4 Capa de usuario, comunicación con Matlab

La cuarta y última capa de es la que engloba todos los programas con los que el usuario se va a comunicar con la tarjeta. Está realizada tanto desde el punto de vista funcional para comunicar con la tarjeta, como del usuario para que sea práctico y pueda manejarlo fácilmente.

Desde del vista de programador el problema que se tiene que resolver es que Matlab no dispone de ninguna función para leer directamente el puerto USB cuando tiene conectado a él un dispositivo HID. Las únicas funciones de las que dispone Matlab son para la lectura/escritura de los puertos COM del ordenador asociado a la clase de dispositivos CDC. Una solución sería emplear las toolbox de instrumentación de Matlab pero son incompatibles con el fabricante Microchip por lo que habría que cambiar la clase de dispositivo. Además sería necesario que Matlab pudiera conectarse de manera bidireccional con la tarjeta, es decir, que pueda leer los datos que le envía y la vez escribirlos en la tarjeta.

La primera idea para resolver este problema es que la tarjeta se comportase como un teclado, de tal manera que cada dato enviado emularía una pulsación del teclado, así Matlab solo tendría que leer cada dato como si un usuario lo estuviese introduciendo de verdad por teclado mediante una función tipo *input*. Los datos se enviarían mediante código ASCII representándolos mediante los números del teclado directamente. Pero este modelo tiene tres limitaciones muy importantes:

- Sería imposible escribirle a un teclado un dato desde Matlab. Esto es debido a que Matlab no tiene una función de tipo *output*, para poder enviar datos por puerto USB.
- Al emular un teclado cada vez que se abriera un editor de texto, la tarjeta escribiría los datos representados mediante números en un editor de texto, y esto es extensible tanto si es el bloc de notas, como Word, o como la barra de búsqueda del explorador de Internet.
- La velocidad de envío es muy baja motivado a que se debía respetar un tiempo mínimo de pulsación de la tecla o teclas para luego simular también

que se dejaban de pulsar. Además un envío de datos/teclas demasiado rápido podía bloquear el teclado del ordenador.

A partir de ahí se abandonó la idea de trabajar con los típicos periféricos para que la tarjeta se pudiera comunicar con Matlab y se abordó más el concepto de que el propio programa trabajase directamente con el driver HID que era el que se encargaba de almacenar la información en memoria del PC.

2.7.4.1 HID.dll

La segunda idea para solucionar el problema de la comunicación fue intentar "hablarle" directamente al driver HID el cual ya se encuentra integrado en el sistema operativo.

Windows ofrece documentación referente al funcionamiento del driver que se puede resumir en el siguiente diagrama (imagen 2.7.4.1.1).

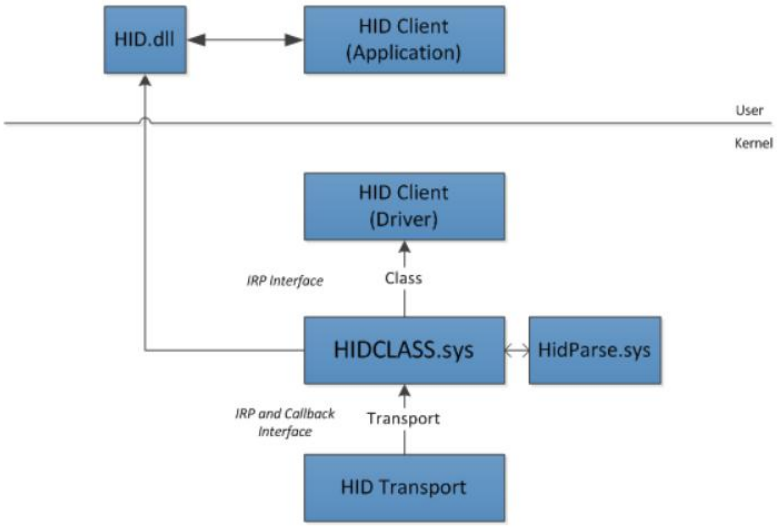


Imagen 2.7.4.1.1 – Diagrama Driver HID

Centrándose en la capa de Usuario, que es la que se necesita para realizar el programa, se puede observar la existencia de un archivo denominado ***HID.dll***. Las ***dll*** (*dynamic link-library*) son bibliotecas que contienen códigos que necesitan ser ejecutados a una velocidad muy elevada y que deben ser ejecutadas por dos

programas al mismo tiempo. Estas *dll* son accesibles tanto para el propio sistema operativo como para aplicaciones de terceros, las cuales pueden ejecutar estas bibliotecas por detrás del programa principal.

Volviendo a la biblioteca *HID.dll*, ésta contiene todas las funciones necesarias para poder interactuar con la parte del *kernel* asociado al *driver* HID. Por lo que si desde Matlab se pudiese llamar a estas funciones para poder emplearlas quedaría resuelto el problema de comunicación con la tarjeta. Pero, aunque existe una función específica de Matlab que permite cargar este tipo librerías, finalmente no se pudo ejecutar obligándonos a buscar otras soluciones.

2.7.4.2 HIDapi

Si no se puede cargar directamente las librerías del sistema operativo en Matlab se optó como tercera solución por usar una librería intermedia que le permitiese al software comunicarse con la *HID.dll*, éste a su vez con el *driver* y finalmente éste con la tarjeta. Esta librería intermedia es la *hidapi.dll* (ver imagen 2.7.4.2.1).

Imagen 2.7.4.2.1 – Diagrama Matlab-HIDapi.dll

HIDapi es una librería multiplataforma que funciona como *interface* permitiendo a una aplicación comunicar dispositivos USB de la clase HID en diversos sistemas operativos como Windows, Linux, y Mac OS X.

Desarrollado por **Signal 11** funciona con compiladores C, en este caso se ha optado por emplear el **Visual Studio v.2011** (ver imagen 2.7.4.2.2). Se recomienda la versión de ese año porque las posteriores no incluyen de serie el *SNK* y suelen dar problemas.

HIDapi permite cargar las funciones de la librería HID que interactúan con el *kernel* de Windows. Para luego ofrecer una interfaz de usuario, que mediante las funciones recogidas en la tabla 2.7.4.2.3 se puedan escribir y leer datos con dispositivos HID.

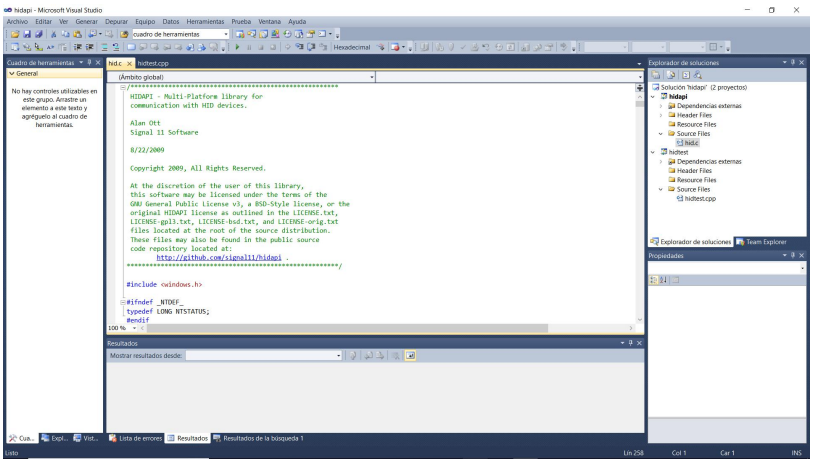


Imagen 2.7.4.2.2 – Visual Studio

Función	Descripción
hid_open	Abre la comunciación con el dispositivo HID
hid_get_manufacuter_string	Solicita el nombre del fabricante
hid_get_product_string	Solicita el nombre del prodicto
hid_get_serial_number_string	Solicita el número de serie
hid_send_feature_report	Envía un feature report al dispositivo
hid_write	Escribe datos al dispositivo mediante el envío de un Output report
hid_read	Lee los datos provenientes del dispositivo mediante un Input report
hid_exit	Cierra la comunicación con el dispositivo HID

Tabla 2.7.4.2.3 – Funciones HIDapi

Generalmente se generaría la *dll* a partir de los archivos, que se pueden descargar desde Github, con el compilador de Microsoft Visual Studio. A partir de ahí, y con el correspondiente *header file,* se podrían cargar en Matlab, el problema es que esta *dll* generada funciona con versiones de Matlab únicamente con 32 bits. Sin embargo, actualmente, todos los programas son mayoritariamente de 64 bits por lo que no funcionaría. Además se ha comprobado que el compilador de Microsoft visual studio a pesar de poder generarlo no lo creaba con todos los archivos que necesitaba por lo que tampoco funcionaría correctamente aunque fuese de 64 bits.

Por lo que llegamos a una última solución que es descargar una versión de *HIDapi* de 64 bits modificada, que se encuentra dentro del paquete **Robotic Toolbox** desarrollada por **Peter Corke**, que además incluye un ejemplo de cómo funciona y como se puede comunicar esta librería con Matlab.

El programa final va a estar basado en la solución anterior que fue descargar la librería *HIDapi* 64 modificada, la cual, aunque está modificada, tiene la mismas funciones que la librería *HIDapi* que vimos anteriormente.

En la siguiente sección se explica las funciones para implementar la capa de usuario del protocolo.

2.7.4.3 Protocolo en Matlab

Una vez solucionado el problema de la comunicación se pueden programar las funciones que soportarán la capa de usuario del protocolo indicado:

- **DAQ_Start()**: Inicia la comunicación con la DAQ.
- **DAQ_Stop()**: Termina la comuniación con la DAQ.
- **DAQ_Read()**: Lectura de datos de la DAQ.
- **DAQ_Write()**: Escritura de datos a la DAQ.

Estas funciones, por motivos de seguridad, siguen una estructura jerárquica como se puede ver en el diagrama de la imagen 2.7.4.3.1.

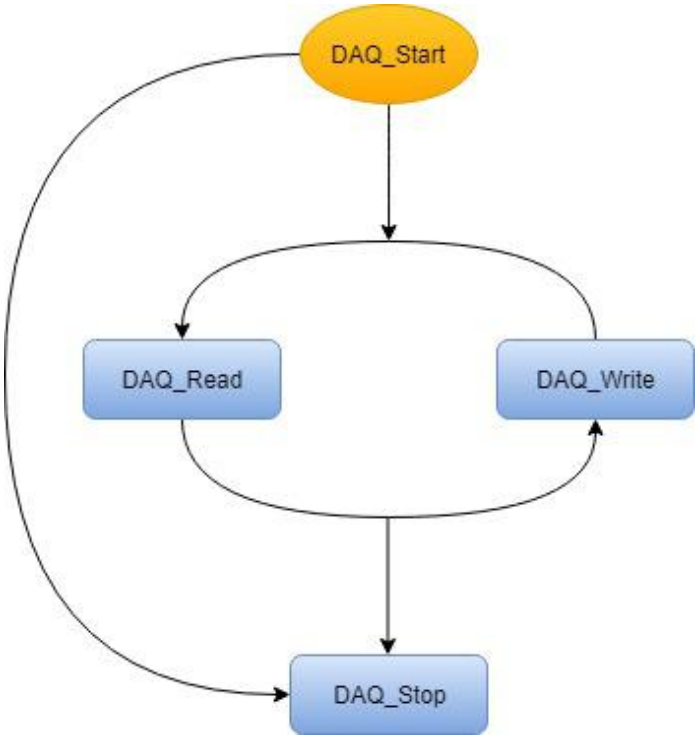


Imagen 2.7.4.3.1– Diagrama funciones Matlab

El orden debe ser siempre el mismo, si se ejecutase una lectura (**DAQ_Read()**) o una escritura (**DAQ_Write()**) sin haber ejecutado el **DAQ_Start()**, se podría genera un error que obliga a reiniciar Matlab, motivado a que no se ha abierto la comunicación con la *dll* y se estaría accediendo a posiciones de memoria que no han sido reservadas para ese cometido. Del mismo modo nunca se debe ejecutar el **DAQ_Stop()** sin haber ejecutado el **DAQ_Start()**.

Para ello se han incorporado controles de error los cuales además informan al usuario, si la llamada a la función viola el orden establecido de ejecución (imagen 2.7.4.3.2).

```
>> DAQ_Read_2()
Error al leer, inicie primero la comunicacion con la tarjeta
```

Imagen 2.7.4.3.2 – Control de errores

En las siguientes secciones se desarrolla cada una de las funciones de forma individual. Hay un factor común a las tres funciones que es el empleo de variables globales, lo que les permite compartir datos entre ellas.

Cada una de estas funciones se puede encontrar en el Anexo III de códigos de programación .

2.7.4.4 DAQ_Start()

Ver código en el anexo III de códigos de programación y ver el flujograma (Figura 2.7.4.4.1).

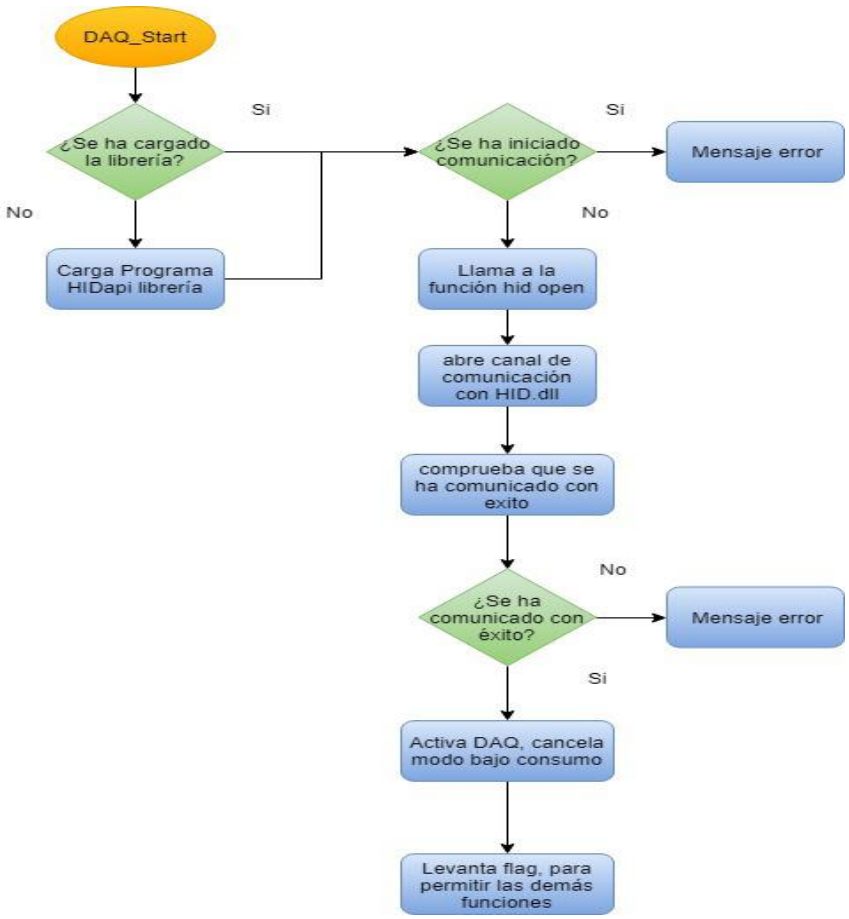


Figura 2.7.4.4.1 – Flujograma DAQ_Start()

Es la función de inicialización de la tarjeta. Abre un canal de comunicación con el programa *HID.dll*, que a su vez le permitirá comunicarse con el *driver* HID, y este en última instancia con el *hardware* para enviar o recibir los datos de la tarjeta.

Para que las funciones puedan comunicarse entre sí se han empleado variables globales. En cada función se debe definir cada variable global, siendo el DAQ_Start la función de inicio en ella se definen todas las variables globales que va usar la aplicación de usuario (ver tabla 2.7.4.4.1).

Variable	Descripción	Inicialización
handle	Puntero que permite pasar los datos a las funciones de la dll y traerlos de vuelta a Matlab	0
VendorID	Valor del identificador del fabricante que posee el dispositivo HID	1240
ProductoID	Valor del identificador del producto que posee el dispositivo HID	7
Flag_permiso	Flag que indica cuando se ha iniciado la comunicación con la tarjeta	0
Endpoint_out	Tamaño del endpoint OUT del host controller	6
ao1	Variable de salida PWM1	0
ao2	Variable de salida PWM2	0
ao3	Variable de salida DAC	0
Buffer_in	Registro para las variables de entrada	Vector de ceros de 4 posiciones
Pointer_readbuffer	Puntero para transmitir a Matlab las variables de entrada	Puntero de 4 posición
hidslib	Variable que almacena la librería para luego poder llamarla.	Libreriahid

Tabla 2.7.4.4.1 – Variables Matlab del protocolo

Lo primero es darle valores a las variables **Vendor ID** y el **Product ID**. Debido a que el algoritmo de *hidapi64.dll* no busca el puerto del ordenador dónde está ubicado la tarjeta de adquisición de datos sino que busca al propio dispositivo.

Mediante el **Vendor ID** y el **Product ID** que envía a la *HID.dll*, ésta interroga al *driver* por si tiene algún dispositivo con estos identificadores conectado.

A continuación se establece los tamaños de los registros de entrada y salida. Se podría afirmar que estos registros son los *endpoints* del ordenador. El tamaño en el caso del registro de entrada no es crítico que sea exactamente igual al número de campos del *report descriptor in*, ya que en el caso de que sea menor él único problema sería que se perderían datos y en el caso de que sea mayor se mostrarían ceros. Pero en el caso del registro de salida aunque se deja un cierto margen de seguridad, debe ser siempre una unidad mayor al número de campos del *report descriptor out*, debido a que debe acomodar el **report ID** (en este caso 0), si es un valor más grande lo esperado podría provocar un *reset* en la DAQ, porque Windows rellena con ceros hasta que se completa el registro de salida provocando un error entre el número de datos que debería contener el *report descriptor out* cuando se envía del ordenador y el número de datos cuando lo recibe el PIC .

El siguiente paso es cargar la librería *hidapi64.dll*, en el Matlab del PC en el que se esté ejecutando. Para ello se almacena la librería *hidapi64* dentro de la variable global **hidslib**, e inmediatamente mediante el comando **loadlibrary** se carga la librería en Matlab.

En vez de usar directamente el archivo de cabecera (*hidapi.h*) se emplea el *@hidapi64_proto*, además se ha nombrado la librería como *libreriahid*.

Se declara la variable local **flag_loadlibrary** para saber mediante el comando de Matlab **libisloadoed** si la librería se ha cargado correctamente, si devuelve un 1 es que en efecto, las funciones de la librería se encuentran listas para usar.

Toda esta operación se debe realizar la primera vez que se ejecute esta función en un equipo con sistema operativo Windows. Para evitar que cada vez que se llama la función se vuelve a realizar todo este proceso se emplea de nuevo el comando **libisloaded**, para interrogar si ya está cargada la librería; en caso afirmativo ya no hará falta recargarla de nuevo. Aun así el comando **libisloaded** tiene una limitación y es que Matlab no guarda información cuando se cierra la

sesión. Por lo que la Matlab no reconoce que existe la librería, la primera vez que se ejecuta la función en una nueva sesión volverá a cargarla lo que genera un *warning* (ver imagen 2.7.4.4.2).

```
Warning: The data type 'hid_device_infoPtr'
used by structure hid_device_info does not
exist.  The structure may not be usable.
> In loadlibrary at 406
  In DAQ_Start at 37
Warning: The library class 'libreriahid'
already exists.  Use a classname alias.
> In loadlibrary at 180
  In DAQ_Start at 38
```

Imagen 2.7.4.4.2 – Warning librería

Una vez que la librería se ha cargado correctamente se puede emplear el comando **libfunctionsview** para visualizar las funciones de HIDapi en Matlab (ver imagen 2.7.4.4.3).

Functions in library libreriahid

Return Type	Name	Arguments
hid_device_Ptr	hid_close	(hid_device_Ptr)
lib.pointer	hid_enumerate	(uint16, uint16)
[lib.pointer, hid_device_Ptr]	hid_error	(hid_device_Ptr)
int32	hid_exit	
hid_device_infoPtr	hid_free_enumeration	(hid_device_infoPtr)
[int32, hid_device_Ptr, uint8Ptr]	hid_get_feature_report	(hid_device_Ptr, uint8Ptr, uint64)
[int32, hid_device_Ptr, uint16Ptr]	hid_get_indexed_string	(hid_device_Ptr, int32, uint16Ptr, uint64)
[int32, hid_device_Ptr, uint16Ptr]	hid_get_manufacturer_string	(hid_device_Ptr, uint16Ptr, uint64)
[int32, hid_device_Ptr, uint16Ptr]	hid_get_product_string	(hid_device_Ptr, uint16Ptr, uint64)
[int32, hid_device_Ptr, uint16Ptr]	hid_get_serial_number_string	(hid_device_Ptr, uint16Ptr, uint64)
int32	hid_init	
[lib.pointer, uint16Ptr]	hid_open	(uint16, uint16, uint16Ptr)
[lib.pointer, cstring]	hid_open_path	(cstring)
[int32, hid_device_Ptr, uint8Ptr]	hid_read	(hid_device_Ptr, uint8Ptr, uint64)
[int32, hid_device_Ptr, uint8Ptr]	hid_read_timeout	(hid_device_Ptr, uint8Ptr, uint64, int32)
[int32, hid_device_Ptr, uint8Ptr]	hid_send_feature_report	(hid_device_Ptr, uint8Ptr, uint64)
[int32, hid_device_Ptr]	hid_set_nonblocking	(hid_device_Ptr, int32)
[int32, hid_device_Ptr, uint8Ptr]	hid_write	(hid_device_Ptr, uint8Ptr, uint64)

Imagen 2.7.4.4.3 – Funciones HIDapi Matlab

A continuación se realiza el control de errores para evitar que se pueda llamar dos veces seguidas a la función, ya que se estaría intentando comunicar con un espacio de memoria que ya estaba reservado, lo que provoca un error que

obligaría a cerrar la aplicación. Se informa al usuario mediante el pertinente mensaje de error.

```
fprintf('Ya se ha establecido la comunicación con la tarjeta\n')
```

Si la librería se ha cargado correctamente, y es la primera ejecución consecutiva de la función, ésta establece la comunicación con la tarjeta. Para ello se emplea el comando *calllib* que permite a Matlab llamar a las funciones contenidas en la *hidapi64.dll* para ello se debe cargar calllib con los siguientes argumentos(ver tabla 2.7.4.4.2):

Librería	Nombre de la función de la dll	vendorID	productID	Puntero para recibir/enviar datos
Hidslib	'hid open'	1240	7	pNull

Tabla 2.7.4.4.2 – Parámetros función calllib

El puntero en este caso es innecesario desde el punto de vista que la función *'hid open'* no devuelve ningún valor, aunque necesario debido a los parámetros que hay que pasarle a la función para que funcione dentro de la *dll*.

Como Matlab no implementa ningún tipo de método para saber si el dispositivo algún dispositivo HID conectado y la ejecución del flag *'hid open'* no devuelve ningún valor que permita saber si se ha conectado correctamente el dispositivo se realiza una comprobación que consiste en solicitar el *product string descriptor*, es decir, pedirle el nombre al dispositivo. Para ello se emplea la función *'hid_get_product_string"*, la cual necesita de los argumentos de la tabla 2.7.4.4.3

Librería	Nombre de la función de la dll	puntero	Registro de almacenamiento	Puntero para recibir/enviar datos
Hidslib	'hid open'	handle	pbuffer	productID

Tabla 2.7.4.4.3 – Parámetros función calllib

El registro *pbuffer* es un vector de ceros de 17 posiciones que almacena el nombre del DAQ. Se realiza una conversión del vector y se compara con el nombre para ver si es el esperado. Si es así, se levanta el *flag* **flag_permiso** y se da permiso al resto de las funciones para operar.

Esta función también se encarga una vez que ya se ha establecido la comunicación con la DAQ, de cancelar su modo de ahorro energético, para que entre en su estado de funcionamiento normal. Para ello se realiza una operación de escritura similar a la explicada en el apartado 1.6.4.5.

Para terminar cabe destacar que la función DAQ_Start modifica dos variables de la función DAQ_Read:

- **Buffer_in**: registro de almacenamiento de los valores que envía la tarjeta.
- **Pointer_readbuffer**: puntero de la librería, que le transmite los valores a Matlab.

2.7.4.5 DAQ_Read()

Ver código en Anexo III códigos de programación y ver flujograma (imagen 2.6.4.5.1).

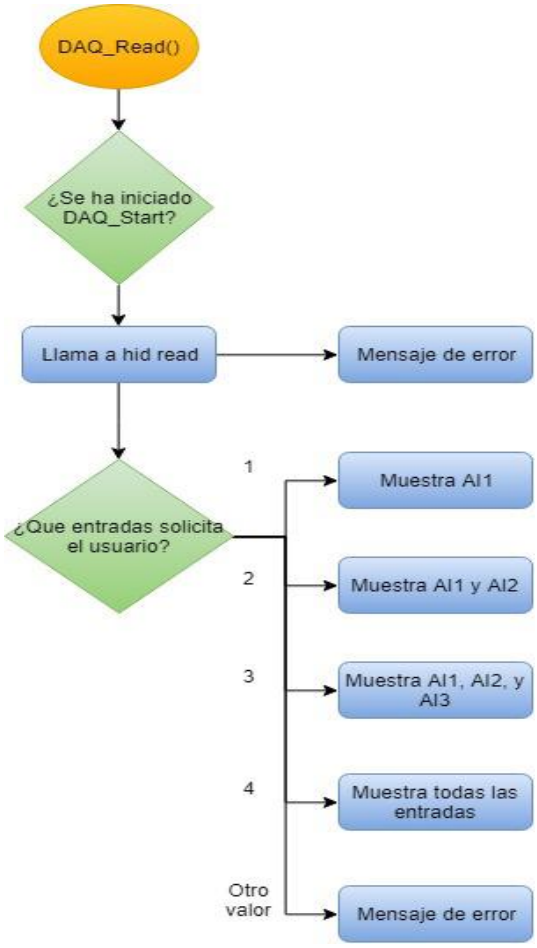


Figura 2.7.4.5.1 – Flujograma DAQ_Read

Se encarga de recibir los datos que le envía la tarjeta. Para ello emplea las variables de la tabla 2.7.4.5.1.

Variables	Tipo	Decripción
handle	Global	Puntero
IsOpen	Global	Flag de permiso
hidslib	Global	Libreria hid
buffer_in	Global	registro de entrada
pointer_readbuffer	Global	puntero para los valores
rmsg	Local	vector que almacena los valores

Tabla 2.7.4.5.1 – Variables DAQ_Read

Lo primero que se hace es un control de errores para asegurar que se ha abierto la comunicación con la tarjeta (que se haya ejecutado primero un DAQ_Start), mediante el *flag* ***isOpen***.

En caso afirmativo se realiza una llamada a la *hidapi64.dll* mediante el comando el *calllib* el cual debe contener los argumentos de la tabla 2.7.4.5.2.

Librería	Nombre de la función de la dll	puntero	Registro de almacenamiento	Tamaño del registro
Hidslib	'hid_read'	handle	Pointer_read_buffer	Buffer_in

Tabla 2.7.4.5.2 – Parámetros calllib

Los valores son transmitidos desde la librería a Matlab mediante el puntero ***Pointer_read_buffer*** y se almacenan en el vector ***rmsg***. Quedando configurado el vector como se indica en la tabla 2.7.4.5.3.

Posición	1	2	3	4
Puerto Tarjeta	Ai1	Ai2	Ai3	Ai4
Puerto PIC	Puerto A4	Puerto C0	Puerto C1	Puerto C2

Tabla 2.7.4.5.3 – Relación puertos vector función de lectura

La función es variable en cuanto a la salida; es decir, si el usuario necesita conocer el valor del puerto Ai1 de la tarjeta, deberá escribir una variable a la función para almacenar el valor. Si necesita el del Ai2 necesitará escribirle dos valores y así sucesivamente. Esto permite optimizar el código y aumentar la velocidad. Se ha implementado mediante la función *nargout* que permite conocer el número de variables que se iguala a la función **DAQ_Read()** (ver imagen

2.7.4.5.1). Si no se encuentra el puerto de la tarjeta seleccionado se emite un mensaje de error.

```
>> DAQ_Read_2()
Error, No se ha podido encontrar el canal seleccionado
>> a1=DAQ_Read_2()

a1 =

    99

>> [a1,a2]=DAQ_Read_2()

a1 =

    99


a2 =

     0
```

Imagen 2.7.4.5.1 – Funcionamiento DAQ_Read

Como se ha nombrado anteriormente toda la comunicación USB se realiza en un 1 *byte*. Por lo que se reciben datos con un valor entre 0 y 255 que se adaptan a una escala de 0 a 100 para que el usuario los puede multiplicar por la tensión de referencia adecuada en función de lo que necesite.

Un punto importante en cuanto a la función de lectura es que, como ya se mencionó anteriormente, el protocolo desarrollado establece que el PIC esté constantemente muestreando y enviando datos al ordenador, de tal manera que Matlab recoge directamente los datos que ya se encuentran en el ordenador, mejorando la velocidad del protocolo.

2.7.4.6 DAQ_Write()

Ver código en Anexo III códigos de programación y ver flujograma (imagen 2.7.4.6.1).

Su función es permitir al usuario escribir las salidas de la DAQ. Para ello hace uso de las siguientes variables recogidas en la tabla 2.7.4.6.1.

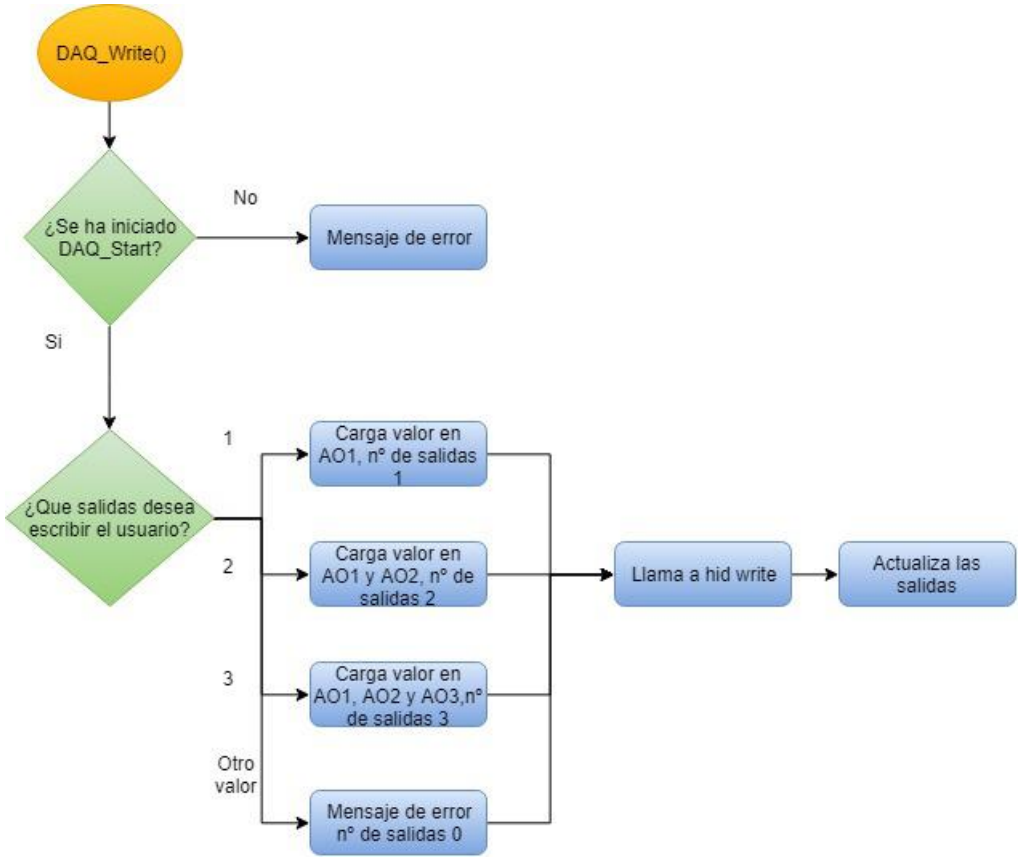


Tabla 2.7.4.6.1 – Flujograma DAQ_Write

Variables	Tipo	Decripción
handle	Global	Puntero
isOpen	Global	Flag de permiso
hidslib	Global	libreria hid
Endpoint_out	Global	registro de salida
AO1	Global	Salida PWM1
AO2	Global	Salida PWM2
AO3	Global	Salida DAC
Output_channel	Local	Número de canales seleccionables
write	Local	Array de 6 posiciones

Tabla 2.7.4.6.1 – Variables DAQ_Write

En el prototipo de la función se han declarado tres argumentos de entrada, que corresponden a las tres salidas de la tarjeta. Del mismo modo que DAQ_Read era variable, la función DAQ_Write es variable en cuanto al número de entradas que

se desea introducir, esto es, si se desea escribir a la salida AO1, únicamente se escribirá un valor de 0 a 100, mientras que si se desea escribir a AO1 y AO2 se escribirán dos salidas. Para ello se emplea *nargin*, que permite determinar el número de salidas que se desea activar. Si no se introduce un número de salidas adecuado (1 a 3), se presenta un mensaje de error por pantalla, y no se actualiza el valor de las salidas (ver imagen 2.7.4.6.1).

```
>> DAQ_Stop
>> DAQ_Start
>> DAQ_Write_2(0,100,100)
>> DAQ_Write_2()
Error, Introduzca al menos un valor
```

Imagen 2.6.4.6.1 – Funcionamiento DAQ_Write

El valor que puede introducir el usuario como se ha nombrado anteriormente es un porcentaje de 0 a 100, pero la comunicación con la DAQ se realiza en un 1*byte* por lo que se realiza una conversión del valor para adaptarlo a una escala de 0 a 255.

También se ha incluido un control de errores para saturar el valor máximo a 100 (255) y el valor mínimo que puede introducir el usuario a 0.

A continuación se selecciona las salidas que el PIC debe actualizar mediante la variable **output_channel**. Esto es debido a que en cada envío Windows actualizará todos las variables que se cargan en el registro de salida, sí no se indicase cuales tiene que modificar actualizaría el valor de todas las salidas de la DAQ. Además hay que tener en cuenta que el tamaño del vector de salida debe permanecer constante para enviar la misma cantidad de *bytes* en el **report out**, sino se puede producir un *reset* de la DAQ.

Para enviarle los datos de escritura se emplea un puntero con los valores previamente almacenados en el vector ***write***. Este puntero se como argumento a **calllib()** permitiendo enviar los valores a *hidapi64.dll*.

Finalmente se emplea **calllib()** con los siguientes parámetros que se pueden observar en la tabla 2.7.4.6.2, para realizar la escritura.

Librería	Nombre de la función de la dll	puntero	Registro de almacenamiento	Tamaño del registro
Hidslib	'hid_write'	handle	Pointer_writebuffer	6

Tabla 2.7.4.6.2 – Parámetros calllib()

Mencionar que pese a que sólo se necesitan enviar cinco valores el *array write* es de 6 posiciones. Esto es debido a que el primer valor es el *Report ID* como se ha comentado anteriormente en la sección 2.6.4.7.5.

2.7.4.7 DAQ_Stop()

Ver código en Anexo III códigos de programación y ver flujograma (figura 2.7.4.7.1).

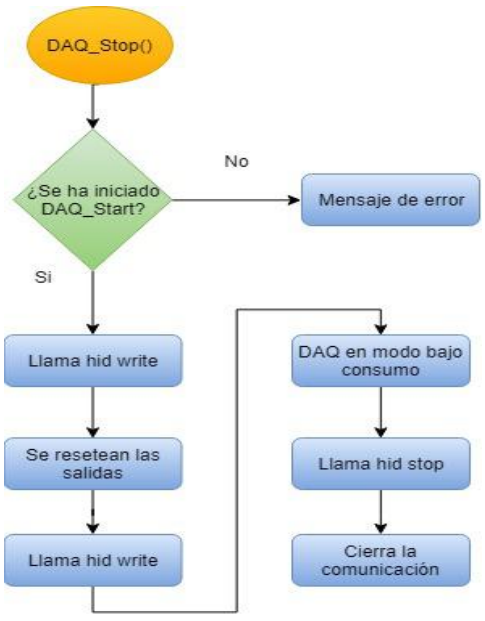


Figura 2.7.4.7.1– Flujograma DAQ_Stop()

Su función es la de finalizar la comunicación, resetear las salidas y activar el modo de ahorro energético de la DAQ . Las variables empleadas se recogen en la tabla 2.7.4.7.1.

Variables	Tipo	Decripción
handle	Global	Puntero
isOpen	Global	Flag de permiso
hidslib	Global	libreria hid
Endpoint_out	Global	registro de entrada
pointer_readbuffer	Global	puntero para los valores
rmsg	Local	vector que almacena los valores

Tabla 2.7.4.7.1– Variables DAQ_Stop()

Para desactivar las salidas, basta con escribir un cero en los módulos DAC y PWM, y luego indicar que se deben desactivar las 3 salidas. En el fondo se está realizando la operación de escritura del **DAQ_Write()**.

De manera homónima se realiza otra operación de escritura para activar el modo de bajo consumo.

A continuación se emplea la función de cierre mediante un **calllib()**, al cual se le pasan los parámetros que se pueden observar en la tabla 2.7.4.7.2 .

Librería	Nombre de la función de la dll	puntero
Hidslib	‘hid_close'	handle

Tabla 2.7.4.7.2 – Parámetros calllib()

Una vez cerrada la comunicación con la DAQ, se resetean todas las variables globales para que no consuman memoria.

2.7.4.8 Help_DAQ()

Como ayuda al usuario se han incluido en cada función un texto que contiene las instrucciones de uso de la función y los valores que devuelve (ver imagen 2.7.4.8.1).

```
>> help DAQ_Stop
 Función DAQ_Stop
    Finaliza la comuncación con la tarjeta
    See also calllib

calllib - Call function in shared library

    This MATLAB function calls function, funcname, in library, libname, passing
    input arguments, argl,...,argN, and returns output values obtained from funcname
    in xl,...,xN.

    [xl,...,xN] = calllib(libname,funcname,argl,...,argN)

    Reference page for calllib

    See also libfunctionsview, loadlibrary
```

Imagen 2.7.4.8.1 – Funcionamiento Help_DAQ()

2.8 Resultados finales

Al final se ha desarrollado un protocolo de comunicación rápido y robusto, que dota al usuario de una DAQ que integra todos las funcionalidades del puerto USB sin necesidad de transmitir los datos mediante puerto Serie emulado y que podrá operar a través de forma eficiente y sencilla (ver imagen 2.8.1).

La tarjeta de adquisición de datos desarrollada para este protocolo tendrá las siguientes especificaciones (ver imagen 2.8.2):

- 4 puertos de entradas: AI1, AI2, AI3, AI4. Con un rango de tensiones de entrada de 0 a 5V y una resolución de 10 *bits* en la medida.
- 3 puertos de salidas analógicas: AO1, AO2, AO3. Con un rango de tensiones a la salida de 0 a 5V y una resolución de 8 *bits* y 5 *bits*.
- *Plug and play*.

- Modo de ahorro de energía: la DAQ no comenzará a transmitir hasta que el usuario no la inicie el programa en Matlab.
- Modo *remote Wake-up*: para control de procesos que obliguen a estar funcionando la DAQ durante mucho tiempo sin que haya cambios que obliguen al intervención del usuario, evita que el ordenador entre en suspensión. Sólo disponible de momento en versiones de Windows 7, Vista y Xp.
- Señalización del estado de alimentación y transmisión de datos.
- Dispositivo sin necesidad de instalación de *drivers* adicionales en el sistema operativo.
- Microcontrolador extraible lo que permite una rápida reparación en caso de fallo o que sea necesario modificar el *firmware*.

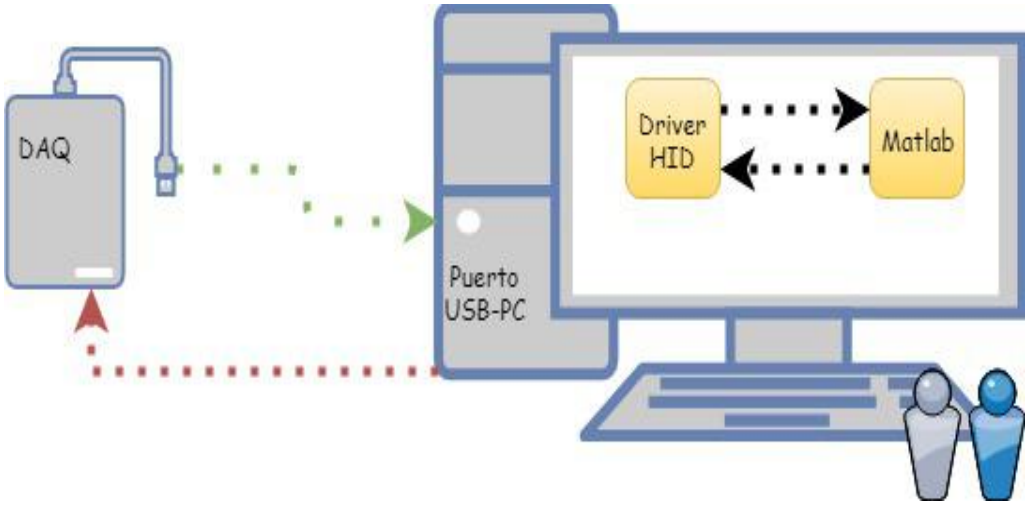


Imagen 2.8.1 – Diagrama resumen del protocolo

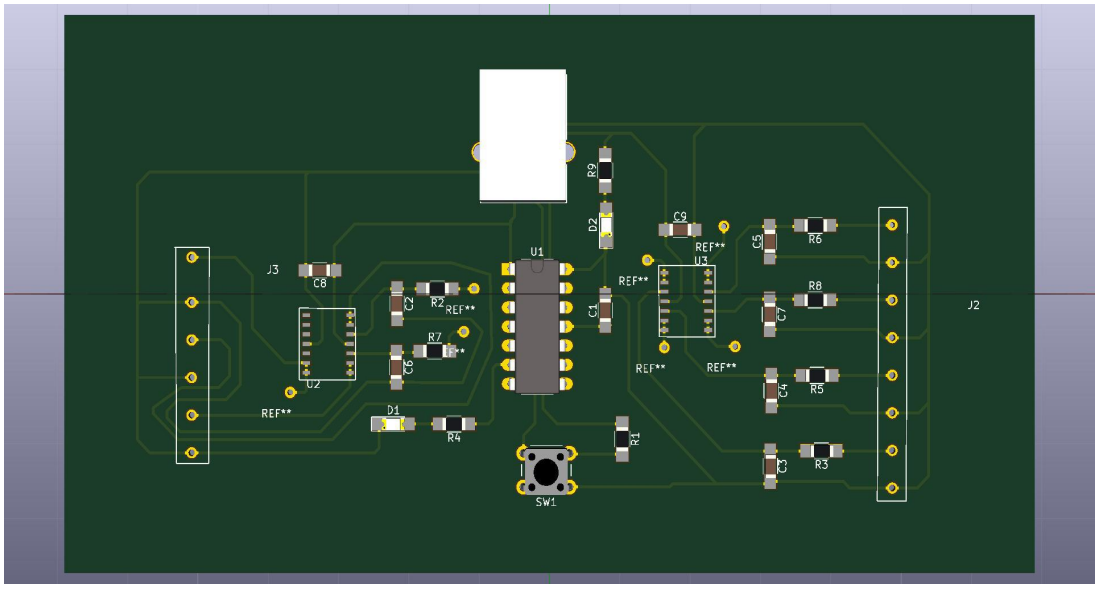


Imagen 2.8.2 – Placa DAQ

Por la parte de software el usuario contará con un programa muy útil como Matlab para comunicarse con la DAQ. Dentro de Matlab, como se ha visto en el análisis de soluciones, contará con cuatro funciones, las cuales le permitirán interactuar fácilmente con la tarjeta de adquisición de datos:

- **DAQ_Start()**: inicia la DAQ.
- **DAQ_Write()**: activa las salidas en función del voltaje deseado.
- **DAQ_Read()**: lee las entradas.
- **DAQ_Stop()**: finaliza la comunicación con la DAQ.

Además cada una de las funciones contienen control de errores para que las pueda ejecutar sin temor ha que se produzca ningún fallo y una pequeña guía para saber como usarlas correctamente.

Con esto se cubre lo que es el desarrollo y las funciones del protocolo.

2.8.1 Conectando la tarjeta al PC

El primer paso para emplear este protocolo es conectar la DAQ al PC. En el momento que se conecta se escuchará el sonido conforme se está ejecutando el

plug and play y se inicia la enumeración. Si se configura correctamente la tarjeta de adquisición de datos, y Windows le asigna el *driver* HID, éste aparecerá en la pantalla del Administrador de dispositivos (ruta:*"Administración de equipos> Administrador de dispositivos> Dispositivos de interfaz de usuario (HID)"*) en el grupo de Dispositivos de interfaz Humana (HID) (ver imagen 2.8.1.1).

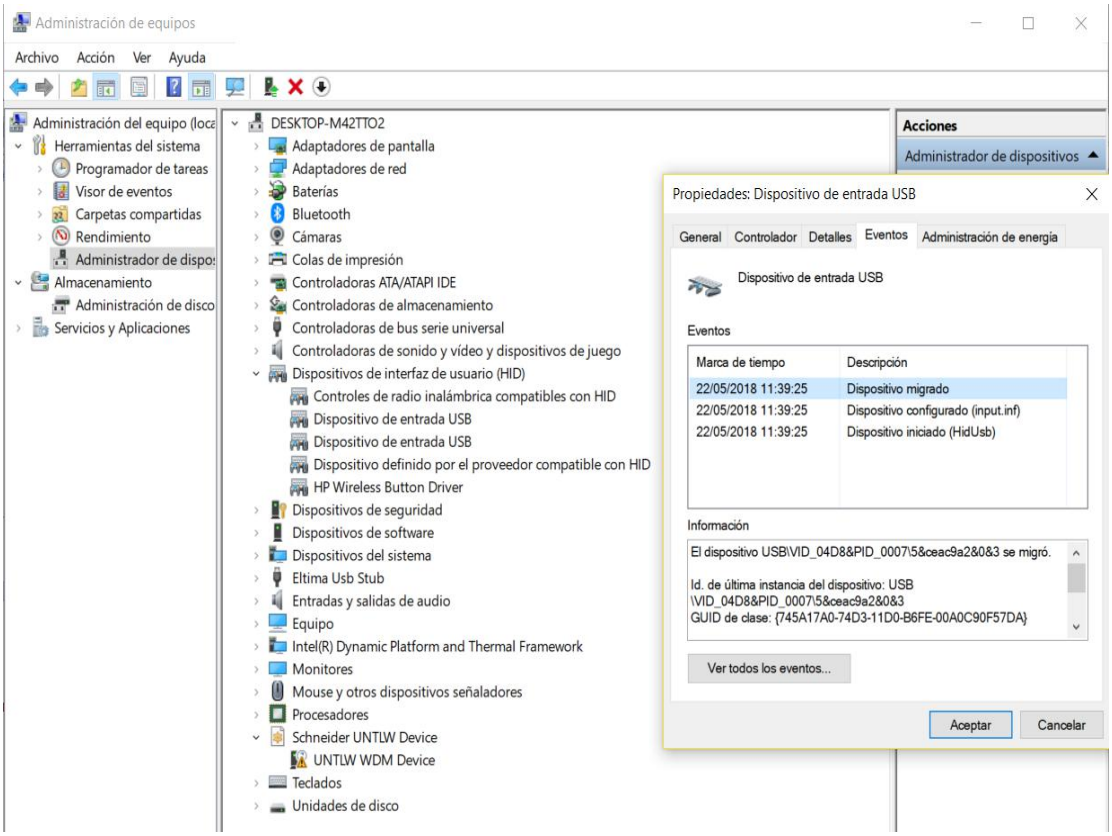


Imagen 2.8.1.1 – Administrador de dispositivos

La tarjeta se puede identificar por su VID y su PID (0x04D8 y 0x0007), como se puede observar en la imagen 2.7.1.1. Además se puede ver como la DAQ está perfectamente configurada e iniciada lista para su uso, sin que reporte ningún problema (ver imagen 2.8.1.2).

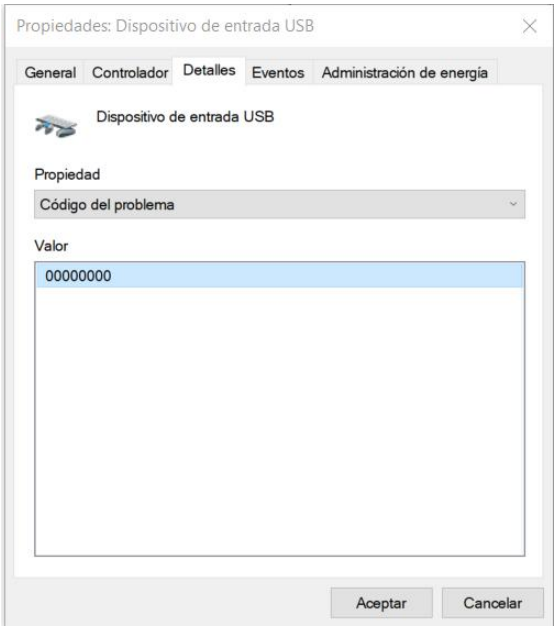


Imagen 2.8.1.2 – Código error.

En configuración Bluetooth y otros dispositivos (ruta: *"Configuración > Bluetooth y otros dispositivos"*) el sistema operativo identificó la DAQ como un dispositivo genérico, que no lleva asociado ninguna subclase ni protocolo (a diferencia del ratón que también se encuentra conectado), ver imagen 2.8.1.3. En la imagen también se puede observar el nombre de la tarjeta dada por el *String Descriptor*.

Imagen 2.8.1.3 – Bluetooth y otros dispositivos.

2.8.2 Comprobando el funcionamiento de la DAQ

Se debe comprobar que la DAQ es capaz de transmitir los paquetes de datos al PC, para descartar un fallo en el *firmware* o en el propio circuito de la tarjeta. **DAQ_Start()** inicia la comunicación con la tarjeta de adquisición de datos. En ese momento la DAQ conforme al protocolo creado comienza a enviar los datos al PC de manera continúa. Si es correcto el envío de datos, se puede ver mediante el empleo de un *sniffer* para puertos USB, el cual muestrea los paquetes de datos que transitan a los puertos del ordenador.

El *sniffer* seleccionado es el USBCap que funciona conjuntamente con el programa Wireshark para mostrar los datos.

No.	Time	Source	Destination	Protocol	Length	Info
4502	13.535044	1.15.1	host	USB	35	URB_INTERRUPT in
4503	13.536941	host	1.15.0	USB	36	GET DESCRIPTOR Request STRING
4504	13.536987	1.15.1	host	USB	35	URB_INTERRUPT in
4505	13.537930	1.15.0	host	USB	62	GET DESCRIPTOR Response STRING
4506	13.537940	1.15.0	host	USB	28	GET DESCRIPTOR Status
4507	13.538257	host	1.15.0	USB	36	GET DESCRIPTOR Request STRING
4508	13.539236	1.15.0	host	USB	62	GET DESCRIPTOR Response STRING
4509	13.539245	1.15.0	host	USB	28	GET DESCRIPTOR Status
4510	13.541033	1.15.1	host	USB	35	URB_INTERRUPT in
4511	13.543034	1.15.1	host	USB	35	URB_INTERRUPT in
4512	13.544037	1.15.1	host	USB	35	URB_INTERRUPT in
4513	13.546035	1.15.1	host	USB	35	URB_INTERRUPT in
4514	13.548030	1.15.1	host	USB	35	URB_INTERRUPT in
4515	13.550069	1.15.1	host	USB	35	URB_INTERRUPT in
4516	13.550988	1.15.1	host	USB	35	URB_INTERRUPT in
4517	13.553032	1.15.1	host	USB	35	URB_INTERRUPT in
4518	13.555029	1.15.1	host	USB	35	URB_INTERRUPT in
4519	13.556032	1.15.1	host	USB	35	URB_INTERRUPT in
4520	13.558032	1.15.1	host	USB	35	URB_INTERRUPT in
4521	13.560014	1.15.1	host	USB	35	URB_INTERRUPT in

Imagen 2.8.2.1 – Wireshark

En la imagen 2.8.2.1 se pueden observar los paquetes de datos y también algunos de los descriptores (exactamente los *String Descriptor*).

2.8.3 Análisis del consumo de memoria en el PIC

Una vez compilado el *firmware* con todas las librerías y programas, se puede ver en la imagen 2.8.3 que el consumo de memoria de programa es bastante elevado. Aunque no es un dato del todo exacto, debido a que se emplea la versión gratuita del compilador, el cual carece del optimizador de código, si que constituye una desventaja debido a que del total del 50% de memoria consumida aproximadamente el 50% se debe a la configuración e implementación del USB mientras que el 10% restante a la configuración de los demás periféricos. Lo que

sólo deja un espacio libre del 40% para añadir más funcionalidades a la tarjeta en un futuro.

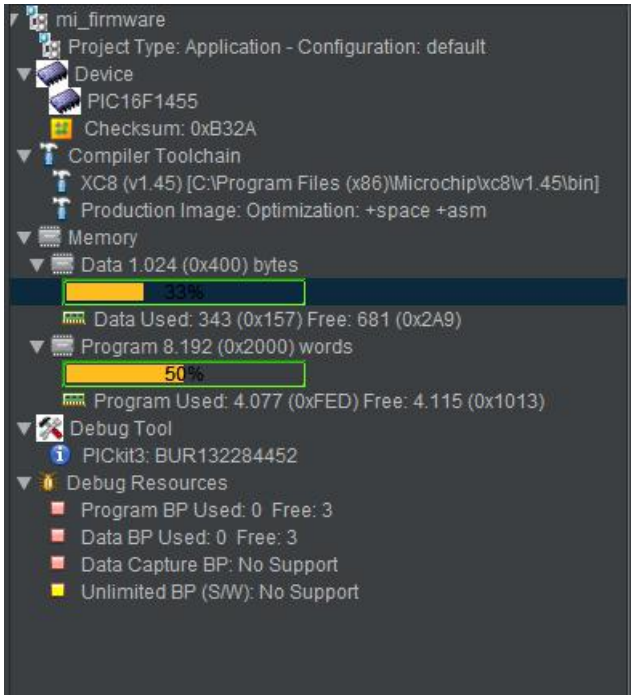


Imagen 2.8.3 – Consumo de memoria

2.8.4 Velocidad del protocolo

Por último queda probar el protocolo de Matlab y la velocidad que puede alcanzar la tarjeta. Como se dijo al comienzo del análisis de soluciones también se buscaba que éste fuera rápido. Por lo que se ha procedido a medir su velocidad. Para probar la velocidad se emplea el algoritmo que se puede encontrar en el programa test tarjeta (ver anexos IV) (ver imagen 2.8.4.1). Se emplea el osciloscopio para poder ver la velocidad.

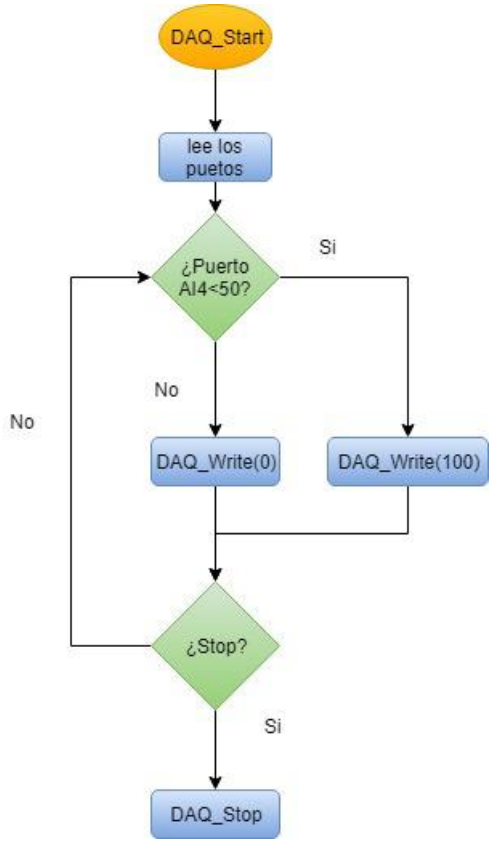


Imagen 2.8.4.1 – Flujograma test velocidad

La velocidad máxima que es capaz de alcanzar la tarjeta es de 6ms superior a la velocidad máxima que puede alcanzar Arduino o algunas tarjetas de adquisición de datos comerciales.

Para medir la velocidad del protocolo se ha creado un programa en Matlab empleando ya las funciones del protocolo, que se basa en ir escribiendo y leyendo los puertos de la placa. De tal forma que se va a generar una señal cuadrada en función de lo que vaya leyendo los puertos de DAQ. El período de esa función cuadrada será la velocidad de la tarjeta de adquisición de datos.

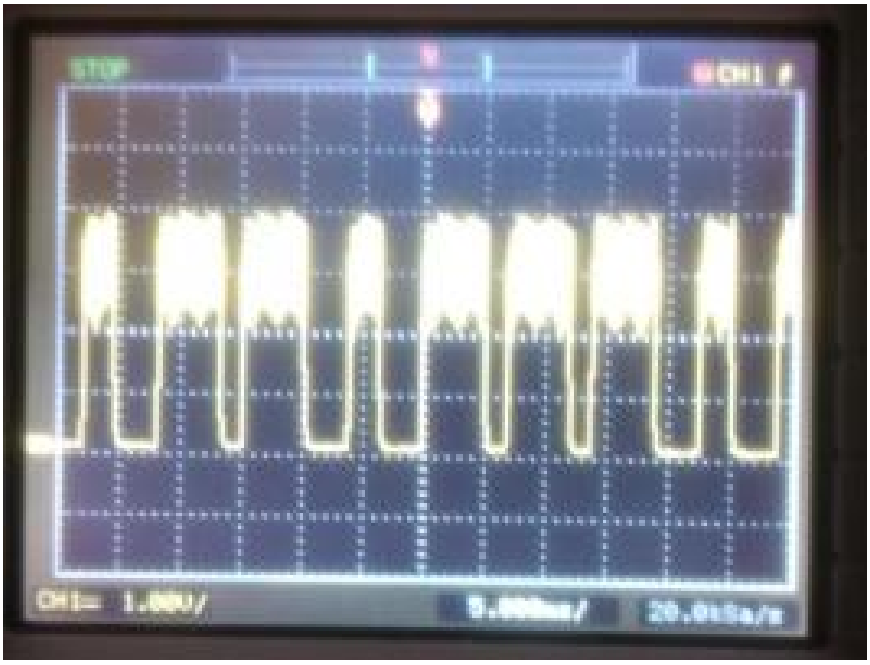


Imagen 2.8.4.2 – Test de velocidad osciloscopio

Como se puede observar en la imagen 2.8.4.2, la velocidad es de aproximadamente 6ms. Se puede observar una pequeña latencia debido a los tiempos de atención de Windows a los envíos de la DAQ.

No es una mejora de la velocidad muy grande pero permite bajar de los 10ms donde se encuentran otras tarjetas y *hardware* como Arduino. También hay que tener en cuenta que no se ha empleado el modo de velocidad más alto de la especificación USB 2.0, que es el *high-speed,* debido a que el SIE del PIC no lo soporta. La diferencia principal es que el modo *full-speed* empleado no puede bajar de 1ms a la hora de acceder al bus, mientras que el modo *high-speed* puede bajar a los microsegundos. Lo que se traduce en una tasa de transferencia teórica para la DAQ de este protocolo de 12Mbps, ni una décima parte de la tasa de transferencia máxima en modo *high-speed* que es de 480Mbps.

2.8.5 Revisión del protocolo

En este apartado se comentarán una serie de aspectos que se pueden mejorar del protocolo pero que por falta de tiempo o de medios no se pudieron llevar a cabo.

- Se debería realizar una modificación de la *HIDapi.dll* para adaptarla a las necesidades de Matlab. Faltan por ejemplo funciones que permitan detectar el momento en el que el dispositivo se conecta.
- En vez de trabajar llamando a una librería externa se podría crear un nuevo programa que comunique con el *HID.dll* de la misma forma que *HIDapi,* pero generarlo en Java para poder integrarlo mediante *Mexfiles* en Matlab.
- Mejorar el modo de velocidad del PIC, pasarlo de un *full-speed* a un modo *high-speed.*
- Dado que el rango de tensión de la DAQ de entrada son muy limitados (0 a 5V), se podrían mejorar también.

2.8.6 Conclusiones

Los resultados finales están centrados en el protocolo de comunicación, en esta última sección del TFG, quiero centrarme más en el empleo del protocolo USB, y sobre todo el tema del empleo del puerto Serie y porque, desde mi punto de vista, no se implementan puertos USB “puros” como el que se propuso en el análisis de soluciones.

Lo primero que me di cuenta es la enorme complejidad de la comunicación USB a nivel tanto de concepto como de algoritmo, sí que las funcionalidades que ofrece son ilimitadas comparado con el puerto Serie o el puerto Paralelo, pero esto supone una inversión de recursos enormes para sólo conseguir la comunicación entre un dispositivo y el ordenador. Luego, por otra parte, está el alto consumo de memoria en el PIC, comparado con otros periféricos. Y por último, habría que preguntarse si existen aplicaciones que de verdad vayan a necesitar semejante velocidad como la que puede ofrecer el USB 2.0, aplicando la tasa de transferencia máxima.

Desde el punto de vista técnico, queda demostrado con este TFG que sí que es viable llevarlo a cabo, sin emplear ni una décima parte de la tasa de transferencia máxima del USB 2.0, se mejora *hardware* como Arduino, y que se puede comunicar con entornos como Matlab o Visual Studio. Pero desde un punto de vista más comercial, existen soluciones más viables como el puerto Serie emulado.

TÍTULO: **CREACIÓN DE UN PROTOCOLO DE COMUNICACIÓN ENTRE UN PC Y UN PUERTO USB DE UN MICROCONTROLADOR Y SU INTEGRACIÓN EN MATLAB**

ANEXOS

PETICIONARIO: **ESCUELA UNIVERSITARIA POLITÉCNICA**

AVDA. 19 DE FEBRERO, S/N

15405 - FERROL

FECHA: **SEPTIEMBRE DE 2018**

AUTOR: EL ALUMNO

Fdo.: CARBALLAS CHAS SERGIO

Índice ANEXOS

3 ANEXOS

3.1 Anexo I: DATOS DE PARTIDA

ESCUELA UNIVERSITARIA POLITÉCNICA

ASIGNACIÓN DE TRABAJO FIN DE GRADO

En virtud de la solicitud efectuada por:
En virtud da solicitude efectuada por:

APELLIDOS, NOMBRE: *Carballas Chas,Sergio*
APELIDOS E NOME:

DNI: [redacted] **Fecha de Solicitud:** Feb2018
DNI: *Fecha de Solicitude:*

Alumno de esta escuela en la titulación de Grado en Ingeniería en Electrónica Industrial y Automática, se le comunica que la Comisión de Proyectos ha decidido asignarle el siguiente Trabajo Fin de Grado:

O alumno de esta escola na titulación de Grado en Enxeñería en Electrónica Industrial e Automática, comunícaselle que a Comisión de Proxectos ha decidido asignarlle o seguinte Traballo Fin de Grado:

Título T.F.G:Creación de un protocolo de comunicación entre un PC y el puerto USB de un microcontrolador, y su integración en MatLab

Número TFG: *770G01A145*

TUTOR:(Titor) *Casteleiro Roca,José Luis*

COTUTOR/CODIRECTOR: *Esteban Jove Pérez*

La descripción y objetivos del Trabajo son los que figuran en el reverso de este documento*:*

A descrición e obxectivos do proxecto son os que figuran no reverso deste documento.

Ferrol a Miercoles, 22 de Agosto del 2018

Retirei o meu Traballo Fin de Grado o dia _____ de _____ do ano _____

Fdo: Carballas Chas,Sergio

DESCRIPCIÓN Y OBJETIVO:OBJETO:
Este Trabajo Final de Grado abordará la implementación de un protocolo completo de comunicación con un microcontrolador. Se estudiará inicialmente el protocolo base USB, y se analizará la forma de implementarlo en un microcontrolador que disponga de puerto USB (de la casa Microchip). Posteriormente, se implementará el protocolo necesario para poder mandar y recibir datos del microcontrolador a través del programa MatLab, para poder adquirir y generar señales a modo de tarjeta de adquisición de datos.

ALCANCE:
- Estudio del protocolo USB.
- Análisis de las opciones disponibles para comunicar un microcontrolador con un PC.
- Estudio del funcionamiento de los puertos USB en los microcontroladores de la casa Microchip.
- Creación de un protocolo para la comunicación efectiva entre el PC y el microcontrolador a través del USB.
- Creación de los programas de MatLab necesarios para implementar el protocolo y poder adquirir/generar señales desde el programa.
- Comparación de la velocidad de conexión por medio del protocolo creado.

3.2 Anexo II: Cálculos

3.2.1 Período PWM

El período de la PWM se calcula con la ecuación que proporciona el fabricante en su datasheet.

$$periodo = [(PR2) + 1] * 4 * Tosc * (TMR2\,\mathrm{Pr}escaler) \quad \textbf{(3.2.1.1)}$$

Dónde:

- PR2: es el período del *Timer 2* que será igual a 255 para conseguir que la frecuencia de la PWM sea máxima.
- Tosc: es el período del oscilador interno del PIC. Siendo su frecuencia de 16MHz, su periodo será de 1/16MHz.
- TMR2 *prescaler*: prescaler del Timer 2, para que el período de la PWM sea máximo, su valor debe ser el mínimo posible: 1/1.

Se sustituyen los valores anteriores en la ecuación (1.1.2).

$$periodo = (255 + 1) * 4 * \frac{1}{16\,MHz} * \frac{1}{1} \quad \textbf{(3.2.1.2)}$$

El período de la PWM será de 0.64μs lo que supone una frecuencia de aproximadamente 16KHz (ver ecuación (1.1.3)).

$$frecuencia\_PWM = \frac{1}{0.64\,\mu s} = 15,625\,KHz \quad \textbf{(3.2.1.3)}$$

3.2.2 Ciclo PWM

El ciclo de trabajo de la PWM variará en función del valor que el usuario cargue en los registros PWMxDCH y PWMxDCL, pero lo que se ha garantizado es que la resolución de la PWM sea de 10 bits, para ello empleando la ecuación (1.2.1) el

período debe ser 255 para que al calcular el ciclo de trabajo pueda alcanzar el máximo valor, ver ecuación (1.2.2).

$$Dutycicle = \frac{(PWMxDCH : PWMxDCL <7:6>)}{4(PR\,2+1)} \quad \textbf{(3.2.1.4)}$$

$$Dutycicle = \frac{(1024)}{4(255+1)} = 1 \quad \textbf{(3.2.1.5)}$$

3.2.3 Filtro R-C

Los seis filtros que se emplean en la DAQ tienen todos el mismo valor, y están calculados en función de la frecuencia de salida de la PWM, 16KHz (se supone que a la entrada se conectará un microcontrolador de la misma frecuencia a la que trabaja el PIC).

El filtro es un R-C paso bajo (ver imagen 1.3.1). Permitirá que en las salidas asociadas a los pines PWM se obtenga una tensión continua, con un pequeño rizado, y eliminará el ruido que pueda haber en las entradas de la DAQ.

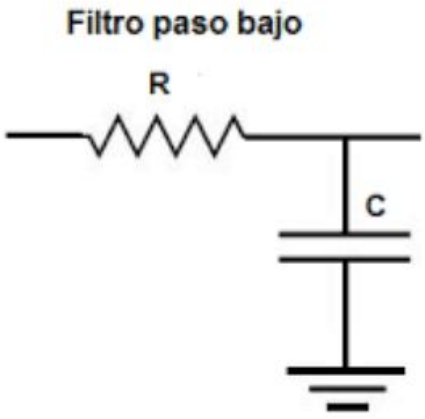


Imagen 3.2.3.1 – Filtro RC

En función de la frecuencia de la tensión de entrada, se ha seleccionado una resistencia de 3KΩ y un condensador de 47KpF. Por lo que de acuerdo con la ecuación (1.3.1) la frecuencia de corte del filtro será de 1.31KHz.

$$fc = \frac{1}{2\pi RC} = 1.31\ KHz \qquad \textbf{(3.2.3.1)}$$

3.3 Anexo III: códigos de programación

3.3.1 Funciones Matlab

3.1.1.1 DAQ_Start():

```
function [] = DAQ_Start()

%Función DAQ_Start

%   Inicia la comuncación con la tarjeta, no devuelve ningún valor. Antes de
    iniciar que la tarjeta está correctamente conectada al puerto USB

%   See also DAQ_Stop, DAQ_Write_2, DAQ_Read_2


global handle                           %Puntero que almacena la conexión con

global debug                            %DAQ

global vendorID                         %Vendor ID

global productID                        %Product ID

global flag_permiso

global hidslib                          %Variable para almacenar la librería

global endpoint_out                     %Tamaño registro de salida

global buffer_in                        %Tamaño registro de entrada

global pointer_readbuffer               %Puntero para la lectura

global ao1                              % Valores salida

global ao2

global ao3

global start_error                      %Variable doble inicio

global sleep                            %Variable bajo consumo
```

```
vendorID = 1240;

productID = 7;

endpoint_in = 4;

flag_name=0;                %control de errores

endpoint_out= 6;

ao1=0;

ao2=0;                      %inicializa las salidas

ao3=0;


if start_error==1    %control de errores para evitar doble inicio

    inicia=0;

else

    inicia=1;

end


if  ~libisloaded('libreriahid')  %comprueba librería

    loadlibrary(hidslib,'hidapi.h', 'alias','libreriahid');

    loadlibrary(hidslib,@hidapi64_proto,'alias','libreriahid');

End

hidslib = 'libreriahid'; %carga librería para pasarla al resto de funciones

flag_loadlibrary=libisloaded('libreriahid');


if (inicia==1 && flag_loadlibrary) %comprueba librería y doble inicio

    start_error = 1;  %no se permite otro inicio antes de DAQ_Stop


    %Inicio de la comunicación

    popen = libpointer('uint16Ptr');   %puntero para retorno valor de hid_open
```

```
    [handle,value] =
     calllib(hidslib,'hid_open',uint16(vendorID),uint16(productID),popen);

    %Comprobar que el dispositivo se ha conectado correctamente

     buffer = zeros(1,17); %registro de almacenamiento del nombre de la DAQ

     pbuffer = libpointer('uint16Ptr', uint16(buffer)); %puntero valor nombre

    [res,h] = calllib(hidslib,
     'hid_get_product_string',handle,pbuffer,uint64(length(buffer)));

    str = sprintf('%s',char(pbuffer.Value)); %cambio formato cadena

    str1= cellstr('Generic HID DAQ7 ');

    b =cellstr(str); %convierto en un cell array

    flag_name = strcmp(b,str1); %comparación

    buffer_in=zeros(1,endpoint_in);

    pointer_readbuffer = libpointer('uint8Ptr', uint8(buffer_in));

    %las dos innstrucciones de arriba crean el puntero del registro IN



    %Cancela modo de bajo consumo

    sleep=1;

    write = [0  sleep ao1  ao2  ao3 0];   %cargo vector escritura

    write(end+(endpoint_out-length(write))) = 0; %relleno de ceros el report

    pointer_writebuffer = libpointer('uint8Ptr', uint8(write)); %puntero

    [res,h] =
calllib(hidslib,'hid_write',handle,pointer_writebuffer,uint64(length(writ
e))); %realizo escritura con hid_write

    if (flag_name==1)

        flag_permiso = 1;     %si ha pasado el control del nombre permito

    else                       %usar las demás funciones

        flag_permiso = 0;

    end

else

    fprintf('Ya se ha establecido la comunicacion con la tarjeta\n')
```

```
End
```

3.1.1.2 DAQ_Write_2():

```
function [] = DAQ_Write_2(a,b,c)

%Función DAQ_Write_2

%   Envía datos a la tarjeta, no devuelve ningún valor. La relación de las
    salidas y las valores de entrada a la función sigue la siguiente relación

    DAQ_Write_2(AO1,AO2,AO3)

%   See also DAQ_Stop, DAQ_Start, DAQ_Read_2



global handle               %conexión con la DAQ

global flag_permiso         %permiso inicio

global hidslib              %librería

global endpoint_out         %registro almacenamiento salida

global ao1                  %variables salida

global ao2

global ao3

aux_ao1=0;                  %variables auxiliares salida control errores

aux_ao2=0;

aux_ao3=0;

output_channel=0;           %número de salidas


if (flag_permiso==1)        %comprueba inicio DAQ_Start

   switch nargin

      case 0

         fprintf('Error, Introduzca al menos un valor');

         output_channel=0; %error no se ha seleccionado salida

      case 1

         aux_ao1=(a*255)/100; %salida ao1 convierto para enviar
```

```
            output_channel=1;
        case 2
            aux_ao1=(a*255)/100;   %salida ao1 convierto para enviar
            aux_ao2=(b*255)/100;   %salida ao2 convierto para enviar
            output_channel=2;
        case 3
            aux_ao1=(a*255)/100;    %salida ao1 convierto para enviar
            aux_ao2=(b*255)/100;    %salida ao2 convierto para enviar
            aux_ao3=(c*255)/100;    %salida ao3 convierto para enviar
            output_channel=3;
        otherwise
            fprintf('Error, No se ha podido encontrar el canal
seleccionado\n');
            output_channel=0;
    end
    if aux_ao1>0 || aux_ao1<255    %control de errores no pueden ser valores
        ao1=aux_ao1;               %más altos de 255 (100%)
    end
    if aux_ao2>0 || aux_ao2<255
        ao2=aux_ao2;
    end
    if aux_ao3>0 || aux_ao3<255
        ao3=aux_ao3;
    end

    write = [0  0 ao1  ao2  ao3 output_channel];   %carga vector escritura
    write(end+(endpoint_out-length(write))) = 0; %rellena de ceros
    pointer_writebuffer = libpointer('uint8Ptr', uint8(write));
```

```
    [res,h]
     calllib(hidslib,'hid_write',handle,pointer_writebuffer,uint64(length(
     write)));

else

    fprintf('Error al escribir, inicie primero la comunicacion con la
tarjeta\n'); %mensaje de error si no se ha iniciado DAQ_Start

end

end
```

3.1.1.3 DAQ_Read_2():

```
function [ai1,ai2,ai3,ai4] = DAQ_Read_2()

%Función DAQ_Read_2

%   Lectura de datos de la tarjeta, no devuelve 4 valores. La relación de las
    entradas de la tarjeta y las valores de salida de la función es:

   DAQ_Read_2(AI1,AI2,AI3,AI4)

%   See also DAQ_Stop, DAQ_Start, DAQ_Write_2


global handle               %conexión con la DAQ

global flag_permiso         %permiso inicio

global hidslib              %librería

global buffer_in            %registro de entrada tamaño

global pointer_readbuffer   %puntero que devuelve las variables de entrada

                            %de hid read

if (flag_permiso==1)

    [res,h] =
     calllib(hidslib,'hid_read',handle,pointer_readbuffer,uint64(length(bu
     ffer_in)));

    rmsg = pointer_readbuffer.Value;  %almaceno valores puntero

else

    fprintf('Error al leer, inicie primero la comunicacion con la tarjeta\n');

end



switch nargout

    case 0

      fprintf('Error, No se ha podido encontrar el canal seleccionado\n');

    case 1

      ai1 = round((double(rmsg(1,1))*100/255);  %entrada AI1 adapto escala

    case 2
```

```
      ai1 = round((double(rmsg(1,1))*100/255);  %entrada AI1 adapto escala
      ai2 = round((double(rmsg(1,2))*100/255);  %entrada AI2 adapto escala
    case 3
      ai1 = round((double(rmsg(1,1))*100/255);  %entrada AI1 adapto escala
      ai2 = round((double(rmsg(1,2))*100/255); %entrada AI2 adapto escala
      ai3 = round((double(rmsg(1,3))*100/255); %entrada AI3 adapto escala
    case 4
      ai1 = round((double(rmsg(1,1))*100/255); %entrada AI1 adapto escala
      ai2 = round((double(rmsg(1,2))*100/255); %entrada AI1 adapto escala
      ai3 = round((double(rmsg(1,3))*100/255); %entrada AI3 adapto escala
      ai4 = round((double(rmsg(1,4))*100/255); %entrada AI4 adapto escala
    otherwise
       fprintf('Error, No se ha podido encontrar el canal seleccionado\n');
end
```

3.1.1.4 DAQ_Stop():

```
%Función DAQ_Stop

%   Finaliza la comuncación con la tarjeta

%   See also DAQ_Start, DAQ_Write_2, DAQ_Read_2


global handle               %conexión con la DAQ

global flag_permiso         %permiso inicio

global hidslib              %librería

global endpoint_out         %almacenamiento salida

global sleep


if (flag_permiso==1)

    sleep = 1;    %reseteo salida

    write = [0 sleep 0 0 0 3];      %vector escritura

    write(end+(endpoint_out-length(write))) = 0;

    pbuffer = libpointer('uint8Ptr', uint8(write)); %puntero escritura

    [res,h] =
    calllib(hidslib,'hid_write',handle,pbuffer,uint64(length(write)));

    %desactivo bajo consumo

    sleep = 0;

    write = [0 sleep 0 0 0 3];

    write(end+(endpoint_out-length(write))) = 0;

    pbuffer = libpointer('uint8Ptr', uint8(write));

    [res,h] =
    calllib(hidslib,'hid_write',handle,pbuffer,uint64(length(write)));



    calllib(hidslib,'hid_close',handle); %cierro la comunicación

    flag_permiso = 0;
```

```
else

   fprintf('Error al cerrar la comunicacion, inicie primero la comunicacion
con la tarjeta\n');

end



clear global   %reseteo las variables globales

end
```

3.3.2 Códigos PIC

3.1.1.5 DAC.h

```
#ifndef DAC_H

#define DAC_H

#include <xc.h> // processor files - each processor file is guarded.

#include "dac.h"


void init_dac (int volt);  // modifica valor salida dac

void config_dac (void);    // configura DAC


#endif /* XC_HEADER_TEMPLATE_H */
```

3.1.1.6 DAC.c

```
#include <xc.h>

void init_dac (int volt){

if (volt>=0 && volt<31){ // control de errores

    DACCON1 = volt;    //00000110 DACR solo 5 bits maximo valor 32

}

}

void config_dac (void){

   DACCON0 = 0x90;         //10010000 DACEN enable DACOUT2 enable en el pin RC3

}
```

3.1.1.7 ADC.h

```
#ifndef ADC_H

#define ADC_H

#include <xc.h> // include processor files - each processor file is guarded.


void adc_initialize(void);  // configura adc

void adc_start(void);       // inicia adc

void adc_stop(void);        // para el convertidor

int  adc_get_conversion(int channel);   // devuelve valor de conversión en
                                           función del canal seleccionado


#endif /* XC_HEADER_TEMPLATE_H */
```

3.1.1.8 Adc_template.c

```
#include <xc.h>
#include "adc.h"
#include <stdbool.h>

#define _XTAL_FREQ 16000000 // para la instrucción delay
#define ACQ_US_DELAY 5
void adc_initialize(void){

    //ADCON1 = 0xD0;              // 10110000
    ADCON1bits.ADPREF=00;          //Ref pos=VDD
    ADCON1bits.ADCS=0b101;         //Fuente de reloj oscilador int FOSC/4.
    ADCON1bits.ADFM=1;             //justificamos a la derecha

    ADCON2 = 0x00;                // auto_trigger desahabilitadp;
    ADRESL = 0x00;                // ADRESL resteado;
    ADRESH = 0x00;                // ADRESH resteado;
    PIE1bits.ADIE = 0;             // ADC interrupt enable.
}

void adc_start(void){
    ADCON0bits.ADON = 1;           // inicia ADC
}

void adc_stop(void){
    ADCON0bits.ADON = 1;           // para ADC
```

```
}


int adc_get_conversion(int channel){

    int adc_value;

    ADCON0bits.CHS= channel;        // selección del canal

    __delay_us (ACQ_US_DELAY);      // retardo para el convertido

    ADCON0bits.GO_nDONE = 1;        // inicia conversion

    while( ADCON0bits.GO_nDONE);    // esperamos a que termine la conversión,
el micro limpia el flag

    adc_value =(ADRESH << 8) + ADRESL; //resteado valor de 10 bits

    return adc_value;               // Conversion finalizada, devuelve valor

    //return ((adc_result_t)((ADRESH << 8) + ADRESL));

}
```

3.1.1.9 PWM.h:

```
#ifndef pwm_H
#define pwm_H
#include <xc.h> // include processor files - each processor file is guarded.
#include <stdint.h>

//Prototipos pwm1 RC5
void pwm1_initialize(void);              //configura PWM1
void load_duty_pwm1(uint16_t dutyValue);//actualiza salida PWM1

//Prototipos pwm2 RA5
void pwm2_initialize(void);              //configura PWM2
void load_duty_pwm2(uint16_t dutyValue);//actualiza salida PWM2
#endif /* XC_HEADER_TEMPLATE_H */
```

3.1.1.10 PWM_template.c

```
#include <xc.h>

#include "pwm.h"


void pwm1_initialize(void){

    LATCbits.LATC5  = 0;            //reseta

    TRISCbits.TRISC5 = 1;           //00010000 ponemos RC5 como entrada


    PWM1CON = 0x00;                //limpiamos registros

    PWM1DCH = 0x00;

    PWM1DCL = 0x00;


    TRISCbits.TRISC5 = 0;           // 00000000 ponemos RA4 y RA5 como salidas

    PWM1CON = 0xC0;                //1100 modulo PWM habilitado bit salida del
PWM habilitado

}


void load_duty_pwm1(uint16_t dutyValue){

   PWM1DCH = (dutyValue & 0x03FC)>>2;            // PWM1DCH 127;

   PWM1DCL = (dutyValue & 0x0003)<<6;            // PWM1DCL 3 00110000

}


void pwm2_initialize(void){

    LATAbits.LATA5 = 0;

    TRISAbits.TRISA5 = 1;           // 00110000 ponemos RA5 como entrada

    WPUAbits.WPUA5 = 0;            //desactivamos la resistencia de pull-up

    OPTION_REGbits.nWPUEN = 1;      //todas las resistencias de pull-up estan
desahabilitadas
```

```
    APFCONbits.P2SEL = 1;          // Seleccionamos RA5 como salida PWM si
estuviese a 0 la salida sería RC3



    PWM2CON = 0x00;

    PWM2DCH = 0x00;               //limpiamos registros

    PWM2DCL = 0x00;



    TRISAbits.TRISA5 = 0;          // 00000000 ponemos RA4 y RA5 como salidas

    PWM2CON = 0xC0;               //1100 modulo PWM habilitado bit salida del
PWM habilitado

}



void load_duty_pwm2(uint16_t dutyValue){

   PWM2DCH = (dutyValue & 0x03FC)>>2;           // PWM1DCH 127;

   PWM2DCL = (dutyValue & 0x0003)<<6;           // PWM1DCL 3 00110000

}
```

3.1.1.11 Timer2.h:

```
#ifndef timer2_H

#define timer2_H


#include <xc.h> // include processor files - each processor file is guarded.


void timer2_initialize(void); //configuramos timer

void timer2_start(void); //iniciamos timer

void timer2_stop(void); //paramos timer

#endif /* XC_HEADER_TEMPLATE_H */
```

3.1.1.12 Timer2_template.c:

```
#include <xc.h>
#include "timer2.h"

void timer2_initialize (void){
    PR2 = 0xFF;                    // cargamos PERIODO PWM
    TMR2 = 0x00;                    // limpiamos
    T2CON = 0b00000100;             //  seleccion prescaler 1 (10) y timer 2 off
}

void timer2_start (void){
    T2CONbits.TMR2ON = 1;           // inicio temporización
}

void timer2_stop (void){
    T2CONbits.TMR2ON = 0;           // paro temporización
}
```

3.1.1.13 Timer1.h:

```
#ifndef timer1_H

#define timer1_H


#include <xc.h> // include processor files - each processor file is guarded.

#include <stdint.h>

void timer1_initialize(void); // configuramos timer

void timer1_start(void); //inicio timer

void timer1_stop(void);  //paro timer

void timer1_reload(uint16_t load_value); //recargo timer

#endif /* XC_HEADER_TEMPLATE_H */
```

3.1.1.14 Timer1_template.c:

```
#include <xc.h>
#include "timer1.h"

void timer1_initialize(void){
    //Timer1 inicializacion
    TMR1H = 0x00; //TMR1H resetea;
    TMR1L = 0x00; //TMR1L resetea;

    INTCON = 0xC0; //11000000 habilito interrupciones globales
    PIR1bits.TMR1IF = 0; //limpaimos
    PIE1bits.TMR1IE = 1; // TMR1 interrupción enable.

    T1CON = 0x40; // preescaler 1/4
}
void timer1_start(void){
    T1CONbits.TMR1ON=1; // inicio timer
}
void timer1_stop(void){
    T1CONbits.TMR1ON=0; // paro timer
}
void timer1_reload(uint16_t load_value){
    TMR1H = (load_value >> 8);
    TMR1L = load_value;
    PIR1bits.TMR1IF = 0; // limpio TMR1 interrupt flag
}
```

3.1.1.15 Usb_config.h:

```
#ifndef USBCFG_H

#define USBCFG_H


/** DEFINITIONS ****************************************************/

#define USB_EP0_BUFF_SIZE        8

#define USB_MAX_NUM_INT          1

#define USB_MAX_EP_NUMBER        1


//Device descriptor

#define USB_USER_DEVICE_DESCRIPTOR &device_dsc

#define USB_USER_DEVICE_DESCRIPTOR_INCLUDE extern ROM USB_DEVICE_DESCRIPTOR
device_dsc


//Configuration descriptors

#define USB_USER_CONFIG_DESCRIPTOR USB_CD_Ptr

#define USB_USER_CONFIG_DESCRIPTOR_INCLUDE extern ROM BYTE *ROM USB_CD_Ptr[]


/** DEFINITIONS ****************************************************/

#define USB_PING_PONG_MODE USB_PING_PONG__FULL_PING_PONG

#define USB_INTERRUPT

#define USB_PULLUP_OPTION USB_PULLUP_ENABLE

#define USB_TRANSCEIVER_OPTION USB_INTERNAL_TRANSCEIVER

#define USB_SPEED_OPTION USB_FULL_SPEED

#define USB_ENABLE_STATUS_STAGE_TIMEOUTS

#define USB_STATUS_STAGE_TIMEOUT     (BYTE)45

#define USB_SUPPORT_DEVICE
```

```
#define USB_NUM_STRING_DESCRIPTORS 3
#define USB_ENABLE_ALL_HANDLERS

/** DEVICE CLASS USAGE *********************************************/
#define USB_USE_HID
#define MY_VID 0x04D8
#define MY_PID 0x0007

/** ENDPOINTS ALLOCATION ********************************************/
/* HID */
#define HID_INTF_ID            0x00
#define HID_EP                  1
#define HID_NUM_OF_DSC          1
#define HID_RPT01_SIZE          47

// Start Generic HID specific
#define HID_INPUT_REPORT_BYTES    4
#define HID_OUTPUT_REPORT_BYTES   5
#define HID_FEATURE_REPORT_BYTES  2

#define USER_GET_REPORT_HANDLER User_Get_Report_Handler
#define USER_SET_REPORT_HANDLER User_Set_Report_Handler
// End Generic HID specific
/** DEFINITIONS ****************************************************/
#endif //USBCFG_H
```

3.1.1.16 Main.c:

```
#ifndef MAIN_C

#define MAIN_C


/** INCLUDES *******************************************************/

#include "usb.h"

#include "HardwareProfile.h"

#include "usb_function_hid.h"

#include "Led.h"

#include "buttons.h"

#include <stdio.h>

#include "adc.h"

#include "pwm.h"

#include "timer2.h"

#include "dac.h"


/** CONFIGURATION **************************************************/

    #pragma config FOSC = INTOSC       //Intrenal RC oscillator Enabled

    #pragma config WDTE = OFF          //Watch Dog Timer Disabled.

    #pragma config PWRTE= ON           //Power Up Timer Enabled

    #pragma config MCLRE = ON          //MCLR pin function Disabled

    #pragma config CP = OFF            //Flash Program Memory Code Protection

    #pragma config BOREN = ON          //Brown-out Reset disabled

    #pragma config CLKOUTEN = OFF      //CLKOUT function is disabled.

    #pragma config IESO = OFF          //Internal/External Switchover Mode is

    #pragma config FCMEN = OFF         //Fail-Safe Clock Monitor is disabled
```

```
    #pragma config WRT = OFF           //Flash Memory Self-Write Protection
Off

    #pragma config CPUDIV = CLKDIV3     //CPU system clock divided by 3

    #pragma config USBLSCLK = 16MHz     //System clock expects 16 MHz, FS/LS
USB CLKENs divide-by is set to 8

    #pragma config PLLMULT = 3x         //3x Output Frequency Selected

    #pragma config PLLEN = 1            //3x or 4x PLL Enabled

    #pragma config STVREN = OFF        //Stack Overflow or Underflow will not
cause

    #pragma config BORV = HI           //Brown-out Reset Voltage (Vbor), high
trip

    #pragma config LPBOR = OFF          //Low-Power BOR is disabled

    #pragma config LVP = OFF           //High-voltage on MCLR/VPP must be used
for programming



/** VARIABLES ******************************************************/

#define RX_DATA_BUFFER_ADDRESS @0x2050

#define TX_DATA_BUFFER_ADDRESS @0x20A0

unsigned char ReceivedDataBuffer[64] RX_DATA_BUFFER_ADDRESS;

unsigned char ToSendDataBuffer[64] TX_DATA_BUFFER_ADDRESS;



USB_HANDLE USBOutHandle = 0;    //USB handle.  Must be initialized to 0 at
startup.

USB_HANDLE USBInHandle = 0;     //USB handle.  Must be initialized to 0 at
startup.



volatile unsigned char hid_report_out[HID_INT_OUT_EP_SIZE];

volatile unsigned char hid_report_in[HID_INT_IN_EP_SIZE];

unsigned char hid_report_feature[USB_EP0_BUFF_SIZE];
```

```
int adc=0, output_channel,
timerVal,pwm_cicle_ao1,pwm_cicle_ao2,pwm_cicle_ao3;

int volt_out,volt_out1,volt_out2, ad_channel, an3=0, an4=0,an5=0,
an6=0,count;

float volt;

int flag_input,sleep;



/** PRIVATE PROTOTYPES *********************************************/

static void InitializeSystem(void);

void ProcessIO(void);

void UserInit(void);

void USBCBSendResume(void);

void ProcessUserInput(void);

void ProcessUserOutput(void);

BYTE ReportSupported(void);

void USBCBWakeFromSuspend(void);

void USBCBSuspend(void);




#define OUTPUT_REPORT &hid_report_out[0]

#define FEATURE_REPORT &hid_report_feature[0]


/** VECTOR REMAPPING ***********************************************/

void interrupt ISRCode()  //These are your actual interrupt handling routines.

{

       timer1_reload(3200);

       flag_input= 1;

        USBDeviceTasks();

}
```

```
/** DECLARATIONS ***************************************************/
 * Function:        void main(void)
 * PreCondition:    None
 * Input:           None
 * Output:          None
 * Side Effects:    None
 * Overview:        Main program entry point.
 * Note:            None
 *******************************************************************/
int main(void)
{
    InitializeSystem();
    #if defined(USB_INTERRUPT)
        USBDeviceAttach();
    #endif
    while(1)
    {
        if(sleep==1){
              LED_On();
              ProcessUserInput();
              ProcessUserOutput();
        }else {
              LED_Off();
             timer2_stop();
             timer1_stop();
              adc_stop();
```

```
        }
    }//end while
}//end main


/********************************************************************
 * Function:        static void InitializeSystem(void)
 * PreCondition:    None
 * Input:           None
 * Output:          None
 * Side Effects:    None
 * Overview:        InitializeSystem is a centralize initialization
 *                routine. All required USB initialization routines
 *                are called from here.
 *                User application initialization routine should
 *                also be called from here.
 * Note:            None
 *******************************************************************/
static void InitializeSystem(void)
{
    OSCCON = 0x7C;   // PLL enabled, 3x, 16MHz internal osc, SCS external
    OSCCONbits.SPLLMULT = 1;   // 1=3x, 0=4x
    ACTCON = 0x90;   // Clock recovery on, Clock Recovery enabled; SOF packet


    LATC = 0x00;
    LATA =  0x00;
    PORTA = 0x00;                   // limpio pines del puerto A
    PORTC = 0x00;                   // limpio pines del puerto C
    ANSELCbits.ANSC0 = 1;            // RC0 entrada analógica ADAN4
```

```
    TRISCbits.TRISC0 = 1;

    ANSELAbits.ANSA4 = 1;           // RA4 entrada analógica ADAN3

    TRISAbits.TRISA4 = 1;

    ANSELCbits.ANSC1 = 1;           // RC1 entrada analógica ADAN5

    TRISCbits.TRISC1 = 1;

    ANSELCbits.ANSC2 = 1;           // RC2 entrada analógica ADAN6

    TRISCbits.TRISC2 = 1;

    TRISCbits.TRISC3 = 0;           // RC3 como salida

    ANSELCbits.ANSC3 = 0;           // RC3 como analog pin DAC


    timer1_initialize();          //se llama a todas las funciones de inicio

    timer1_reload(3200);

    timer1_start();

    adc_initialize();

    pwm1_initialize();

    pwm2_initialize();

    timer2_initialize();

    config_dac();

    adc_start();


    USBOutHandle = 0;        // incializo Handles

    USBInHandle = 0;


    USBDeviceInit();  //usb_device.c.  Initializes USB module SFRs and
firmware

                      //variables to known states.

}//end InitializeSystem
```

```
/********************************************************************
 * Function:       void ProcessIO(void)
 * PreCondition:    None
 * Input:          None
 * Output:         None
 * Side Effects:    None
 * Overview:       This function is a place holder for other user
 *                routines. It is a mixture of both USB and
 *                non-USB tasks.
          Sends a received Input report back
 *                to the host in an Output report.
 *                Both directions use interrupt transfers.
 * Note:           None
 *******************************************************************/

void ProcessUserInput(){


   int j;
    for (j=3; j<=6; j++){
      volt = adc_get_conversion(j);
      volt_out = volt>>2;
          switch (j){
          case 3:
              an3=volt_out;
              break;
          case 4:
              an4=volt_out;
              break;
```

```
            case 5:
                an5=volt_out;
                break;
            case 6:
                an6=volt_out;
                break;
            }
        }
}

void ProcessUserOutput(){
    if (pwm_cicle_ao1 >= 0 && pwm_cicle_ao1 <= 255) {
        cicle_ao1 = pwm_cicle_ao1<<2;
    }
    if (pwm_cicle_ao2 >= 0 && pwm_cicle_ao2 <= 255) {
        cicle_ao2 = pwm_cicle_ao2<<2;
    }
    if (pwm_cicle_ao3 >= 0 && pwm_cicle_ao3 <= 255) {
        cicle_ao3 = pwm_cicle_ao3>>3;
    }

switch (output_channel){
        case 1:
            load_duty_pwm1(cicle_ao1);
            break;
        case 2:
            load_duty_pwm1(cicle_ao1);
            load_duty_pwm2(cicle_ao2);
```

```
            break;
        case 3:
            load_duty_pwm1(cicle_ao1);
            load_duty_pwm2(cicle_ao2);
            init_dac(cicle_ao3);
            break;
        default:
 break;
 }
}

void ProcessIO(void)
{
 if((USBDeviceState < CONFIGURED_STATE)||(USBSuspendControl==1)) return;

WORD len = 0;

if (flag_input==1 && !HIDTxHandleBusy(USBInHandle)){

          USBInHandle=HIDTxPacket(HID_EP,(BYTE*)&ToSendDataBuffer[0],HID_I
          NPUT_REPORT_BYTES); //puntero para cargar valores en el endpoint
        ToSendDataBuffer[0]= an3;
        ToSendDataBuffer[1]= an4;
        ToSendDataBuffer[2]= an5;
        ToSendDataBuffer[3]= an6;
        flag_input=0;
}

if(!HIDRxHandleBusy(USBOutHandle))
```

```
{
    len = USBHandleGetLength(USBOutHandle); //comprueba estado bus
    if (len > 0)
    {//puntero para cargar valores en el endpoint
        USBOutHandle = HIDRxPacket(HID_EP,(BYTE*)&ReceivedDataBuffer,64);
        sleep = ReceivedDataBuffer[0];
        pwm_cicle_ao1  = ReceivedDataBuffer[1];
        pwm_cicle_ao2  = ReceivedDataBuffer[2];
        pwm_cicle_ao3  = ReceivedDataBuffer[3];
        output_channel  = ReceivedDataBuffer[4];
    }
}
}//end ProcessIO


// ************** USB Callback Functions
*****************************************************************

/*********************************************************************
******

 * Function:       void USBCBSuspend(void)
 * PreCondition:    None
 * Input:          None
 * Output:         None
 * Side Effects:    None
 * Overview:       Call back that is invoked when a USB suspend is detected
 * Note:           None

*********************************************************************
****/
void USBCBSuspend(void)
{
```

```
    OSCCON = 0x50; //switching to 62.5 KHz

    LED_Off();

 }


/*********************************************************************
******

 * Function:        void USBCBWakeFromSuspend(void)

 * PreCondition:    None

 * Input:           None

 * Output:          None

 * Side Effects:    None

 *

 * Overview:        The host may put USB peripheral devices in low power suspend
mode (by "sending" 3+ms of idle).  Once in suspend mode, the host may wake
the device back up by sending non idle state signalling.This call back is
invoked when a wakeup from USB suspend is detected.

 * Note:            None


*********************************************************************
****/

void USBCBWakeFromSuspend(void)

{

    OSCCON = 0x7C;   // PLL enabled, 3x, 16MHz internal osc, SCS external

    OSCCONbits.SPLLMULT = 1;   // 1=3x, 0=4x

    while(OSCSTATbits.PLLRDY != 1);   //Wait for PLL lock

    ACTCON = 0x90;   // Clock recovery on, Clock Recovery enabled; SOF packet

    timer2_start();

    adc_start(); //reinicia sistema

}


/*********************************************************************
```

```
 * Function:        void USBCB_SOF_Handler(void)
 * PreCondition:    None
 * Input:           None
 * Output:          None
 * Side Effects:    None
 * Overview:        The USB host sends out a SOF packet to full-speed
 *                  devices every 1 ms. This interrupt may be useful
 *                  for isochronous pipes. End designers should
 *                  implement callback routine as necessary.
 * Note:            None
 *******************************************************************/
void USBCB_SOF_Handler(void)
{
    // No need to clear UIRbits.SOFIF to 0 here.
    // Callback caller is already doing that.
}
/*******************************************************************
 * Function:        void USBCBErrorHandler(void)
 * PreCondition:    None
 * Input:           None
 * Output:          None
 * Side Effects:    None
 * Overview:        The purpose of this callback is mainly for
 *                  debugging during development. Check UEIR to see
 *                  which error causes the interrupt.
 * Note:            None
 *******************************************************************/
void USBCBErrorHandler(void)
```

```
{

}

void USBCBCheckOtherReq(void)

{

    USBCheckHIDRequest();

}//end

/******************************************************************

 * Function:        void USBCBStdSetDscHandler(void)

 * PreCondition:    None

 * Input:           None

 * Output:          None

 * Side Effects:    None

 * Overview: The USBCBStdSetDscHandler() callback function is called when a
SETUP, bRequest: SET_DESCRIPTOr request arrives.  Typically SET_DESCRIPTOR
requests are not used in most applications, and it isoptional to support this
type of request.

 * Note:            None

 ******************************************************************/

void USBCBStdSetDscHandler(void)

{

    // Must claim session ownership if supporting this request

}//end



/******************************************************************

 * Function:        void USBCBInitEP(void)

 * PreCondition:    Non

 * Input:           None

 * Output:          None

 * Side Effects:    None
```

```
 * Overview:        This function is called when the device becomes

 * initialized, which occurs after the host sends a SET_CONFIGURATION (wValue
    not = 0) request.  This callback function should initialize the
    endpointsfor the device's usage according to the curren configuration.

 * Note:            None

 ********************************************************************/

void USBCBInitEP(void)

{

 //enable the HID endpoint


USBEnableEndpoint(HID_EP,USB_IN_ENABLED|USB_OUT_ENABLED|USB_HANDSHAKE_ENA
BLED|USB_DISALLOW_SETUP);

//Re-arm the OUT endpoint for the next packet

 USBOutHandle = HIDRxPacket(HID_EP,(BYTE*)&ReceivedDataBuffer,64);

}



/********************************************************************

 * Function:        void USBCBSendResume(void)

 * PreCondition:    None

 * Input:           None

 * Output:          None

 * Side Effects:    None

 * Overview:        The USB specifications allow some types of USB peripheral
devices to wake up host PC (such   as if it is in a low power suspend to RAM
state). This can be a very useful feature in some USB applications, such as
an Infrared remote control receiver.  If a user presses the "power" button
on a remote control, it is nice that the IR receiver can detect this signalling,
and thensend a USB "command" to the PC to wake up. The USBCBSendResume()
"callback" function is used to send this special USB signalling which wakes
up the PC.  This function may be called by application firmware to wake up
the PC.

 ********************************************************************/

void USBCBSendResume(void)

{
```

```
static WORD delay_count;

//First verify that the host has armed us to perform remote wakeup.
//It does this by sending a SET_FEATURE request to enable remote wakeup,
//usually just before the host goes to standby mode (note: it will only
//send this SET_FEATURE request if the configuration descriptor declares
//the device as remote wakeup capable, AND, if the feature is enabled
//on the host (ex: on Windows based hosts, in the device manager
//properties page for the USB device, power management tab, the
//"Allow this device to bring the computer out of standby." checkbox
//should be checked).
if(USBGetRemoteWakeupStatus() == TRUE)
{
    //Verify that the USB bus is in fact suspended, before we send
    //remote wakeup signalling.
    if(USBIsBusSuspended() == TRUE)
    {
        USBMaskInterrupts();
        //Clock switch to settings consistent with normal USB operation.
        USBCBWakeFromSuspend();
        USBSuspendControl = 0;
        USBBusIsSuspended = FALSE;
         //So we don't execute this code again,
        //until a new suspend condition is detected.
        //Section 7.1.7.7 of the USB 2.0 specifications indicates a USB
        //device must continuously see 5ms+ of idle on the bus, before it
          sends
        //remote wakeup signalling.  One way to be certain that this
          parameter
        //gets met, is to add a 2ms+ blocking delay here (2ms plus at
```

```
            //least 3ms from bus idle to USBIsBusSuspended() == TRUE, yeilds
            //5ms+ total delay since start of idle).
            delay_count = 3600U;
            do
            {
                delay_count--;
            }while(delay_count);
            //Now drive the resume K-state signalling onto the USB bus.
            USBResumeControl = 1;       // Start RESUME signaling
            delay_count = 2000U;        // Set RESUME line for 1-13 ms
            do
            {
                delay_count--;
            }while(delay_count);
            USBResumeControl = 0;       //Finished driving resume signalling
            USBUnmaskInterrupts();
        }
    }
}

/********************************************************************
 * Function:       BOOL USER_USB_CALLBACK_EVENT_HANDLER(
 *                     USB_EVENT event, void *pdata, WORD size)
 * PreCondition:    None
 * Input:          USB_EVENT event - the type of event
 *                void *pdata - pointer to the event data
 *                WORD size - size of the event data
 * Output:          None
```

```
 * Side Effects:    None
 * Overview:        This function is called from the USB stack to
 *                notify a user application that a USB event
 *                occured.  This callback is in interrupt context
 *                when the USB_INTERRUPT option is selected.
 * Note:            None
 *******************************************************************/
BOOL USER_USB_CALLBACK_EVENT_HANDLER(int event, void *pdata, WORD size)
{
    switch(event)
    {
        case EVENT_TRANSFER:
            break;
        case EVENT_SOF:
            USBCB_SOF_Handler();
            break;
        case EVENT_SUSPEND:
            USBCBSuspend();
            break;
        case EVENT_RESUME:
            USBCBWakeFromSuspend();
            break;
        case EVENT_CONFIGURED:
            USBCBInitEP();
            break;
        case EVENT_SET_DESCRIPTOR:
            USBCBStdSetDscHandler();
            break;
```

```
        case EVENT_EP0_REQUEST:
            USBCBCheckOtherReq();
            break;
        case EVENT_BUS_ERROR:
            USBCBErrorHandler();
            break;
        case EVENT_TRANSFER_TERMINATE
            break;
        default:
            break;
    }
    return TRUE;
}

/********************************************************************
******

 * Function:        BOOL ReportSupported(void)

 * PreCondition:    None

 * Input:           None

 * Output:          2 if it's a supported Output report

 *                 3 if it's a supported Feature report

 *                0 for all other cases

 * Side Effects:    None

 * Overview:        Checks to see if the HID supports a specific Output or
Feature

 *                 report.

 * Note:            None


********************************************************************
****/

BYTE ReportSupported(void)
```

```
{
switch (SetupPkt.W_Value.byte.HB)
{
      case 0x02: // Output report
      {
                  // Setup packet low byte
                switch(SetupPkt.W_Value.byte.LB)
                    {
                          case 0x00: // Report ID 0
                          {
                              return 2;
                          }
                          default:
                          {


                            return 0;
                          }
                } // end switch(SetupPkt.W_Value.byte.LB))
        }
        case 0x03: // Feature report
        {
                  // Setup packet low byte

                  switch(SetupPkt.W_Value.byte.LB)
                     {
                          case 0x00: // Report ID 0
                          {
```

```
                            return 3;
                        }
                        default:
                        {
                            // Other report IDs not supported.
                          return 0;
                        }
                    } // end switch(SetupPkt.W_Value.byte.LB))
        }
        default:
        {
            return 0;
        }
    } // end switch(SetupPkt.W_Value.byte.HB))

} // end ReportSupported

/*********************************************************************
 * Function:       void User_Get_Report_Handler(void)
 * PreCondition:    A Setup packet with a HID Get_Report request
 *                has been received.
 * Input:          None
 * Output:         None
 * Side Effects:    None
 * Overview:       If the HID supports the requested report, provides data
 *                for an Input or Feature report. Otherwise the endpoint
 *                returns STALL.
 *                This code is adapted from the code for GET_MAX_LUN in
```

```
 *               usb_function_msd.c, USBCheckMSDRequest() function.
 * Note:           None
 *******************************************************************/
void User_Get_Report_Handler(void)
{
    #if defined(USER_GET_REPORT_HANDLER)
        if ((SetupPkt.W_Value.byte.HB) == 0x01)
        {
            // Input report
            if (SetupPkt.W_Value.byte.LB == 0x00)
            {
                // Report ID = 0, supported
    USBEP0SendRAMPtr((BYTE*)&hid_report_out[0],HID_INPUT_REPORT_BYTES,USB
_EP0_INCLUDE_ZERO);
            }
        }

        else if (SetupPkt.W_Value.byte.HB == 0x03)
        {
            // Feature report
            if (SetupPkt.W_Value.byte.LB == 0x00)
            {
                // Report ID = 0, supported
    USBEP0SendRAMPtr((BYTE*)&hid_report_feature[0],HID_FEATURE_REPORT_BYT
ES,USB_EP0_INCLUDE_ZERO);
            }
        }
    #endif
```

```
}//end User_Get_Report_Handler


/*****************************************************************
 * Function:        void User_Set_Report_Handler(void)
 * PreCondition:    A Setup packet with a HID Set_Report request
 *                has been received.
 * Input:           None
 * Output:          None
 * Side Effects:    None
 * Overview:        If the HID supports the report the host is requesting
 *                to send, prepares to receive the report in the
 *                Data stage of the control transfer.
 *                Otherwise the endpoint returns STALL.
 *                This code is adapted from the code for SET_LINE_CODING in
 *                usb_function_cdc.c, USBCheckCDCRequest() function.
 * Note:            None
 *****************************************************************/


void User_Set_Report_Handler(void)
{
    #if defined(USER_SET_REPORT_HANDLER)
Nop();
                    switch (ReportSupported())
        {
             case 0x02: // Output report
             {
        if (SetupPkt.W_Value.byte.LB == 0x00){
```

```
                                    // Report ID = 0, supported
                                    outPipes[0].wCount.Val = SetupPkt.wLength;
                                    outPipes[0].pDst.bRam =
    (BYTE*)OUTPUT_REPORT;
                                    outPipes[0].pFunc = SET_REPORT_PFUNC;
                                    outPipes[0].info.bits.busy = 1;
        }
        break;
             }
        case 0x03: // Feature report
        {
        if (SetupPkt.W_Value.byte.LB == 0x00)
        {
                                   // Report ID = 0, supported
                                    outPipes[0].wCount.Val = SetupPkt.wLength;
                                    outPipes[0].pDst.bRam =
(BYTE*)FEATURE_REPORT;
                                    outPipes[0].pFunc = SET_REPORT_PFUNC;
                                    outPipes[0].info.bits.busy = 1;
                                }
                               break;
        }
       }
    #endif

}//end User_Set_Report_Handler
/** EOF main.c *************************************************/
#endif
```

3.1.1.17 Descriptors.c:

```
#ifndef __USB_DESCRIPTORS_C
#define __USB_DESCRIPTORS_C

/** INCLUDES *******************************************************/
#include "usb.h"
#include "usb_function_hid.h"

/* Device Descriptor */
ROM USB_DEVICE_DESCRIPTOR device_dsc=
{
    0x12,                   // Size of this descriptor in bytes
    USB_DESCRIPTOR_DEVICE,  // DEVICE descriptor type
    0x0200,                 // USB Spec Release Number in BCD format
    0x00,                   // Class Code
    0x00,                   // Subclass code
    0x00,                   // Protocol code
    USB_EP0_BUFF_SIZE,      // Max packet size for EP0, see usb_config.h
    MY_VID,                 // Vendor ID (Lakeview Research)
    MY_PID,                 // Product ID: generic HID device demo
    0x0001,                 // Device release number in BCD format
    0x01,                   // Manufacturer string index
    0x02,                   // Product string index
    0x00,                   // Device serial number string index
    0x01                    // Number of possible configurations
};
```

```
/* Configuration 1 Descriptor */
ROM BYTE configDescriptor1[]={
    /* Configuration Descriptor */
    0x09,//sizeof(USB_CFG_DSC),    // Size of this descriptor in bytes
    USB_DESCRIPTOR_CONFIGURATION,   // CONFIGURATION descriptor type
    0x29,0x00,                      // Total length of data for this cfg
    1,                              // Number of interfaces in this cfg
    1,                              // Index value of this configuration
    0,                             // Configuration string index
    _DEFAULT | _RWU,               // Attributes, see usb_device.h
    250,                          // Max power consumption (2X mA)

    /* Interface Descriptor */
    0x09,//sizeof(USB_INTF_DSC),    // Size of this descriptor in bytes
    USB_DESCRIPTOR_INTERFACE,      // INTERFACE descriptor type
    0,                              // Interface Number
    0,                              // Alternate Setting Number
    2,                              // Number of endpoints in this intf
    HID_INTF,                      // Class code
    0,                             // Subclass code
    0,                             // Protocol code
    0,                             // Interface string index

    /* HID Class-Specific Descriptor */
    0x09,//sizeof(USB_HID_DSC)+3,   // Size of this descriptor in bytes
    DSC_HID,                      // HID descriptor type
```

```
    0x11,0x01,                  // HID Spec Release Number in BCD format
(1.11)

    0x00,                       // Country Code (0x00 for Not supported)

    HID_NUM_OF_DSC,              // Number of class descriptors, see usbcfg.h

    DSC_RPT,                    // Report descriptor type

    HID_RPT01_SIZE,0x00,//sizeof(hid_rpt01),     // Size of the report
descriptor



    /* Endpoint Descriptor */

    0x07,                       /*sizeof(USB_EP_DSC)*/

    USB_DESCRIPTOR_ENDPOINT,        //Endpoint Descriptor

    HID_EP | _EP_IN,              //EndpointAddress

    _INTERRUPT,                   //Attributes

    0x40,0x00,                    //size

    0x01,                         //Interval



    /* Endpoint Descriptor */

    0x07,/*sizeof(USB_EP_DSC)*/

    USB_DESCRIPTOR_ENDPOINT,          //Endpoint Descriptor

    HID_EP | _EP_OUT,               //EndpointAddress

    _INTERRUPT,                     //Attributes

    0x40,0x00,                      //size

    0x01                            //Interval

};



//Language code string descriptor

ROM struct{BYTE bLength;BYTE bDscType;WORD string[1];}sd000={

sizeof(sd000),USB_DESCRIPTOR_STRING,{0x0409

}};
```

```
//Manufacturer string descriptor

ROM struct{BYTE bLength;BYTE bDscType;WORD string[21];}sd001={

sizeof(sd001),USB_DESCRIPTOR_STRING,

{'C','a','r','b','a','l','l','a','s','
','T','e','c','h','n','o','l','o','g','y','.'

}};

//Product string descriptor

ROM struct{BYTE bLength;BYTE bDscType;WORD string[16];}sd002={

sizeof(sd002),USB_DESCRIPTOR_STRING,

{'G','e','n','e','r','i','c',' ','H','I','D',' ','D','A','Q','7'

}};


//Class specific descriptor - HID

ROM struct{BYTE report[HID_RPT01_SIZE];}hid_rpt01={

{

  0x06, 0xA0, 0xFF,             // Usage page (vendor defined)

  0x09, 0x01,           // Usage ID (vendor defined)

  0xA1, 0x01,           // Collection (application)UA

   // The Input report

      0x09, 0x03,           // Usage ID - vendor defined

      0x15, 0x00,           // Logical Minimum (0)

      0x26, 0xFF, 0x00,     // Logical Maximum (255)

      0x75, 0x08,           // Report Size (8 bits)

      0x95, 0x08,           // Report Count (2 fields)

      0x81, 0x02,           // Input (Data, Variable, Absolute)

    // The Output report

      0x09, 0x04,           // Usage ID - vendor defined

      0x15, 0x00,           // Logical Minimum (0)
```

```
      0x26, 0xFF, 0x00,     // Logical Maximum (255)
      0x75, 0x08,           // Report Size (8 bits)
      0x95, 0x06,           // Report Count (2 fields)
      0x91, 0x02,           // Output (Data, Variable, Absolute)
   // The Feature report
      0x09, 0x05,           // Usage ID - vendor defined
      0x15, 0x00,           // Logical Minimum (0)
      0x26, 0xFF, 0x00,     // Logical Maximum (255)
      0x75, 0x08,           // Report Size (8 bits)
      0x95, 0x02,           // Report Count (2 fields)
      0xB1, 0x02,           // Feature (Data, Variable, Absolute)

   0xc0,                          // end collection
}
};

//Array of configuration descriptors
ROM BYTE *ROM USB_CD_Ptr[]=
{
    (ROM BYTE *ROM)&configDescriptor1
};

//Array of string descriptors
ROM BYTE *ROM USB_SD_Ptr[]=
{
    (ROM BYTE *ROM)&sd000,
    (ROM BYTE *ROM)&sd001,
    (ROM BYTE *ROM)&sd002
```

```
};

/** EOF usb_descriptors.c
*************************************************/

#endif
```

3.3.3 Scripts Matlab:

3.1.1.18 Test_tarjeta.m:

```
DAQ_Start();    %inicia comunicación con la tarjeta
while 1   %bucle de lectura
Puertos[a1,a2,a3,a4]=DAQ_Read(); %comprueba los cuatro puertos
stop=puertos(1); %si se pulsa el botón para salir del bucle
if puertos(4)<50 %evalúa y genera señal cuadrada
    DAQ_Write(100);
else
    DAQ_Write(0);
end
if(stop==0) %sal del bucle
   break
end
end
DAQ_Stop(); %finaliza la comunicación
```

3.4 Anexo IV: Guía Programación PIC

3.4.1 MPAB X IDE v4.15

Para crear y compilar el firmware visto en el análisis se ha utilizado el programa de MPLAB X IDE con el compilador XC8.

El compilador XC8 es para PICS de 8bits y se debe descargar a parte del MPLAB X IDE. Se encuentra disponible en la página web de Microchip.

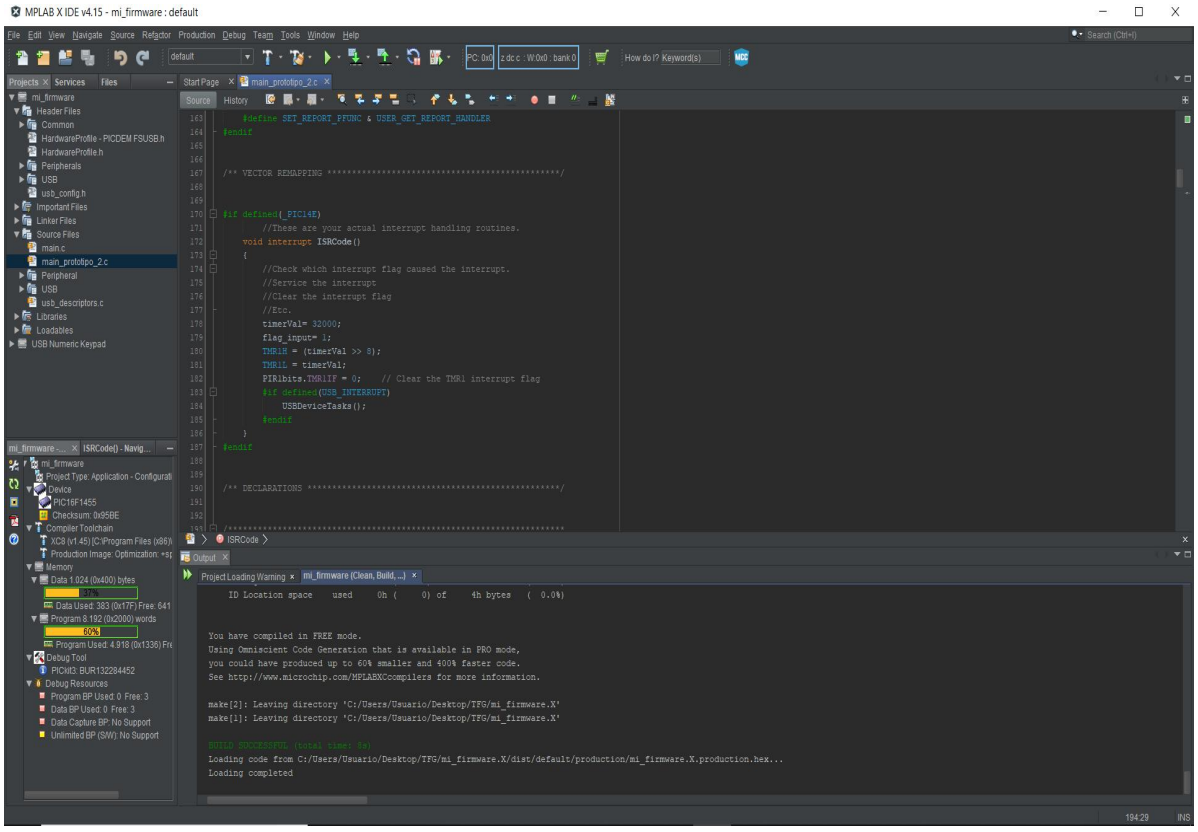


Imagen 3.4.1.1 – Página principal del compilador

El programa presenta el aspecto que se puede ver en la imagen 3.4.1.1. En la cual tenemos arriba a la izquierda, la ventana del explorador de proyecto y abajo, la configuración del programa que se está desarrollando: tipo de microcontrolador, compilador, consumo de memoria y *debugger*. Abajo la ventana de O*utput* donde

se puede ver las y los errores, y si el programa ha compilado correctamente. Por último se encuentra el editor de texto donde se escribe el programa.

Para construir el *firmware*, lo primero es crear un nuevo proyecto, en el menú de *File>New Project.* A continuación se selecciona *Microchip Embedded > Standalone Project*.

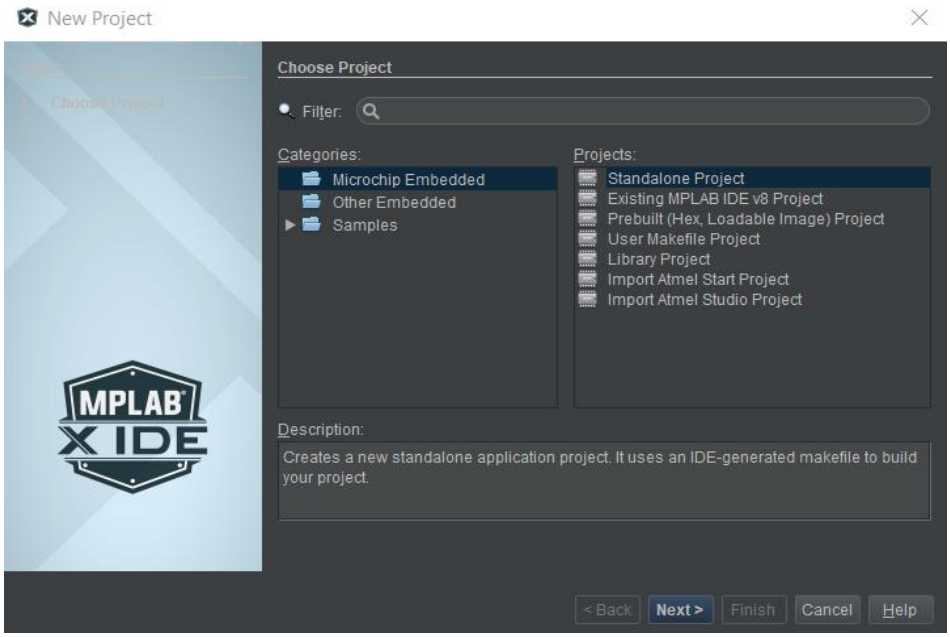


Imagen 3.4.1.2 – Pantalla uno de configuración del proyecto

Lo siguiente es indicar el tipo de PIC que se va a emplear, en este caso el PIC16F1455. Es una configuración que se puede modificar más adelante en cualquier momento.

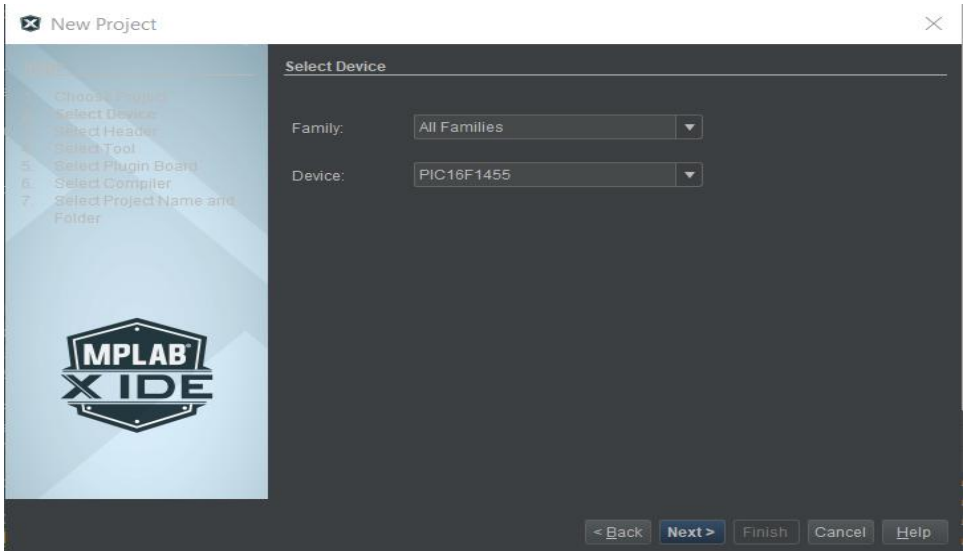


Imagen 3.4.1.3 – Pantalla dos de configuración del proyecto

Selección de la herramienta en este caso, se marcará el Pickit3.

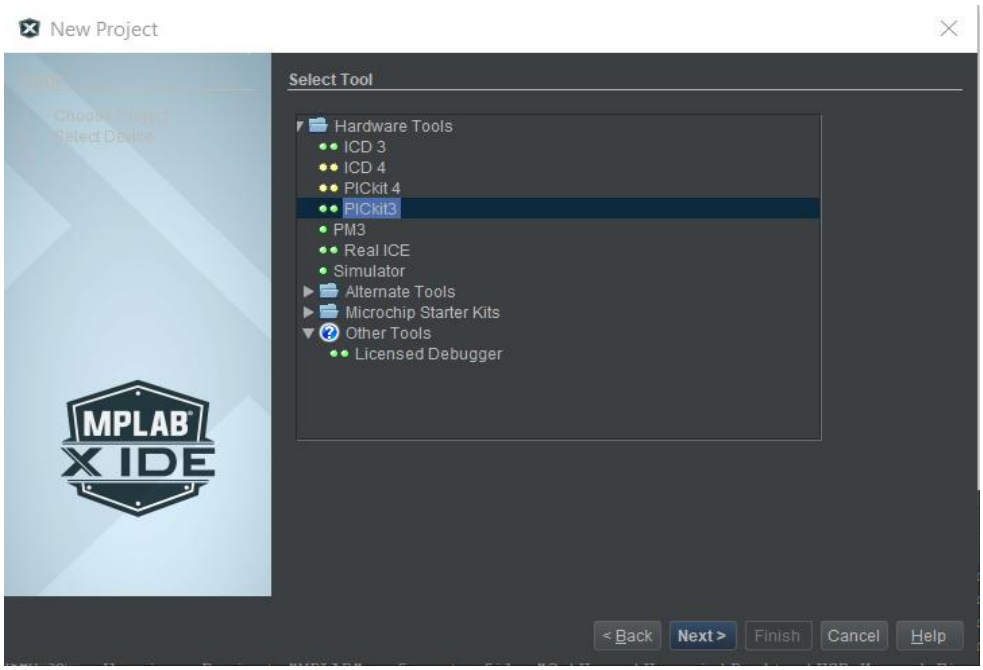


Imagen 3.4.1.4 – Pantalla tres de configuración del proyecto

Selección del compilador, al PIC16F1455 le corresponde el compilador XC8, se puede elegir cualquiera de las dos versiones.

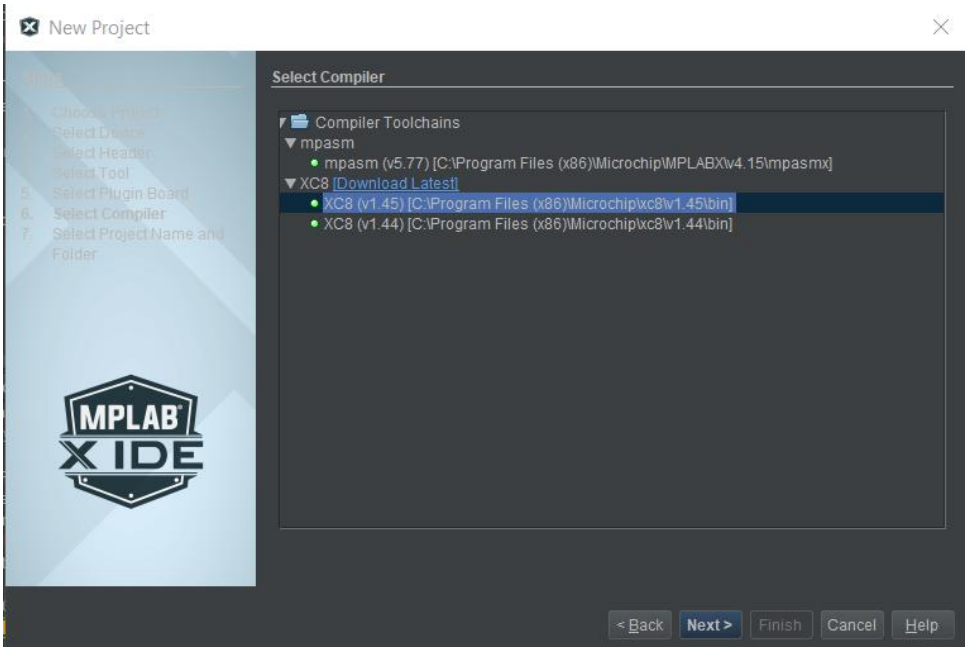


Imagen 3.4.1.5 – Pantalla cuatro de configuración del proyecto

Para finalizar se le da un nombre a la carpeta y ya se puede comenzar a trabajar en el proyecto.

3.4.2 MPLAB IPE v4.15

Una vez compilado el *firmware* hay que cargarlo en el PIC, para ello se va emplear el programador **Pickit 3** junto con el programa MPLAB IPE.

Este programa se puede descargar desde la página web de Microchip, aunque normalmente viene incluido en el paquete de instalación del MPLAB IDE.

Una vez que se ejecuta se abre una ventana como la de la imagen 3.4.2.1. En ella se pueden ver tres *commandbox:* la primera es para seleccionar la familia (AVR, PIC, etc.), la segunda de el microcontrolador (para este proyecto el PIC16F1455) y la tercera se rellena automáticamente una vez que se ha detectado el programador (en este caso el Pickit 3).

A la derecha de las *commandbox* se encuentran una serie de casillas, que funcionan a modo de registro, de tal forma que el programa lleva la cuenta del número de veces que se ha programado los PIC, errores, etc.

Abajo están los comandos que los que se puede interaccionar con el PIC:

- *Program*: programa el PIC con el, código seleccionado.
- *Erase*: borra el programa actual del PIC.
- *Read*: lee el valor de los registros del PIC.
- *Verify*: comprueba que el PIC se ha conectado correctamente.

Debajo de estos botones de comandos se encuentra los directorios para cargar seleccionar el programa que se desea cargar en el PIC.

Por último está la ventana *Output,* dónde se puede observar si el proceso se ha realizado éxito (ver imagen 3.4.2.1).

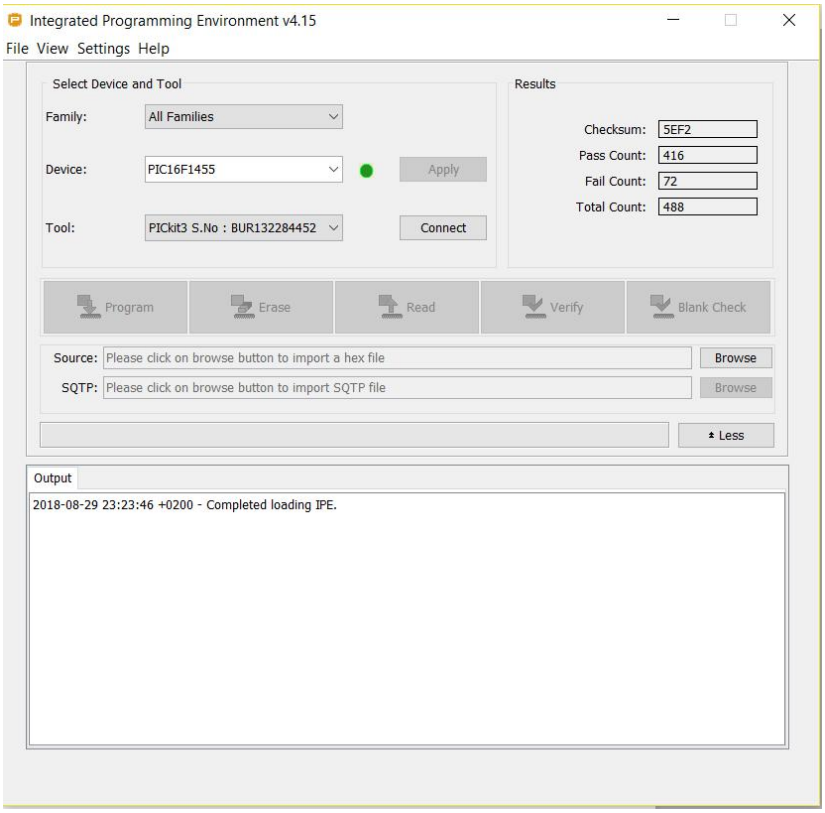


Imagen 3.4.2.1 – MPLAB IPE

3.4.2.1 Pickit 3

Como se ha comentado anteriormente la herramienta seleccionada para programar el PIC es el Pickit 3 (ver imagen 3.4.2.1.1).

Imagen 3.4.2.1.1 – Pickit 3

Esta herramienta programación permite a través del puerto de ordenador alimentar el PIC y a la vez programarlo, debido a que el microcontrolador soporta programación ICSP (*In Circuit Serial Programming*). Para ello dispone de los siguiente pines (ver imagen 3.4.2.1.1):

- Vpp: su función es configurar el PIC en el estado de programación.
- PGD: *Program Clock,* señal de reloj.
- PGC: *Program Data*, pin para programación en serie de datos. También realiza las otras funciones asociadas a los comandos vistos anteriormente: lectura, verificación, etc.
- Vdd: voltaje de alimentación. Se puede configurar el programador para que alimente al PIC.
- Vss: masa.
- PGM: *Low-Voltage Programming Mode*. Se emplea para programar con bajo voltaje ignorando Vpp.

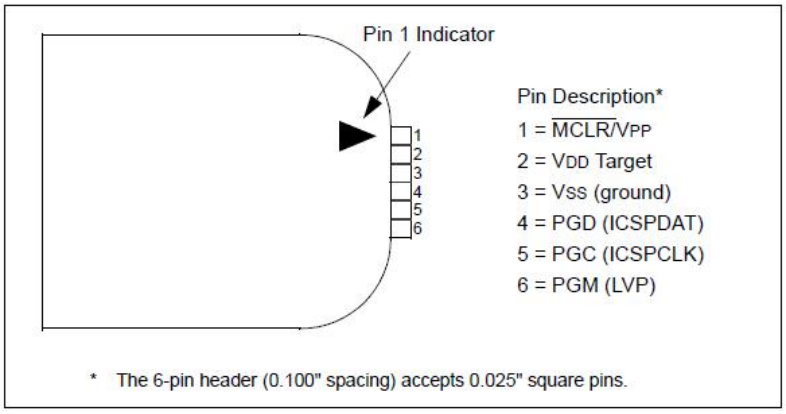


Imagen 3.4.2.1.1 – Pines Pickit 3

Para programar el microcontrolador de esta aplicación se descartará el modo de programación de bajo voltaje (LVP), se programará en un circuito a parte del de la DAQ, por lo que no se integrarán el circuito de programación del PIC y el USB de la tarjeta de adquisición de datos.

3.4.2.2 Programación

Primero se conecta los pines del Pickit 3 con los del PIC según la tabla 3.4.2.2.1.

Pickit 3	PIC 16F1455
Vpp	Pin 4 /RA3
Vdd	Pin 1 /Vdd
Vss	Pin 3 /Vss
PGD	Pin 10 /RC0
PGC	Pin 9 /RC1
PGM	-

Tabla 3.4.2.2.1 – Relación Pickit 3 - PIC

Una vez que se realizado correctamente el circuito se conecta al puerto USB del ordenador. En ese momento se deberían encender los leds del programador. Si se encienden se ejecuta el programa MPLAB IPE.

El programa ya reconoce automáticamente el programador. El siguiente paso será configurar el programador para que alimente el PIC a través de *Vdd*. Para ello hay que ir a la pestaña de *Settings*, seleccionar *Advance Mode*, dentro de este modo la ventana de *Power* y por último marcar la casilla de *Power Target Circuit from Tool* (ver imagen 3.4.2.2.2).

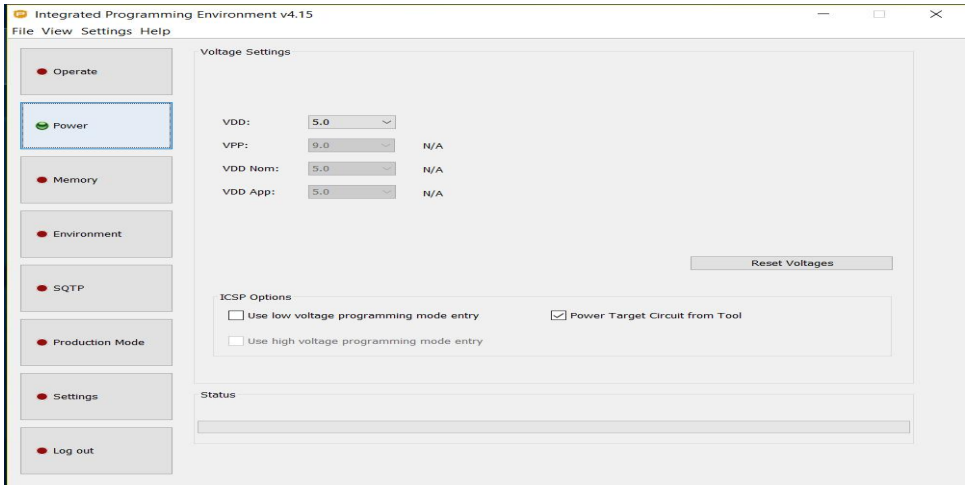


Imagen 3.4.2.2.2 – Settings MPLAB IPE

En ese momento se *clickea* en el botón de *Connect.* Si se ha conectado correctamente la venta de *Output* de verá presentar el aspecto de la imagen 3.4.2.2.3.

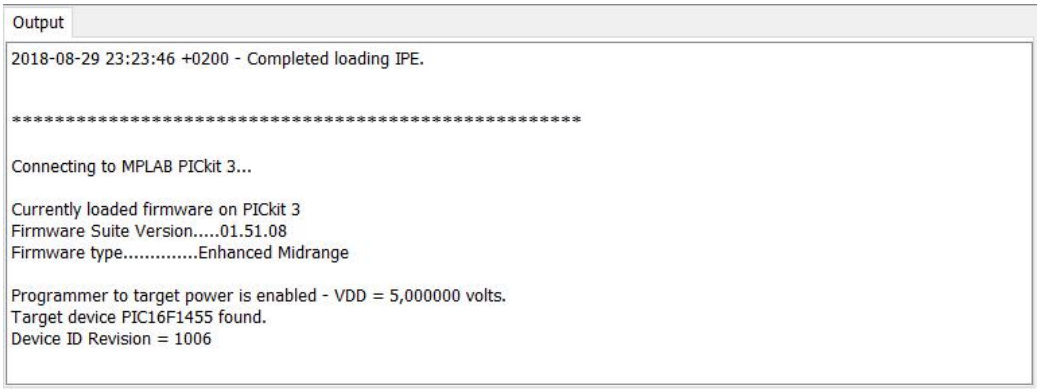


Imagen 3.4.2.2.3 – Conexión del PIC

A continuación se carga el archivo del programa. Para ello se debe buscar un archivo de extensión .Hex, el cual es el programa compilado y traducido para el

PIC. Generalmente se encuentra en la siguiente dirección: "*Carpeta_del_projecto.X<dist<default<production*".

Una vez que se tiene cargado el programa, se puede escribir el PIC. Si se ha realizado con éxito la programación la ventana *Output* presentará el aspecto de la imagen 3.4.2.2.4.

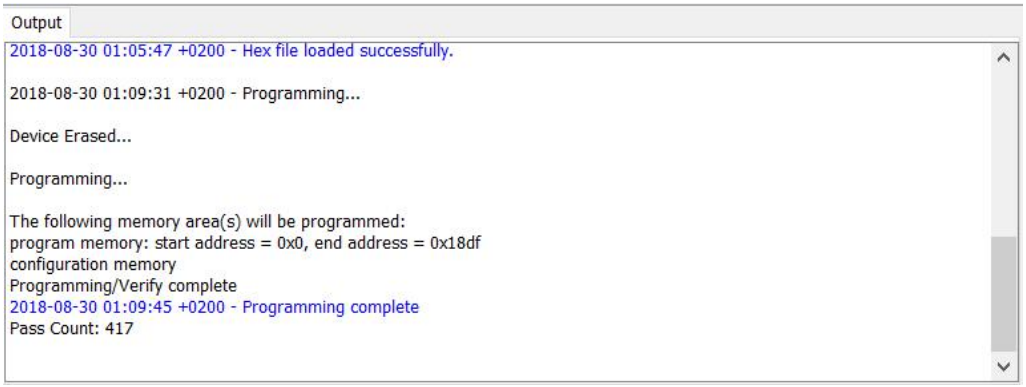


Imagen 3.4.2.2.4 – Programación exitosa

3.5 Anexo V: Datasheets

3.5.1 Datasheets PIC16F1455

“https://www.microchip.com/wwwproducts/en/PIC16F1455”

3.5.2 Datasheets AD8659

“http://www.analog.com/en/products/ad8659.html”.

PIC16(L)F1454/5/9

14/20-Pin, 8-Bit Flash USB Microcontroller with XLP Technology

High-Performance RISC CPU:

- C Compiler Optimized Architecture
- Only 49 Instructions
- 14 Kbytes Linear Program Memory Addressing
- 1024 Bytes Linear Data Memory Addressing
- Operating Speed:
  - DC – 48 MHz clock input
  - DC – 83 ns instruction cycle
  - Selectable 3x or 4x PLL for specific frequencies
- Interrupt Capability with Automatic Context Saving
- 16-Level Deep Hardware Stack with Optional Overflow/Underflow Reset
- Direct, Indirect and Relative Addressing modes:
  - Two full 16-bit File Select Registers (FSRs) capable of accessing both data or program memory
  - FSRs can read program and data memory

Special Microcontroller Features:

- Operating Voltage Range:
  - 1.8V to 3.6V (PIC16LF145X)
  - 2.3V to 5.5V (PIC16F145X)
- Self-Programmable under Software Control
- Power-on Reset (POR)
- Power-up Timer (PWRT)
- Programmable Brown-Out Reset (BOR)
- Low-Power BOR (LPBOR)
- Extended Watchdog Timer (WDT):
  - Programmable period from 1 ms to 256s
- Programmable Code Protection
- In-Circuit Serial Programming™ (ICSP™) via Two Pins
- Enhanced Low-Voltage Programming (LVP)
- Power-Saving Sleep mode
- 128 Bytes High-Endurance Flash
  - 100,000 write Flash endurance (minimum)

Universal Serial Bus (USB) Features:

- Self-Tuning from USB Host (eliminates need for external crystal)
- USB V2.0 Compliant SIE
- Low Speed (1.5 Mb/s) and Full Speed (12 Mb/s)
- Supports Control, Interrupt, Isochronous and Bulk Transfers
- Supports up to Eight Bidirectional Endpoints
- 512-Byte Dual Access RAM for USB
- Interrupt-on-Change (IOC) on D+/D- for USB Host Detection
- Configurable Internal Pull-up Resistors for use with USB

Extreme Low-Power Management PIC16LF145X with XLP:

- Sleep mode: 25 nA @ 1.8V, typical
- Watchdog Timer Current: 290 nA @ 1.8V, typical
- Timer1 Oscillator: 600 nA @ 32 kHz, typical
- Operating Current: 25 μA/MHz @ 1.8V, typical

Flexible Oscillator Structure:

- 16 MHz Internal Oscillator Block:
  - Factory calibrated to ±0.25%, typical
  - Software selectable frequency range from 16 MHz to 31 kHz
  - Tunable to 0.25% across temperature range
  - 48 MHz with 3x PLL
- 31 kHz Low-Power Internal Oscillator
- Clock Switching with run from:
  - Primary Oscillator
  - Secondary Oscillator (SOSC)
  - Internal Oscillator
- Clock Reference Output:
  - Clock Prescaler
  - CLKOUT

Analog Features(1):

- Analog-to-Digital Converter (ADC):
  - 10-bit resolution
  - Up to nine external channels
  - Two internal channels:
    - Fixed Voltage Reference channel
    - DAC output channel
  - Auto acquisition capability
  - Conversion available during Sleep
- Two Comparators:
  - Rail-to-rail inputs
  - Power mode control
  - Software controllable hysteresis
- Voltage Reference module:
  - Fixed Voltage Reference (FVR) with 1.024V, 2.048V and 4.096V output levels
- Up to One Rail-to-Rail Resistive 5-Bit DAC with Positive Reference Selection

Note 1: Analog features are not available on PIC16(L)F1454 devices.

TÍTULO: **CREACIÓN DE UN PROTOCOLO DE COMUNICACIÓN ENTRE UN PC Y UN PUERTO USB DE UN MICROCONTROLADOR Y SU INTEGRACIÓN EN MATLAB**

PLANOS

PETICIONARIO: **ESCUELA UNIVERSITARIA POLITÉCNICA**

AVDA. 19 DE FEBRERO, S/N

15405 - FERROL

FECHA: **SEPTIEMBRE DE 2018**

AUTOR: EL ALUMNO

Fdo.: CARBALLAS CHAS SERGIO

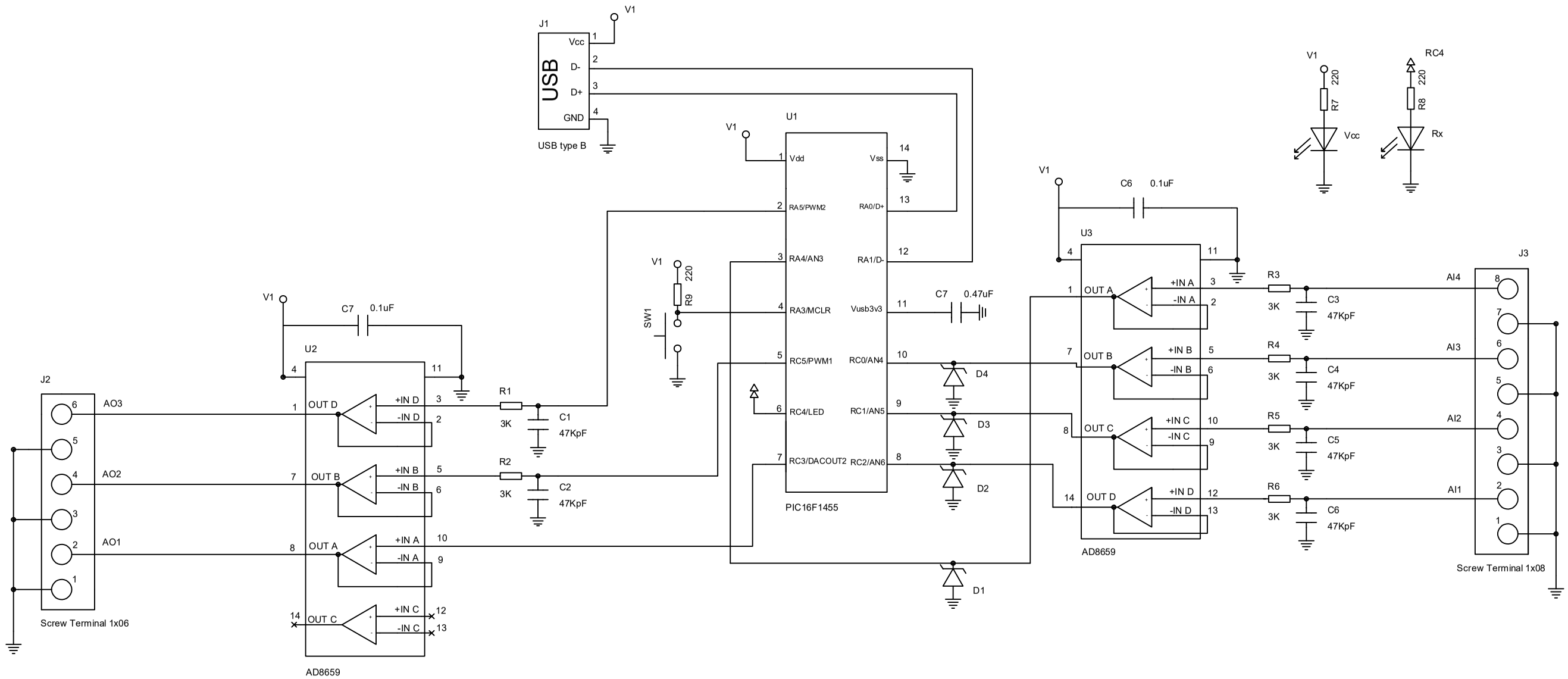
Índice PLANOS

4 PLANOS

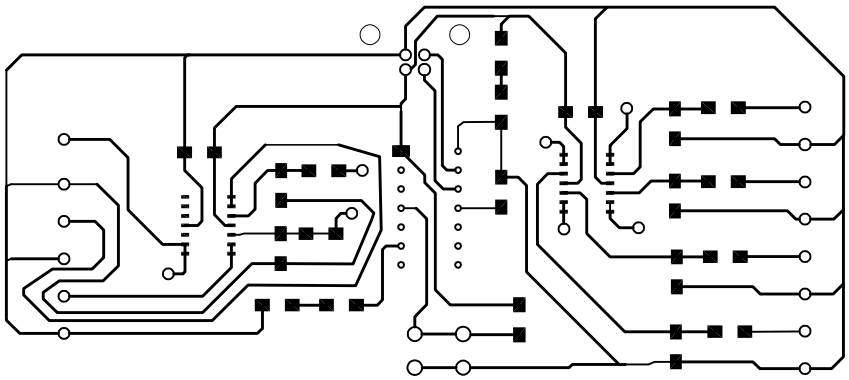
4.1 Esquemático de la placa

4.2 PCB

4.3 Esquemático programador



UNIVERSIDADE DA CORUÑA — ESCUELA UNIVERSITARIA POLITÉCNICA GRADO EN INGENIERÍA ELECTRÓNICA INDUSTRIAL Y AUTOMÁTICA	TFG Nº: 770G01A145
TÍTULO DEL TFG: CREACIÓN DE UN PROTOCOLO DE COMUNICACIÓN	
TÍTULO DEL PLANO: ESQUEMÁTICO DAQ	FECHA: SEPTIEMBRE 2018
AUTOR: SERGIO CARBALLAS CHAS — FIRMA:	ESCALA: - PLANO Nº: 01



UNIVERSIDADE DA CORUÑA

ESCUELA UNIVERSITARIA POLITÉCNICA

GRADO EN INGENIERÍA ELECTRÓNICA INDUSTRIAL Y AUTOMÁTICA

TFG Nº: 770G01A145

TÍTULO DEL TFG:

CREACIÓN DE UN PROTOCOLO DE COMUNICACIÓN

TÍTULO DEL PLANO:

PCB

FECHA: SEPTIEMBRE 2018

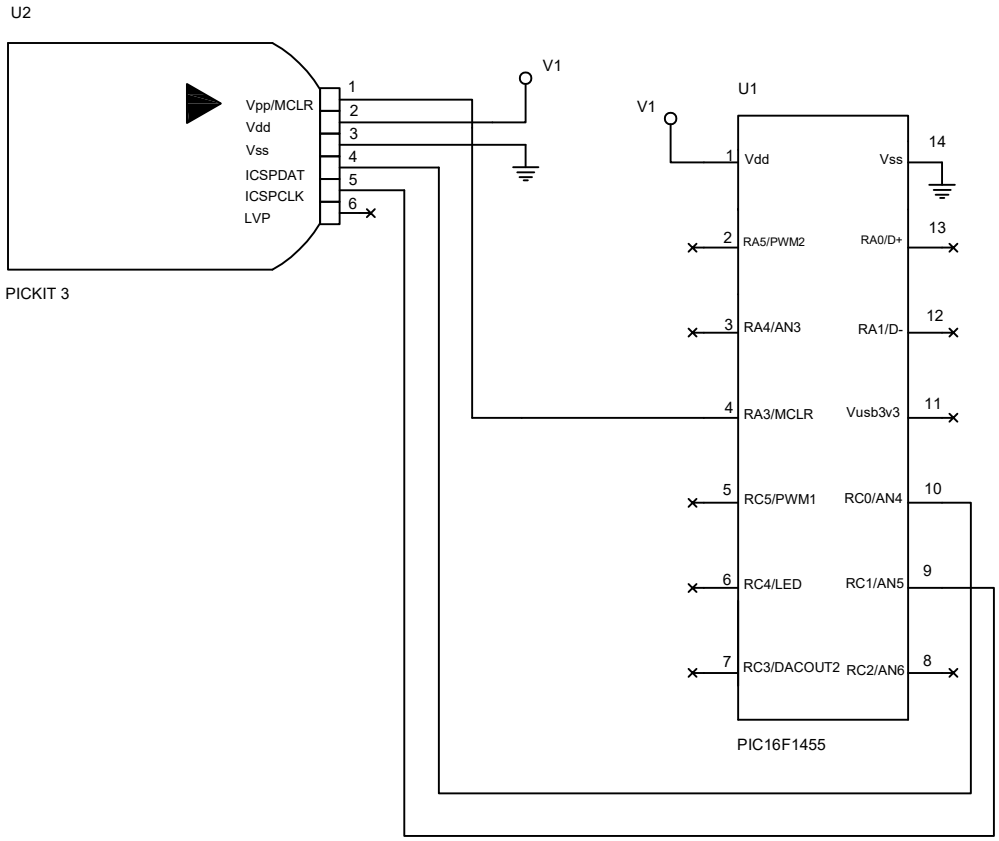
ESCALA: 1:1

AUTOR:

SERGIO CARBALLAS CHAS

FIRMA:

PLANO Nº: 02



UNIVERSIDADE DA CORUÑA

ESCUELA UNIVERSITARIA POLITÉCNICA

GRADO EN INGENIERÍA ELECTRÓNICA INDUSTRIAL Y AUTOMÁTICA

TFG Nº: 770G01A145

TÍTULO DEL TFG:

CREACIÓN DE UN PROTOCOLO DE COMUNICACIÓN

TÍTULO DEL PLANO:

ESQUEMÁTICO PROGRAMADOR Y PIC16F1455

FECHA: SEPTIEMBRE 2018

ESCALA: -

AUTOR:

SERGIO CARBALLAS

FIRMA:

PLANO Nº: 03

TÍTULO: **CREACIÓN DE UN PROTOCOLO DE COMUNICACIÓN ENTRE UN PC Y UN PUERTO USB DE UN MICROCONTROLADOR Y SU INTEGRACIÓN EN MATLAB**

PLIEGO DE CONDICIONES

PETICIONARIO: **ESCUELA UNIVERSITARIA POLITÉCNICA**

AVDA. 19 DE FEBRERO, S/N

15405 - FERROL

FECHA: **SEPTIEMBRE DE 2018**

AUTOR: El ALUMNO

Fdo.: CARBALLAS CHAS SERGIO

Índice PLIEGO DE CONDICIONES

5 PLIEGO DE CONDICIONES

5.1 Condiciones de uso de la placa

El presente documento tiene como objeto establecer las condiciones técnicas en las que se podrá utilizar y se garantiza el correcto funcionamiento del protocolo desarrollado.

- La DAQ debe estar permanente conectada al PC una vez que se inicia el DAQ_Start. Y se recomienda asegurarse que esté correctamente conectada antes de iniciar cualquier operación.
- La tensión máxima que soporta la DAQ es de 5V. Una tensión superior a ésta podría dañar los integrados.
- No se recomienda su uso en ambientes húmedos.
- El *firmware* está preparado para funcionar con un PIC16F1455, el empleo de otros modelos requerirá realizar modificaciones en la capa de entradas y salidas.
- La versión de Matlab debe ser inferior o igual al año 2015. Esto es debido a que versiones posteriores a esta, no incluyen el compilador de C necesario para que se puede llamar a las funciones del *HIDapi*.
- Es necesario que los archivos *hidapi64.dll* y *hidapi64_proto* se encuentren en la misma carpeta que las funciones de Matlab.
- Se garantiza el correcto funcionamiento en Sistemas Operativos basados en Windows.
- Se recomienda modificar en lo mínimo posible las funciones DAQ_ de Matlab, en especial los argumentos de las funciones que interactuan con la librería *hidapi64.dll*.
- Aunque el protocolo sea flexible, cualquier cambio en las funciones de Matlab, sobre todo en aquellas partes relacionadas con la comunicación USB, tendrá consecuencias de cara al *firmware* de la DAQ. Por lo que probablemente habría que modificar este último.

- En caso de que se produzca un error en el funcionamiento de la DAQ, no se recomienda desconectarla, simplemente con resetearla con el pulsador de *reset* es suficiente. Una vez que se ha reseteado se recomienda cerrar la ejecución del protocolo en Matlab mediante la función DAQ_Stop().

TÍTULO: **CREACIÓN DE UN PROTOCOLO DE COMUNICACIÓN ENTRE UN PC Y UN PUERTO USB DE UN MICROCONTROLADOR Y SU INTEGRACIÓN EN MATLAB**

ESTADO DE LAS MEDICIONES

PETICIONARIO: **ESCUELA UNIVERSITARIA POLITÉCNICA**

AVDA. 19 DE FEBRERO, S/N

15405 - FERROL

FECHA: **SEPTIEMBRE DE 2018**

AUTOR: EL ALUMNO

Fdo.: CARBALLAS CHAS SERGIO

Índice ESTADO DE LAS MEDICIONES

6 Estado de las mediciones

En este documento se incluirán el listado de los materiales para realizar la placa, así como la distribución de horas invertidas en desarrollar el protocolo.

6.1 Listado de materiales

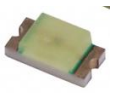
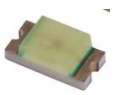
Ver tabla 6.1.1.

6.2 Distribución de horas

Ver tabla 6.2.1.

Elemento	Tiempo(h)
Firmware (Programación MPLAB X IDE)	100
Programa de usuario (Programación Matlab)	70
Desarrollo de la placa	20
Pruebas funcionamiento	10
Documentación	60

Tabla 6.2.1 – Distribución de horas

Componente	Imagen	Descripción	Referencia	Nº u
PIC16F1455		Microcontrolador 8 bits, PDIP, casa Microchip.	E/P	1
AD8659		Cuádruple amplificador de bajo consumo.	ARZ	2
Resistencia SMD 1206		Tamaño 1206, valor: 3KΩ	-	7
Resistencia SMD 1206		Tamaño 1206, valor: 220Ω	-	2
Condensador SMD 1206		Tamaño 1206, valores: 0.1µF	-	2
Condensador SMD 1206		Tamaño 1206, valores: 47nF	-	4
Condensador SMD 1206		Tamaño 1206, valores: 47nF	-	1
LED SMD 1206 Verde		Tamaño 1206, color Verde, tensión 5V	LSMD1206V	1
LED SMD 1206 Ambar		Tamaño 1206, color Ambar, tensión 5V	LSMD1206A	1
Pulsador switch		Pulsador agujero pasante, 4 pines, 12mm	CMP-0039	1
Terminal de tornillo		Conector de tornillo 5mm, 2 pines, 10A	B01MT4LC0F	7
Conector USB		Conector tipo B, agujero pasante.	NI5003	1

Placa de cobre para PCB		Simple cara, grosor 35µm, 100x160x1,6 mm	AD16	1
Diodo Zener		0,5W; 5V1;	BZX55	4
Cable USB		Tipo A-B, 30cm, Azul, USB 2.0 para Arduino	ACC_0931	1

Tabla 6.1.1 – Listado de materiales

TÍTULO: **CREACIÓN DE UN PROTOCOLO DE COMUNICACIÓN ENTRE UN PC Y UN PUERTO USB DE UN MICROCONTROLADOR Y SU INTEGRACIÓN EN MATLAB**

PRESUPUESTO

PETICIONARIO: **ESCUELA UNIVERSITARIA POLITÉCNICA**

AVDA. 19 DE FEBRERO, S/N

15405 - FERROL

FECHA: **SEPTIEMBRE DE 2018**

AUTOR: EL ALUMNO

Fdo.: CARBALLAS CHAS SERGIO

Índice PRESUPUESTO

7 Presupuesto

7.1 Placa

Coste de la tarjeta de adquisición de datos (ver tabla 7.1.1).

Componente	Coste unidad (€/u)	Nº unidades	Coste total (€)
PIC16F1455	1,24€	1	1,24€
AD8659	4,98€	2	9,96€
Resistencia SMD	0,17€	9	1,53€
Condensador SMD	0,28€	7	1,96€
LED SMD	0,24€	2	0,48€
Pulsador switch	0,50€	1	0,50€
Terminal de tornillo de 2 pines	0,18€	7	1,26€
Conector USB tipo B hembra	0,63€	1	0,63€
Placa de cobre para PCB	4,40€	1	4,40€
Diodo Zener BZX55	1,25€	4	5,00€
Cable USB tipo A-B	1,37€	1	1,37€
***Total** (con IVA incluído)*			**28,33€**

Tabla 7.1.1 – Presupuesto DAQ

7.2 Licencias software

En la tabla 7.1.2 se han incluido los programas necesarios para el diseño del protocolo. IVA incluido.

Programa	Coste (€)
Matlab	2.000€
MPLAB XC8 Compiler (versión gratuíta)	0,00€
***Total** (IVA incluído)*	**2.000,00€**

Tabla 7.2.1 – Presupuesto licencias

7.3 Mano de obra

Se ha fijado el salario en 40 €/h, presupuesto en la tabla 7.3.1.

Elemento	Tiempo(h)	Coste (€/h)	Total (€)
Firmware (Programación MPLAB X IDE)	100	40,00€	4.000,00 €
Programa de usuario (Programación Matlab)	70	40,00€	2.800,00€
Desarrollo de la placa	20	40,00€	800,00€
Pruebas funcionamiento	10	40,00€	400,00€
Documentación	60	40,00€	2.400,00€
Total (sin IVA)			10.400,00€
Total (con IVA 21%)			**12.584,00€**

Tabla 7.3.1 – Presupuesto Mano de obra

7.4 Presupuesto total

Coste total del protocolo ver tabla 7.4.1.

Presupuesto	Coste (€)
Placa	28,33€
Licencias	2.000,00€
Mano de obra	12.584,00€
Total	**14.612,33€**

Tabla 7.4.1 – Presupuesto total