ESCUELA TÉCNICA SUPERIOR DE INGENIEROS DE TELECOMUNICACIÓN

Diseño de un dispositivo HID mejorado con data-logger y pulsioxímetro

Proyecto final de carrera

Realizado por: Juan Domingo Jiménez Jerez

Dirigido por: Miguel Ángel Mateo Plá

Titulación: Ingeniería en Telecomunicaciones

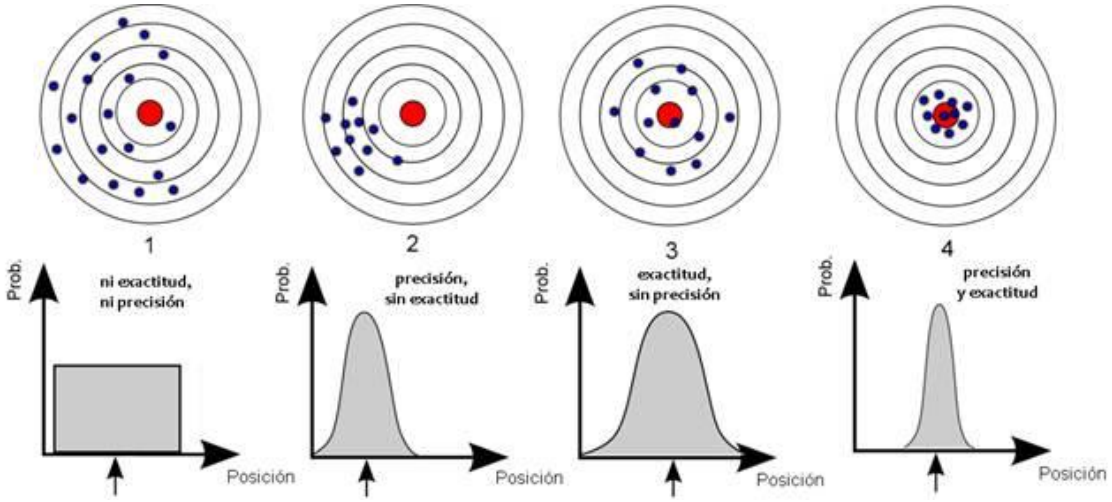
Valencia, 2015

Índice

1 Introducción.

El objeto principal del presente proyecto es el desarrollo de un dispositivo para experimentos psicológicos, este dispositivo está dirigido a psicólogos o neurólogos que realizan análisis a pacientes y necesitan registrar el tiempo en que ocurren diferentes eventos con una gran exactitud (milisegundos o incluso decimas de milisegundo) para realizar un análisis correcto. Para el desarrollo de estos experimentos la precisión y la exactitud son factores críticos.

El Hardware de las computadoras modernas puede ser cada vez más rápido, pero la precisión de milisegundos es muy difícil de lograr. Un error común es decir que "precisión en milisegundos" es igual a "exactitud en milisegundos" (1) . Precisión significa simplemente que los tiempos se toman y se indican en unidades de una milésima de segundo, pero esto no quiere decir que sean exactos. La precisión depende únicamente de la distribución de los resultados y no está relacionada con el valor convencionalmente "verdadero" de la medición. Por su parte, la exactitud viene definida como la proximidad entre el valor medido y el valor "verdadero" del mensurando. Así pues, una medición es más exacta cuanto más pequeño es el error de medida (2).



Por ejemplo, todos los monitores TFT y proyectores tienen retardos de entrada. Cada vez que se quiere presentar una imagen se necesita más tiempo para que aparezca en la pantalla de lo que se piensa. En algunos modelos esto puede ser más de 100 milisegundos. En cualquier experimento que se utilice, sólo es posible saber cuándo se solicitó que una imagen de estímulo se mostrará y no el momento en que apareció físicamente esto conduce a errores de exactitud.

Muchos psicólogo, neurólogo y otros investigadores usan ordenadores para ejecutar experimentos y necesitan ser informados con exactitud del momento en que se producen los eventos (en milésimas de segundo), como se ha dicho anteriormente si solo usan un ordenador es probable que sus tiempos sean incorrectos. Esto puede conducir a errores de replicación, resultados espurios y conclusiones cuestionables. Actualmente existen en el mercado dispositivos (*pad* USB) muy costosos con precisión en milisegundos como el que se puede encontrar en la web (3). El objeto del presente proyecto es la realización de un dispositivo HID con dos botones que mejore la precisión (en decimas de milisegundo) de los dispositivos ya existentes, además de añadir un pulsioxímetro para que el especialista disponga de más información para la realización de su análisis, esto es, el especialista también podrá saber la pulsación del paciente y el porcentaje de oxígeno en sangre del mismo en cada instante del test.

No es objeto del presente proyecto la realización de circuito impreso y encapsulado así como la realización de un software informático que sirva de interfaz entre el dispositivo y el especialista ya que se trata de un proyecto final de carrera en el que el director determinó que el objetivo principal era realizar un dispositivo USB HID con *data-logger* al que se le podría añadir un pulsioxímetro como suplemento.

El interfaz entre el usuario y el dispositivo que ofrece el presente proyecto es el proporcionado por cualquier programa que tenga acceso al puerto serie del PC (Matlab, programas de terminal, etc.). Se entiende que este no es un interfaz ideal para un psicólogo o neurólogo normal que no tendría por qué saber utilizar estos programas, pero como ya se ha comentado en el párrafo anterior, no es objeto de este proyecto.

Finalmente se decidió medir únicamente la pulsación ya que se carecía de los medios necesarios para verificar si la medida del oxígeno en sangre era correcta, no obstante, en la memoria se indicarán tanto la teoría como la implementación que se debería usar en el microcontrolador para la obtención de esta medida, pero no habrá nada en el apartado de prueba ya que finalmente se decidió no incluirlo en el proyecto.

1.1 Objetivos concreto:

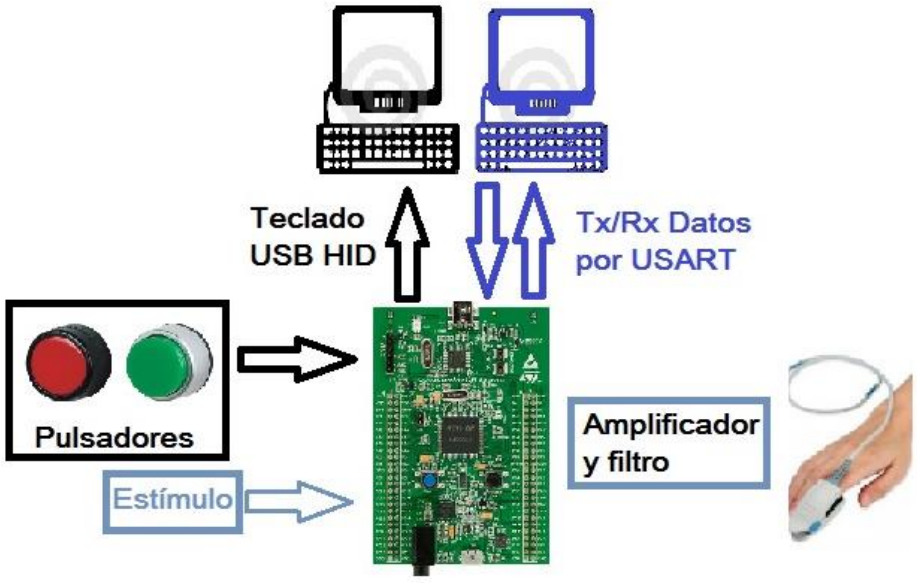
- Implementar el microcontrolador como teclado USB HID.
- Añadir dos pulsadores externos a la placa de pruebas que envíen un determinado carácter al ser pulsados.
- Implementar una consola de control en el microcontrolador con la que se pueda trabajar mediante puerto serie.
- Almacenar, leer y borrar la información de los eventos en la memoria Flash del microcontrolador.
- Sincronizar relojes.
- Adecuar la señal procedente de una sonda estándar de pulsioxímetro a los niveles de tensión y corriente del de la placa Discovery STM32F407D.
- Muestrear la señal procedente de la sonda de pulsioxímetro después de ser filtrada y amplificada.
- Realizar la FFT (transformada rápida de Fourier) de la señal muestreada y obtener la frecuencia cardiaca.

2 Background

El objeto del presente proyecto, como ya se ha comentado en el apartado de introducción, es la realización de un dispositivo HID con dos botones que mejore la precisión de los dispositivos para análisis psicológicos ya existentes, además de añadir un pulsioxímetro para que el especialista disponga de más información para la realización de su análisis.

Para ello se utiliza un microcontrolador STM32-F407 de ARM, ya que es un dispositivo de bajo coste y gran precisión con el que incluso se podría conseguir una precisión mayor a decimas de milisegundo.

Con esta información ya podemos ir modelando a grandes rasgos cómo será nuestro proyecto (en su hardware) y representarlo en un diagrama de bloques:



Teclado USB HID y pulsadores

La placa se conecta al PC mediante USB HID *Device* como *keyboard*.

Se han incluido dos pulsadores:

-Botón1 en PD14

-Botón2 en PA0

El microcontrolador testea los pulsadores mediante *polling*.

Al pulsar se envía un *keyboard report* USB con la tecla que esté asignada en ese momento en el pulsador, se ha añadido un retardo de 0.1s para quitar el "rebote" del pulsador, 0.1s es el tiempo que se ha considerado apropiado para el correcto funcionamiento después de testearse.

Cuando se deja de pulsar se envía un *keyboard report* con el valor 0x00 (no *key*).

Consola de control USART

Las órdenes de la consola están programadas en el microcontrolador y podemos usar cualquier tipo de programa que trabaje con puerto serie (Matlab, programas de terminal, etc.) como interfaz para interactuar con la consola.

Se han implementado las siguientes órdenes que se explicaran con detalle en el apartado de diseño e implementación:

-*SetKeyX*

-*GetKey*

-*echo*

-*h*: Para sincronizar relojes.

-*t*: Muestra tabla de eventos.

-*write*: Guarda datos en la flash.

-*clear*: Borra la memoria flash.

-*read*: Lee la memoria flash.

-*a*: Activa pulsímetro.

-*p*: Muestra la frecuencia cardiaca actual.

-*s*: Para pulsímetro.

Amplificador y filtro

La señal proporcionada por el foto-transistor es muy pequeña por lo que debemos filtrarla y amplificarla. En el proceso de filtrado se quitan las frecuencias que no nos interesan y que pueden interferir en la medición, por ejemplo, la frecuencia generada por la red eléctrica. En el proceso de amplificado la señal se aumenta a los niveles en los que el microcontrolador pueda trabajar con la señal.

Sonda Pulsioxímetro

Para tomar las medidas se decide usar una sonda de medición Nellcor desechable con el fin de abaratar los costes de desarrollo del proyecto, ya que la conexión y pinout siguen un estándar seguido por la gran mayoría de fabricantes, el usuario final podrá usar cualquier tipo de sonda de pulsioxímetro que solo tendrá que conectar a el conector DB-9 del dispositivo. Esta sonda contiene en su interior dos diodos LED, uno trabaja en 660 nanómetros (luz roja visible) y otro a 920 nanómetros (Infrarrojos). También dispone de un foto-transistor el cual es saturado en su base mediante luz, produciendo en su emisor una señal semejante a la de la base.

3 Diseño y Análisis

3.1 Tx/Rx Datos, Comunicación Unidad de proceso - PC (consola de control)

Para la transmisión de datos que necesitamos entre el microcontrolador y el PC necesitamos un sistema de comunicación que nos sea compatible con lo que buscamos, a la vez que nos resulte económico pero que no comprometa la transmisión, tanto en velocidad como en pérdida de datos. Vamos a barajar diversas opciones.

Estándar RS-232: Este estándar ha sido el más común para los puerto serie de un PC. Está pensado para comunicaciones Punto a Punto half-duplex o full duplex, en otras palabras el ordenador (maestro) comunica con el elemento que realiza la función industrial (en nuestro caso el microcontrolador) y pueden comunicarse en los dos sentidos simultáneamente o no.

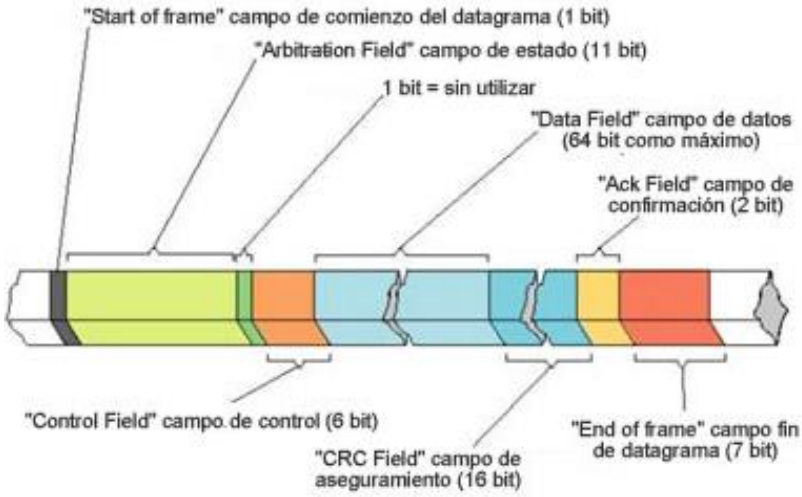
Ejemplo del cable RS-232

También existen los estándar RS-422 y RS-485, pero están más pensados en la utilización de varios nodos controlados por un PC, que no es nuestro caso. A continuación podemos ver una tabla comparativa.

Specifications	RS-232	RS-422	RS-485
Mode of Operation	Single-Ended	Differential	Differential
Total Number of Drivers and Receivers on One Line (One driver active at a time for RS-485 networks)	1 Driver 1 Receiver	1 Driver 10 Receivers	32 Drivers 32 Receivers
Maximum Cable Length	50 ft. (2500 pF)	4000 ft.	4000 ft.
Maximum Data Rate (40 ft.-4000 ft. for RS-422/RS-485)	20kB/s (by spec-can be higher)	10 Mbits/s	10 Mbits/s

Tabla comparativa RS-232, RS-422 y RS-485

Bus CAN: El Bus CAN (Controller Area Network) es un protocolo de comunicaciones basado en una arquitectura de bus para transferencia de mensajes en ambientes distribuidos, es decir, donde no existe ningún tipo de jerarquía entre los nodos. Este protocolo también está más destinado a aplicaciones con varios nodos, el cual como hemos comentado, no es nuestro caso.



Formato de la trama de transmisión con el protocolo Bus CAN

Universal Serial Bus (USB): Es el estándar más utilizado hoy en día para conectar cualquier periférico a los ordenadores. Su velocidad de transferencia más baja es de 1'5 Mbps (192 KB/s) .Actualmente su precio es muy reducido por lo tanto es muy bueno para cualquier aplicación en general.

Otros: Actualmente existen muchísimos modos de comunicar el microcontrolador con el PC como por ejemplo: red Ethernet, Wi-Fi, ZigBee, Bluethoot, Infrarrojos… Pero no los vamos a comentar porque supone un costo extra considerable, además de que necesitaría bastantes recursos del PC y del microcontrolador. No nos hace falta, ya que nuestro sistema no es demasiado exigente en esta parte de transmisión de datos.

Una vez analizados las distintas opciones que tenemos debemos mencionar que el USB probablemente sería la mejor opción a día de hoy, pero eso supondría que el tiempo de desarrollo sería mucho mayor por la complejidad de dicho estándar. Por tanto, para facilitar el diseño, hemos elegido el estándar RS-232 para que comunique el PC con el microcontrolador.

Para la conexión con el PC se ha utilizado un convertidor serie a USB por comodidad ya que es más usual encontrar un conector USB que un RS-232 en los ordenadores actuales. También se podría haber usado una conexión USB pero es mucho más sencillo de implementar la conexión serie y con un simple convertidor obtenemos los mismos resultados.

El conversor que se ha utilizado es Arduino USB 2 Serial Converter.

Esta placa convierte la conexión USB en los 5 voltios TX y RX que se pueden conectar directamente a la

placa, a través de un cable compatible FTDI. Cuenta con Atmega8U2 programado como un convertidor desde USB a serial. El módulo USB serial dispone de una interfaz ISCP, lo que permite reprogramar el chip cuando se coloca en modo DFU.

Los pines del conector son compatibles con el estándar FTDI.

Es necesario instalar un driver para que funcione. El driver es un archivo *.inf* que se puede encontrar en la página de arduino.

3.2. Microcontrolador

El microcontrolador es el elemento principal del proyecto. Para la realización del presente proyecto se plateaba la duda de usar un microcontrolador PIC o un ARM, la elección fue utilizar un dispositivo ARM. Ya se tenía experiencia previa con ambas familias de microcontroladores pero el factor determinante de la decisión fue que en la asignatura de Ingeniería de Telecomunicaciones SEA (Sistemas Electrónicos Avanzados) en la parte de microcontroladores la docencia estaba enfocada a dispositivos ARM y puesto que el presente proyecto es un proyecto de final de carrera para Ingeniería en Telecomunicaciones se ha decidido utilizar un microcontrolador de la familia ARM.

STM32 es una familia de microcontroladores de 32-bits de STMicroelectronics, basada en el procesador ARM ® Cortex®-M, está diseñada para ofrecer nuevos grados de libertad para los usuarios de MCU. Ofrece una gama de productos de 32 bits que combina alto rendimiento, tiempo real, procesamiento de señales digitales, y bajo consumo de energía, operación de baja tensión, mientras se mantiene la plena integración y facilidad de desarrollo.

La amplia gama de dispositivos STM32, basado en un núcleo estándar en la industria y acompañada de una amplia selección de herramientas y software, hace que esta familia de productos sea la elección ideal para pequeños y grandes proyectos.

> Los chips STM32 se agrupan en series relacionadas que se basan en el mismo núcleo de 32-bit del procesador ARM, como el Cortex-M7, Cortex-M4F, Cortex-M3, Cortex-M0 + o el Cortex-M0. Internamente, cada microcontrolador está compuesto de un núcleo procesador, memoria RAM estática, memoria flash, interfaz de depuración, y varios periféricos.

La familia STM32 se compone de siete series de microcontroladores: F4, F3, F2, F1, F0, L4, L1, L0, W. Cada serie de microcontroladores STM32 se basa en un procesador de ARM Cortex-M4F, Cortex-M3, Cortex-M0 + o Cortex-M0. El Cortex-M4F es conceptualmente un Cortex-M3 al que se añade DSP e instrucciones de precisión simple y coma flotante.

STM32 F4

STM32 F4 Series	
Producción	desde 2011 hasta la actualidad
Max. CPU clock rate	84 a 180 MHz
Instrucción set	Thumb, Thumb-2, Sat Math, DSP, FPU
Microarquitectura	ARM Cortex-M4F

El F4-series STM32 está basado en el núcleo ARM Cortex-M4F. La serie F4 es también la primera serie STM32 en tener DSP e instrucciones de coma flotante. El F4 es compatible pin-to-pin con el F2 y añade una mayor velocidad de reloj, RAM estática de 64K CCM RAM, I²S full duplex, mejora de reloj en tiempo real, y ADCs más rápidos.

Core:

Procesador ARM Cortex-M4F a una velocidad de reloj máxima de 84/168/180 MHz.

Memoria:

RAM estática: compuesta de hasta 192 KB de uso general, 64 KB de memoria acoplada de núcleo (CCM), 4 KB respaldados por batería, 80 bytes respaldados por batería con borrado ante la detección de manipulaciones.

Flash: consta de 512/1024/2048 KB de propósito general, 30 KB de arranque del sistema, 512 bytes programables una sola vez (OTP), 16 bytes opcionales.

Periféricos:

Los periféricos comunes incluidos en todos los paquetes de circuitos integrados son USB 2.0 OTG HS y FS, dos CAN 2.0B, un SPI + dos SPI o I²S full-duplex, tres I²C, cuatro USART, dos UART, SDIO para tarjetas SD / MMC, doce temporizadores de 16-bits, dos temporizadores de 32 bits, dos temporizadores watchdog, sensores de temperatura, 16 o 24 canales en tres ADCs, dos DAC, 51-140 GPIO, dieciséis DMA, mejora de reloj en tiempo real (RTC), comprobación de redundancia cíclica (CRC), generador de números aleatorios (RNG). Los paquetes de circuitos integrados más grandes añaden 8 / 16 bits de capacidad de bus de memoria externa.

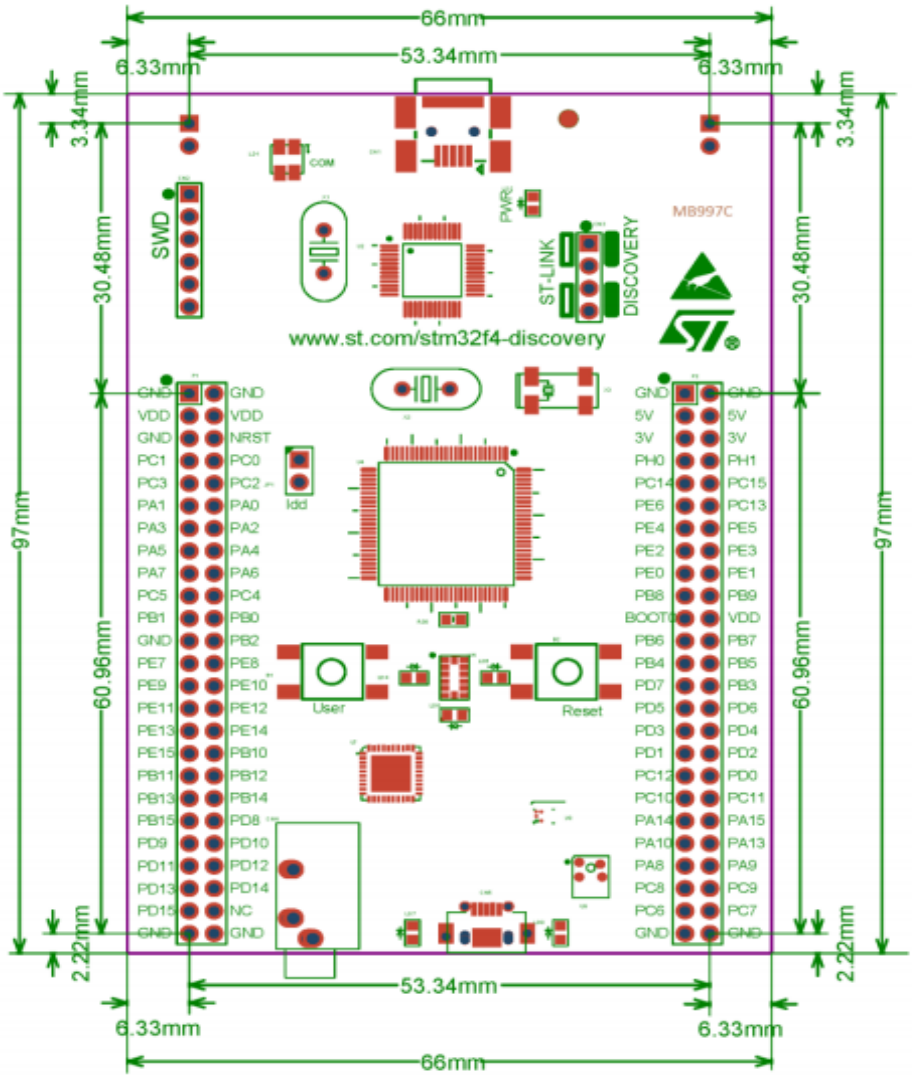
- Los modelos STM32F4x7 añaden Ethernet MAC e interfaz de la cámara.
- Los modelos STM32F41x / 43x añaden un procesador criptográfico para DES / TDES / AES, y un procesador de hash para SHA-1 y MD5.
- Los modelos STM32F4x9 añaden un controlador de LCD-TFT.

Osciladores internos (16 MHz, 32 kHz), opcional externo (4 a 26 MHz, 32,768 a 1000 kHz).

Paquetes IC: WLCSP64, LQFP64, LQFP100, LQFP144, LQFP176, UFBGA176. STM32F429 / 439 también ofrece LQFP208 y UFBGA216.

La tensión de funcionamiento es de 1.8 hasta 3.6 voltios.

Se ha decidido usar la placa Discovery STM32F407 por motivos que se explicaran más adelante. La placa Discovery STM32F407 viene en un formato de circuito impreso con dimensiones de una longitud igual a 97mm y con anchura de 66mm, lo que la hace de fácil implantación por su reducido tamaño. En la siguiente imagen podemos observar un esquemático de la placa y sus dimensiones.



Esquemático de la placa STM32F4 Discovery.

La conexión se podrá realizar a través de los 100 pines de los que dispone el dispositivo y que están provistas de diferentes señales del microprocesador, en este caso un ARM 32-bit Cortex M4, y también mediante las dos conexiones USB 2.0 de que dispone, así como para el caso que fuera necesario, de conexión Ethernet MAC. Al conectarse mediante USB el consumo de energía será bastante pequeño, alrededor de 5V. El funcionamiento del dispositivo no debe verse influenciado por las condiciones climatológicas de temperatura, ya que las características del mismo no difieren de algún otro aparato electrónico que esté en uso hoy en día, teniendo un rango de valores de temperatura o humedad similar al normal en estos casos.

En lo referente a la programación del microprocesador, podemos encontrar diferentes entornos de desarrollo, quedando a nuestra elección cuál de ellos usar, así como un completo modo debug. Para nuestro caso, ante las diferentes posibilidades ofrecidas se usará el entorno IAR por experiencia previa.

Por último y no menos importante, centrándonos en el aspecto económico, comparando los diferentes precios de los dispositivos disponibles en el mercado, pudimos observar el bajo coste que supone la adquisición de una placa del modelo elegido y que ronda los 14 €, un precio más que asumible por la empresa o particular que quisiera llevar a cabo el proyecto.

Hasta aquí hemos ido satisfaciendo los requisitos que creíamos indispensables cubrir para la correcta elección del hardware, aun así podemos incluir otros aspectos favorables que nos hicieron tomar esta decisión:

1. El programador aporta una gran biblioteca de manejo de periféricos, al instalarlo se puede acceder al código de estos programas de manera sencilla, lo que hace que el aprendizaje del manejo del mismo sea mucho más rápido, habiendo también tutoriales y ejemplos prácticos en la página del fabricante, así como herramientas software de gran utilidad.

2. Al contener todos los elementos de los que queríamos disponer, de una manera sencilla podemos hacer una primera aproximación al prototipo, con lo que se puede ver *in situ*, un funcionamiento aproximado al que tendría en el momento de su implantación.

3. Al tratarse de un microprocesador insertado en una placa con una gran diversidad de periféricos, lo que le da mucha versatilidad, y combinado con su reducido coste, se trata de un dispositivo muy popular, por lo que la ayuda en línea, por medio de foros o blogs está muy extendida y es de fácil consulta.

4. Ya se disponía de la placa, la cual se había comprado previamente para utilizarla en la asignatura Sistemas Electrónicos Avanzados y en otros proyectos, por lo que la compra no fue necesaria y además ya se tenía cierta experiencia con ella, lo cual agilizaría considerablemente el proyecto.

Todas estas razones hicieron de la placa *STM32F4 Discovery* la elección óptima para la creación y programación de nuestro proyecto.

3.3Teclado USB HID

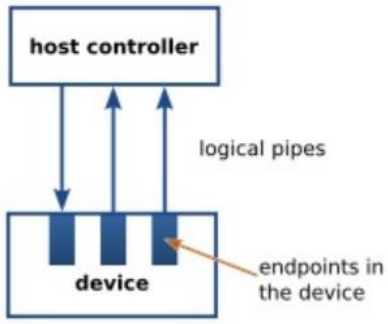
En este apartado se exponen todos los datos teóricos necesarios para la implementación de un dispositivo USB HID en un mincrocontrolador. Se comenzará con una breve introducción y algunas definiciones, seguido de un resumen del documento "*Device Class Definition for Human Interface Devices (HID)*" (4) incluyendo y resumiendo los apartados que guardan relación con el objetivo del presente proyecto.

Por último se incluye un resumen del documento "*SMT32F USB On-The-Go host and device library"* (5), este documento es un manual de usuario que proporciona STMicroelectronics para implementar los microcontroladores STM32F como USB, en dicho apartado se incluye un resumen de los apartados del documento que guardan relación con el presente proyecto.

3.3.1 INTRODUCCIÓN

Antes de entrar en el desarrollo propiamente dicho, es necesario explicar algunos de los fundamentos en los que se basa USB, ya que es necesario conocer ciertos detalles sobre la comunicación y el proceso de enumerado.

3.3.1.1. Endpoints y pipes



El flujo de datos que se produce entre el *Host* y un dispositivo USB se organiza de forma lógica en pipes o tuberías. Una tubería es un canal lógico de datos entre ambos. Estas se conectan entre un *endpoint* o punto final del dispositivo y la capa de software que controla el hardware USB en el *Host*. Los endpoints vienen configurados de fábrica, están numerados de forma única por cada dispositivo y se agrupan en conjuntos que se llaman interfaces, de manera que el control de un periférico USB puede llevarse a cabo mediante la comunicación a través de varios *endpoints*.

Físicamente, el *Host* se comunica con los dispositivos cada 1 ms en USB 1.x y cada 125 µs en USB 2.0 (tiempo de trama USB) a través del par de señales diferenciales D+/D- que pasan por el cable. A nivel lógico el software USB configura y administra los dispositivos mediante la tubería por defecto de control. Esta tubería utiliza el *endpoint* 0, que debe estar soportado por todos los dispositivos USB para su funcionamiento. Durante el proceso de enumeración que tiene lugar inmediatamente después de conectar el periférico, el *Host* obtiene a través de la tubería por defecto de control la información del dispositivo y el resto de puntos finales disponibles en él. Finalmente le asigna una dirección única en el bus. En el nivel más alto, el software o driver que hace uso del dispositivo USB se comunica con él a través de las tuberías conectadas a puntos finales. Como ejemplo se puede definir un dispositivo de almacenamiento en el que se dedique un *endpoint* para el envío de comandos de lectura y escritura, y otro para la transferencia de datos.

3.3.1.2 Descriptores

Los descriptores contienen toda la información que el *Host* necesita para identificar al dispositivo y sus características, y así ser capaz de buscar y cargar el driver adecuado para su puesta en funcionamiento. Los descriptores se organizan en una jerarquía que contiene los siguientes elementos:

• Descriptor de dispositivo: Indica la versión de USB con la que es compatible el dispositivo, el código del fabricante (asignado por el USB-IF), y el código del producto. También el número de descriptores de configuración presentes.

• Descriptor de configuración: Una configuración es un modo de operación del dispositivo (por ejemplo una cámara podría operar como webcam o como cámara de fotos). El *Host* debe seleccionar una configuración y sólo puede haber una activa. Este descriptor especifica la potencia requerida, y el número de interfaces asociados con esa configuración. En la práctica son raros los dispositivos que ofrezcan más de una configuración.

• Descriptor de interfaz: Estos descriptores contienen el número de puntos finales asociado con el interfaz, y la clase de dispositivo a la que pertenece (almacenamiento masivo, HID, impresión,...).

• Descriptor de *endpoint*: Especifica el número de *endpoint*s, el tipo de transferencias que se realizarán a través de él (interrupción, masiva, isócrona,...), su longitud y su dirección desde el punto de vista del *Host*: (*IN*=Del dispositivo al *Host*, *OUT*=Del *Host* al dispositivo). El punto final 0 es necesario y predefinido, por tanto no necesita descriptor.

• Descriptor de *string*: Contiene una cadena de texto que puede ser referenciada por los descriptores anteriores, ejemplos comunes de cadenas son el nombre del fabricante o el número de serie.

3.3.2 Definición de la clase USB *Device* para HID (4)

El bus universal serie (USB) es una arquitectura de comunicaciones que da a los ordenadores personales (PC) la capacidad de interconectar una gran variedad de dispositivos.

USB es un enlace de comunicación serie cuya velocidad de transmisión puede ser de 1.5 o 12 megabits por segundo (mbs). Los protocolos USB pueden configurar dispositivos en el arranque (*startup*) o cuando ya están en funcionamiento, en tiempo de ejecución. Estos dispositivos están distribuidos en varias clases de dispositivos. Cada clase de dispositivo define el comportamiento y protocolos comunes para los dispositivos que sirven funciones similares. Algunos ejemplos de dispositivos USB se muestran a continuación:

Clase de dispositivo	Ejemplo de dispositivo
Display	Monitor
Comunicación	Modem
Audio	Altavoces
Almacenamiento masivo	Disco duro
Interfaz Humana	*Data glove*

En relación al presente proyecto se describirá a continuación la clase de Interfaz Humana o *Human Interface Device* (HID) para su uso con USB.

La clase HID consiste principalmente en dispositivos que son usados por humanos para el control de operaciones en sistemas informáticos. Algunos ejemplos típicos de la clase HID son:

- Teclados, ratones y *joysticks*.
- Paneles de control: Mandos rotatorios, botones y controles deslizantes.
- Controles para teléfonos, controles remotos VCR, juegos o dispositivos de simulación: *Data gloves*, volates y pedales de simulación.

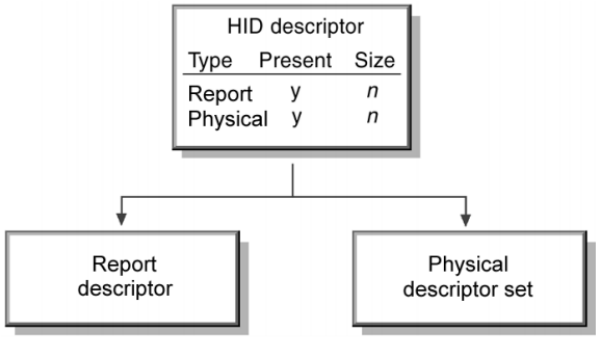
3.3.2.1 Descripción General

La información sobre un dispositivo USB se almacena en segmentos de su ROM (memoria de sólo lectura). Estos segmentos son llamados descriptores. Un descriptor de interfaz puede identificar un dispositivo como perteneciente a uno de un número finito de clases. La clase HID es el objetivo principal de este documento.

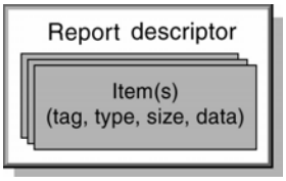
Un dispositivo de clase USB/HID utiliza su correspondiente driver de clase HID para recuperar y encaminar todos los datos.

El enrutamiento y la recuperación de los datos se llevan a cabo mediante el examen de los descriptores del dispositivo y los datos que proporciona.

El descriptor del dispositivo de clase HID identifica qué otros descriptores de clase HID están presentes e indica su tamaño. Por ejemplo, *report* y descriptores físicos.



Un **descriptor de *report*** describe cada fragmento de datos que el dispositivo genera y lo que los datos están realmente midiendo.



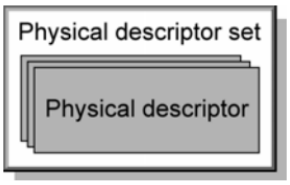
Por ejemplo, un descriptor de *report* define los *ítems* (elementos) que describen una posición o un estado de un botón. Los *item* de información se utiliza para:

* Determinar dónde enrutar una entrada, por ejemplo, enviar una entrada al ratón o joystick de una API.

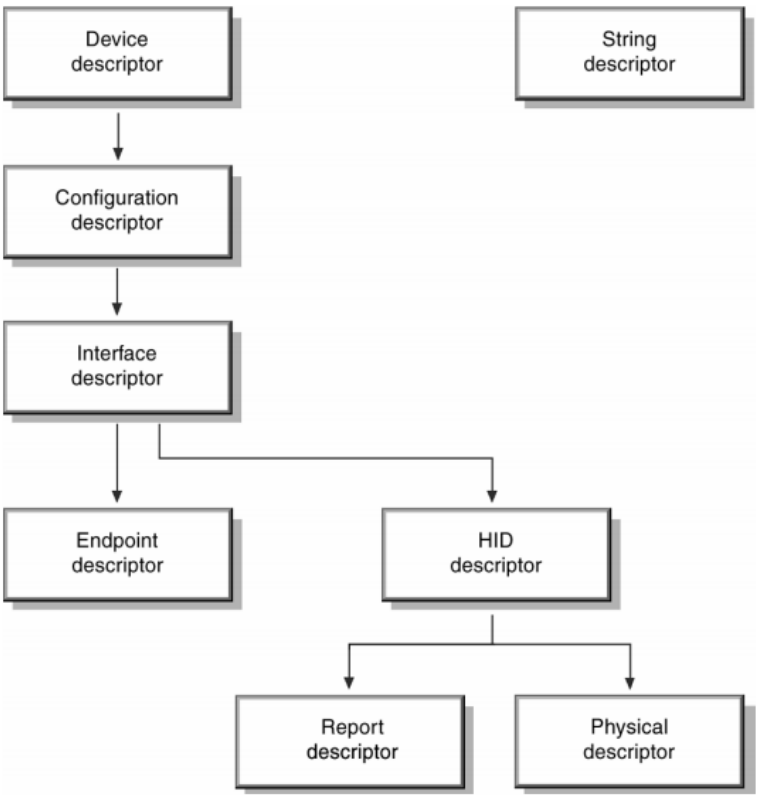
* Permitir al software asignar funcionalidad a una entrada – por ejemplo, utilizar una entrada de joystick para posicionar un tanque.

Mediante un examen de elementos (descriptor *report*) el driver de clase HID es capaz de determinar el tamaño y la composición de los *reports* de datos desde el dispositivo de clase HID.

Los **descriptores físicos** son descriptores opcionales que proporcionan información sobre la parte o partes del cuerpo humano utilizada para activar los controles de un dispositivo.



Todas estas cosas se pueden combinar para ilustrar la estructura del descriptor.



El resto de este apartado documenta los detalles de implementación, advertencias y restricciones para el desarrollo de dispositivos de clase HID y drivers.

3.3.2.2 Características Funcionales.

Esta sección describe las características funcionales de HID:

- Clase
- Subclase
- Interfaces

La clase HID

Los dispositivos USB están divididos en clases de dispositivos que:

- Tienen requisitos de transporte de datos similares.
- Comparte un único driver de clase.

Un dispositivo USB puede tener un único tipo de clase o puede estar compuesta de varias clases. Por ejemplo, un aparato telefónico podría utilizar las funciones de HID, Audio y clases

de telefonía. Esto es posible porque la clase se especifica en el descriptor de interfaz y no en el descriptor de dispositivo.

Las especificaciones de *USB Core* definen el código de clase HID. El miembro *bInterfaceClass* de un descriptor de interfaz es siempre 3 para los dispositivos de clase HID.

Subclase

Al comienzo del desarrollo de las especificaciones del HID, las subclases fueron destinadas a ser utilizados para identificar los protocolos específicos de diferentes tipos de dispositivos de clase HID. Si bien esto refleja el modelo actualmente en uso por la industria (todos los dispositivos utilizan protocolos definidos por los dispositivos populares similares), rápidamente se hizo evidente que este enfoque era demasiado restrictivo. Es decir, los dispositivos necesitarían encajar en subclases estrictamente definidas y no serían capaces de proporcionar ninguna funcionalidad más allá de un apoyo para la subclase.

Muchos dispositivos conocidos tienen múltiples clasificaciones, por ejemplo, teclados con localizadores, o localizadores que proporcionan las pulsaciones de teclado. En consecuencia, la clase HID no utiliza subclases para definir la mayoría de los protocolos. En cambio, un dispositivo de clase HID identifica su protocolo de datos y el tipo de datos facilitados en su descriptor de *report.*

El descriptor *report* es cargado y analizado por el driver de clase HID tan pronto como se detecta el dispositivo. Los protocolos para los dispositivos existentes y nuevos se crean mezclando tipos de datos dentro del descriptor *report*.

El miembro *bInterfaceSubClass* declara si un dispositivo es compatible con una interfaz de inicio, de lo contrario es 0.

Códigos de subclase:

0 → No hay subclase

1 → Interfaz de subclase de arranque.

2-255 → Reservado.

Protocolos

Una gran variedad de protocolos son soportados por los dispositivos HID. El miembro *bInterfaceProtocol* de un descriptor de interfaz sólo tiene sentido si el miembro *bInterfaceSubClass* declara que el dispositivo es compatible con una interfaz de inicio, de lo contrario es 0.

Códigos de Protocolo:

0 → No

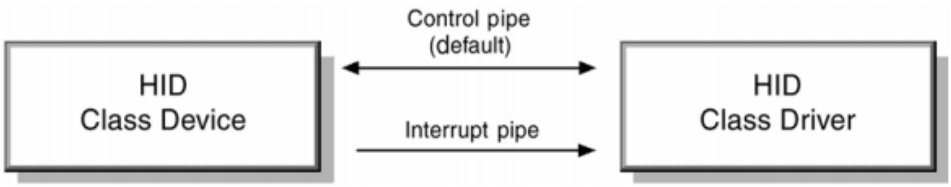
1 → Teclado

2 → Ratón

3-255 → Reservado

Interfaces

Un dispositivo de clase HID se comunica con el driver de clase HID utilizando una tubería de control (por defecto) o una tubería de interrupción.



- La tubería de control se utiliza para:
  - Recibir y atender las solicitudes de control de USB y datos de la clase.
  - La transmisión de datos cuando es consultados por el driver de clase HID (utilizando la solicitud *Get_Report*).
  - Recepción de datos desde el *Host*.
- La tubería de interrupción se utiliza para:
  - Recepción de datos asíncrona (no solicitada) desde el dispositivo.
  - La transmisión de datos de baja latencia en el dispositivo.

Nota: *Endpoint 0* es una tubería de control que siempre está presente en los dispositivos USB. Por lo tanto, sólo la tubería de interrupción de entrada esta descrita por el descriptor de interfaz utilizando un descriptor *Endpoint*. De hecho, varios descriptores de interfaz pueden compartir *Endpoint 0*. La interrupción de salida es opcional y requiere un descriptor *Endpoint* adicional.

Tubería	Descripción
Control (Endpoint 0)	Control USB, código de petición de clase, datos consultados (datos de mensaje).
Interrupción Entrada	Datos de entrada, datos del dispositivo (datos compartidos)
Interrupción Salida	Datos de salida, datos para el dispositivo (datos compartidos)

Limitaciones del dispositivo

Esta especificación se aplica tanto a alta velocidad como a baja velocidad en dispositivos de clase HID.

Cada tipo de dispositivo tiene varias limitaciones, como se define en el Capítulo 5 de "*The Universal Serial Bus Specification"* (6).

3.3.2.3 Modelo de operación

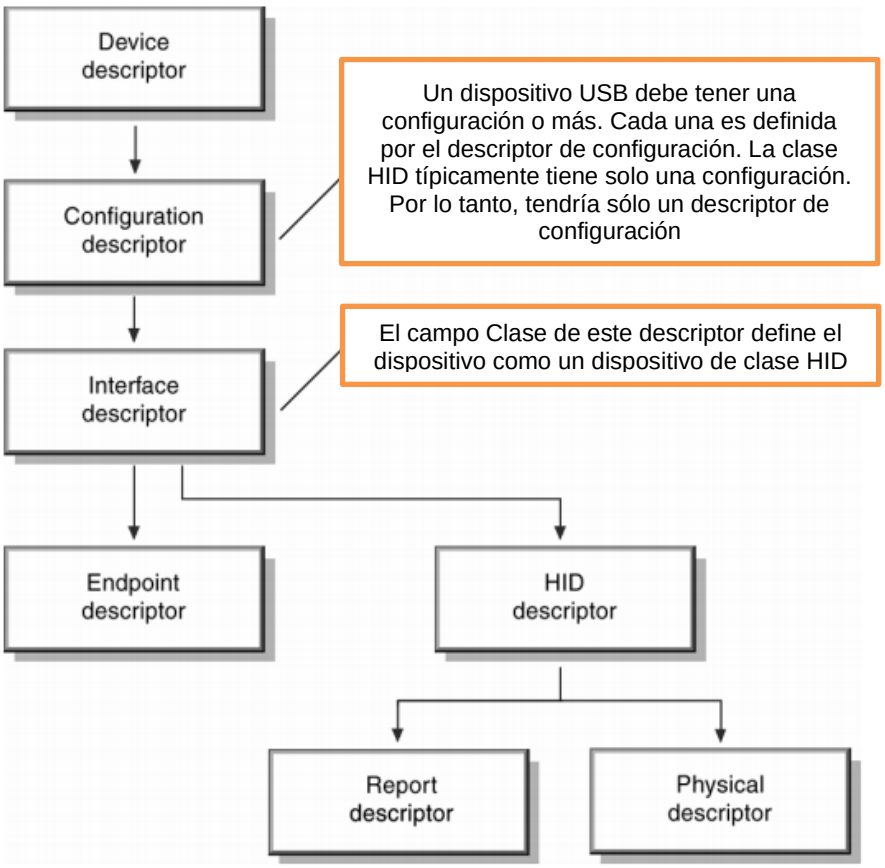
En esta sección se describe el modelo de funcionamiento básico de un dispositivo de clase HID. Los elementos del diagrama de flujo representan tablas de información con el *firmware*.

Estructura de descriptores del dispositivo.

En el nivel superior, un descriptor incluye dos tablas de información, el descriptor de dispositivo y el descriptor *String*. Un descriptor de dispositivos USB estándar especifica el ID del producto y otra información acerca del dispositivo. Por ejemplo, los campos de los descriptores de dispositivos incluyen principalmente:

- Clase
- Subclase
- Fabricante

- Producto
- Versión



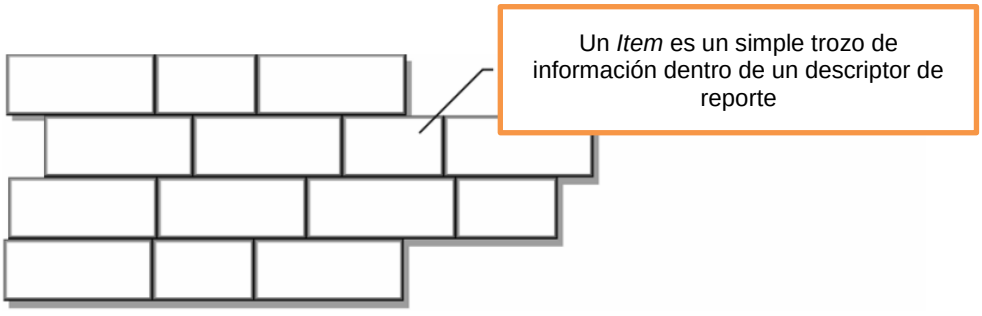
Para los dispositivos de clase HID:

- El tipo de la clase no se define en el nivel de descriptor de dispositivos. El tipo de clase para un dispositivo de clase HID se define por el descriptor de interfaz.
- El Campo subclase se utiliza para identificar los dispositivos de arranque.

Descriptores de *Report*

Estos descriptores se ilustran mediante *items* de un diagrama de flujo que representan tablas de información. Cada tabla de información puede ser considerada como un bloque de datos.

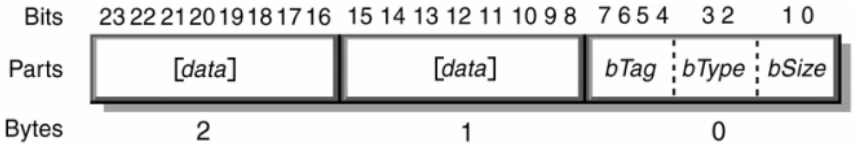
En lugar de un bloque de datos, los descriptores *report* están compuestos de piezas de información. Cada pieza de información se denomina como *Item*.



Formato genérico de los *Item*

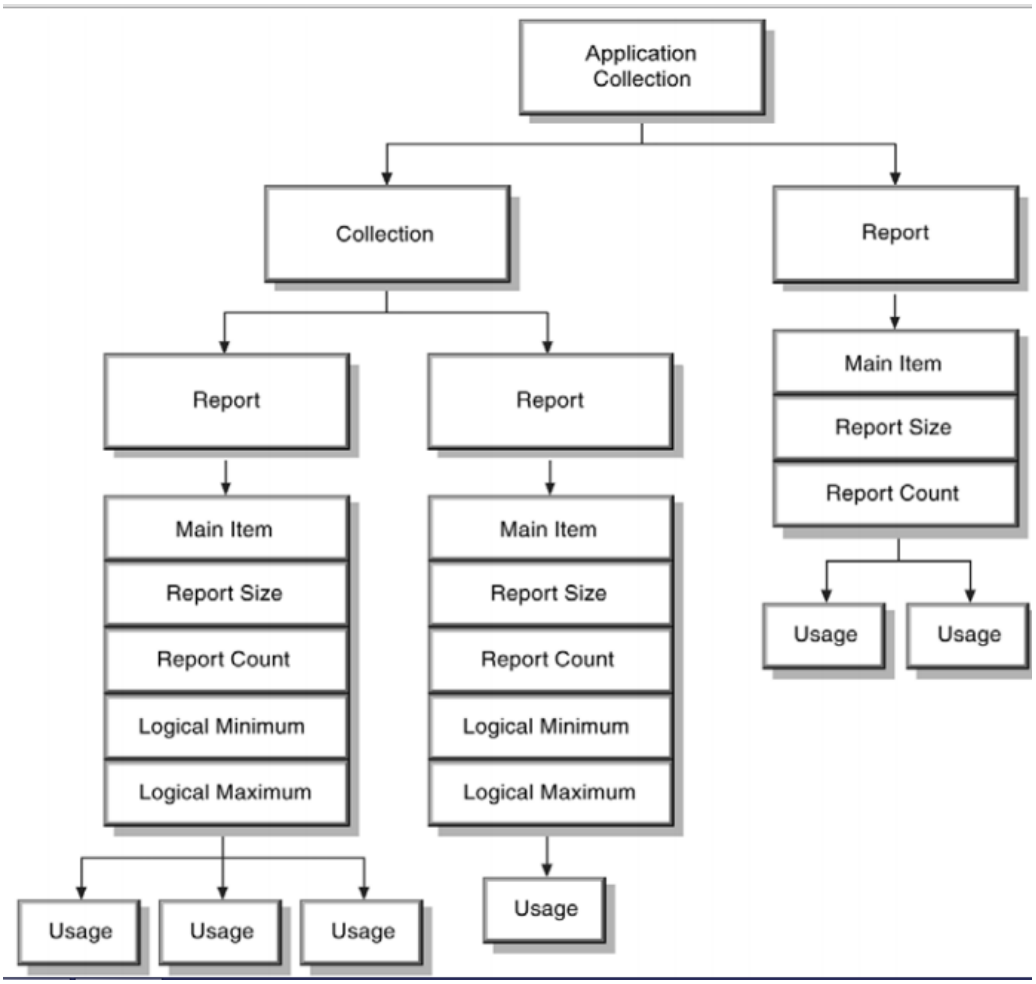
Un Item es un trozo de información del dispositivo. Todos los *Items* tienen un byte prefijado que contiene la etiqueta de ítem, tipo de ítem y tamaño de ítem.

Un ítem puede incluir un dato de ítem opcional. El tamaño de la porción de datos de un ítem está determinado por su tipo fundamental. Hay dos tipos básicos de ítems: cortos y largos. Si el ítem es corto, su tamaño de dato opcional puede ser de 0,1, o 4 bytes. Si se trata de un ítem largo su *bSize* es siempre 2. El siguiente ejemplo muestra posible valores para un *long ítem*.



Analizador de ítem

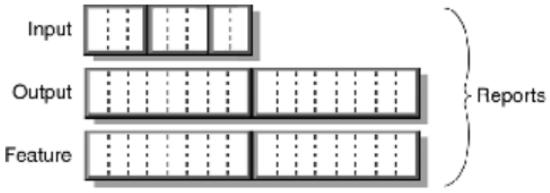
El driver de la clase HID tiene un analizador de ítems encontrados en el descriptor *report*. El analizador extrae información procedente del descriptor en forma lineal. El analizador recolecta el estado de cada ítem conocido como si caminara a través del descriptor y lo almacena en una tabla de estado de ítem. La tabla de estado de ítem contiene el estado de cada ítem individual. Desde el punto de vista del analizador, un dispositivo de clase HID se parece a lo que muestra la siguiente figura:



Cuando se encuentran algún item, el contenido de la tabla de estado del item se modifica. Estos items incluyen todos los *Main*, *Push*, y *Pop* items.

Reports

Utilizando la terminología USB, un dispositivo puede enviar o recibir una transacción cada *frame* USB (1 milisegundo). Una transacción puede estar compuesta de múltiples paquetes (*token*, datos, *handshake*), pero está limitado en tamaño a 8 bytes para dispositivos de baja velocidad y 64 bytes para dispositivos de alta velocidad. Una transferencia es una o más transacciones que crean un conjunto de datos que sean significativos para el dispositivo – por ejemplo, entrada, salida y *report* de características. Transferencia es sinónimo de *report*.



La mayoría de los dispositivos generan *reports*, o transferencias, devolviendo una estructura en la que cada campo de datos se representa de forma secuencial. Sin embargo, algunos dispositivos pueden tener múltiples estructuras de reporte en un solo *endpoint*, cada uno representando unos pocos campos de datos.

Por ejemplo, un teclado con un dispositivo de puntero integrado podría reportar de manera independiente datos "*key press*" y "apuntando" sobre el mismo *endpoint*. Los ítems de *Report ID* se utilizan para indicar qué campos de datos están representados en cada estructura de reporte. Una etiqueta de item de informe ID asigna un prefijo de identificación de 1 byte para cada reporte de transferencia. Si no hay etiqueta de ítem de *Report ID* en el descriptor *report*, se puede suponer que existe una estructura de entrada, salida, y existe la estructura de *report* de características y juntos representa todos los datos del dispositivo.

Si un dispositivo tiene múltiples estructuras de *report*, todas las transferencias de datos comenzarán con un prefijo identificador de 1 byte que indica que estructura de *report* se aplica a la transferencia. Esto permite al driver de clase distinguir datos de puntero entrantes de los datos del teclado mediante el examen del prefijo de transferencia.

Strings

Un recolección de campos de datos puede tener una etiqueta particular (índice de *string*) asociado con él. Las cadenas (*strings*) son opcionales.

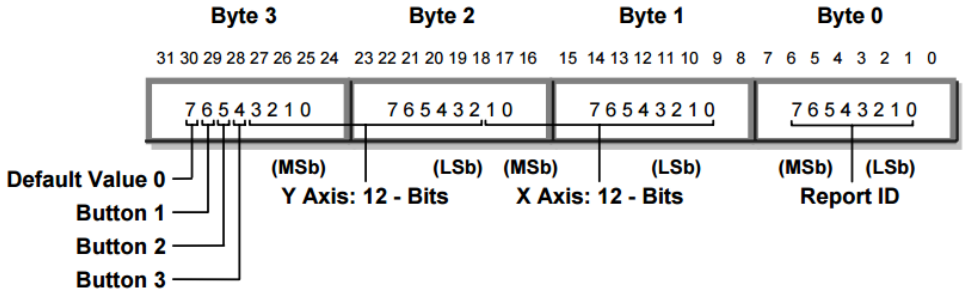
La etiqueta de uso de un elemento no es necesariamente lo mismo que una cadena asociada con el elemento principal. Sin embargo, las cadenas pueden ser útiles cuando se requiere un uso definido por el proveedor. El descriptor de cadena contiene una lista de cadenas de texto para el dispositivo.

Formato de valores numéricos Multibyte

Los valores numéricos de varios bytes se representan en los *reports* con formato *little-endian*, con el byte menos significativo en la dirección más baja. Los valores mínimos y máximos lógicos identifican el rango de valores que se encontró en un *report*. Si el valor lógico mínimo y máximo es positivo entonces es innecesario el bit de signo en el campo de *report* y el contenido de un campo puede ser asumido como un valor sin signo. De lo contrario, todos los valores

enteros son valores representados en formato de complemento a 2. No se permiten los valores de coma flotante.

El bit menos significativo en un valor se almacena en el bit 0, el siguiente bit más significativo en 1 y así sucesivamente hasta el tamaño del valor. El siguiente ejemplo ilustra una representación de bits con valores *long integer*.



Byte	Bits
0	0-7
1	8-15
2	16-23
3	24-31

3.3.2.4 Descriptores

3.3.2.4.1 Descriptores estándard

La clase dispositivo HID usa los siguientes descriptores USB estándard:

- Dispositivo
- Configuración
- Interface
- Endpoint
- String

3.3.2.4.2 Descriptores de clase especifica

Cada clase de dispositivo incluye un descriptor de clase especifica o más. Estos descriptores son distintos a los descriptores standard USB. La clase de dispositivo HID usa los siguientes descriptores de clase específica:

- HID
- *Report*
- Físico

Descriptores HID

El descriptor HID identifica la longitud y el tipo de descriptores subordinado para un dispositivo.

Parte	Offset/Tamaño (Bytes)	Descripción
bLenght	0/1	Tamaño total del descriptor HID
bDescriptorType	1/1	Tipo específico de descriptor HID
bcdHID	2/2	Identifica la clase HID
bCountryCode	4/1	Código de país del hardware
bNumDescriptors	5/1	Especifica el número de descriptor de clase
bDescriptorType	6/1	Identifica tipo de descriptor de clase.
wDescriptorLenght	7/2	Tamaño total del descriptor *report*
[bDescriptorType]	9/1	Especifica tipo de descriptor opcional
[wDescriptorLength]	10/12	Tamaño total de descriptor opcional

Descriptor de *report*

El descriptor de *report* se diferencia de otros descriptores en que no es simplemente una tabla de valores. La longitud y el contenido de un descriptor *report* varían en función del número de campos de datos necesarios para el r*eport* o *reports* del dispositivo. El descriptor *report* se compone de items que proporcionan información sobre el dispositivo. La primera parte de un item contiene tres campos: tipo de item, etiqueta item y tamaño del item. En conjunto, estos campos identifican el tipo de información que el item ofrece.

Hay tres tipos de items: *Main*, *Global* y *Local*. Hay cinco etiquetas de ítem actualmente definidas:

- Etiqueta de ítem de entrada: Se refiere a los datos de uno o más controles similares en un dispositivo. Por ejemplo, los datos variables tales como la lectura de la posición de un solo eje o un grupo de palancas o una matriz de datos como uno o más botones pulsadores o interruptores.
- Etiqueta de ítem de salida: Se refiere a los datos de uno o más controles similares en un dispositivo como la posición de un eje único o un grupo de palancas (datos variables). O bien, puede representar los datos a uno o más LEDs (matriz de datos).
- Etiqueta de ítem de características: Describe la entrada del dispositivo y la salida no destinada al consumo por el usuario final; por ejemplo, una función de software o de palanca de Panel de control.
- Etiqueta de ítem de colección: Una agrupación significativa de entrada, de salida e ítems de característica Por ejemplo, ratón, teclado, joystick, y puntero.
- Etiqueta de ítem de fin de colección: Un item de terminación utilizado para especificar el final de una colección de items.

El descriptor de *reports* proporciona una descripción de los datos proporcionados por cada control en un dispositivo. Cada etiqueta de un *Main item* (entrada, salida, o Característica) identifica el tamaño de los datos devueltos por un control en particular, e identifica si los datos son absolutos o relativos y otra información. Los ítem *locales* y *globales* definen los valores máximo y mínimo de datos. Un descriptor *report* es el conjunto completo de todos los ítems de un dispositivo. Al mirar a un descriptor *report*, una aplicación sabe cómo manejar los datos entrantes, así como para lo que los datos podrían ser utilizados.

Uno o más campos de datos de los controles se definen por un *Main item* y se describen adicionalmente por los *items Globales y Locales* precedentes. Los *ítem Locales* sólo describen los campos de datos definidos por el siguiente *Main item*. Los *ítem Globales* se convierten en los atributos predeterminados para todos los campos de datos subsiguientes en ese descriptor.

Descriptores físicos

Un descriptor físico es una estructura de datos que proporciona información sobre la parte específica o partes del cuerpo humano que están activando un control o controles.

Por ejemplo, un descriptor físico podría indicar que el pulgar de la mano derecha utiliza el botón 5. Una aplicación puede utilizar esta información para asignar funciones a los controles de un dispositivo.

NOTA: Los Descriptores físicos son totalmente opcionales. Añaden complejidad y ofrecen muy poco a cambio en la mayoría de los dispositivos. Sin embargo, algunos dispositivos, en especial, aquellos con un gran número de controles idénticos (por ejemplo: botones) encontrarán que los Descriptores físicos ayudan a diferentes aplicaciones asignando funciones a estos controles de una manera más consistente. Como en este proyecto no se utilizan descriptores físicos damos por terminado este apartado.

3.3.2.5 Solicitudes (*Requests)*

3.3.2.5.1 Solicitudes standard

La clase HID utiliza la solicitud estándar *Get_Descriptor* como se describe en las Especificaciones USB. Cuando se emite una solicitud *Get_Descriptor(Configuración)*, devuelve el descriptor de configuración, todos los descriptores de interfaz, todos los descriptores de *Endpoint*, y el descriptor HID para cada interfaz. No se devolverá el descriptor de *string*, descriptor de reporte HID o cualquiera de los descriptores opcionales de la clase HID. El descriptor HID se intercala entre la interfaz y los descriptores de *Endpoint* para interfaces HID. Es decir, el orden será:

```
Descriptor de configuración
  Descriptor de interfaz (especificando clase HID)
      Descriptor HID (asociado con interfaz anterior)
            Descriptor Endpoint (for HID Interrupt In Endpoint)
            Descriptor opcional Endpoint (para interrupciones de salida
            HID Endpoint)
```

Para obtener más información, consulte el Capítulo 9 de la especificación USB, *"USB Device Class Framework"*.

A continuación se describen brevemente los dos tipos de solicitudes estándar, para más información consultar el capítulo 7 de "*Device Class Definition for Human Interface Devices (HID)*" (4).

Solicitud ***Get_Descriptor:*** Devuelve un descriptor al dispositivo.

Part	Standard USB Descriptor	HID Class Descriptor
bmRequestType	100 *xxxxx*	10000001
bRequest	GET_DESCRIPTOR (0x06)	GET_DESCRIPTOR (0x06)
wValue	Descriptor Type and Descriptor Index	Descriptor Type and Descriptor Index
wIndex	0 (zero) or Language ID	Interface Number
wLength	Descriptor Length	Descriptor Length
Data	Descriptor	Descriptor

Solicitud ***Set_Descriptor:*** Deja los descriptores de cambio del host en los dispositivos.

Part	Standard USB Descriptor	HID Class Descriptor
bmRequestType	00000000	00000001
bRequest	SET_DESCRIPTOR (0x07)	SET_DESCRIPTOR (0x07)
wValue	Descriptor Type (high) and Descriptor Index (low)	Descriptor Type and Descriptor Index
wIndex	0 (zero) or Language ID	Interface
wLength	Descriptor Length	Descriptor Length
Data	Descriptor	Descriptor

3.3.2.5.2 Solicitudes de clase especifica

Las solicitudes de clase específica permiten al host informarse acerca de las capacidades y el estado de un dispositivo y para establecer el estado de la salida y los ítems de característica. Estas operaciones se realizan sobre la tubería por defecto, por lo que siguen el formato de las solicitudes de tubería por defecto como se define en las Especificaciones USB (6).

Part	Offset/Size (Bytes)	Description
bmRequestType	0/1	Bits specifying characteristics of request. Valid values are 10100001 or 00100001 only based on the following description: 7 Data transfer direction 0 = Host to device 1 = Device to host 6..5 Type 1 = Class 4..0 Recipient 1 = Interface
bRequest	1/1	A specific request.
wValue	2/2	Numeric expression specifying word-size field (varies according to request.)
wIndex	4/2	Index or offset specifying word-size field (varies according to request.)
wLength	6/2	Numeric expressions specifying number of bytes to transfer in the data phase.

Solicitud ***Get_Report***: permite al host recibir un *report* procedente de la tubería de control.

Part	Description
bmRequestType	10100001
bRequest	GET_REPORT
wValue	Report Type and Report ID
wIndex	Interface
wLength	Report Length
Data	Report

Solicitud ***Set_Report***: Permite al host enviar un *report* al dispositivo, estableciendo el estado de entrada, salida, o controles de característica.

Part	Description
bmRequestType	00100001
bRequest	SET_REPORT
wValue	Report Type and Report ID
wIndex	Interface
wLength	Report Length
Data	Report

Solicitud ***Get_Idle***: lee la tasa de inactividad actual de un *report* de entrada.

Part	Description
bmRequestType	10100001
bRequest	GET_IDLE
wValue	0 (zero) and Report ID
wIndex	Interface
wLength	1 (one)
Data	Idle rate

Solicitud ***Set_Idle***: silencia un *report* en tubería de interrupción de entrada hasta que un nuevo evento ocurra o finalice una cantidad de tiempo específico.

Part	Description
bmRequestType	00100001
bRequest	SET_IDLE
wValue	Duration and Report ID
wIndex	Interface
wLength	0 (zero)
Data	Not applicable

Solicitud ***Get_Protocol***: lee que protocolo está actualmente activo (ya sea el protocolo de arranque o el protocolo de *report*.)

Part	Description
bmRequestType	10100001
bRequest	GET_PROTOCOL
wValue	0 (zero)
wIndex	Interface
wLength	1 (one)
Data	0 = Boot Protocol 1 = Report Protocol

Solicitud ***Set_Protocol***: cambia entre el protocolo de arranque y el protocolo de *report* (y viceversa).

Part	Description
bmRequestType	00100001
bRequest	SET_PROTOCOL
wValue	0 = Boot Protocol 1 = Report Protocol
wIndex	Interface
wLength	0 (zero)
Data	Not Applicable

3.3.2.6 Protocolo de report

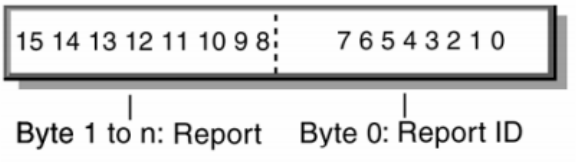
Tipos de reports

Los *reports* contienen datos de uno o más items. Las transferencias de datos son enviados desde el dispositivo al *Host* a través de la interrupción de la tubería de entrada en forma de *reports*. Los *reports* también se pueden solicitar (por *polling*) y se envían a través de la tubería de control o a través de una interrupción de la tubería de salida opcional. Un *report* contiene el estado de todos los elementos (de entrada, de salida o de característica) que pertenecen a un *report ID* particular. La aplicación de software se encarga de extraer los items individuales de *report* basado en el descriptor de *reports*.

La longitud de bits de los datos de un item se obtiene a través del descriptor *report* (Tamaño de report * contador de report). Los datos de item se ordenan al igual que los item se ordenan en el descriptor *report*. Si se utiliza una etiqueta ID de *repor*t en el descriptor *report*, todos los *reports* incluirán un único prefijo de ID de bytes. Si no se utiliza la etiqueta ID de *report*, todos los valores se devolverán en un único r*eport* y la ID prefijada no estará incluida en ese *report*.

Formato de report para ítems estándar

El formato del informe se compone de un identificador de *report* de 8 bits seguido por los datos que pertenecen a ese *report*.



Report ID: Este campo es de 8 bits. Si no se usa una etiqueta *Report ID* en el descriptor de *report*, solo habrá un r*eport* y *Report ID* se omitirá.

Report Data: Los campos de datos son campos de longitud variable que informan el estado de un item.

Formato de *report* para arrays de ítems

Cada botón en una matriz reporta un número asignado llamado índice de matriz. Esto se puede traducir en un código clave buscando los elementos de la matriz *Usage Page* y *Usage*. Cuando no haya ninguna transición entre botón abierto y cerrado, la lista completa de los índices para los botones actualmente cerrados en la matriz se transmite al host.

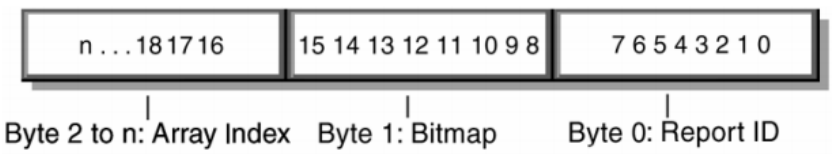
Como sólo se puede reportar un elemento de la matriz en cada campo de matriz, las teclas modificadas deben ser reportadas como datos de mapa de bits (un grupo de campos variables de 1 bit). Por ejemplo, las teclas como CTRL, MAYÚS, ALT y teclas GUI conforman el byte modificador de 8 bits en un *report* de teclado estándar. Aunque estos códigos de uso se definen en la Tabla de Uso como E0-E7, no se envía como datos de la matriz. El byte modificador se define como sigue.

Bit	Key
0	LEFT CTRL
1	LEFT SHIFT
2	LEFT ALT
3	LEFT GUI
4	RIGHT CTRL
5	RIGHT SHIFT
6	RIGHT ALT
7	RIGHT GUI

El siguiente ejemplo muestra los reports generados por un usuario que escribe ALT + CTRL + DEL, utilizando un mapa de bits para los modificadores y una única matriz para todas las demás teclas.

Transition	Modifier Byte	Array Byte
LEFT ALT down	00000100	00
RIGHT CTRL down	00010100	00
DEL down	00010100	4C
DEL up	00010100	00
RIGHT CTRL up	00000100	00
LEFT ALT up	00000000	00

Si hay varios *reports* para un dispositivo, cada *report* estará precedido por su *report* único de ID.



Si un conjunto de teclas o botones no pueden ser mutuamente exclusivos, deben ser representados ya sea como un mapa de bits o como array múltiple. Por ejemplo, las teclas de función en un teclado de 101 teclas se utilizan a veces como teclas modificadas—por ejemplo F1 A. En este caso, al menos dos campos de la matriz deben ser reportados en un *array item*, es decir, *Report Count(2).*

Limitaciones en los report

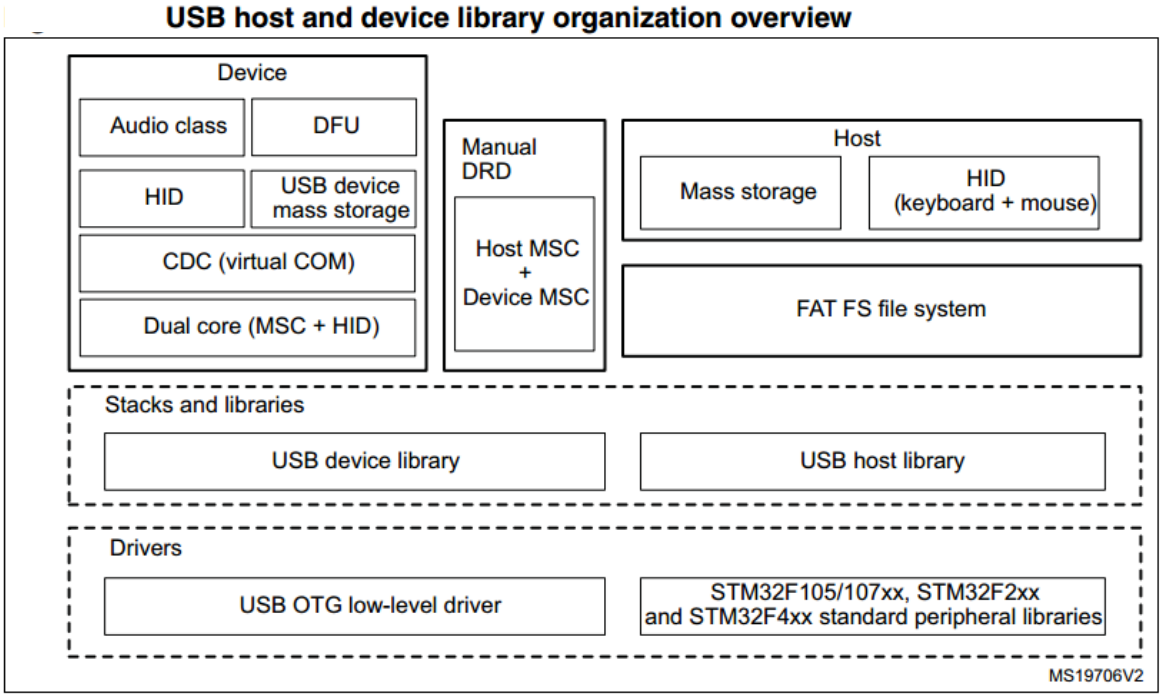
Las siguientes limitaciones se aplican a los *reports* y al controlador de item:

- Un campo de item no puede abarcar más de 4 bytes en un *report*. Por ejemplo, un item de 32 bits debe comenzar en un límite de byte para satisfacer esta condición.
- Sólo se permite un *report* en una única transferencia USB.
- Un *report* puede abarcar una o más transacciones USB. Por ejemplo, una aplicación que tiene *reports* de 10 bytes tendrá una duración de, al menos, dos operaciones USB en un dispositivo de baja velocidad.
- Todos los *reports*, excepto el más largo que excedan *wMaxPacketSize* en el *endpoint* deben terminar con un breve paquete. El informe más largo no requiere una terminación de paquete corto.
- Cada colección de nivel superior debe ser una colección de aplicaciones y los *reports* no puede abarcar más de una colección de nivel superior.
- Un *report* es siempre *byte-aligned*. Si es necesario, los r*eports* se rellenan con bits (0) hasta que se alcanza el siguiente límite de bytes.

3.3.3 Librería USB Device de ARM STM32F4 (5)

3.3.3.1 Visión general de las librerías

La siguiente figura ofrece una visión general de las bibliotecas *Host* y *USB device*.



Las bibliotecas de *Host* y *USB Device* se construyen alrededor del driver común de bajo nivel STM32 USB OTG y las librerías *USB Device* y *Host*.

Características generales

Las librerías *USB device* y *Host* son:

- Compatibles con dispositivos STM32F4xx en modos USB HS y FS.
- Totalmente compatibles con USB 2.0
- Optimizadas para trabajar con los periféricos USB OTG (HS, alta velocidad y FS, velocidad máxima) y se puede utilizar todas sus funciones.
- Espacio ocupado reducido, alto rendimiento de transferencia, robustez y código y documentación de paquete de alta calidad
- Fácilmente ampliables para soportar funciones USB OTG.
- Construidas siguiendo una arquitectura genérica y fácil de usar
  - capaz de añadir clases específicas de proveedor adicionales.
  -soporta aplicaciones de interfaz múltiple (dispositivos compuestos)
- Capaz de soportar múltiples núcleos USB OTG permitiendo el uso de varios núcleos con la misma librería.

3.3.3.2. Núcleo USB OTG

3.3.3.2.1 Núcleo USB OTG *full speed* (máxima velocidad)

OTG_FS es un controlador de dispositivo de doble función (DRD) que soporta tanto las funciones de Dispositivo como de *Host*. Es totalmente compatible con el suplemento *On-The-Go* de las especificaciones USB 2.0.

También se puede configurar como un controlador de sólo *Host* o solo dispositivo, totalmente compatible con la especificación USB 2.0. En el modo Host, OTG_FS soporta transferencias máximas de velocidad (12 Mbps) y de poca velocidad (1,5 Mbps), mientras que en modo dispositivo, sólo es compatible con transferencias de velocidad máxima.

El OTG_FS soporta tanto HNP (Protocolo de Negociación Host) como SRP (Protocolo de Solicitud de Sesión). El único dispositivo externo requerido es un *charge pump* para la fuente de alimentación VBUS en modo Host.

El interfaz OTG_Fs tiene las siguientes características:

- Cumple con el suplemento On-The-Go de las especificaciones USB 2.0
- Opera en los modos Full Speed (12 Mbps) y Low Speed (1,2 Mbps)
- Soporta Protocolo de solicitud de sesión (SRP)
- Soporta Protocolo de Negociación de Host (PNH)
- Cuatro endpoints bidireccionales, incluyendo 1 control endpoint1 y 3 endpoints de dispositivos que soportan *bulk*, interrupciones y transferencias isócronas
- Todos los *endpoints* de entrada al dispositivo pueden soportar transferencias periódicas
- Ocho canales de *Host* con apoyo periódico de salida
- Transmisión FIFO para cada uno de los 4 *endpoints* de entrada al dispositivo. Cada FIFO puede contener múltiples paquetes
- Rx y Tx FIFO combinada de tamaño 320 x 35 bits con FIFO dinámica de tamaño (1,25 Kbytes)
- Ocho entradas en cola periódica de Tx, 8 entradas en la cola no periódica de Tx
- Controles PHY FS en el chip para operaciones OTG de Dispositivo USB o *Host* USB.
- Requiere una bomba de carga externa (*charge pump*) para fuente la de alimentación VBUS.
- Interfaz AHB Esclavo de 32 bits para el acceso a los registros de control y de estado (CSR) y la FIFO de datos.

3.3.3.2.2 Núcleo USB OTG *high speed* (alta velocidad)

El OTG_HS es un controlador de dispositivo de doble función (DRD) que soporta funciones periféricas y de host. Es totalmente compatible con el suplemento *On-The-Go* de USB 2.0.

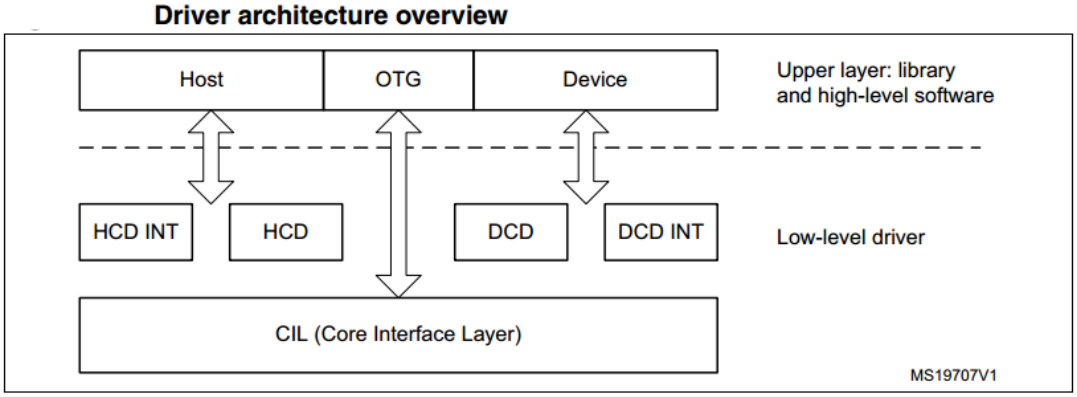
También se puede configurar como un controlador de sólo host o de sólo periférico, totalmente compatible con las especificaciones USB 2.0 (6). En el modo Host, OTG_HS soporta transferencias de alta velocidad (480 Mbps), de máxima velocidad (12 Mbps) y de baja velocidad (1,5 Mbps), mientras que en el modo periférico, sólo es compatible con transferencias de alta velocidad y máxima velocidad.

El interface OTG_HS tiene las siguientes características:

- Certificado USB-IF con las especificaciones de USB 2.0
- Soporta dos interfaces PHY:
  - Un PHY *Full Speed* en el chip
  - Un interfaz para PHY externa de alta velocidad ULPI (UTMI + *low pin interface*)
- Soporta el protocolo de negociación de *Host* (PNH) y el protocolo de solicitud de sesión (SRP)
- Puede ser usado por el monitor VBUS para ahorrar energía en aplicaciones OTG, sin necesidad de componentes externos
- Se puede utilizar para controlar los niveles de VBUS con comparadores internos
- Software configurable para operar como:
  - Un SRP USB HS / FS periférico (B-dispositivo)
  - Un SRP USB HS / FS / host de baja velocidad (A-dispositivo)
  - Un dispositivo USB OTG FS *dual-role*.
- Soporta pulsos HS / FS SOF (start-of-frame), así como *tokens keep_alive* con:
  - Pulso SOF de salida.
  - Pulso SOF conexión interna a Timer 2 (TIM2)
  - Período *framing* configurable.
  - Interrupción configurable de fin de trama *end-of-frame*.
- DMA interno con soporte de umbral y software seleccionable AHB tipo ráfaga en el modo DMA
- características *powersaving*, como parada de reloj del sistema mientras USB está suspendido, desconectar el ámbito del reloj interno del núcleo digital, administración de energía PHY y DFIFO.
- RAM dedicada de datos de 4 Kbytes con gestión avanzada FIFO:
  - Se puede configurar una partición de memoria en diferentes FIFO para permitir un uso flexible y eficiente de RAM
  - Cada FIFO puede contener múltiples paquetes
  - La asignación de memoria se realiza de forma dinámica
  -El tamaño de FIFO se puede configurar para valores que no sean potencias de 2 para permitir el uso de posiciones de memoria contiguas.
- Asegura un máximo ancho de banda USB de hasta un fotograma sin intervención de aplicación

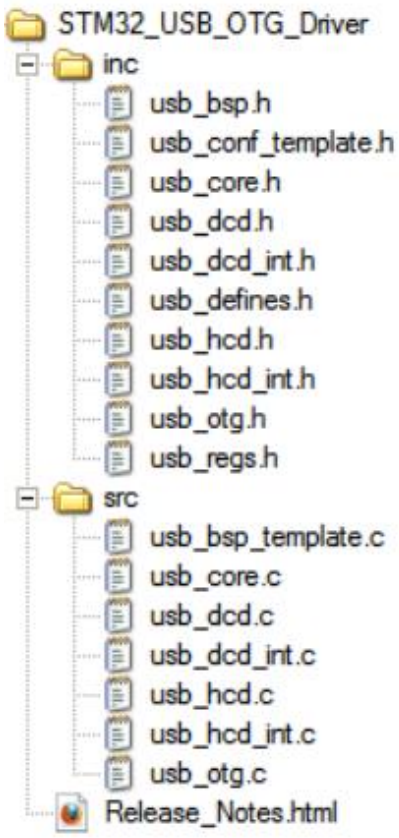
3.3.3.3 USB OTG driver de nivel bajo

3.3.3.3.1 Arquitectura



El driver de bajo nivel se puede utilizar para conectar el núcleo USB OTG con la pila de alto nivel. El usuario puede desarrollar una capa de interfaz por encima del driver de bajo nivel para proporcionar las API adecuadas que necesita la pila.

3.3.3.3.2 Archivos



Modo	Archivos	Descripción
Común	Usb_core.c/h	Este archivo contiene la capa de abstracción de hardware y las operaciones de comunicación USB.
	Usb_core.c/h	Configuración básica para *Host*, *Device* y modos de OTG: Tamaño Tx FIFO, tamaño FIFO Rx, modo de núcleo y características seleccionadas... etc. Este archivo se debe copiar en la carpeta de aplicaciones y modificar en función de las necesidades de la aplicación
	Usb_bsp_templace.c	Este archivo contiene la configuración básica de bajo nivel (interrupciones, GPIO). Este archivo se debe copiar en la carpeta de la aplicación y modificar en función de las necesidades de la aplicación.
Host	Usb_hcd.c/h	Capa de interfaz de *Host* utilizado por la librería para acceder al núcleo.
	Usb_hcd_int.c/h	Subrutinas de interrupción para el modo *Host*.
Device	Usb_dcd.c/h	Capa de interfaz de dispositivo utilizada por la librería para acceder al núcleo.
	Usb_dcd_int.c/h	Subrutinas de interrupción para el modo de *Device*.
OTG	Usb_otg.c/h	Contiene la implementación de los protocolos SRP y HNP y las interrupciones relativas al modo OTG.

3.3.3.3.3 Configuración

La configuración de los núcleos USB OTG (de alta y máxima velocidad) se define en el archivo de configuración común (usb_conf.h). El usuario puede activar o desactivar ciertas características fundamentales, definir Tx y Rx FIFO para el dispositivo, el periodo y la FIFO de transmisión no periódica así como la FIFO de Rx para el *Host*. Este archivo también se utiliza para seleccionar *Host*, dispositivo o modos OTG o para seleccionar ambos modos (*Device* y *Host*) para aplicaciones de dispositivos *dual-role*.

La siguiente tabla detalla las configuraciones básicas definidas en el archivo *usb_conf.h*.

DEFINE	Descripción
`USB_OTG_FS_CORE`	Permite el uso del núcleo de velocidad máxima.
`USB_OTG_HS_CORE`	Permite el uso del núcleo de alta velocidad.
`RX_FIFO_FS_SIZE`	Establece el tamaño de recepción FIFO para el núcleo de velocidad máxima.
`RX_FIFO_HS_SIZE`	Establece el tamaño de recepción FIFO para el núcleo de alta velocidad.
`TXn_FIFO_FS_SIZE`	Establece el tamaño de transmisión FIFO para un *endpoint* de dispositivo (Full Speed) donde n es el índice del *endpoint* que se utilizará
`TXn_FIFO_HS_SIZE`	Establece el tamaño de transmisión FIFO para un *endpoint* de dispositivo (núcleo *High Speed*), donde n es el índice de *endpoint* que se utilizará.
`TXH_NP_FS_FIFOSIZ`	Establece el tamaño de Transmisión FIFO no periódica para el modo *Host* (*Full Speed*).
`TXH_NP_HS_FIFOSIZ`	Establece el tamaño de transmisión FIFO no periódica para el modo *Host* (núcleo *High Speed*).
`TXH_P_FS_FIFOSIZ`	Establece el tamaño de transmisión FIFO periódica para el modo *Host* (*Full Speed*).

TXH_P_HS_FIFOSIZ	Establece el tamaño de transmisión FIFO periódica para el modo Host (núcleo *High Speed*).
USB_OTG_ULPI_PHY_ENABLED	Activa PHY ULPI para el núcleo *High Speed*.
USB_OTG_EMBEDDED_PHY_ENABLED	Activa PHY FS embebido para el núcleo *High Speed*
USB_OTG_HS_LOW_PWR_MGMT_SUPPORT	Habilita administración de energía *low power* para el núcleo de alta velocidad (*USB Core clock gating*, etc.).
USB_OTG_FS_LOW_PWR_MGMT_SUPPORT	Habilita administración de energía *low power* para el núcleo *Full Speed* (*USB Core clock gating*, etc.) ..
USB_OTG_HS_INTERNAL_DMA_ENABLED	Activa la función de DMA interno para el núcleo de alta velocidad.
USB_OTG_HS_DEDICATED_EP1_ENABLED	Activa la función de *endpoint* 1 para el modo de dispositivo de núcleo de alta velocidad.

3.3.3.3.4 Manual de programación

Estructuras del driver de bajo nivel

El driver de bajo nivel no tiene ninguna variable exportable. Una estructura global (USB_OTG_CORE_HANDLE) que mantiene todas las variables, el estado y *buffers* utilizados por el núcleo para manejar su estado interno y flujo de transferencia, se debe utilizar para asignar en la capa de aplicación la instancia manejada por el núcleo.

Este método permite que la aplicación utilice el mismo driver de bajo nivel para ambos núcleos de alta y máxima velocidad en el mismo proyecto.

La estructura de núcleo USB global se define como sigue:

```
typedef struct USB_OTG_handle
{
USB_OTG_CORE_CFGS cfg;
USB_OTG_CORE_REGS regs;
 #ifdef USE_DEVICE_MODE
      DCD_DEV dev;
 #endif
#ifdef USE_HOST_MODE
      HCD_DEV host;
#endif
#ifdef USE_OTG_MODE
      OTG_DEV otg;
#endif
 }
USB_OTG_CORE_HANDLE, *PUSB_OTG_CORE_HANDLE;
```

Consideraciones a la hora de programar cuando se usa DMA interna

Cuando se utiliza DMA interno con el núcleo USB OTG de alta velocidad, todas las estructuras relacionadas con DMA (*buffers* de datos) durante el proceso de la transacción deben ser 32 bits alineados.

En consecuencia, la estructura `USB_OTG_handle` se define para mantener todos los *buffers* internos y variables utilizadas para mantener los datos a transferir de 32 bits alineados.

Cuando se utiliza DMA interno, la estructura *global Core USB* debe declararse de la siguiente manera:

USB core structure

```
#ifdef USB_OTG_HS_INTERNAL_DMA_ENABLED
  #if defined ( __ICCARM__ ) /*!< IAR Compiler */
    #pragma data_alignment=4
  #endif
#endif /* USB_OTG_HS_INTERNAL_DMA_ENABLED */

__ALIGN_BEGIN USB_OTG_CORE_HANDLE     USB_OTG_dev __ALIGN_END ;
```

Nota: `__ALIGN_BEGIN` Y `__ALIGN_END` son palabras clave específicas del compilador definidas en el archivo *usb_conf.h* y se utilizan para alinear las variables en un límite de32-bit.

C compiler-dependant keywords (defined in usb_conf.h file)

```
/******************* C Compilers dependant keywords ****************************/
/* In HS mode and when the DMA is used, all variables and data structures dealing
   with the DMA during the transaction process should be 4-bytes aligned */
#ifdef USB_OTG_HS_INTERNAL_DMA_ENABLED
  #if defined   (__GNUC__)        /* GNU Compiler */
    #define __ALIGN_END    __attribute__ ((aligned (4)))
    #define __ALIGN_BEGIN
  #else
    #define __ALIGN_END
    #if defined   (__CC_ARM)      /* ARM Compiler */
      #define __ALIGN_BEGIN    __align(4)
    #elif defined (__ICCARM__)    /* IAR Compiler */
      #define __ALIGN_BEGIN
    #elif defined  (__TASKING__)  /* TASKING Compiler */
      #define __ALIGN_BEGIN    __align(4)
    #endif /* __CC_ARM */
  #endif /* __GNUC__ */
#else
  #define __ALIGN_BEGIN
  #define __ALIGN_END
#endif /* USB_OTG_HS_INTERNAL_DMA_ENABLED */

/* __packed keyword used to decrease the data type alignment to 1-byte */
#if defined (__CC_ARM)          /* ARM Compiler */
  #define __packed    __packed
#elif defined (__ICCARM__)      /* IAR Compiler */
  #define __packed    __packed
#elif defined   ( __GNUC__ )    /* GNU Compiler */
  #define __packed    __attribute__ ((__packed__))
#elif defined   (__TASKING__)   /* TASKING Compiler */
  #define __packed    __unaligned
#endif /* __CC_ARM */
```

Selección de interfaz física USB

Como se describe en la configuración del controlador USB OTG de bajo nivel, el usuario puede seleccionar la interfaz física USB (PHY) a utilizar

- Para el *USB OTG Full Speed Core*, se utiliza el PHY *Full Speed* embebido.

- Cuando se utiliza el *USB OTG High Speed Core*, el usuario puede seleccionar una de las dos interfaces PHY:
  - Una interfaz ULPI para el PHY de alta velocidad externa: *USB HS Core* funcionará en modo de alta velocidad
- Un PHY de máxima velocidad en el chip: *USB HS Core* operará en modo de máxima velocidad.

La librería ofrece la posibilidad de seleccionar el PHY para ser utilizado mediante uno de estos dos define (en el archivo *usb_conf.h*):

- USE_ULPI_PHY: si *USB OTG HS Core* se va a utilizar en el modo de alta velocidad
- USE_EMBEDDED_PHY: si *USB OTG HS Core* se va a utilizar en el modo de máxima velocidad.

Nota: Con la interfaz ULPI, el usuario puede forzar al núcleo a trabajar en modo *Full Speed* modificando el archivo *usb_core.c* través del bit ULPIFSLS en el Registro OTG_HS_GUSBCFG.

En el modo *Host*, la velocidad del núcleo puede ser modificada cuando se conecta un dispositivo con una velocidad más baja.

Programación de los drivers del dispositivo.

- **Inicialización del dispositivo**

El dispositivo se inicializa utilizando la siguiente función:

```
DCD_Init (USB_OTG_CORE_HANDLE *pdev, USB_OTG_CORE_ID_TypeDef coreID)
```

El tamaño Rx y Tx FIFO y la dirección de inicio se establecen dentro de esta función para utilizar usar un *endpoints* más además del *control endpoint(0)*. El usuario puede cambiar la configuración FIFO mediante la modificación de los valores por defecto y cambiar la profundidad de FIFO para cada Tx FIFO en el archivo *usb_conf.h*.

- **Configuración *Endpoint***

Una vez que el núcleo USB OTG se inicializa, se selecciona el modo de dispositivo. La capa superior puede llamar al driver de bajo nivel para abrir o cerrar el *endpoint* activo para iniciar la transferencia de datos. Las dos APIs siguientes son usadas:

```
uint32_t DCD_EP_Open (USB_OTG_CORE_HANDLE *pdev ,
uint8_t ep_addr,
uint16_t ep_mps,
uint8_t ep_type)
uint32_t DCD_EP_Close (USB_OTG_CORE_DEVICE *pdev,
uint8_t ep_addr)
```

- **Estructura del núcleo de dispositivo**

Las estructuras DCD_DEV contienen todas las variables y estructuras utilizadas para mantener en tiempo real toda la información relativa a los dispositivos, la máquina de estado de transferencia de control, así como la información y el estado de *endpoint*.

```
typedef struct _DCD
```

```
{
      uint8_t device_config;
      uint8_t device_state;
      uint8_t device_status;
      uint8_t device_address;
      uint32_t DevRemoteWakeup;
      USB_OTG_EP in_ep [USB_OTG_MAX_TX_FIFOS];
      USB_OTG_EP out_ep [USB_OTG_MAX_TX_FIFOS];
      uint8_t setup_packet [8*3];
      USBD_Class_cb_TypeDef *class_cb;
      USBD_Usr_cb_TypeDef *usr_cb;
      uint8_t *pConfig_descriptor;
}
DCD_DEV , *DCD_PDEV;
```

En esta estructura, `device_config` mantiene la configuración del dispositivo USB actual y `device_state` controla la máquina de estados con los siguientes estados:

```
/* EP0 State */
#define USB_OTG_EP0_IDLE 0
#define USB_OTG_EP0_SETUP 1
#define USB_OTG_EP0_DATA_IN 2
#define USB_OTG_EP0_DATA_OUT 3
#define USB_OTG_EP0_STATUS_IN 4
#define USB_OTG_EP0_STATUS_OUT 5
#define USB_OTG_EP0_STALL 6
```

En esta estructura, `device_status` define el estado de la conexión, la configuración y el *power status:*

```
/* Device Status */
#define USB_OTG_DEFAULT 0
#define USB_OTG_ADDRESSED 1
#define USB_OTG_CONFIGURED 2
```

- **Flujo de transferencia de datos USB**

La capa DCD ofrece al usuario todas las API necesarias para iniciar y controlar un flujo de transferencia utilizando el siguiente conjunto de funciones:

```
uint32_t    DCD_EP_PrepareRx ( USB_OTG_CORE_HANDLE *pdev,
                               uint8_t ep_addr,
                               uint8_t *pbuf,
                               uint16_t buf_len);
uint32_t    DCD_EP_Tx (USB_OTG_CORE_HANDLE *pdev,
                               uint8_t ep_addr,
```

```
                                uint8_t *pbuf,
                                uint32_t buf_len);
uint32_t    DCD_EP_Stall (USB_OTG_CORE_HANDLE *pdev,
                                uint8_t epnum);
uint32_t    DCD_EP_ClrStall (USB_OTG_CORE_HANDLE *pdev,
                                uint8_t epnum);
uint32_t    DCD_EP_Flush (USB_OTG_CORE_HANDLE *pdev,
                                uint8_t epnum);
```

La capa de DCD del driver de bajo nivel de USB OTG tiene una función que debe ser llamada por la interrupción USB (alta velocidad o máxima velocidad):

```
uint32_t    DCD_Handle_ISR (USB_OTG_CORE_HANDLE *pdev)
```

El archivo *dcd_int.h* contiene los prototipos de funciones de las funciones llamadas desde la capa central de la librería para manejar los eventos USB.

- **Definición de la estructura de driver USB**

```
typedef struct _USBD_DCD_INT
{
      uint8_t (* DataOutStage) (USB_OTG_CORE_HANDLE *pdev , uint8_t
epnum);
      uint8_t (* DataInStage) (USB_OTG_CORE_HANDLE *pdev , uint8_t
epnum);
      uint8_t (* SetupStage) (USB_OTG_CORE_HANDLE *pdev);
      uint8_t (* SOF) (USB_OTG_CORE_HANDLE *pdev);
      uint8_t (* Reset) (USB_OTG_CORE_HANDLE *pdev);
      uint8_t (* Suspend) (USB_OTG_CORE_HANDLE *pdev);
      uint8_t (* Resume) (USB_OTG_CORE_HANDLE *pdev);
      uint8_t (* IsoINIncomplete) (USB_OTG_CORE_HANDLE *pdev);
      uint8_t (* IsoOUTIncomplete) (USB_OTG_CORE_HANDLE *pdev);
      uint8_t (* DevConnected) (USB_OTG_CORE_HANDLE *pdev);
      uint8_t (* DevDisconnected) (USB_OTG_CORE_HANDLE *pdev);
}USBD_DCD_INT_cb_TypeDef;
```

En la capa de librería, una vez que se define la estructura `USBD_DCD_INT_cb_TypeDef`, debe ser asignado al puntero `USBD_DCD_INT_fops`.

Ejemplo:

```
USBD_DCD_INT_cb_TypeDef *USBD_DCD_INT_fops = &USBD_DCD_INT_cb;
```

- Interrupción IN y OUT

El núcleo OTG USB de alta velocidad incorpora dos interrupciones independientes para *endpoint 1 IN* y *endpoint 1 OUT*. En consecuencia, `USBD_OTG_EP1OUT_ISR_Handler` y `USBD_OTG_EP1IN_ISR_Handler` se pueden utilizar juntas para aligerar la interrupción global de OTG USB.

La función de *endpoint* específico se selecciona activando el *define*

`USB_OTG_HS_DEDICATED_EP1_ENABLED` en el archivo *usb_conf.h*.

- Uso de DMA interno en el modo de alta velocidad

El núcleo OTG de alta velocidad USB incorpora un DMA interno capaz de manejar la solicitud FIFO I/O de forma automática sin uso de la CPU. Sin embargo, las estructuras de datos utilizadas en el modo DMA deben ser de 32 bits alineados.

La característica de DMA interna se selecciona activando

`USB_OTG_HS_INTERNAL_DMA_ENABLED` en el archivo usb_conf.h.

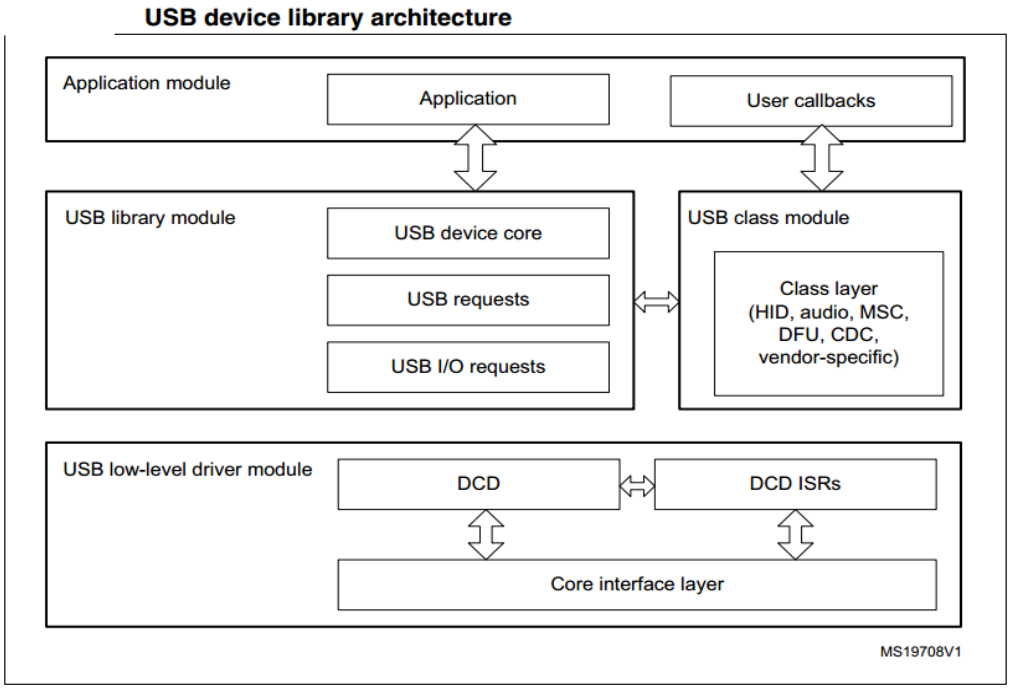
Nota: Las características del DMA Interno y la interrupción IN y OUT se pueden utilizar conjuntamente para mejorar el rendimiento de transferencia de datos.

3.3.3.4 Librería *USB device*

La librería *USB device*:

- Soporta funciones de transferencia de múltiples paquetes para que una gran cantidad de datos se pueden enviar sin tener que dividirlos en transferencias de tamaño máximo de paquete.
- Soporta hasta tres transferencias *back-to-back* en *endpoint* de control (compatible con controladores OHCI).
- Utiliza archivos de configuración para cambiar el núcleo y la configuración de la librería sin cambiar el código de la librería (sólo lectura).
- Estructuras de datos de 32 bits alineados para manejar la transferencia de DMA en los modos de alta velocidad.
- Soporta instancias de núcleo OTG USB múltiples desde un nivel de usuario.

3.3.3.4.1Visión general

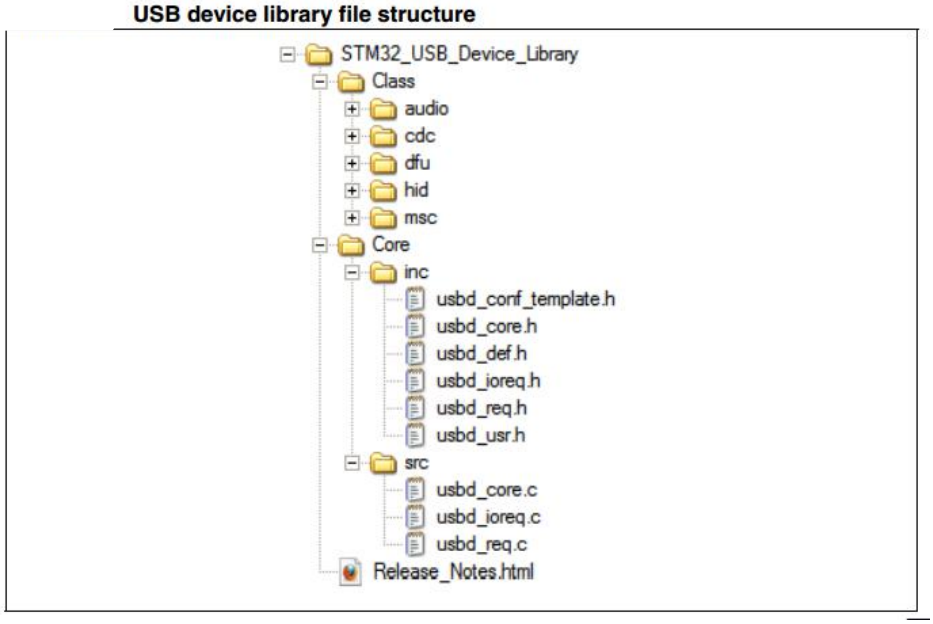


Como se muestra en la figura anterior, la librería de dispositivos USB se compone de dos partes principales: el núcleo de la librería y los drivers de clase.

El núcleo de la librería se compone de tres bloques principales:

- USB device Core
- USB requests
- USB I/O requests

3.3.3.4.2 Archivos



La librería de dispositivos USB se basa en el controlador de nivel bajo genérico USB OTG que soporta modos de *Host*, dispositivos y OTG y trabaja para alta velocidad, velocidad máxima y baja velocidad (en el modo de *Host*).

La carpeta ***Core*** contiene las máquinas de librerías *USB device* como se define en las especificaciones de USB 2.0 (6).

La carpeta ***Class*** contiene todos los archivos en relación a la implementación de la clase.

3.3.3.4.3 Descripción de la librería USB device

Flujo de la librería USB device

- **Manejador de *control endpoint 0***

Las especificaciones USB definen cuatro tipos de transferencia: control, interrupción, *bulk* y transferencias isócronas. El *Host USB* envía solicitudes al dispositivo a través de la variable de control (en este caso, *endpoint* de control es *endpoint 0*). Las solicitudes se envían al dispositivo en forma de paquetes de configuración. Estas solicitudes se pueden clasificar en tres categorías: estándar, de clase específica y específicos del proveedor.

Dado que las solicitudes estándar son genéricas y comunes a todos los dispositivos USB, la librería recibe y maneja todas las solicitudes estándar en *endpoint 0* de control.

Las respuestas de la librería solicitan sin la intervención de la aplicación de usuario si la librería tiene suficiente información acerca de estas solicitudes. De lo contrario, la librería llama a las funciones de devolución de llamada de aplicación definidas por el usuario para cumplir las peticiones cuando se necesitan algunas acciones de aplicación o información de la aplicación. El formato y el significado de las peticiones específicas de clase y las peticiones específicas de los proveedores no son comunes para todos los dispositivos USB.

La librería no maneja ninguna de las solicitudes en estas categorías. Cada vez que la librería recibe una solicitud que desconoce, la librería llama a una función de devolución de llamada definida por el usuario y pasa la petición al código de la aplicación de usuario. Todas las solicitudes *SETUP* se procesan con una máquina de estado implementada en un modelo de interrupción.

Se genera una interrupción al final de una transferencia USB correcta. El código de la librería recibe esta interrupción. En la rutina de procesado de interrupción, se identifica el *endpoint* solicitante. Si el evento es una configuración de *endpoint 0*, el contenido de la configuración recibida se guarda y la máquina de estado comienza.

- **Las transacciones en *non-control endpoint***

El núcleo de la clase específica utiliza *non-control endpoints* llamando a un conjunto de funciones para enviar o recibir datos a través de la etapa de devolución de llamadas *data IN* y *OUT*.

- **Estructura de datos para el paquete CONFIGURACIÓN**

Cuando llega un nuevo paquete de configuración, todos los ocho bytes del paquete de configuración se copian en un estructura interna `USB_SETUP_REQ req`, por lo que el siguiente paquete de configuración no puede sobrescribir al anterior durante el proceso. Esta estructura interna se define como:

```
typedef struct usb_setup_req
{
uint8_t bmRequest;
uint8_t bRequest;
uint16_t wValue;
uint16_t wIndex;
uint16_t wLength;

} USB_SETUP_REQ;
```

- **Solicitudes estándar**

La mayoría de las solicitudes especificadas en la siguiente tabla de las especificaciones USB se manejan como solicitudes estándar en la librería. La tabla enumera todas las solicitudes estándar y sus parámetros válidos en la librería. Las solicitudes que no están en esta tabla se consideran como solicitudes no estándar.

	State	BmRequestT	Low byte of	High byte of	Low byte of	High byte of windex	wLenght	Comentarios
GET_STATUS	A,C	80	00	00	00	00	2	Obtiene el estado del dispositivo.
	C	81	00	00	N	00	2	Obtiene el estado de la interfaz, donde N es el número de interfaz válido.
	A,C	82	00	00	00	00	2	Obtiene el estado de *Endpoint 0 OUT direction*.
	A,C	82	82	00	00	00	2	Obtiene el estado de *Endpoint 0 IN direction.*
	C	82	00	00	EP	00	2	Obtiene el estado del *endpoint* EP.
CLEAR_FEATURE	A,C	00	01	00	00	00	00	Borra la función de activación remota de dispositivos.
	C	02	00	00	EP	00	00	Establece la condición STALL (bloqueo) del *endpoint* EP. EP no se refiere *endpoint 0*.
SET_FEATURE	A,C	00	01	00	00	00	00	Establece la función de activación remota de dispositivos.
	C	02	00	00	EP	00	00	Establece la condición STALL en *endpoin*t EP. EP no se refiere a *endpoint0*
SET_ADRESS	D,A	00	N	00	00	00	00	Establece la dirección de dispositivo, N es la dirección de dispositivo válida
GET_DESCRIPTOR	All	80	00	01	00	00	Non-0	Obtiene el descriptor de dispositivo
	All	80	N	02	00	00	Non-0	Obtiene el descriptor de configuración; donde N es el índice configuración válida.
	All	80	N	03	LanID		Non-0	Establece la configuración del dispositivo; donde N es el número de configuración válido.
GET_CONFIGURATION	A,C	80	00	00	00	00	1	Obtiene la configuración del dispositivo.
SET_CONFIGURATION	A,C	80	N	00	00	00	00	Establece la configuración del dispositivo, donde N es un número valido de configuración
GET_INTERFACE	C	81	00	00	N	00	1	Obtiene la configuración alternativa de la interfaz N; donde N es el número de interfaz válido.
SET_INTERFACE	C	01	M	00	N	00	00	Establece configuración alternativa M de la interfaz N; donde N es el número de interfaz válido y M es la configuración alternativa válida del interfaz N

Nota: En la columna Estado:
D = estado por defecto;
A = Estado dirección;
C = Estado configurado;

ALL = Todos los estados.
EP: D0-D3 = dirección de *endpoint*;
D4-D6 = Reservado como cero;
D7 = 0: *endpoint OUT*, 1: *endpoint IN*

- **Solicitudes no estándar**

Todas las solicitudes no estándar se pasan al código específico de clase a través de funciones de devolución de llamada.

- Etapa de configuración
  La librería pasa todas las peticiones no estándar en el código específico de clase con la función de devolución de llamadas *pdev-> dev.class_cb-> Setup (PDEV, req)*. Las solicitudes no estándar incluyen las solicitudes *user-interpreted* y las solicitudes no válidas. Las solicitudes *user-interpreted* son solicitudes específicas de clase, solicitudes específicas del proveedor o solicitudes que la librería considera como solicitudes no válidas que la aplicación quiere interpretar como peticiones válidas (por ejemplo, la librería no admite la función *Halt* en *endpoint 0*, pero la aplicación de usuario sí).

  Solicitudes no válidas son las peticiones que no son solicitudes estándar y no son solicitudes *user-interpred*. Desde *pdev-> dev.class_cb-> Setup (PDEV, req)* se llama después de la fase de *SETUP* y antes de la fase de datos, el código del usuario es responsable, en el *pdev-> dev.class_cb-> Setup (PDEV, req)* , de analizar el contenido del paquete *SETUP (req)*. Si una solicitud no es válida, el código de usuario tiene que llamar a *USBD_CtlError (PDEV, req)* y devolver la llamada *pdev-> dev.class_cb-> Setup (PDEV, req).*

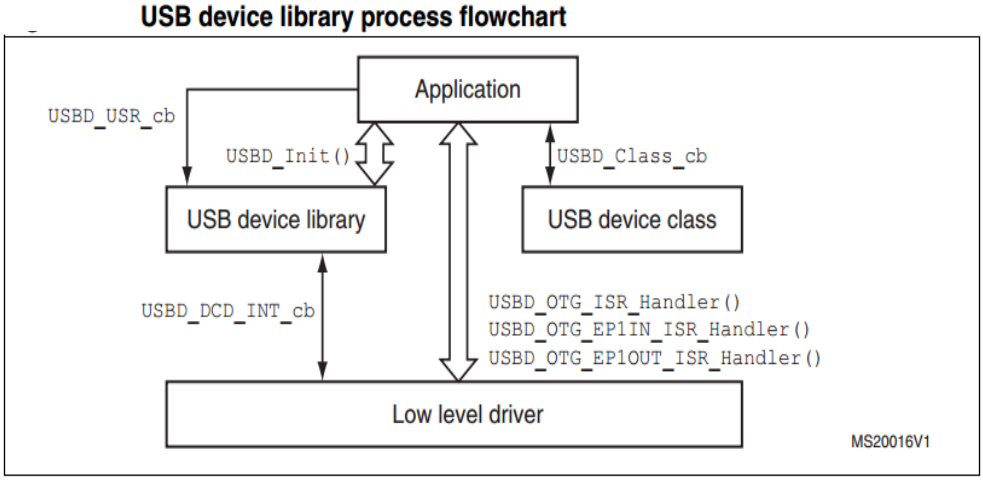
  Para una solicitud *user-interpreted*, el código de usuario tiene que preparar el *buffer* de datos para el etapa siguiente de datos si la solicitud tiene una etapa de datos; de lo contrario el código de usuario ejecuta la solicitud y devuelve la llamada *pdev-> dev.class_cb-> SETUP(PDEV, req).*

- Etapa de Datos
  La capa de clase utiliza el estándar `USBD_CtlSendData` y `USBD_CtlPrepareRx` para enviar o recibir datos, el flujo de transferencia de datos es manejado internamente por la librería y el usuario no tiene que dividir los datos en paquetes ep_size.

- Etapa de estado.
  La etapa de estado es manejada por de la librería después de regresar de la devolución de llamada *pdev-> dev.class_cb->SETUP (PDEV, req).*

Proceso de la librería *USB device*

La siguiente figura muestra las diferentes capas de la interacción entre los drivers de bajo nivel, la librería *USB device* y la capa de aplicación.



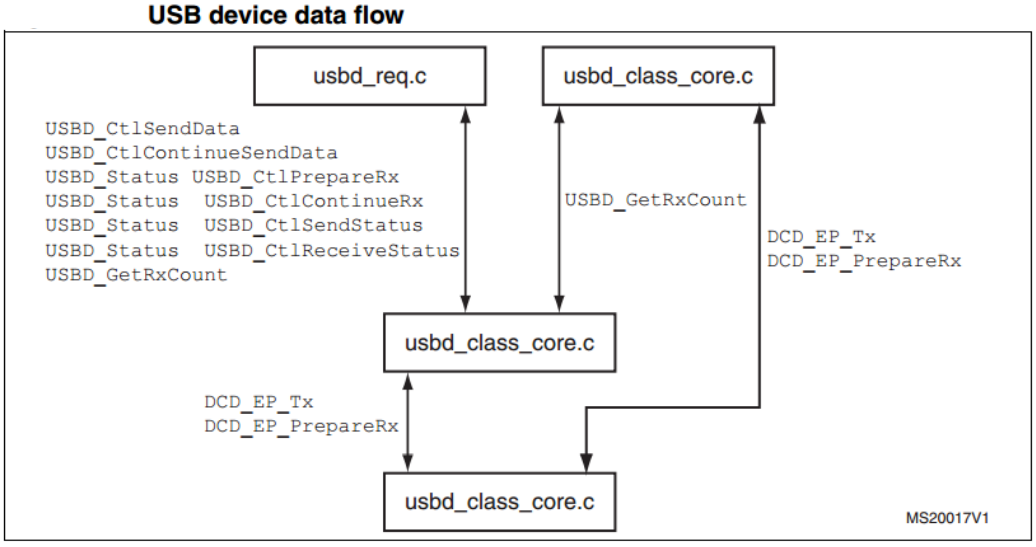
La capa de aplicación sólo ha de llamar a una función (`USBD_Init`) para inicializar el driver de nivel bajo del USB, la librería *USB device*, el hardware en la tarjeta utilizada (BSP) e iniciar la librería. La aplicación también tiene que utilizar el USB ISR general y las subrutinas EP1 cuando él define `USB_OTG_HS_DEDICATED_EP1_ENABLED` se descomenta en el archivo *usb_conf.h*.

La función `USBD_Init` necesita sin embargo, la estructura de devolución de llamada de usuario para informar a la capa de usuario de los diferentes estados y mensajes de la librería y la estructura de clases de devolución de llamada para iniciar la interfaz de clase.

El driver USB de nivel bajo puede estar vinculado a la librería *USB device* a través de la estructura `USBD_DCD_INT_cb`. Esta estructura garantiza una total independencia entre la librería *USB device* y el driver de nivel bajo; permitiendo al driver de nivel bajo ser utilizado por cualquier otra librería de dispositivos.

Flujo de datos *USB device*

La librería USB (núcleo USB y la capa de clase USB) maneja el procesamiento de datos en *endpoint 0* (EP0) a través de la capa de solicitud IO cuando se necesita una envoltura para gestionar la función de múltiples paquetes en el *endpoint* de control o directamente desde la capa *usb_dcd.c* cuando se utilizan los otros *endpoints* desde el núcleo USB OTG compatible con la función multi-paquete. La siguiente figura ilustra este esquema de flujo de datos.



Configuración de la librería *USB device*

La librería *USB device* se puede configurar mediante el archivo *usbd_conf.h* (un archivo *template* de configuración está disponible en el directorio "*Libraries*\ *STM32_USB_Device_Library* \ *Core* \" de la librería).

```
#define USBD_CFG_MAX_NUM            1
#define USB_MAX_STR_DESC_SIZ        64

/**** USB_MSC_Class_Layer_Parameter *********/
#define MSC_IN_EP                   0x81
#define MSC_OUT_EP                  0x01
#define MSC_MAX_PACKET              512
#define MSC_MEDIA_PACKET            4096

/**** USB_HID Class_Layer_Parameter *********/
#define HID_IN_EP                   0x81
#define HID_OUT_EP                  0x01
#define HID_IN_PACKET               4
#define HID_OUT_PACKE               4
```

Manejador de transferencia de datos USB

El manejador de transferencia de datos USB es compatible con las funciones de transferencia de múltiples paquetes para que una gran cantidad de datos puedan ser enviados sin dividirlos en transferencias de tamaño máximo de paquete. El proceso multi-paquete es manejado por el driver de nivel bajo a través de `DCD_HandleRxStatusQueueLevel_ISR` y `DCD_HandleInEP_ISR` cuando el núcleo USB OTG se ejecuta en modo Esclavo y por el DMA interno cuando se utiliza el modo DMA (modo DMA disponible sólo con USB núcleo OTG HS).

Utilizando la función multi-packet

Para transmitir datos, se llama a la función `DCD_EP_Tx ()` y para recibir datos se llama a la función `DCD_EP_PrepareRx ()`, ambas tienen una longitud de datos ilimitada. Internamente, el núcleo USB OTG comprueba el espacio disponible en la FIFO y procesa la transferencia, respetando el tamaño de *endpoint*. Por ejemplo, si el tamaño de *endpoint* está configurado para trabajar con 64 bytes de datos y el usuario quiere transmitir / recibir N bytes de datos, el núcleo USB envía / recibe varios paquetes de 64 bytes cada uno.

Funciones de control USB

Las aplicaciones de usuario pueden beneficiarse de algunas otras funciones USB incluidas en un dispositivo USB.

- ***Device reset***

Cuando el dispositivo recibe una señal de *reset* del USB, se restablece la librería e inicializa la aplicación de software y hardware.

Esta función es parte de la rutina de interrupción. Se aplican restricciones de rutina de interrupción.

- ***Device suspend***

Cuando el dispositivo detecta una condición de suspender en el USB, se detienen todas las operaciones de la librería y pone el sistema en estado de suspensión (si el modo de baja potencia está habilitado en el archivo *usb_conf.h*).

- ***Device resume***

Cuando el dispositivo detecta una señal de reanudación en el USB, la librería restaura el reloj de núcleo USB y pone el sistema al estado de reposo (si el modo de baja potencia está habilitado en el archivo *usb_conf.h*).

FIFO de tamaño personalizado

Para utilizar un nuevo *endpoint* o cambiar un *endpoint* ya utilizado en la aplicación, el usuario tiene que tener cuidado con dos cosas:

1. Inicialización del *Endpoint*: Esta fase se realiza generalmente en la capa usbd_class_core a través de la siguiente función:

```
uint32_t DCD_EP_Open (USB_OTG_CORE_HANDLE *pdev ,
      uint8_t ep_addr,
      uint16_t ep_mps,
      uint8_t ep_type)
```

El ep_addr debe mantener la dirección de *endpoint*, hay que tener en cuenta que la dirección *endpoint* se identifica por el bit MSB (es decir, índice "0x80 | ep" para *ep IN endpoint*) y ep_mps mantiene el tamaño de paquete máximo de los endpoints.

2. La configuración FIFO se realiza en *usb_core.c* en el driver de nivel bajo USB, la configuración de la FIFO podría ser modificada por el usuario a través del archivo *usb_conf.h*

```
#ifdef USB_OTG_FS_CORE
      #define RX_FIFO_FS_SIZE128
      #define TX0_FIFO_FS_SIZE64
      #define TX1_FIFO_FS_SIZE128
      #define TX2_FIFO_FS_SIZE0
      #define TX3_FIFO_FS_SIZE0
#endif
#ifdef USB_OTG_HS_CORE
      #define RX_FIFO_HS_SIZE512
      #define TX0_FIFO_HS_SIZE128
      #define TX1_FIFO_HS_SIZE384
      #define TX2_FIFO_HS_SIZE0
      #define TX3_FIFO_HS_SIZE0
      #define TX4_FIFO_HS_SIZE0
```

```
      #define TX5_FI
#endif
```

La configuración de la FIFO se describe con detalle en los manuales que proporciona la empresa STMicroelectronics: RM0033 y RM0008.

La Rx y TXs FIFO se pueden calcular como sigue:

1. Tamaño FIFO de datos Rx = RAM para paquetes de configuración + información de *control endpoint* de datos de salida + paquetes de datos de salida + *Miscellaneous*

Nota: Espacio = Una palabra de 32-bit

- RAM para paquetes de configuración = 10
- Información de *control endpoint* de datos de salida= 1espacio (un espacio para la información de estado escrita en la FIFO junto con cada paquete recibido).
- Paquetes de datos de salida = (máximo tamaño de paquete/4) + 1 espacio (Mínimo para recibir paquetes) o paquetes de datos de salida = al menos 2*(máximo tamaño de paquete/4) + 1espacio (si *endpoint* de alto ancho de banda está habilitado o varios *endpoints* isócronos)
- *Miscellaneous* = 1espacio por paquete de datos de salida (un espacio por cada información de estado de transferencia completa y también con cada *endpoin*t de último paquete)

2. MÍNIMO espacio de RAM requerido para cada *Data In endpoint* Tx FIFO = MAX tamaño de paquete para cada *Data IN endpoint* particular. Más espacio asignado en los *Data In endpoint* Tx FIFO se traduce en un mejor rendimiento en el USB y puede ocultar las latencias en AHB.
3. Txn tamaño mínimo = 16 palabras. (donde, n es el índice de transmisión FIFO).
4. Cuando no se usa Tx FIFO, la configuración debe ser la siguiente:
   Caso 1: n > m y Txn no se usa (donde, n, m son los índices de transmisión FIFO)
   - Txm puede usar el espacio asignado a Txn.

   Caso 2: n < m y Txn no se usa (donde, n, m son los índices de transmisión FIFO)

   - Txm deber ser configurado con un espacio mínimo de 16 palabras.
5. La FIFO se usa óptimamente cuando las TxFIFOs producidas están alojadas en límite superior de la FIFO. Por ejemplo, utilizar EP1 y EP2 como entrada en lugar de EP1 y EP3 como entradas.

   El tamaño total FIFO para el núcleo USB OTG utilizado es: para el núcleo USB OTG FS, el tamaño total FIFO es de 320 * 32 bits mientras que para el núcleo USB OTG HS, el tamaño total FIFO es 1024 * 32 bits.

Ejemplo

Si la aplicación utiliza *1 IN endpoint* para el control con MPS = 64 Bytes, 1 *OUT endpoint* de control con MPS = 64 Bytes y 1 *IN endpoint* de ***Bulk*** para la clase con MPS = 512 Bytes. Los EP0 IN y OUT están configurados por la librería *USB Device*. Sin embargo, el usuario debe abrir el *endpoint IN* 1 en la capa de clase, como se muestra a continuación:

```
DCD_EP_Open (pdev,
             0x81,
             512,
             USB_OTG_EP_BULK)
```

y configurar el `TX1_FIFO_FS_SIZE` utilizando la fórmula descrita en los manuales de referencia RM0033, RM0090 y RM0008.

3.3.3.4.4 Funciones de la librería USB device

La capa ***Core*** contiene las máquinas de la librería USB device como se define en las especificaciones USB 2.0 (6). La siguiente tabla presenta los archivos principales de USB device.

Archivos	Descripción
`usbd_core (.c, .h)`	Este archivo contiene las funciones para manejar todas las comunicaciones USB y máquinas de estado.
`usbd_req( .c, .h)`	Este archivo incluye las implementaciones de solicitudes (request)
`usbd_ioreq (.c, .h)`	Este archivo maneja los resultados de las transacciones USB.
`usbd_conf.h`	Este archivo contiene la configuración del dispositivo: ID de proveedor, Id de producto, Strings, etc....

- **Funciones de archivos *usbd_core(.c, .h)***

Funciones	Descripción
`void USBD_Init (USB_OTG_CORE_HANDLE *pdev, USB_OTG_CORE_ID_TypeDef coreID, USBD_Class_cb_TypeDef *class_cb, USBD_Usr_cb_TypeDef *usr_cb)`	Inicializa la librería de dispositivos y carga el *driver* de clase y las devoluciones de llamadas del usuario.
`USBD_Status USBD_DeInit (USB_OTG_CORE_HANDLE *pdev)`	Des-inicializa la librería de dispositivos.
`uint8_t USBD_SetupStage (USB_OTG_CORE_HANDLE *pdev)`	Maneja la etapa de configuración.
`uint8_t USBD_DataOutStage (USB_OTG_CORE_HANDLE *pdev , uint8_t epnum)`	Maneja la etapa de datos de salida.
`uint8_t USBD_DataInStage (USB_OTG_CORE_HANDLE *pdev , uint8_t epnum`	Maneja la etapa de datos de entrada
`uint8_t USBD_Reset (USB_OTG_CORE_HANDLE *pdev`	Maneja el evento de reseteo.
`uint8_t USBD_Resume (USB_OTG_CORE_HANDLE *pdev`	Maneja el evento reanudación.
`uint8_t USBD_Suspend (USB_OTG_CORE_HANDLE *pdev)`	Maneja el evento de suspensión.
`uint8_t USBD_SOF (USB_OTG_CORE_HANDLE *pdev)`	Maneja el evento SOF.
`USBD_Status USBD_SetCfg (USB_OTG_CORE_HANDLE *pdev, uint8_t cfgidx)`	Configura el dispositivo y empieza la interfaz.

`USBD_Status USBD_ClrCfg (USB_OTG_CORE_HANDLE *pdev, uint8_t cfgidx)`	Borra la configuración actual.
`uint8_t USBD_IsoINIncomplete(USB_OTG_CORE_HANDLE *pdev)`	Maneja la transferencia IN isócrona incompleta
`uint8_t USBD_IsoOUTIncomplete(USB_OTG_CORE_HANDLE *pdev)`	Maneja la transferencia OUT isócrona incompleta
`uint8_t USBD_DevConnected(USB_OTG_CORE_HANDLE *pdev)`	Maneja evento de conexión del dispositivo.
`static uint8_t USBD_DevDisconnected(USB_OTG_CORE_HANDLE *pdev`	Maneja evento desconexión dispositivo.

- **Funciones del archivo *usbd_ioreq (.c, .h)***

Funciones	Descripción
`USBD_Status USBD_CtlSendData ( USB_OTG_CORE_HANDLE *pdev, uint8_t *pbuf, uint16_t len)`	Envía los datos en la tubería de control.
`USBD_Status USBD_CtlContinueSendData (USB_OTG_CORE_HANDLE *pdev, uint8_t *pbuf, uint16_t len`	Continúa enviando los datos en la tubería de control.
`USBD_Status USBD_CtlPrepareRx (USB_OTG_CORE_HANDLE *pdev, uint8_t *pbuf, uint16_t len)`	Prepara el núcleo para la recepción de datos en la tubería de control.
`USBD_Status USBD_CtlContinueRx (USB_OTG_CORE_HANDLE *pdev, uint8_t *pbuf, uint16_t len`	Continúa la recepción de datos en la tubería de control.
`USBD_Status USBD_CtlSendStatus (USB_OTG_CORE_HANDLE *pdev`	Envía un paquete de longitud cero en la tubería de control.
`USBD_Status USBD_CtlReceiveStatus (USB_OTG_CORE_HANDLE *pdev`	Recibe un paquete de longitud cero en la tubería de control.

- **Funciones del archivo *usbd_req (.c, .h)***

Funciones	Descripción
`void USBD_GetString(uint8_t *desc, uint8_t *unicode, uint16_t *len)`	Convierte una cadena ASCII a *Unicode* para dar formato a un descriptor de *String*.
`static uint8_t USBD_GetLen(uint8_t *buf)`	Devuelve la longitud de la cadena.
`USBD_Status USBD_StdDevReq (USB_OTG_CORE_HANDLE *pdev,`	Maneja peticiones de dispositivos USB estándar.

`USB_SETUP_REQ *req)`	
`USBD_Status USBD_StdItfReq (USB_OTG_CORE_HANDLE *pdev, USB_SETUP_REQ *req)`	Maneja peticiones de interfaz USB estándar.
`USBD_Status USBD_StdEPReq (USB_OTG_CORE_HANDLE *pdev, USB_SETUP_REQ *req)`	Maneja peticiones de *endpoint* USB estándar.
`static void USBD_GetDescriptor (USB_OTG_CORE_HANDLE *pdev, USB_SETUP_REQ *req)`	Maneja obtener peticiones de descriptor.
`static void USBD_SetAddress (USB_OTG_CORE_HANDLE *pdev, USB_SETUP_REQ *req)`	Establece nueva dirección de dispositivo USB.
`static void USBD_SetConfig (USB_OTG_CORE_HANDLE *pdev, USB_SETUP_REQ *req)`	Maneja el establecimiento solicitud de configuración del dispositivo.
`static void USBD_GetConfig (USB_OTG_CORE_HANDLE *pdev, USB_SETUP_REQ *req)`	Maneja Obtener solicitud de configuración del dispositivo.
`static void USBD_GetStatus (USB_OTG_CORE_HANDLE *pdev, USB_SETUP_REQ *req)`	Maneja la recepción de solicitud de estado.
`static void USBD_SetFeature (USB_OTG_CORE_HANDLE *pdev, USB_SETUP_REQ *req)`	Maneja el establecimiento de solicitud de función dispositivo.
`static void USBD_ClrFeature (USB_OTG_CORE_HANDLE *pdev, USB_SETUP_REQ *req)`	Maneja Borrar solicitud de función dispositivo.
`void USBD_ParseSetupRequest ( USB_OTG_CORE_HANDLE *pdev, USB_SETUP_REQ *req)`	Copias solicitan *buffer* en la estructura de configuración
`void USBD_CtlError ( USB_OTG_CORE_HANDLE *pdev, USB_SETUP_REQ *req)`	Maneja Errores USB en la tubería de control.

3.3.3.4.5 Interface de clase *USB Device*

La clase USB se elige durante la inicialización de la librería *USB Device* mediante la selección de la estructura de clases de devolución de llamada correspondiente. La estructura de clase se define como sigue:

```
typedef struct _Device_cb
{
      uint8_t (*Init) (void *pdev , uint8_t cfgidx);
      uint8_t (*DeInit) (void *pdev , uint8_t cfgidx);
      /* Control Endpoints*/
      uint8_t (*Setup) (void *pdev , USB_SETUP_REQ *req);
```

```
      uint8_t (*EP0_TxSent) (void *pdev);
      uint8_t (*EP0_RxReady) (void *pdev );
      /* Class Specific Endpoints*/
      uint8_t (*DataIn) (void *pdev , uint8_t epnum);
      uint8_t (*DataOut) (void *pdev , uint8_t epnum);
      uint8_t (*SOF) (void *pdev);
      uint8_t (*IsoINIncomplete) (void *pdev);
      uint8_t (*IsoOUTIncomplete) (void *pdev);
      uint8_t *(*GetConfigDescriptor)( uint8_t speed , uint16_t
*length);
       #ifdef USB_OTG_HS_CORE
      uint8_t *(*GetOtherConfigDescriptor)( uint8_t speed , uint16_t
*length);
      #endif
      #ifdef USB_SUPPORT_USER_STRING_DESC
      uint8_t *(*GetUsrStrDescriptor)( uint8_t speed ,uint8_t index,
      uint16_t *length);
      #endif
      } USBD_Class_cb_TypeDef;
```

Init: esta devolución de llamada se llama cuando el dispositivo recibe una solicitud de configuración de puesta en marcha; en esta función los *endpoints* utilizados por la interfaz de clase están abiertos.

DeInit: Este *callback* se llama cuando se ha recibido una solicitud de configuración de borrado; esta función cierra los *endpoint* utilizados por la interfaz de clase.

Setup: Este *callback* es llamado para manejar las solicitudes de configuración de clase específica.

EP0_TxSent: Este *callback* es llamado cuando se ha termina el estado del envío.

EP0_RxSent: Este *callback* es llamado cuando se termina el estado de recepción.

DataIn: Este *callback* es llamado para llevar a cabo la etapa *data IN* relativa a los *endpoints non-control*.

DataOut: Este *callback* es llamado para llevar a cabo la etapa *data OUT* relativa a los *endpoints non-control*.

SOF: Este *callback* es llamado cuando se recibe una interrupción SOF; este *callback* se puede utilizar para sincronizar algunos procesos con el inicio de trama.

IsoINIncomplete: Este *callback* es llamado cuando la última entrada isócrona en la transferencia está incompleta.

IsoOUTIncomplete: Este *callback* es llamado cuando la última salida isócrona en la transferencia está incompleta.

GetConfigDescriptor: Esta *callback* devuelve el descriptor de configuración USB.

GetOtherConfigDescriptor: Este *callback* devuelve el otro descriptor de configuración de la clase utilizada en el modo de alta velocidad.

GetUsrStrDescriptor: Este *callback* devuelve el descriptor de *string* definido por el usuario.

Interface de usuario de USB device

La librería proporciona una estructura de *callbacks* de usuario para permitir al usuario añadir código especial para manejar los eventos USB. Esta estructura de usuario se define como sigue:

```
typedef struct _USBD_USR_PROP
{
      void (*Init)(void);
      void (*DeviceReset)(uint8_t speed);
      void (*DeviceConfigured)(void);
      void (*DeviceSuspended)(void);
      void (*DeviceResumed)(void);
      void (*DeviceConnected)(void);
      void (*DeviceDisconnected)(void);
}
 USBD_Usr_cb_TypeDef;
```

Init: se llama cuando la biblioteca de dispositivos se inicia.

DeviceReset: se llama cuando el dispositivo ha detectado un evento de restablecimiento del *Host*.

DeviceConfigured: se llama cuando el dispositivo recibe una solicitud de configuración.

DeviceSuspended: se llama cuando el dispositivo ha detectado un evento de suspender desde el *Host*.

DeviceResumed: se llama cuando el dispositivo ha detectado un evento de reanudación desde el *Host*.

DeviceConnected: se llama cuando el dispositivo está conectado al *Host*.

DeviceDisconnected: se llama cuando el dispositivo se desconecta de la *Host*.

La librería ofrece estructuras de *callbacks* de descriptores para permitir al usuario gestionar los dispositivos y los descriptores de *string* en tiempo real. Esta estructura de descriptores se define como sigue:

```
typedef struct _Device_TypeDef
{
      uint8_t *(*GetDeviceDescriptor)( uint8_t speed , uint16_t
      *length); uint8_t *(*GetLangIDStrDescriptor)( uint8_t speed ,
      uint16_t *length); uint8_t *(*GetManufacturerStrDescriptor)(
      uint8_t speed , uint16_t *length);
      uint8_t *(*GetProductStrDescriptor)( uint8_t speed , uint16_t
      *length);
```

```
      uint8_t *(*GetSerialStrDescriptor)( uint8_t speed , uint16_t
      *length);
      uint8_t *(*GetConfigurationStrDescriptor)( uint8_t speed ,
      uint16_t *length);
      uint8_t *(*GetInterfaceStrDescriptor)( uint8_t speed , uint16_t
      *length);
      #ifdef USB_SUPPORT_USER_STRING_DESC
            uint8_t* (*Get_USRStringDesc) (uint8_t speed, uint8_t idx ,
            uint16_t *length);
      #endif /* USB_SUPPORT_USER_STRING_DESC */
} USBD_DEVICE, *pUSBD_DEVICE;
```

GetDeviceDescriptor: Devuelve el descriptor de dispositivo.

GetLangIDStrDescriptor: Devuelve el descriptor de *string* del ID de idioma.

GetManufacturerStrDescriptor: Devuelve el descriptor de *string* del fabricante

GetProductStrDescriptor: Devuelve el descriptor de *string* del producto.

GetSerialStrDescriptor: Devuelve el descriptor de *string* del número de serie.

GetConfigurationStrDescriptor: Devuelve el descriptor de *string* de configuración.

GetInterfaceStrDescriptor: Devuelve el descriptor de *string* de interface.

Get_USRStringDesc: Devuelve el descriptor de *string* definido por el usuario.

Nota: El archivo *usbd_desc.c* proporcionada ejemplos de dispositivos USB que implementan estos cuerpos de devolución de llamada.

3.3.3.4.6 Clases *USB device*.

El módulo de clase contiene todos los archivos relativos a la implementación de la clase. Cumple con la especificación del protocolo incorporado en estas clases.

La siguiente tabla presenta los archivos de clase de dispositivo USB para MSC y clases HID.

Clases	Archivos	Descripción
HID	`usbd_hid (.c, .h)`	Este archivo contiene las devoluciones de llamada de la clase HID (driver) y los descriptores de configuración relativos a esta clase.
MSC	`usbd_msc( .c, .h)`	Este archivo contiene las devoluciones de llamada clase MSC (driver) y los descriptores de configuración relativos a esta clase.
	`usbd_bot (.c, .h)`	Este archivo maneja el único protocolo de transferencia *bulk* (masivo).
	`usbd_scsi (.c, .h)`	Este archivo se encarga de los comandos SCSI.
	`usbd_info (.c,.h)`	Este archivo contiene las páginas de consulta vitales y los datos sensados de los dispositivos de almacenamiento masivo.
	`usbd_mem.h`	Este archivo contiene los prototipos de las funciones llamadas de la capa SCSI para tener acceso a los

		medios físicos
DFU	`usbd_dfu_core (.c,.h)`	Este archivo contiene las devoluciones de llamada clase DFU (driver) y los descriptores de configuración relativos a esta clase.
	`usbd_flash_if (.c,.h)`	Este archivo contiene las *callbacks* clase DFU en relación con la interfaz de memoria flash interna.
	`usbd_otp_if (.c,.h)`	Este archivo contiene las *callbacks* clase DFU con respecto a la interfaz de memoria OTP.
	`usbd_template_if (.c,.h)`	Este archivo proporciona un driver de plantilla que le permite implementar interfaces de memoria adicionales.
Audio	`usbd_audio_core (.c,.h)`	Este archivo contiene las devoluciones de llamada clase AUDIO (driver) y los descriptores de configuración relativos a esta clase.
	`usbd_audio_out_if (.c,.h)`	Este archivo contiene la capa más baja del driver de salida de audio (de host USB para altavoz de salida).
CDC	`usbd_cdc_core (.c,.h)`	Este archivo contiene las devoluciones de llamada clase CDC (driver) y los descriptores de configuración relativos a esta clase.
	`usbd_cdc_if_templat e (.c,.h)`	Este archivo proporciona un driver de plantilla que le permite implementar funciones de la capa baja para CDC terminal.

Clase HID

Implementación de la clase HID

Este módulo gestiona la clase HID siguiendo el documento "*Device Class Definition for Human Interface Devices (HID) Version 1.11 Junio 27,2001*" (4), en esta memoria podemos encontrar un resumen de este documento en el apartado 3.3.2. Este driver implementa los siguientes aspectos de las especificaciones:

- La subclase interfaz de arranque
- El protocolo de ratón
- *Usage Page* escritorio genérico
- *Usage*: palanca de mando
- Colección: aplicación

Interfaz de usuario HID

El `USBD_HID_SendReport` puede ser utilizado por la aplicación para enviar reports HID, el driver HID, en esta versión, maneja solo tráfico de entrada. Un ejemplo de uso de esta función se muestra a continuación:

```
static uint8_t HID_Buffer [4];
USBD_HID_SendReport (&USB_OTG_FS_dev, USBD_HID_GetPos(), 4);
static uint8_t *USBD_HID_GetPos (void)
{
      HID_Buffer[0] = 0;
      HID_Buffer[1] = GetXPos();
      HID_Buffer[2] = GetXPos();
```

```
      HID_Buffer[3] = 0;
      return HID_Buffer;
}
```

Archivos del núcleo HID

Funciones	Descripción
`static uint8_t USBD_HID_Init (void *pdev, uint8_t cfgidx)`	Inicializa la interfaz HID y abre los *endpoints* utilizados.
`static uint8_t USBD_HID_DeInit (void *pdev, uint8_t cfgidx)`	Des-Inicializa la capa de HID y cierra los *endpoints* utilizados.
`static uint8_t USBD_HID_Setup (void *pdev, USB_SETUP_REQ *req)`	Maneja las solicitudes específicas de HID.
`uint8_t USBD_HID_SendReport (USB_OTG_CORE_HANDLE *pdev, uint8_t *report, uint16_t len)`	Envía información de HID.

3.4 Pulsioxímetro

3.4.1 CONCEPTOS DE LA OXIMETRÍA DE PULSO O PULSIOXIMETRÍA

La oximetría de pulso, o pulsioximetría, es actualmente un importante método para la monitorización, no invasiva, de la saturación de oxígeno en sangre (SaO2) porque ofrece una lectura fiable y constante de la saturación de la hemoglobina arterial (oxihemoglobina). Además de ofrecernos información del ritmo cardíaco (7).

El control continuo de los signos vitales, las condiciones hemodinámicas, respiratorias y hemogasométricas constituyen pilares básicos de la vigilancia intensiva a la que se someten los pacientes.

3.1.1.1 Conceptos básicos

Con el fin de poder entender con mayor detalle la finalidad de esta parte del proyecto, en este apartado se ofrece información sobre todos los conceptos que se han creído relevantes para el buen entendimiento de este.

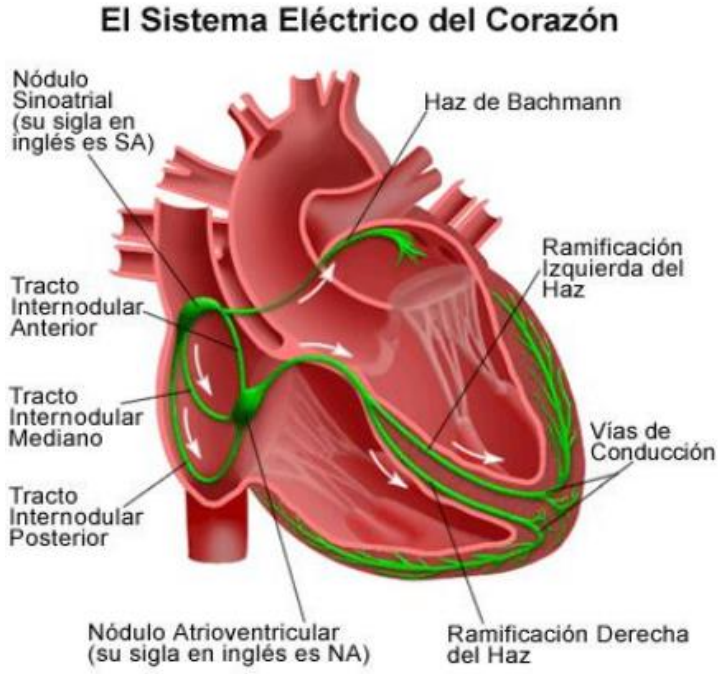
3.1.1.2 La Hemoglobina

El cuerpo humano está compuesto de diferentes sistemas que se encargan de regular el correcto funcionamiento del organismo. Uno de ellos es el sistema circulatorio el cual se encarga de hacer llegar, a través de las venas y las arterias, la sangre a todas las partes del cuerpo. La sangre saturada se encarga de llevar el oxígeno (O2), a través de las arterias, hasta los tejidos y células del organismo donde lo descarga y recoge productos de desecho, como el dióxido de carbono (CO2), generados por el metabolismo, regresando a los pulmones a través de las venas y los capilares donde cede el CO2 para su eliminación y se satura nuevamente de O2 . Cuando la hemoglobina está saturada de O2 se denomina oxihemoglobina o hemoglobina oxigenada (HbO2) dando el color rojo vivo a la sangre arterial. Cuando la hemoglobina ha perdido el O2 se la conoce como hemoglobina reducida (Hb) lo que ocasiona el color rojo azulado oscuro típico de la sangre que circula a través de las venas. Esta diferencia de color entre la HbO2 y la Hb es la que nos permitirá más adelante calcular la SaO2.

3.1.1.3 Frecuencia cardiaca

La frecuencia cardiaca, o pulso, es el número de latidos cardíacos por minuto. Estos latidos son generados por el corazón y se deben a que es una bomba de tejido muscular y como cualquier bomba, el corazón, necesita una fuente de energía para poder funcionar (8).

La acción de bombeo del corazón proviene de un sistema integrado de conducción eléctrica como el que se muestra en la siguiente figura obtenida de (8).



En condiciones normales, genera un impulso eléctrico cada vez que el corazón late: entre 60 y 190 veces por minuto; en función de la edad del individuo y de su grado de actividad (ver siguiente cuadro).

Latidos del corazón, en reposo, en mujeres y hombres

Edad	FCME`	Pulsación habitual (50%-75% de la FCME)
60 años	160 latidos por minuto	80 - 120 lpm
50 años	170 latidos por minuto	85 - 127 lpm
40 años	180 latidos por minuto	90 - 135 lpm
30 años	190 latidos por minuto	95 - 142 lpm
20 años	200 latidos por minuto	100 - 150 lpm
10 años	210 latidos por minuto	110 - 155 lpm

FCME= Frecuencia cardiaca máxima.

Debido a este estímulo la sangre se comporta como un fluido pulsátil, lo que permite que se realice la medición usando el pulsioxímetro.

3.4.2 Fundamentos de la Oximetría de Pulso

El oxímetro de pulso es, probablemente, una de la mejores herramientas de monitoreo que hayan sido desarrolladas en los últimos años y brinda información no solo de la saturación de la hemoglobina, sino también de la frecuencia y ritmo del pulso periférico.

El pulso oximétrico a menudo se considera la quinta muestra vital, después del ritmo cardíaco, la presión arterial, temperatura y frecuencia respiratoria. Sirve como herramienta importante para el asistente sanitario proporcionando un control continuo de la saturación arterial del oxígeno del paciente (SaO2) (9).

La oximetría de pulso se fundamenta en la espectrofotometría y la pletismografía. La espectrofotometría, permite calcular la concentración de una sustancia en solución, a partir de su absorción óptica, a una longitud de onda determinada; y la pletismografía, permite medir los cambios en el flujo sanguíneo o el volumen de aire en diferentes partes del cuerpo.

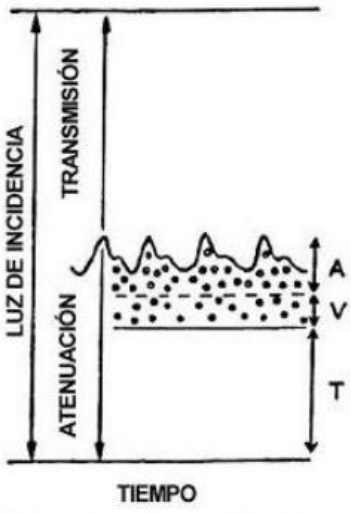
La sustancia que se está analizando se ilumina y se mide la absorción de luz de longitudes de onda específicas, al pasar por un lecho vascular arterial pulsátil y a partir de esa medida se calcula la concentración de oxígeno.

Dicha técnica analítica también establece que para analizar dos sustancias en solución se necesitan, como mínimo, dos longitudes de onda.

En el caso de la sangre, hay dos sustancias relevantes a la oxigenación que son: la hemoglobina reducida (Hb) y la oxihemoglobina (HbO2). Como son dos, los oxímetros requieren de cómo mínimo dos longitudes de onda: típicamente una roja y otra infrarroja.

En la espectrofotometría clásica, el análisis se realiza en cubetas de vidrio en las cuales se deposita la muestra de sangre arterial. En cambio el oxímetro de pulso utiliza, por ejemplo, el dedo mismo del paciente como cubeta.

Esto conlleva a que la luz tiene que atravesar a parte de la sangre oxigenada (arterial) y la reducida (venosa), otros tejidos, como huesos, uñas y la piel.

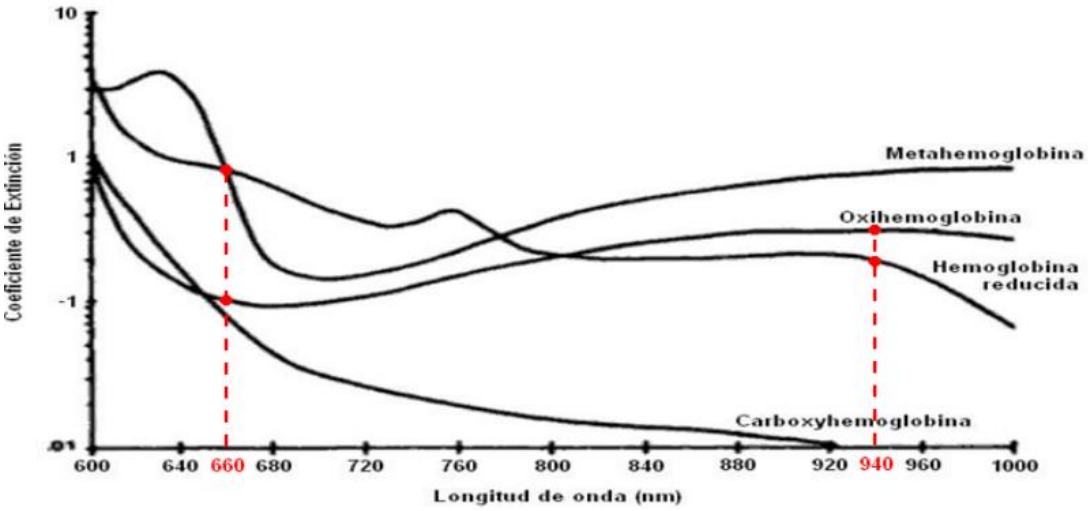


Transmisión de la luz a través de un dedo cuando la atenuación es debida a la sangre arterial (A), sangre venosa (V) y tejidos (T)

Para distinguir la sangre arterial entre todos estos tejidos, el oxímetro de pulso utiliza la técnica descubierta en 1974 por el japonés Takuo Aoyagi que observó que la variación del volumen de la sangre arterial con cada pulso se podría utilizar para obtener una señal dependiente sólo de las características de dicho tipo sangre. Por ello analiza únicamente la parte pulsátil de la señal óptica que se debe exclusivamente a la sangre arterial.

Es por esta razón que se le agrega la palabra “PULSO” cuando nos referimos al oxímetro. Si no hay ritmo cardíaco, el oxímetro no puede distinguir la sangre arterial, que es el objeto de su análisis.

Con anterioridad se ha comentado que los oxímetros de pulso utilizan dos longitudes de onda una roja (660nm) y una infrarroja (940nm). Esto se debe a que a esas dos longitudes de onda, los coeficientes de extinción de la HbO2 y de la Hb, tienen valores más dispares entre ellos como podemos apreciar en la siguiente figura.



Coeficientes de extinción de la hemoglobina.

Por lo tanto, la desoxigenación de la sangre presenta una extinción óptica superior en la región del rojo del espectro de luz que la oxihemoglobina y una menor absorción óptica en la región del infrarrojo cercano. Estas direcciones opuestas en la absorción de los dos colores al variar la oxigenación contribuyen a que haya cambios ópticos apreciables más fáciles de medir.

El modelo matemático para el pulsioxímetro se basa en medir el tiempo en que la intensidad de luz pasa a través del tejido fino como por ejemplo la extremidad del dedo o del lóbulo de la oreja. El procesado de la señal se basa en este modelo simple y en la ley de Beer-Lambert.

3.4.2.1 Espectrofotometría

La espectrofotometría es el método de análisis cuantitativo, más usado en las investigaciones químicas y biológicas, que utiliza los efectos de la interacción de las radiaciones electromagnéticas con la materia para medir la absorción o la transmisión de luz por las sustancias y de esta forma medir la concentración de estas.

3.4.2.2 Pletismografía

La pletismografía es una técnica de diagnóstico que consiste en determinar las variaciones del flujo sanguíneo (volumen o presión) en una arteria o vena mediante el uso de un pletismógrafo. Hay diversos métodos pletismográficos que valoran los cambios de volumen de forma indirecta mediante la utilización de diversos principios físicos:

- Pletismografía de agua.
- Pletismografía de aire.
- Pletismografía de impedancia.
- Pletismografía de anillos de mercurio.

Cuando, como en nuestro caso, nos servimos de técnicas ópticas para el cálculo de dichas variaciones, hablamos de fotopletismografía (PPG).

3.4.2.3 Ley de Beer-Lambert

El principio en el que se basa la determinación de la saturación de O2, con el oxímetro de pulso, es la ley de Beer-Lambert. La ley de Beer declara que la cantidad de luz absorbida por un cuerpo, depende de la concentración, de este, en la solución y la ley de Lambert declara que la cantidad de luz absorbida por un objeto, depende de la distancia recorrida por la luz.

La ley de Beer-Lambert (ver siguiente cuadro), es la combinación de las dos leyes y declara que la absorbancia de una radiación monocromática a través de una sustancia en solución, depende de la concentración del compuesto absorbente, de la distancia recorrida por la radiación y del tipo de radiación monocromática. Y su ecuación matemática es [1.1]:

$$A = -log_{10}\, T = -log_{10}\left(\frac{I}{I_0}\right) = -log_{10}\, 10^{-abc} = a \cdot b \cdot c \qquad [1.1]$$

Ley de Beer-Lambert

$A = abc$	$A = \varepsilon bc'$
A = Absorbancia	A = Absorbancia
a = Longitud de onda del coef. de absorción	ε = Longitud de onda dependiente del coef. de absorción molar ($Lmol^{-1}cm^{-1}$)
b = Longitud de la trayectoria (cm)	b = Longitud de la trayectoria (cm)
c = Concentración del compuesto (g/L)	c' = Concentración del compuesto ($mol\ L^{-1}$)

Gracias a esta ley, se puede relacionar la cantidad de luz incidente en el dedo, la de O2 presente en la sangre y la de luz captada por el fotodetector.

3.4.3 Funcionamiento del pulsioxímetro

La operación principal de un oxímetro de pulso es la determinación de la saturación de oxígeno en sangré en una persona. La saturación de oxígeno arterial, o SaO2, es el porcentaje de hemoglobina arterial funcional que se oxigena.

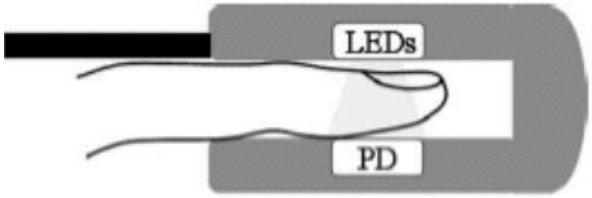
La Hemoglobinas funcional es un tipo de hemoglobina que es capaz de unirse con el oxígeno. Las Hemoglobinas no funcionales no pueden unirse al oxígeno. Un ejemplo de hemoglobina no funcional es carboxihemoglobina (COHb), que se une fácilmente con monóxido de carbono.

Cuando una hemoglobina funcional se une con cuatro moléculas de oxígeno, se considera una hemoglobina oxigenada (HbO2). Cuando se une a menos de cuatro moléculas de oxígeno, se considera reducida (Hb) (10).

La Saturación de oxígeno medida con un pulsioxímetro se llama a menudo SpO2 porque es una estimación basada en mediciones periféricas y una suposición de que sólo HbO2 y Hb están presentes en la sangre. La presencia de hemoglobinas no funcionales tales como COHb puede causar mediciones erróneas. Por lo tanto, SpO2 es una medida diferente a SaO2 (10).

$$S_pO_2 = \frac{HbO_2}{Hb + HbO_2}$$

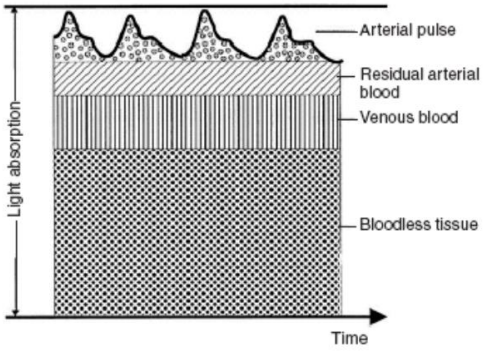
La hemoglobina oxigenada y reducida difieren en su absorción de la luz, la pulsioximetría se basa en encontrar los niveles relativos de las dos hemoglobinas. La oximetría de pulso más común utiliza un LED rojo y un LED infrarrojo para emitir luz a través de un lecho capilar pulsátil, por ejemplo un dedo. Un fotodetector se coloca en el lado opuesto para medir la luz roja e infrarroja transmitida.



Los LED rojos e infrarrojos emiten luz alternativamente , cuando uno emite el otro está apagado, de este modo se puede utilizar un único foto-detector para medir la intensidad de luz de ambos LEDs.

Con una medida conocida de luz roja e infrarroja transmitida a través del dedo, se puede determinar una estimación de la hemoglobina oxigenada y la hemoglobina reducida en base a la absorción de las distintas longitudes de onda de la luz transmitida. Un oxímetro típico trabaja con la luz roja 660nm, 940nm y la luz infrarroja. En 660nm, hemoglobina reducida absorbe unas diez veces más luz que no oxigenada.

Debido a que el flujo de la sangre es pulsátil por naturaleza, la luz transmitida varía con el tiempo. Un dedo normal tiene absorción de luz a través de tejido sin sangre, sangre venosa y arterial. El volumen de sangre arterial cambia con el pulso, por lo tanto la absorción de luz también cambia. Por ello, el detector de luz verá una gran señal DC que representa la sangre arterial residual, la sangre venosa, y el tejido sin sangre. Una pequeña porción de la señal detectada (~ 1%), será una señal de AC que representa el pulso arterial. Debido a que esta es la única señal de corriente alterna, la parte arterial de la señal puede ser diferenciada. Esta señal de AC se separa con el filtrado simple y un se puede calcular su valor RMS (10).

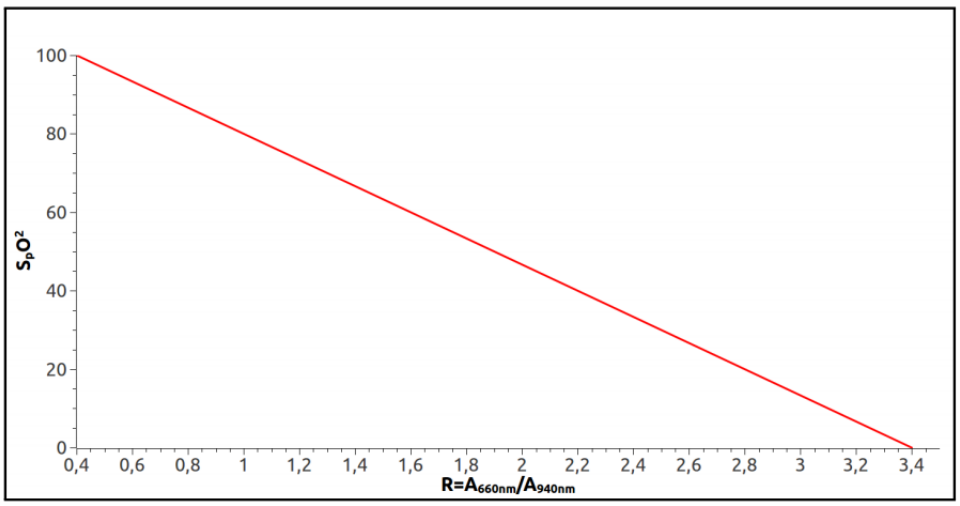


Light Absorption by Tissue Type

Un valor intermedio, conocido como ratio normalizado R, se calcula desde las señales leídas de la sonda pulsioximétrica (11).

$$R = \frac{(\frac{ACrms66nm}{DC660nm})}{(\frac{ACrms940nm}{DC940nm})}$$

Este valor representa una relación de reducción de la hemoglobina arterial oxigenada. Usando este valor, y en base a unos datos empíricos, se puede calcular un valor de saturación de oxígeno.



: Curva empírica sobre la saturación del oxígeno (vs. R)

Esta curva empírica puede ser aproximada por una ecuación lineal:

$$S_pO_2 = 110 - 25 \times R$$

Llegados a este punto pueden surgir dos interrogantes:

- ¿Cómo obtener ACrms66nm, DC660nm , ACrms940nm, DC940nm?
- ¿Cómo se determina la frecuencia cardiaca (pulsación)?

La respuesta a ambas preguntas es: mediante una FFT (Transformada Rápida de Fourier)

3.4.4 Obtención de resultados mediante FFT (Transformada Rápida de Fourier).

3.4.4.1 FFT: Definición, Características y Limitaciones.

FFT es la abreviatura usual (del inglés Fast Fourier Transform) de un eficiente algoritmo que permite calcular la transformada de Fourier discreta (DFT) y su inversa. La FFT es de gran importancia en una amplia variedad de aplicaciones, desde el tratamiento digital de señales y filtrado digital en general a la resolución de ecuaciones en derivadas parciales o los algoritmos de multiplicación rápida de grandes enteros. El algoritmo pone algunas limitaciones en la señal y en el espectro resultante. Por ejemplo: la señal de la que se tomaron muestras y que se va a transformar debe consistir de un número de muestras igual a una potencia de dos.

Fourier en el siglo XVIII desarrolla la ecuación que explica la naturaleza compleja de las forma de onda. En su forma más básica expone que cualquier forma de onda compleja puede caracterizarse como una combinación de ondas sinusoidales individuales con componentes de amplitud y fase definidas. La Transformada de Fourier convierte la onda de amplitud-tiempo en amplitud-frecuencia y a la inversa.

Características y limitaciones

- **FFT**: convierte los datos grabados en el tiempo en datos de respuesta frecuencial.

- **Frecuencia de muestreo**: es el reloj de frecuencia de la conversión analógico-digital.Es decir si utilizamos una frecuencia de muestreo de 48KHz estamos tomando 48000 muestras por segundo lo que equivale a tomar una muestra cada 1/48000 = 0,02ms, todo dato que caiga entre 2 muestras el analizador no lo podrá leer.

- **Frecuencia de Nyquist**: Es la frecuencia más alta que puede ser capturada por el analizador (la mitad de la frecuencia de muestreo).

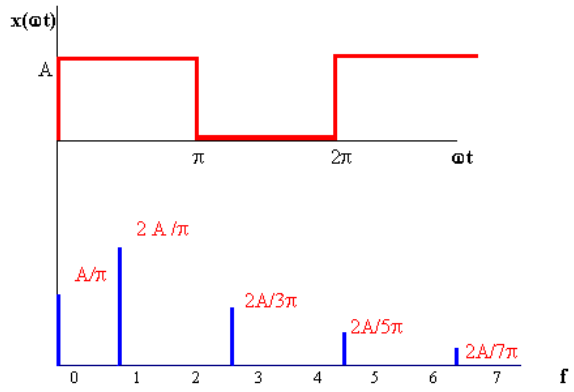
- **Ventana de tiempo (TC)**: Es el periodo de tiempo sobre el cual se muestrea una forma de onda. la duración de la ventana de tiempo tiene un factor decisivo en cuál será la menor frecuencia medible. **TC=1/FR**

- **Tamaño de FFT**: son el número de datos de la ventana de tiempo, la cantidad de datos y la frecuencia de muestreo determinará la constante de tiempo **(TC)** y la frecuencia de resolución **(FR)**

- **Frecuencia de resolución**: Es el cociente entre la frecuencia de muestreo y el tamaño del FFT y su valor se da en Hz.

3.4.4.2 Obtención de resultados

Mediantes la FFT se obtiene el espectro en frecuencia de la señal muestreada. Por ejemplo, si hacemos la FFT de la siguiente señal.



Podemos observar en el espectro que el primer elemento corresponde con el nivel de DC, el segundo elemento corresponde a la frecuencia fundamental (frecuencia de la señal) y el resto de elementos son los armónicos.

Por lo tanto en nuestro proyecto, bastará con obtener el elemento predominante (mayor amplitud) al realizar la FFT para determinar la frecuencia de cualquiera de las señales que recibe el fotodetector (Producida por LED rojo o infrarrojo), que corresponde a la frecuencia cardiaca del sujeto bajo test.

Para la obtención del porcentaje de oxígeno en necesitamos conocer los valores de ACrms66nm, DC660nm , ACrms940nm, DC940nm. Que son los valores de AC y DC de cada señal (LED rojo e infrarrojo). Para ello bastará con realizar la FFT de cada señal obteniendo en el primer elemento el nivel de DC y en el elemento de mayor amplitud la amplitud máxima AC, con estos datos ya se pueden calcular ACrms66nm, DC660nm , ACrms940nm, DC940nm y por lo tanto el Ratio normalizado, con el que se puede obtener en una tabla ponderada el porcentaje de oxígeno en sangre.

3.4.5 Limitaciones de los pulsioxímetros

Debido a que la oximetría de pulso mide la saturación de O2 por métodos espectrofotométricos, existen factores que limitan su uso. Ciertas condiciones pueden resultar en lecturas no reales, incorrectas o poco informativas.

- Dishemoglobinemias: Otras moléculas presentes en la sangre afectan la exactitud de los valores obtenidos. Tales como:
    - Anemias.
    - Hemoglobina fetal.
    - Carboxihemoglobina y metahemoglobina.
- Colorantes. El radio de absorción puede ser afectado por cualquier sustancia presente en la sangre que absorba luz entre 660 y 940 nm.

Existe también un número importante de interferencias a considerar cuando se usa el pulsioxímetro. Las causas más comunes incluyen:

- Movimiento.
- Luz quirúrgica.
- Esmalte de uñas.
- Micosis ungulares e hiperpigmentación de la piel.
- Vasoconstricciones e hipotermia.
- Arritmias cardíacas.

4. Implementación y prueba

4.1 Software y flujogramas

En este apartado se presenta los flujogramas del gestor de interrupciones y del programa principal del software del microcontrolador, así como una pequeña explicación de los mismos.

Este apartado se ha incluido en primer lugar para tener una visión más general del proyecto ya que en los siguientes apartados se hará referencia a algunas partes del código y puede ser de ayuda a la hora situarse y entender mejor cada apartado.

El código que compone el programa del dispositivo está dividido en varios ficheros que podemos clasificar en grupos:

- Librerías de periféricos y de la placa STM32F4 discovery.
- Librerías USB OTG y USB HID.
- Programa principal "*main*"
- Gestor de interrupciones "*stm32f4xx_it*"

No se incluirán los flujogramas de algunas funciones auxiliares sencillas cuyo nombre es suficientemente descriptivo para entender su función así como funciones de configuración de periféricos ya que las configuraciones de los periféricos se explicaran en apartados posteriores.

Para más detalle se puede consultar el apartado 7 Apendice, donde podemos encontrar todo el código del programa principal "*main.c*" y del gestor de interrupciones "*stm32f4xx_it.c*".

Programa Principal "*main.c*"

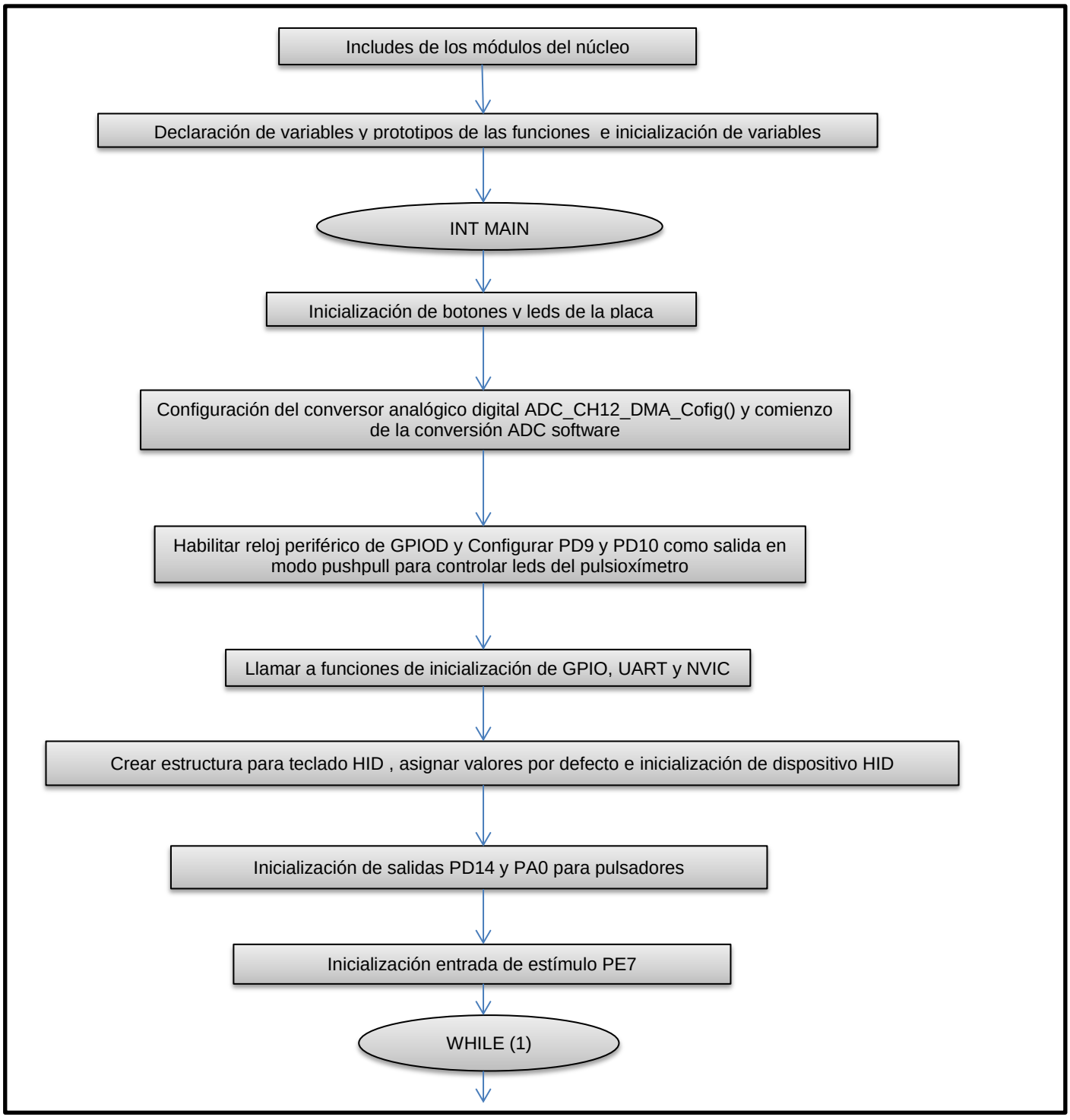
En este fichero es donde se configuran e inicializan todos los periféricos del proyecto y se comprueba por polling distintas situaciones.

Se comprueba continuamente si el dispositivo está conectado como USB HID y si los drivers son correctos, de ser así el programa comprueba continuamente si se pulsa o suelta un pulsador o si se inicia o acaba una entrada de estímulo, de ser así, se almacena en el vector de eventos el tiempo en que se produjo el evento.

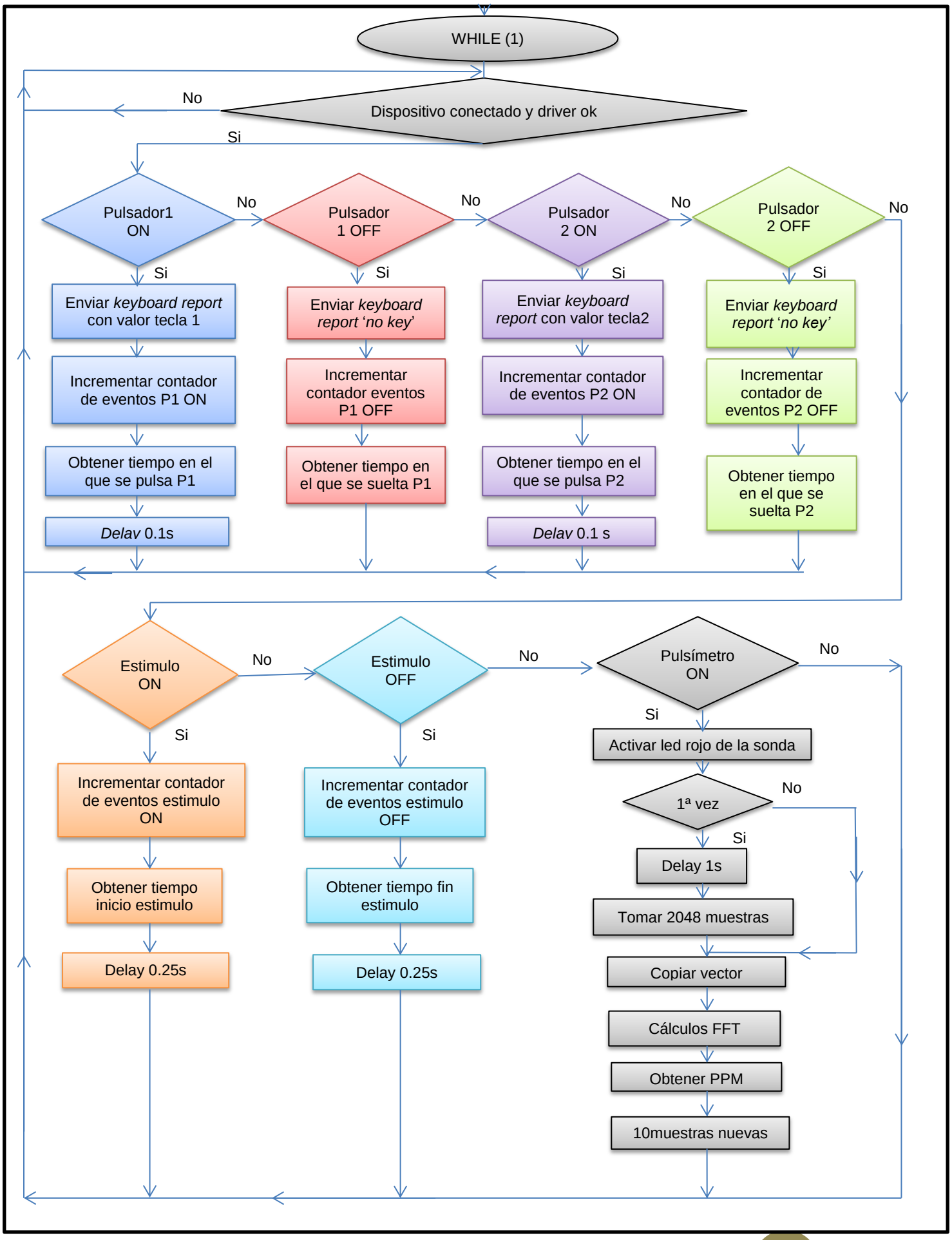
También comprueba si está activado el pulsímetro y de ser así ejecuta el código para calcular la frecuencia cardiaca.

Se incluyen varios *delay* (retardos) para quitar el rebote de los pulsadores, así como de la entrada de estímulo y en el pulsímetro, ya que en los primeros instantes se toman muestras erróneas de la luz que recibe el fotosensor producida por el led rojo de la sonda de pulsioxímetro.

main.c (declaraciones y configuraciones)



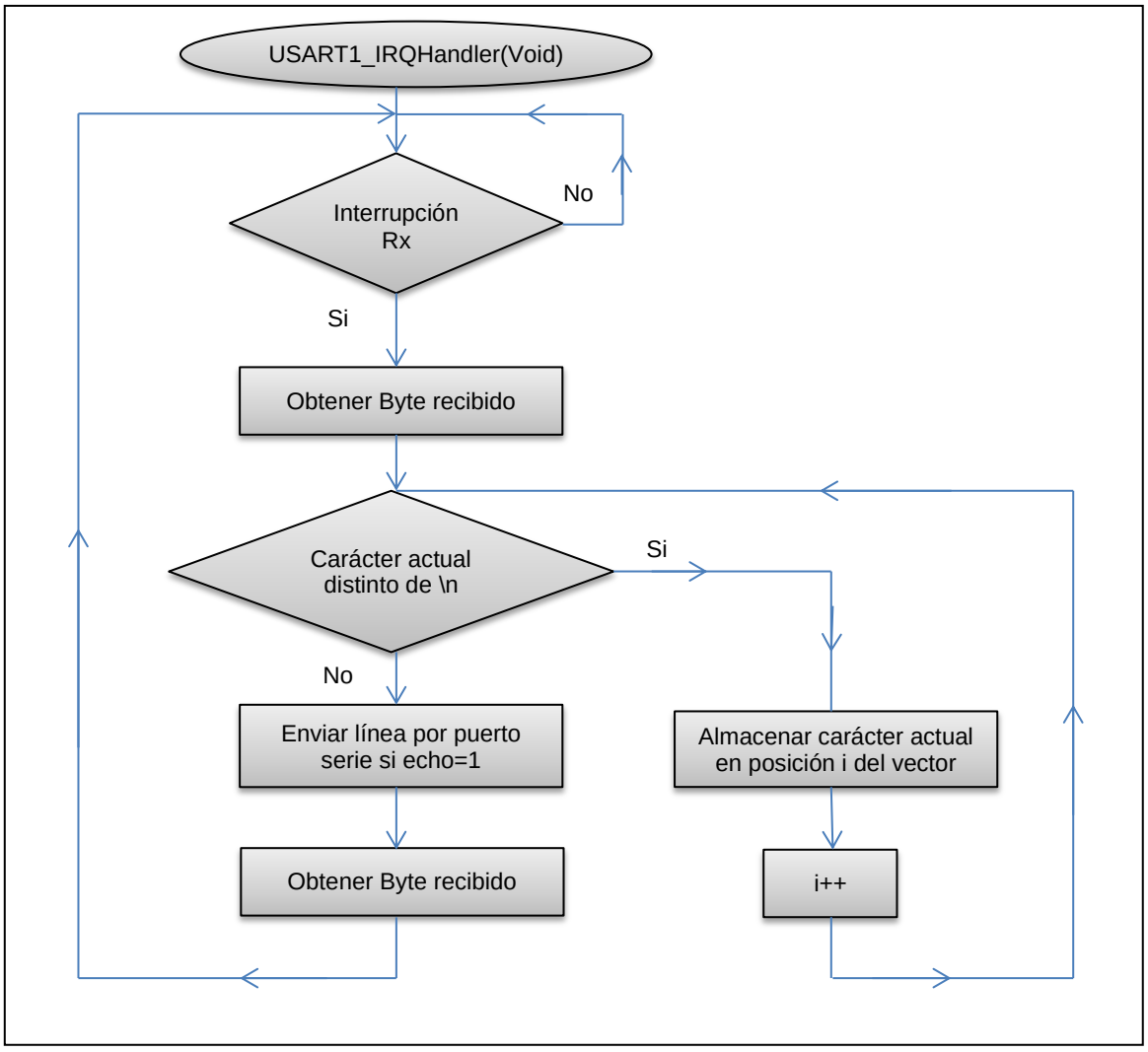
Main.c (bucle infinito while (1))


WHILE (1)
Dispositivo conectado y driver ok
No
Si
Pulsador1 ON
Enviar keyboard report con valor tecla 1
Incrementar contador de eventos P1 ON
Obtener tiempo en el que se pulsa P1
Delay 0.1s
Pulsador 1 OFF
Enviar keyboard report 'no key'
Incrementar contador eventos P1 OFF
Obtener tiempo en el que se suelta P1
Pulsador 2 ON
Enviar keyboard report con valor tecla2
Incrementar contador de eventos P2 ON
Obtener tiempo en el que se pulsa P2
Delay 0.1 s
Pulsador 2 OFF
Enviar keyboard report 'no key'
Incrementar contador de eventos P2 OFF
Obtener tiempo en el que se suelta P2
Estimulo ON
Incrementar contador de eventos estimulo ON
Obtener tiempo inicio estimulo
Delay 0.25s
Estimulo OFF
Incrementar contador de eventos estimulo OFF
Obtener tiempo fin estimulo
Delay 0.25s
Pulsímetro ON
Activar led rojo de la sonda
1ª vez
Delay 1s
Tomar 2048 muestras
Copiar vector
Cálculos FFT
Obtener PPM
10muestras nuevas

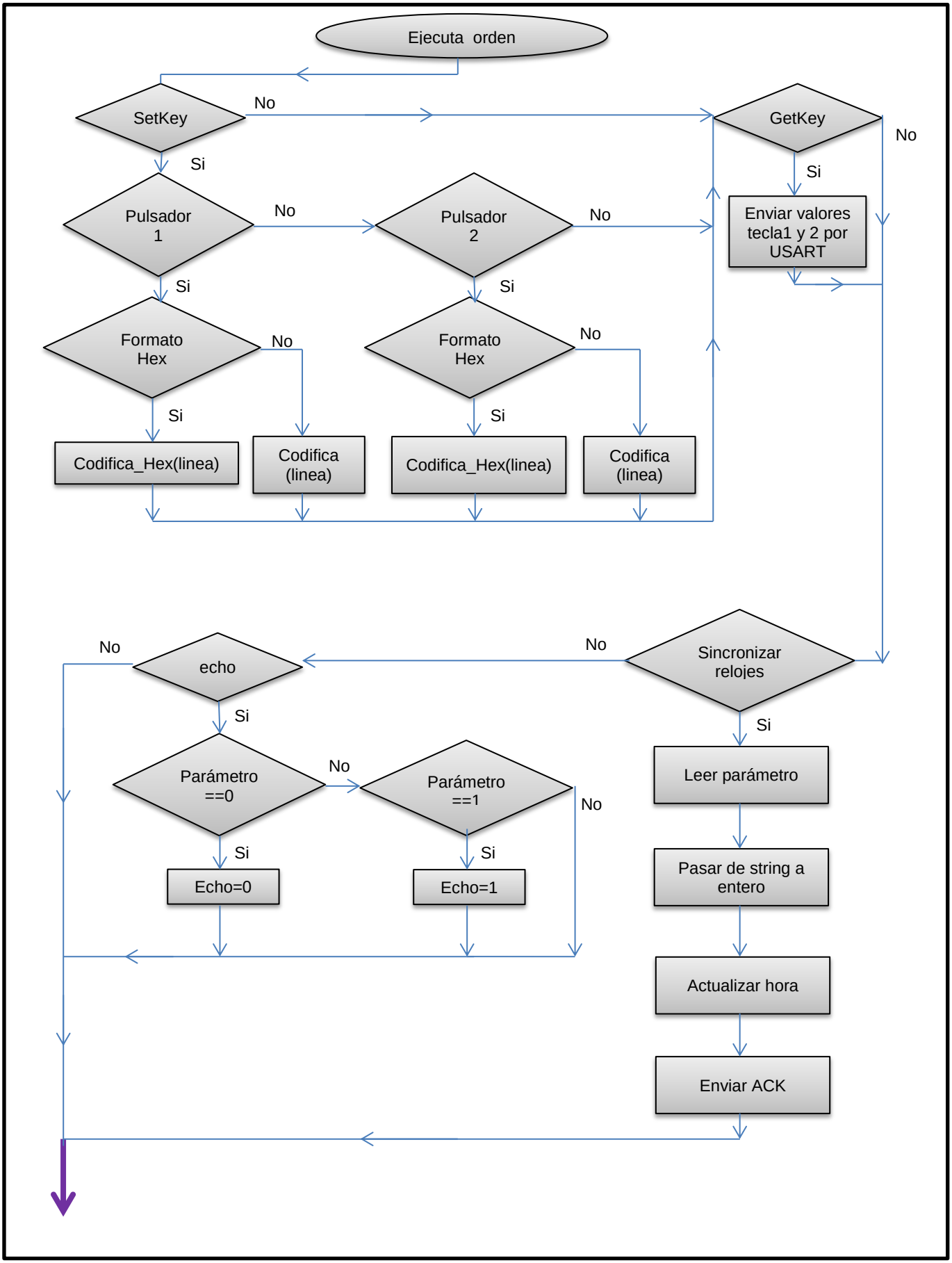
Gestor de interrupciones "*stm32f4xx_it.c*"

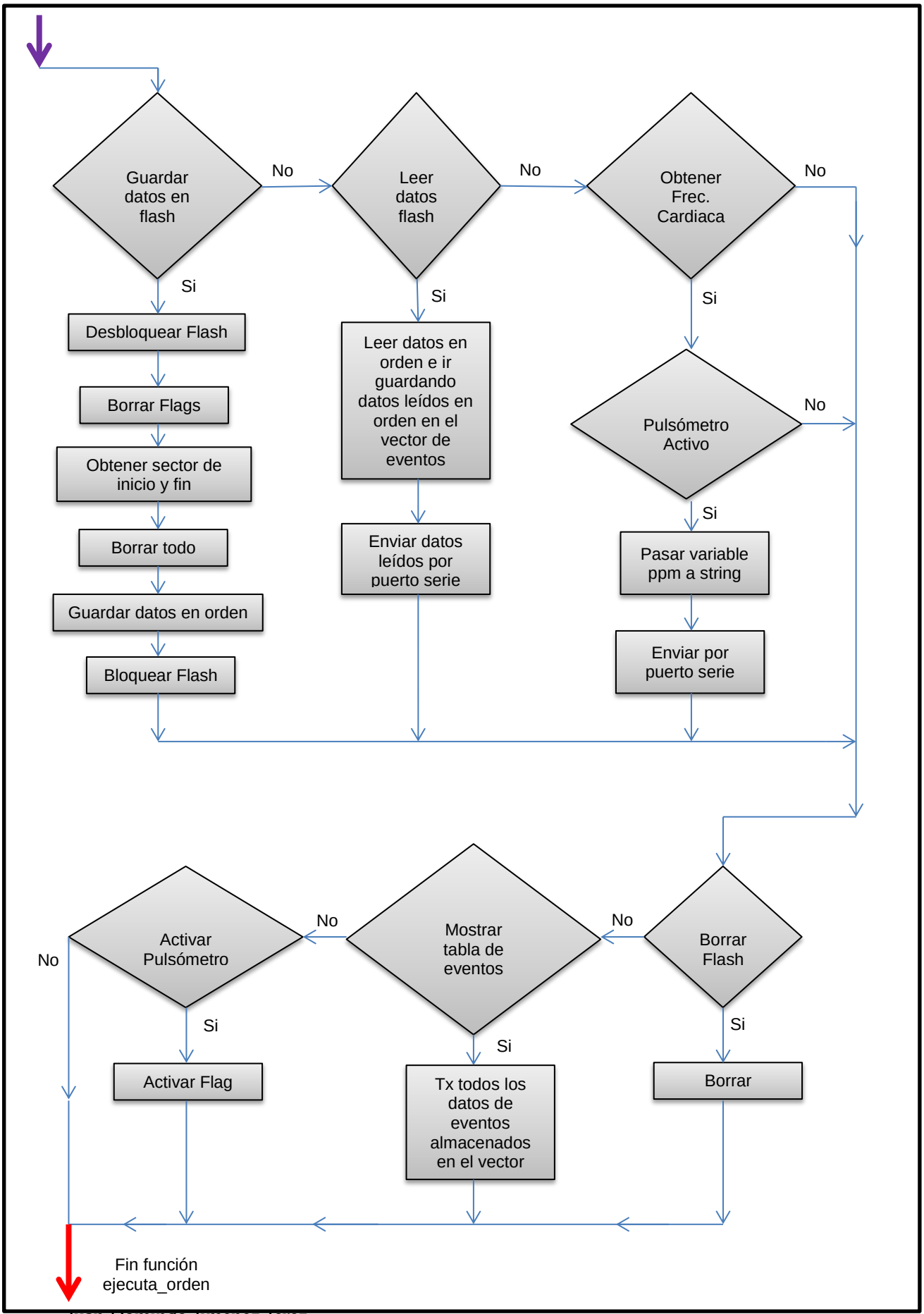
Este fichero contiene las funciones que son llamadas cuando se producen distintas interrupciones tanto internas como interrupciones producidas por los periféricos. No están incluidas algunas funciones auxiliares que se explicaran en apartados posteriores y cuyo código podemos encontrar en el **apartado 7.2**.

USART1_IRQHandler(Void) es la función que gestiona las interrupciones USART1. Por lo tanto esta función será llamada cada vez que se reciban datos por el puerto serie, o lo que es lo mismo, cada vez que se reciba una orden desde la "consola". La función de esta interrupción es almacenar la línea de código recibida por el puerto serie en un vector llamado *received_str[]* y por último se llamará a la función `ejecuta_orden(received_str)`, pasándole como argumento el vector con la línea recibida.



A continuación el flujograma que describe la función *ejecuta_orden():*


Ejecuta orden
SetKey
No
Si
GetKey
No
Si
Pulsador 1
No
Pulsador 2
No
Si
Si
Enviar valores tecla1 y 2 por USART
Formato Hex
No
Formato Hex
No
Si
Si
Codifica_Hex(linea)
Codifica (linea)
Codifica_Hex(linea)
Codifica (linea)
No
echo
No
Sincronizar relojes
Si
Si
Parámetro ==0
No
Parámetro ==1
No
Leer parámetro
Si
Si
Echo=0
Echo=1
Pasar de string a entero
Actualizar hora
Enviar ACK


Guardar datos en flash
No
Leer datos flash
No
Obtener Frec. Cardiaca
No
Si
Si
Si
Desbloquear Flash
Borrar Flags
Obtener sector de inicio y fin
Borrar todo
Guardar datos en orden
Bloquear Flash
Leer datos en orden e ir guardando datos leídos en orden en el vector de eventos
Enviar datos leídos por puerto serie
Pulsómetro Activo
No
Si
Pasar variable ppm a string
Enviar por puerto serie
Activar Pulsómetro
No
Mostrar tabla de eventos
No
Borrar Flash
No
Si
Si
Si
Activar Flag
Tx todos los datos de eventos almacenados en el vector
Borrar
Fin función ejecuta_orden

4.2 Pulsadores y Estímulo

El proyecto tiene dos pulsadores configurables para implementar cualquier tecla de un teclado estándar y una entrada de estímulo.

Los pulsadores se conectan cada uno a un pin de entrada de la placa Discovery STM32F4. Uno de ellos se conecta a PD14 en modo Push Pull, cuando el pulsador este en On en la entrada habrá 5V, el microcontrolador controla por polling si este pin de entrada está en On. El otro pulsador se configura exactamente igual, pero está asociado a la entrada PB0.

La entrada de estímulo es una entrada que se utiliza para saber cuándo empieza y acaba un estímulo introducido por el especialista cuando se esté usando el dispositivo final para un análisis psicológico o cualquier otra aplicación. El estímulo de entrada se ha programado exactamente igual que los pulsadores, esto es, cuando la entrada esté a "1" el estímulo estará activado y cuando esté a "0" estará desactivado.

A continuación se muestra el código de configuración de un pulsador y de la entrada de estímulo.

```
/*Inicializamos botón externo*/
    TM_GPIO_Init(GPIOD, GPIO_PIN_14, TM_GPIO_Mode_IN,
      TM_GPIO_OType_PP, TM_DISCO_BUTTON_PULL, TM_GPIO_Speed_Low);
/*Inicializamos entrada de estímulo*/
    TM_GPIO_Init(GPIOE, GPIO_PIN_7, TM_GPIO_Mode_IN, TM_GPIO_OType_PP,
      TM_DISCO_BUTTON_PULL, TM_GPIO_Speed_Low);
```

4.3 Teclado USB HID

Para la realización de esta parte del proyecto se han seguido punto por punto las especificaciones del manual de usuario UM1021 *On-The-Go device library* **(5)** para STM32F4 que proporciona el fabricante STMicroelectronics. Estás especificaciones se pueden encontrar en esta memoria en el punto 4.3.3.

Se ha configurado la placa Discovery STM32F407 como un dispositivo USB HID que puede trabajar en los modos FS y Hs. El modo utilizado por defecto es Fs pero se puede cambiar en el archivo *defines.h*.

Al conectar la placa discovery STM32F407 a un PC aparece como un dispositivo *Keyboard* (teclado). Este dispositivo puede usar todas las teclas especiales (SHIFT, CTRL, ALT y GUI), en combinación con otras teclas.

4.3.1 Estructura de archivos y de librerías en el proyecto.

Se han incluido en el proyecto todos los archivos de librerías necesarios para realizar un dispositivo USB HID. Los archivos se han obtenido de las librerías CMSIS, *STM32_USB_Device library* y *STM32_USB_OTG_Driver*, también es necesario incluir los archivos *STM32F4xx RCC*, *STM32F4xx GPIO*, *STM32F4xx EXTI* ya que hay que configurar algunos pines para recibir y transmitir datos por USB así como también se hace uso de interrupciones externas.

A continuación se muestra la estructura de archivos en el proyecto. Los archivos que aparecen en rojo son archivos que se han añadido o que se han tenido que modificar para la realización del nuestro dispositivo particular, los archivos que no aparecen en rojo son archivos que o su

código es completamente igual al código que presentan en la librería proporcionada por STMicroelectronics o se han modificado según se refleja en el apartado 3.3.3 de esta memoria, donde podemos encontrar una descripción de cada archivo y sus funciones.

- Discovery STM32F407D
  - Cmsis
    - Core_cm4.h
    - Core_cm4_simd.h
    - Core_cmFunc.h
    - Core_ccmInstr.h
  - User
    - Stm32f4_usb_hid_device
    - Defines.h
    - Main.c
    - Usb_bsp.c
    - Usb_conf.h
    - Usbd_conf.h
    - Usbd_desc.c
    - Usbd_desc.h
    - Usbd_usr.c
  - Usb_Hid_Device
    - Usb_bsp.h
    - Usb_core.c
    - Usb_dcd.c
    - Usb_dcd_int.c
    - Usb_defines.h
    - Usbd_core.c
    - Usbd_ioreq.c
    - Usbd_req.c
    - Usbd_usr.h
    - Usbd_hid_core.c
    - Usbd_hid_core.h

4.3.2 Configuración del archivo *usb_bsp.c*

Este archivo configura los pin de la placa Discovery STM32F4 y habilita la interrupción externa EXTI para la Tx/Rx USB .

USB	FS MODE	HS IN FS MODE	DESCRIPTION
Data +	PA12	PB15	USB Data+ line
Data –	PA11	PB14	USB Data- line
ID	PA10	PB12	USB ID pin
VBUS	PA9	PB13	USB activate

4.3.3 Configuración *usbd_hid_core.c*

Este archivo se ha modificado para que el dispositivo pueda trabajar como *keyboard*, *mouse o gamepad*, con el fin de tener un dispositivo con versatilidad para proyectos futuros, pero en este proyecto siempre que se conecte STM32F4 a un PC solo se configurará como *keyboard*, como podremos ver después el en apartado que describe al fichero *Stm32f4_usb_hid_device*

Para una descripción detallada de cada campo consultar el documento *Device Class Definition for Human Interface Devices (HID), Appendix E* (4).

```
/* USB HID device Configuration Descriptor */
__ALIGN_BEGIN static uint8_t USBD_HID_CfgDesc[USB_HID_CONFIG_DESC_SIZ]
__ALIGN_END =
{
  0x09, /* bLength: Configuration Descriptor size */
  USB_CONFIGURATION_DESCRIPTOR_TYPE, /* bDescriptorType: Configuration
*/
  USB_HID_CONFIG_DESC_SIZ,
  /* wTotalLength: Bytes returned */
  0x00,
  0x01,   /*bNumInterfaces: 1 interface*/
  0x01,   /*bConfigurationValue: Configuration value*/
  0x00,   /*iConfiguration: Index of string descriptor describing the
configuration*/
  0xE0,   /*bmAttributes: bus powered and Support Remote Wake-up */
  0x32,   /*MaxPower 100 mA: this current is used for detecting Vbus*/

  /************** Descriptor of Joystick Mouse interface ***********/
  /* 09 */
  0x09,         /*bLength: Interface Descriptor size*/
  USB_INTERFACE_DESCRIPTOR_TYPE,/*bDescriptorType: Interface
descriptor type*/
  0x00,         /*bInterfaceNumber: Number of Interface*/
  0x00,         /*bAlternateSetting: Alternate setting*/
  0x01,         /*bNumEndpoints*/
  0x03,         /*bInterfaceClass: HID*/
  0x00,   /*0x01*/      /*bInterfaceSubClass : 1=BOOT, 0=no boot*/
  0x00,   /*0x02*/      /*nInterfaceProtocol : 0=none, 1=keyboard,
2=mouse*/
  0,            /*iInterface: Index of string descriptor*/
  /************** Descriptor of Joystick Mouse HID **************/
  /* 18 */
  0x09,         /*bLength: HID Descriptor size*/
  HID_DESCRIPTOR_TYPE, /*bDescriptorType: HID*/
  0x11,         /*bcdHID: HID Class Spec release number*/
  0x01,
  0x00,         /*bCountryCode: Hardware target country*/
```

```
  0x01  /*bNumDescriptors: Number of HID class descriptors to follow*/
  0x22,         /*bDescriptorType*/
  HID_MOUSE_REPORT_DESC_SIZE,/*wItemLength: Total length of Report
descriptor*/
  0x00,
  /*************** Descriptor of Mouse endpoint ****************/
  /* 27 */
  0x07,          /*bLength: Endpoint Descriptor size*/
  USB_ENDPOINT_DESCRIPTOR_TYPE, /*bDescriptorType:*/

  HID_IN_EP,     /*bEndpointAddress: Endpoint Address (IN)*/
  0x03,          /*bmAttributes: Interrupt endpoint*/
  HID_IN_PACKET, /*wMaxPacketSize: 4 Byte max */
  0x00,
  0x0A,          /*bInterval: Polling Interval (10 ms)*/
  /* 34 */
} ;
/* USB HID device Configuration Descriptor */
__ALIGN_BEGIN static uint8_t USBD_HID_Desc[USB_HID_DESC_SIZ]
__ALIGN_END=
{
  /* 18 */
  0x09,         /*bLength: HID Descriptor size*/
  HID_DESCRIPTOR_TYPE, /*bDescriptorType: HID*/
  0x11,         /*bcdHID: HID Class Spec release number*/
  0x01,
  0x00,         /*bCountryCode: Hardware target country*/
  0x01, /*bNumDescriptors: Number of HID class descriptors to follow*/
  0x22,         /*bDescriptorType*/
  HID_MOUSE_REPORT_DESC_SIZE,/*wItemLength: Total length of Report
descriptor*/
  0x00,
};
#endif
```

4.3.4 Configuración de *usbd_desc.c*

Como podemos observar el dispositivo HID trabaja en modos de alta y máxima velocidad y proporciona la siguiente información del dispositivo USB (*usbd_desc.c*).

```
#define USBD_VID                        0x0483
#define USBD_PID                        0x5710

#define USBD_LANGID_STRING              0x409
#define USBD_MANUFACTURER_STRING        "STMicroelectronics"
```

```
#define USBD_PRODUCT_HS_STRING            "USB HID device in HS mode"
#define USBD_SERIALNUMBER_HS_STRING       "00000000011B"

#define USBD_PRODUCT_FS_STRING            "USB HID device in FS mode"
#define USBD_SERIALNUMBER_FS_STRING       "00000000011C"

#define USBD_CONFIGURATION_HS_STRING      "HID Config"
#define USBD_INTERFACE_HS_STRING          "HID Interface"

#define USBD_CONFIGURATION_FS_STRING      "HID Config"
#define USBD_INTERFACE_FS_STRING          "HID Interface"
```

Y a continuación en este archivo se configuran los descriptores del dispositivo

```
/* USB Standard Device Descriptor */
__ALIGN_BEGIN uint8_t USBD_DeviceDesc[USB_SIZ_DEVICE_DESC] __ALIGN_END
=
  {
    0x12,                       /*bLength */
    USB_DEVICE_DESCRIPTOR_TYPE, /*bDescriptorType*/
    0x00,                       /*bcdUSB */
    0x02,
    0x00,                       /*bDeviceClass*/
    0x00,                       /*bDeviceSubClass*/
    0x00,                       /*bDeviceProtocol*/
    USB_OTG_MAX_EP0_SIZE,      /*bMaxPacketSize*/
    LOBYTE(USBD_VID),           /*idVendor*/
    HIBYTE(USBD_VID),           /*idVendor*/
    LOBYTE(USBD_PID),           /*idVendor*/
    HIBYTE(USBD_PID),           /*idVendor*/
    0x00,                       /*bcdDevice rel. 2.00*/
    0x02,
    USBD_IDX_MFC_STR,           /*Index of manufacturer  string*/
    USBD_IDX_PRODUCT_STR,       /*Index of product string*/
    USBD_IDX_SERIAL_STR,        /*Index of serial number string*/
    USBD_CFG_MAX_NUM            /*bNumConfigurations*/
  } ; /* USB_DeviceDescriptor */
```

Descripción de cada campo:

Parte	**Descripción**

bLength	Tamaño del descriptor
bDescriptorType	Tipo de descriptor de dispositivo (asignado por USB)
bcdUSB	USB HID versión especifica 1.0
bDeviceClass	Código de clase (asignado por USB). Hay que tener en cuenta que la clase HID está definida en descriptor de interfaz
bDeviceSubClass	Código de subclase (asignado por USB). Estos códigos son calificados por el valor del campo *bDeviceClass*
bDeviceProtocol	Código de protocolo. Estos códigos son calificados por el valor del campo *bDeviceSubClass*
bMaxPacketSize0	Máximo tamaño de paquete para *endpoint0* (solo 8,16,32 o 64 son válidos)
idVendor	ID del vendedor (asignada por USB). Este valor se define al principio del archivo *usbd_desc.c*
idProduct	ID del producto. Definido en el archivo *usbd_desc.c*
bcdDevice	Número de versión del dispositivo
iManufacturer	Índice del descriptor *String* que describe al fabricante
iProduct	Índice del descriptor *String* que describe al producto
iSerialNumer	Índice del descriptor *String* que describe el número de serie del dispositivo
bNumConfigurations	Numero de posibles configuraciones

Cuando conectamos el dispositivo al Pc y nos vamos al panel de control, ver dispositivos e impresoras encontramos lo siguiente:

Como podemos observar, encontramos nuestro dispositivo en modo teclado y en modo FS (modo por defecto) con el nombre de producto que le hemos indicado en *usbd_desc.c*

4.3.5 Archivo de creación propia *stm32f4_usb_hid_device.*

La librería USB HID *device* está configurado para soportar 1 teclado, 1 ratón y 2 gamepads al mismo tiempo. Se pueden enviar *reports* HID para el ratón y el teclado uno por uno alternativamente. Pero como se ha comentado en apartados anteriores el dispositivo solo se va a configurar (en este archivo) en modo teclado.

Trabaja en modo USB FS o USB HS. Por defecto la librería trabaja en modo USB FS, pero se puede modificar en el archivo *defines.h*.

A continuación, la estructura de cómo *USB HID report* ve los *reports*. Hay que poner el bit a 1 cuando se presiona el botón o a 0 cuando se suelta el botón.

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Byte 0	Report ID = 0x01							
Byte 1	Right GUI	Right ALT	Right SHIFT	Right CTRL	Left GUI	Left ALT	Left SHIFT	Left CTRL
Byte 2	Padding = Always 0x00							
BYTE 3	Key 1							
BYTE 4	Key 2							
BYTE 5	Key 3							
BYTE 6	Key 4							
BYTE 7	Key 5							
BYTE 8	Key 6							

Parámetros de la librería:

- *USB_HIDDEVICE_Status_LibraryNotInitialized*

  No se ha inicializado todavía la biblioteca.

- *USB_HIDDEVICE_Status_Connected*

  El dispositivo está conectado y listo para usar.

- *USB_HIDDEVICE_Status_Disconnected*

  El dispositivo no está conectado.

- *USB_HIDDEVICE_Status_IdleMode*

  El dispositivo está en el modo IDLE.

- *USB_HIDDEVICE_Status_SuspendMode*

  El dispositivo está en modo suspensión.

Función de inicialización

La función `USB_HIDDEVICE_Init(void)` inicializa la librería USB HID para empezar a trabajar llamando a la función de *usbd_core.c* `USBD_Init` que inicializa el driver de bajo nivel del USB, la librería *USB device*, el hardware en la tarjeta utilizada (BSP) y arranca la librería.

```
TM_USB_HIDDEVICE_Init(void) {
	/* Initialize HID device */
	USBD_Init(&USB_OTG_dev,
	#ifdef USE_USB_OTG_HS
			USB_OTG_HS_CORE_ID,
	#else
			USB_OTG_FS_CORE_ID,
	#endif
			&USR_desc,
			&USBD_HID_cb,
			&USR_cb);
	/* Set not connected */
	TM_USB_HIDDEVICE_INT_Status =
	TM_USB_HIDDEVICE_Status_Disconnected;

	/* Device not connected */
	return TM_USB_HIDDEVICE_INT_Status;
}
```

Devuelve siempre como valor de estado “dispositivo no conectado”, hasta que no mandemos el primer *report* no estaremos conectados.

Estructura del teclado

Keyboard tiene 8 botones especiales (CTRL, ALT, SHIFT, GUI (o WIN) Se pueden pulsar 6 teclas al mismo tiempo, por ejemplo *Key1 = 'a'*, *Key2 = 'b'*, y se obtendrá "*ab*” en la pantalla. Si una tecla no está siendo usada, entonces se le debe asignar el valor 0x00.

El teclado se inicializa mediante la siguiente función:

```
TM_USB_HIDDEVICE_Status_t
TM_USB_HIDDEVICE_KeyboardStructInit(TM_USB_HIDDEVICE_Keyboard_t*
Keyboard_Data) {
/* Set defaults */
	Keyboard_Data->L_CTRL = TM_USB_HIDDEVICE_Button_Released;
	Keyboard_Data->L_ALT = TM_USB_HIDDEVICE_Button_Released;
	Keyboard_Data->L_SHIFT = TM_USB_HIDDEVICE_Button_Released;
	Keyboard_Data->L_GUI = TM_USB_HIDDEVICE_Button_Released;
	Keyboard_Data->R_CTRL = TM_USB_HIDDEVICE_Button_Released;
	Keyboard_Data->R_ALT = TM_USB_HIDDEVICE_Button_Released;
	Keyboard_Data->R_SHIFT = TM_USB_HIDDEVICE_Button_Released;
	Keyboard_Data->R_GUI = TM_USB_HIDDEVICE_Button_Released;
	Keyboard_Data->Key1 = 0;
```

```
	Keyboard_Data->Key2 = 0;
	Keyboard_Data->Key3 = 0;
	Keyboard_Data->Key4 = 0;
	Keyboard_Data->Key5 = 0;
	Keyboard_Data->Key6 = 0;
	/* Return currect status */
	return TM_USB_HIDDEVICE_INT_Status;
}
```

Función *KeyboardSend*:

Esta función será llamada cada vez que pulsamos o soltamos un pulsador en nuestro proyecto. Esta función actualiza la estructura de datos del teclado con el valor de cada tecla y envía un *report* con el *buffer* de datos mediante la función `USBD_HID_SendReport` que podemos encontrar en el archivo *usbd_hid_core.c* que a su vez llama a la función `DCD_EP_Tx` del archivo *usb_dcd.c* , esta última función configura e inicia la transmisión.

```
TM_USB_HIDDEVICE_Status_t
TM_USB_HIDDEVICE_KeyboardSend(TM_USB_HIDDEVICE_Keyboard_t*
Keyboard_Data) {
uint8_t buff[9] = {0, 0, 0, 0, 0, 0, 0, 0, 0}; // 9 bytes long report
	/* Check status */
 if (TM_USB_HIDDEVICE_INT_Status != TM_USB_HIDDEVICE_Status_Connected)
	{
		return TM_USB_HIDDEVICE_Status_Disconnected;
	}
	/* Report ID */
	buff[0] = 0x01; /* Keyboard */
	/* Control buttons */
	buff[1] = 0;
	buff[1] |= Keyboard_Data->L_CTRL   << 0; /* Bit 0 */
	buff[1] |= Keyboard_Data->L_SHIFT  << 1; /* Bit 1 */
	buff[1] |= Keyboard_Data->L_ALT    << 2; /* Bit 2 */
	buff[1] |= Keyboard_Data->L_GUI    << 3; /* Bit 3 */
	buff[1] |= Keyboard_Data->R_CTRL   << 4; /* Bit 4 */
	buff[1] |= Keyboard_Data->R_SHIFT  << 5; /* Bit 5 */
	buff[1] |= Keyboard_Data->R_ALT    << 6; /* Bit 6 */
	buff[1] |= Keyboard_Data->R_GUI    << 7; /* Bit 7 */
	/* Padding */
	buff[2] = 0x00;
	/* Keys */
	buff[3] = Keyboard_Data->Key1;
	buff[4] = Keyboard_Data->Key2;
	buff[5] = Keyboard_Data->Key3;
	buff[6] = Keyboard_Data->Key4;
```

```
        buff[7] = Keyboard_Data->Key5;
        buff[8] = Keyboard_Data->Key6;
        /* Send to USB */
        USBD_HID_SendReport(&USB_OTG_dev, buff, 9);
        /* Return connected */
        return TM_USB_HIDDEVICE_Status_Connected;
}
```

4.3.6 USB HID en el programa principal *main.c*

Cuando se envía un *report* al pulsar algún botón, el microcontrolador detectará el botón presionado. Pero después de soltar el botón, se tiene que enviar también un *report* HID que indique que se ha soltado el botón. En una palabra: Hay que enviar el *report HID* cada vez que cambia cada botón (cada vez que se presiona o se suelta).

```
TM_USB_HIDDEVICE_Keyboard_t Keyboard;  /* Set struct */
TM_USB_HIDDEVICE_Init();   /* Initialize USB HID Device */
TM_USB_HIDDEVICE_KeyboardStructInit(&Keyboard); /* Set default values
for keyboard struct */


if ((TM_GPIO_GetInputPinValue(GPIOD, GPIO_PIN_14)!=0) && already1 ==0)
{ // Button1 on
       already1 = 1;
       Keyboard.Key1 = tecla1;
       TM_USB_HIDDEVICE_KeyboardSend(&Keyboard);// Send keyboard report
} else if (!TM_GPIO_GetInputPinValue(GPIOD,GPIO_PIN_14)&&already1== 1)
{ // Button1 release
      already1 = 0;
      /* Release all buttons*/
      Keyboard.L_GUI = TM_USB_HIDDEVICE_Button_Released;
      Keyboard.Key1 = 0x00; // No key
      TM_USB_HIDDEVICE_KeyboardSend(&Keyboard); // Send keyboard report
}
```

4.4 Consola USART

La consola en si misma será cualquier aplicación que soporte comunicación por puerto serie. Las aplicaciones que se han utilizado para el desarrollo de este proyecto son Matlab y un programa típico de terminal llamado *CoolTerm*.

En la consola se pueden escribir distintas órdenes que se enviarán mediante el puerto serie al microcontrolador y este las ejecutará y en algunos casos transmitirá una salida que recibirá el programa de puerto serie que estemos utilizando mostrando el resultado por pantalla.

No es necesario escribir todo el nombre de la orden, puesto que cada orden empieza por una letra diferente, solo será necesario escribir la primera letra de cada orden. Las órdenes son:

-*SetKeyX*

-*GetKey*

-*echo*

-*h*: Para sincronizar relojes.

-*t*: Muestra tabla de eventos.

-*write*: Guarda datos en la flash.

-*clear*: Borra la memoria flash.

-*read*: Lee la memoria flash.

-*a*: Activa pulsómetro.

-*p*: Muestra la frecuencia cardiaca actual.

-*s*: Para pulsómetro.

4.4.1 Configuración USART

De las cuatro USART de la que dispone el STM32f4 se ha decidido utilizar USART1 con la siguiente configuración:

* Tasa en baudios: 9600 baud

* Palabras de 8bits

* 1 Stop Bit

* Sin bit de paridad

* Control de flujo Hardware inhabilitado.

* Recepción y Transmisión habilitado.

* Pin de transmisión: PB6.

* Pin de recepción: PB7.

Función de inicialización en el microcontrolador:

```
void UART_Initialize (void){
 /* Enable peripheral clock for USART1 */
RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1, ENABLE);
USART_InitStructure.USART_BaudRate = 9600;
USART_InitStructure.USART_WordLength = USART_WordLength_8b;
USART_InitStructure.USART_StopBits = USART_StopBits_1;
USART_InitStructure.USART_Parity = USART_Parity_No;
USART_InitStructure.USART_HardwareFlowControl
=USART_HardwareFlowControl_None;    USART_InitStructure.USART_Mode =
USART_Mode_Rx | USART_Mode_Tx;      USART_Init(USART1,
&USART_InitStructure); // USART configuration USART_Cmd(USART1,
ENABLE); // Enable USART
}
```

También necesitamos definir PB6 como *Usart_Tx* y PB7 *como Usart_Rx*, ambas en el puerto B, para ellos recurrimos a la siguiente función de configuración GPIO:

```
void GPIOInitialize(void){
//Habilitar reloj para  GPIOB
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB, ENABLE);
       /* USART1 Tx on PB6 | Rx on PB7 */
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6 | GPIO_Pin_7;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;
GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;
GPIO_Init(GPIOB, &GPIO_InitStructure);
GPIO_PinAFConfig(GPIOB, GPIO_PinSource6, GPIO_AF_USART1);//PB6 to
UsartTx GPIO_PinAFConfig(GPIOB, GPIO_PinSource7, GPIO_AF_USART1);//PB7
Usart Rx
}
```

Es necesario habilitar las interrupciones USART1. Estas interrupciones se producirán cada vez que haya comunicación con la consola. Para ello se configura la unidad dedicada al control de interrupciones (NVIC) de la siguiente forma:

```
void NVICInitialize(void){
NVIC_InitStructure.NVIC_IRQChannel = USART1_IRQn;
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1;
NVIC_InitStructure.NVIC_IRQChannelSubPriority = 1;
NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
NVIC_Init(&NVIC_InitStructure);
USART_ITConfig(USART1, USART_IT_RXNE, ENABLE);    //habilita las
interrup USART
}
```

Para el envío de datos mediante USART se ha decidido implementar una función que recibe un vector de caracteres y si echo (variable que indica si se debe mostrar por pantalla lo que se escribe en la consola) está habilitado irá recorriendo este vector enviando cada carácter. La función que implementa esto es la siguiente:

```
void USART_puts(volatile char *s){
if(echo!=0){
      while(*s){
            while( !(USART1->SR & 0x00000040) );
            USART_SendData(USART1, *s);
            *s++;
      }
 } return; }
```

4.4.2 Ordenes de la consola

Orden *setKeyX*

Donde *X* puede ser 1 o 2. Mediante esta orden establecemos los valores de cada pulsador (1 y 2). Hay dos modos para establecer los valores:

-s*X* "*caracter*":

Donde *X* puede ser 1 o 2, haciendo referencia a los pulsadores.

A continuación habría que introducir un espacio.

Por último, el *caracter* que se desee asignar al pulsador en cuestión.

Con este modo solo se podrán asignar números y teclas, pero no caracteres especiales, ya que los dispositivos HID usan codificación hexadecimal y este modo se ha diseñado básicamente para el desarrollo y testeo del proyecto ya que es más sencillo y rápido.

-s*X* "*0xXX*"

Donde *X* puede ser 1 o 2, haciendo referencia a los pulsadores.

A continuación habría que introducir un espacio.

Por último el carácter en Hexadecimal que se desee asignar al pulsador en cuestión.

Podemos encontrar una tabla con el valor en formato Hexadecimal de cada carácter en la tabla 12 del documento *HID Usage Tables* (12)

A continuación se muestra el código del programa que implementa esta orden, donde *línea[i]* es el vector recibido por el puerto serie donde podemos encontrar toda la información introducida antes de un *Enter* de teclado *(\n)*, lo que sería una línea de la consola.

```
if(linea[0]=='s'){   //orden setkey
    while(linea[i]!=' ') i++;
    if(linea[i-1]=='1'){  //letra antes del espacio ¿tecla1?
      if((linea[i+1]=='0')&&(linea[i+2]=='x'))
codificador_hex1(linea,i); //hexadecimal?
      else { codificador1(linea[i+1]);}  //enviamos la siguiente letra
después del espacio

    }else if(linea[i-1]=='2'){ //¿tecla2?
      if((linea[i+1]=='0')&&(linea[i+2]=='x'))
codificador_hex2(linea,i); //hexadecimal
      else {codificador2(linea[i+1]);}  //enviamos la siguiente letra
después del espacio
    }else{
      USART_puts("orden setkey incorrecta\n introduzca setkeyX donde X
es 1 o 2 \n");
    }
    return;
  }
```

Si se ha establecido el valor de un pulsador mediante el primer modo se llamará a una función que codificará el valor a formato hexadecimal, ya que es el formato con el que trabajan los dispositivos HID.

Si se ha establecido el valor de un pulsador por el modo hexadecimal se llamará a una función (`codificador_hex1` o `codificador_hex2`) que asignará el valor hexadecimal introducido a la variable *tecla1*)

```
void codificador_hex1(volatile char *linea_hex1, int j){
  strHEX1[0]=NULL; strHEX1[1]=NULL; strHEX2[0]=NULL;
  strHEX1[0]=linea_hex1[j+3];
  strHEX2[0]=linea_hex1[j+4];
      strcat(strHEX1, strHEX2);
      tecla1=strtol(strHEX1, NULL, 16);
}
```

La variable *tecla1* (o *tecla2*) contiene el valor actual de cada pulsador, cada vez que pulsemos un pulsador u otro se enviará un *SendReport* con el valor que tenga en ese momento la variable *teclaX* del pulsador X.

Orden *getKey*

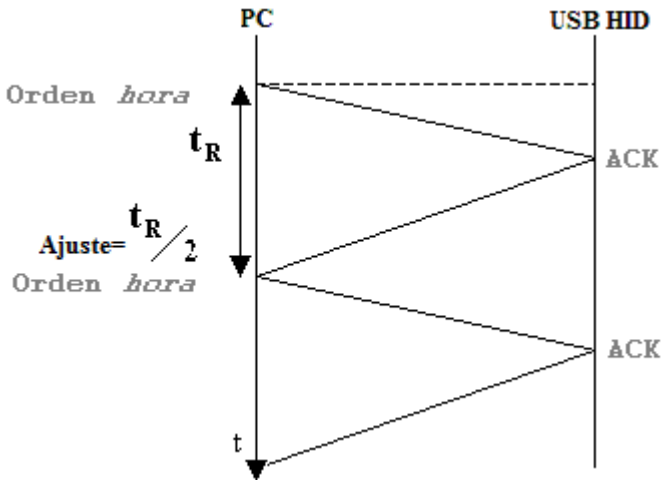
Muestra el valor que está asignado en ese momento en las Teclas 1 y 2.

Orden *echo*

Si la variable *echo* es igual a 1 el programa de puerto serie mostrará caracteres por pantalla, si *echo* es igual a 0 estará desactivado con lo cual no se mostrarán caracteres en el programa de puerto serie.

Orden *hora*

Está orden sirve para sincronizar relojes. El *Host* enviará una determinada hora (en décimas de milisegundo) y el dispositivo USB HID (microcontrolador) almacena este parámetro en una variable y manda un ACK. Después de esto el *Host* puede calcular el tiempo de retardo entre los dos dispositivos y corregirlo.




```
if(linea[0]=='h') { //orden set time  ALMACENA HORA Y ENVIA ACK, el
                        ajuste lo hace el otro dispositivo.
    volatile char nueva_linea[MAX_WORDLEN + 1]="vacia";
    while(linea[i]!=' ') i++;
```

```
    for (int f=i+1;f<=MAX_WORDLEN; f++){
      nueva_linea[f-i-1]=linea[f];  // almacena todo lo que hay
                                    después del espacio (hora en 0.1ms)
    }
    uint32_t hora;
    char str11[15];
    sprintf(str11, "%s", nueva_linea);
    hora = atoi(str11);  // ahora hora es un entero
    /*Enviar ACK*/
    USART_puts("ACK");
    return;
  }
```

Orden *time*

Muestra una tabla con los tiempos (en decimas de milisegundo) en que se ha producido cada evento.

Para obtener el tiempo en decimas de milisegundo se hace uso de la interrupción SysTick. Esta interrupción se dispara cada vez que se vence el tiempo con el que se ha configurado. En este proyecto se ha configurado para que se dispare cada 10ms,

```
/* Set Systick interrupt every 10ms */
SysTick_Config(SystemCoreClock / 100)
```

En la función que maneja la interrupción se ha incluido una variable que se incrementa en una unidad cada vez que vence el SysTick.

```
void SysTick_Handler(void) {  // cada 10ms
      TM_Time++;
}
```

Por lo tanto cada vez que se quiera saber el tiempo, solo habrá que consultar la variable TM_Time. Pero con esto solo se tendrá una precisión de 10 ms y uno de los objetivos principales del proyecto era llegar a decimas de milisegundo. Para ello lo único que hay que hacer es mirar por donde va la cuenta del SysTick en el momento en que se quiere medir el tiempo. Esto se hace mirando el registro VAL, `SysTick->VAL`. Para obtener un valor entre 0 y 99 que indicarían la decimas de cada $1s.10^{-2}$ (valor de vencimiento del SysTick, 10ms) se hace la siguiente operación: VAL*99/168000.

168000 es la frecuencia de reloj dividida entre 10, `SystemCoreClocks/10` y por lo tanto el valor máximo que alcanza el contador del SysTick.

El ultimo pasó es convertir a decimas de milisegundo. Como vemos es un proceso un poco largo y el proyecto realiza muchas consultas de tiempo en diferentes puntos del programa por lo es conveniente implantar una función que automatice todo lo explicado anteriormente y que devuelva el tiempo actual en decimas de milisegundo:

```
*Funcion que retorna decimas de milisegundo*/
uint32_t dms(void)
```

```
{
      VALo=SysTick->VAL;   //Valor actual exacto del SysTick
      cs_decimales=(uint32_t)(VALo*99/168000); //entre 0 y 99
      ms_decimales=(TM_Time*100)+(cs_decimales);
       return ms_decimales;
}
```

La orden *time* muestra una tabla con el tiempo en que se produjo cada evento ordenada cronológicamente, en la primera columna aparece el tiempo en milisegundo. En la segunda columna el tipo de evento:

P1I→ P1 pulsado P1F→ P1 liberado EI→Inicio estímulo

P2I→ P2 pulsado P2F→P2 Liberado EF→Fin estímulo

En la tercera columna aparece un contador de tipo de evento por ejemplo la primera vez que se pulsa P1 será P1I1 y la segunda P1I2.

A continuación su implementación:

```
  if(linea[0]=='t'){   //tabla de tiempos
     TiempoA=dms();
     char str2[15];
     sprintf(str2, "%d", TiempoA);
     USART_puts(str2);
     USART_puts(" ms/10 \n");
     char str3[15],str4[15],str5[15];
     for(int i=0;i<next_slot;i++){
      sprintf(str3, "%d", Tiempos[i]);  //Pasamos de int32 a string
      sprintf(str5, "%d", event_count[i]); //Pasamos de int32 a string
      USART_puts(str3);
      strcpy(str4, "ms/10 -- ");
      strcat(str4, event_type[i]);
      strcat(str4, str5);
       strcat(str4, "\n");
      USART_puts(str4);
     }
  }
```

Orden Write

Almacena información en la Flash. Se ha dividido el área destinada al usuario de la memoria Flash en 12 sectores de 16, 64 y 128Kbytes.

El primer dato se guarda en el sector 2 (uint32_t)0x08008000)) y para los siguientes se va incrementando la dirección según lo que ocupe cada dato, para no dejar huecos libres:

Si el dato es de 32 bits: *addres = addres + 4*

Si el dato es de 16 bits: *addres = addres + 2*

Si el dato es de 8 bits: *addres = addres + 1*

Cada vez que se quiera almacenar en la memoria Flash se realizaran los siguientes pasos:

- Desbloquear la Flash para poder acceder a los registro de control de la Flash.
- Se borra el arrea de Flash utilizada por el usuario.
- Se obtiene la dirección del sector en el que se comenzara a almacenar.
- Se almacena cada variable en la memoria Flash.
- Por último se bloquea la memoria Flash.

```
if(linea[0]=='w'){ //Guardar datos en FLASH 'write flash'
    STM_EVAL_LEDOn(LED3); STM_EVAL_LEDOn(LED4); STM_EVAL_LEDOn(LED5);
STM_EVAL_LEDOn(LED6);
     Address = FLASH_USER_START_ADDR;
     FLASH_Unlock();  // Unlock the Flash to enable the flash control
register access *************
     /* Erase the user Flash area(area defined by
      FLASH_USER_START_ADDR and FLASH_USER_END_ADDR) ***********/
      FLASH_ClearFlag(FLASH_FLAG_EOP | FLASH_FLAG_OPERR |
      FLASH_FLAG_WRPERR |  /* Clear pending flags (if any) */
      FLASH_FLAG_PGAERR | FLASH_FLAG_PGPERR|FLASH_FLAG_PGSERR);
      StartSector = GetSector(FLASH_USER_START_ADDR);  /* Get the
      number of the start and end sectors */
      EndSector = GetSector(FLASH_USER_END_ADDR);
      for (i = StartSector; i < EndSector; i += 8)
      {
    // Device voltage range supposed to be [2.7V to 3.6V], the
      operation will be done by word
          if (FLASH_EraseSector(i, VoltageRange_3) != FLASH_COMPLETE)
            {
              // Error occurred while sector erase. User can add here
                  some code to deal with this error
              while(1){}
             }
      }
       Address = FLASH_USER_START_ADDR;
      uint8_t sstrtoint;
      uint32_t add;
      FLASH_ProgramWord(Address, next_slot); // Guarda Primero el
      Ultimo slot
       Address = Address + 4;
       for(int i=0;i<next_slot;i++){
         FLASH_ProgramWord(Address, Tiempos[i]);
          Address = Address + 4;
         FLASH_ProgramWord(Address, event_count[i]);
```

```
          Address = Address + 4;
         sstrtoint= (int)event_type[i][0];
         FLASH_ProgramByte(Address, sstrtoint);
          Address = Address + 1;
        sstrtoint= (int)event_type[i][1];
         FLASH_ProgramByte(Address, sstrtoint);
          Address = Address + 1;
         sstrtoint= (int)event_type[i][2];
         FLASH_ProgramByte(Address, sstrtoint);
          Address = Address + 1;
          sstrtoint= (int)event_type[i][3];
         FLASH_ProgramByte(Address, sstrtoint);
          Address = Address + 1;
       }
      FLASH_Lock();  // Lock the Flash to disable the flash control
register access (recommended to protect the FLASH memory against
possible unwanted operation) ********
      STM_EVAL_LEDOff(LED3); STM_EVAL_LEDOff(LED4);
STM_EVAL_LEDOn(LED5); STM_EVAL_LEDOff(LED6);
```

Orden Clear

Borra todos los datos de la memoria Flash

Orden Read:

Lee los datos de la memoria Flash en el orden en que se guardaron y los va guardando en el vector de eventos y finalmente envía por el puerto todos los datos del vector de eventos y estos serán mostrados por pantalla.

```
  if(linea[0]=='r'){ //Leer datos en FLASH 'read flash'
      Address = FLASH_USER_START_ADDR;
      next_slot = *(__IO uint32_t*)Address; //primero leemos next_slot
      Address = Address + 4
      for(int i=0;i<next_slot;i++){
          Tiempos[i] = *(__IO uint32_t*)Address;
           Address = Address + 4;
          event_count[i] = *(__IO uint32_t*)Address;
           Address = Address + 4;
          event_type[i][0] = *(char*)Address;
           Address = Address + 1;
          event_type[i][1] = *(char*)Address;
           Address = Address + 1;
          event_type[i][2] = *(char*)Address;
           Address = Address + 1;
          event_type[i][3] = *(char*)Address;
```

```
            Address = Address + 1;
      }  /*Enviar datos por puerto serie*/
     char str3[15],str4[15],str5[15];
     for(int i=0;i<next_slot;i++){
       sprintf(str3, "%d", Tiempos[i]);  //int to string
       sprintf(str5, "%d", event_count[i]);  //int to string
       USART_puts(str3);
       strcpy(str4, "ms/10 -- ");
       strcat(str4, event_type[i]);
       strcat(str4, str5);
       strcat(str4, "\n");
       USART_puts(str4);
     }
  }
```

Orden Activar pulsímetro

Esta orden es la encargada de habilitar el pulsómetro. Cuando se introduce esta orden se pone a "1" un *Flag* que habilita el código para calcular la frecuencia cardiaca

```
if(linea[0]=='a'){ //activar pulsómetro
   activar_p=1;
  }
```

Orden Pulso

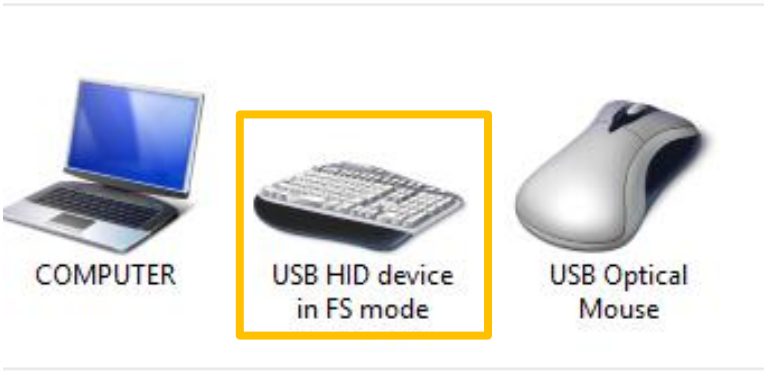
```
    if(linea[0]=='p'){ //pulso
      if(activar_p==0){
        USART_puts("Pulsometro inactivo, activar con orden 'a' \n");
      }
       else{
        char str3[15];
        sprintf(str3, "%f", ppm);  //float to string;
        strcat(str3,"ppm");
        strcat(str3, "\n");
        USART_puts(str3);
       }
      }
```

Orden Parar pulsímetro

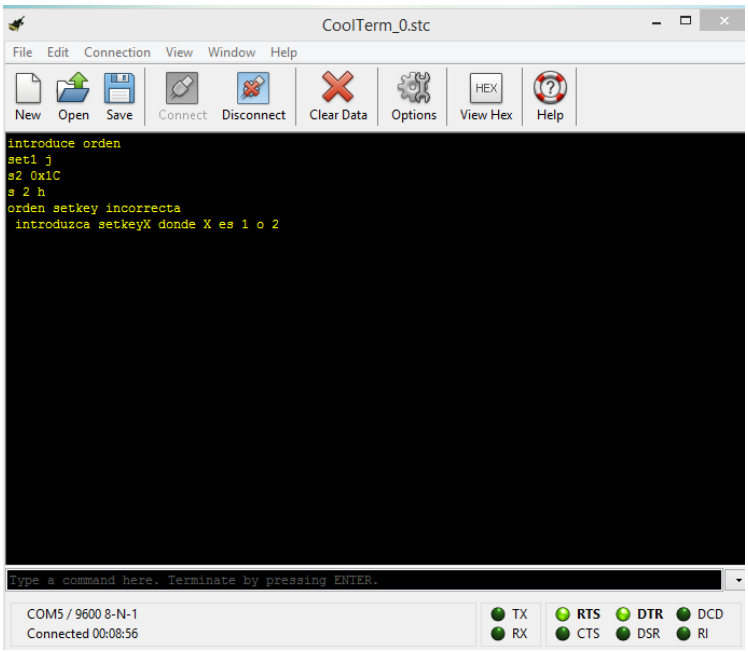
Pone a "0" el Flag que inhabilita el código para calcular la frecuencia cardiaca.

4.4.3 Prueba

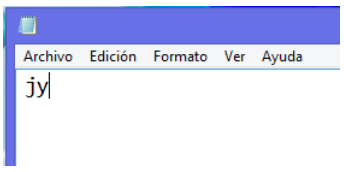
Al conectar el dispositivo a un PC si miramos en dispositivos e impresoras en el panel del control del PC veremos que el dispositivo aparece como teclado.



Para probar las órdenes de la consola de control se ha utilizado el programa de puerto serie CoolTerm. En la primera imagen podemos observar que al conectar el dispositivo aparece un mensaje inicial "introduce orden". Primero se configura el pulsador 1 con la tecla "j", después el pulsador 2 con "0x1C" que corresponde a la letra "y" (mirar tabla 12 de (12)) y por último se usa la orden *setkey* inapropiadamente como vemos en el mensaje de error.



Ahora pulsador 1 está configurado como "j" y pulsador 2 como "y". Si se habré un blog de notas, por ejemplo, y se pulsa el pulsador1 y luego el 2:



Con la orden getkey podemos obtener el valor de cada pulsador.

```
get
el valor de la tecla 1 es: j
el valor de la tecla 2 es: y
```

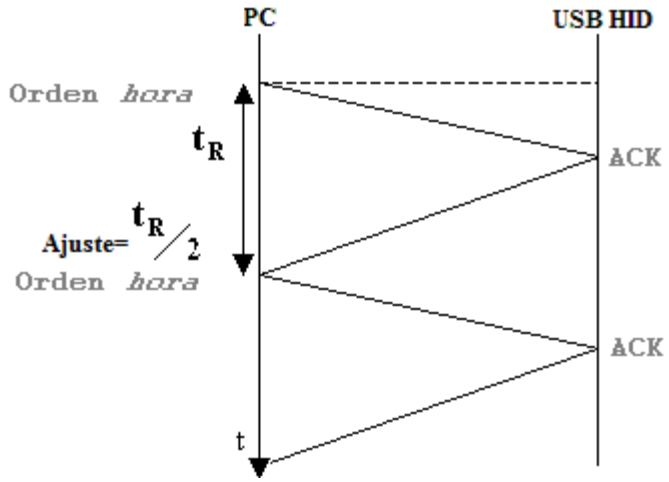
Después de pulsar P1 y P2 se ha activado y apagado la entrada de estímulo, ahora se introduce la orden time (tabla de eventos) y como vemos en la siguiente imagen, aparece la tabla de eventos con los tiempos en decimas de milisegundos. El primer tiempo es momento en que se introdujo la orden y lo demás tiempos son la tabla de eventos. Los dos primeros corresponden al pulsador 1 (on\off), los dos siguientes al pulsador 2 (on\off) y por último los correspondientes a la entrada de estímulo (on\off).

```
t
270138 ms/10
182723ms/10 -- P1I1
184823ms/10 -- P1F1
190019ms/10 -- P2I1
192017ms/10 -- P2F1
225073ms/10 -- EI1
243949ms/10 -- EF1
```

La orden *hora* sirve para sincronizar relojes, hay que introducir la hora en milisegundos, el dispositivo almacenará el valor en una variable y mandará un ACK.

```
h 3102545
ACK
```

Como vemos con este programa de terminal no se puede hacer mucho, pero con un programa como Matlab podríamos calcular el tiempo que ha transcurrido desde que se mandó la orden *hora* hasta que se recibió el ACK, este tiempo corresponde a el tiempo de retardo, después de obtener este valor podríamos ajustar la hora para que cuando llegue al dispositivo HID esté sincronizado con el PC o también podría seguir mandando la orden *hora* y recibiendo ACK para obtener un ajuste más exacto a partir del t_R medio.



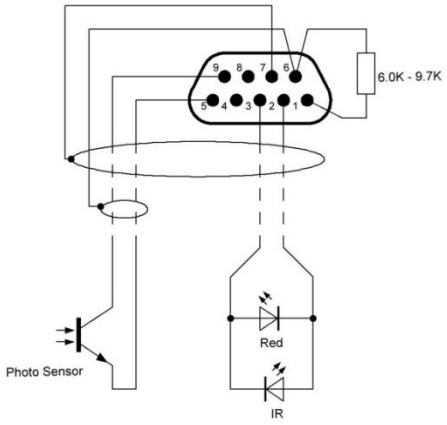
4.5Pulsioxímetro

4.5.1 SONDA

Para el desarrollo del proyecto se decide usar una sonda de medición *Nellcor Maxa* desechable, ya que la conexión y *pinout* son casi un estándar seguido por la gran mayoría de fabricantes, por lo tanto se podrá usar cualquier sonda de pulsioxímetro que siga el estándar, solo habrá que conectarla al conector DB9.

Esta sonda contiene en su interior dos diodos LED, uno trabaja a 660 nanómetros (luz roja visible) y el otro a 920 nanómetros (Infrarrojos). También dispone de un foto-transistor el cual es saturado en su base mediante luz, produciendo en su emisor una señal semejante a la de la base.

El sensor cuenta con un conector DB9 que facilita el procesamiento de datos. En la siguiente figura podemos observar la distribución del sensor así como el *pinout* (13).

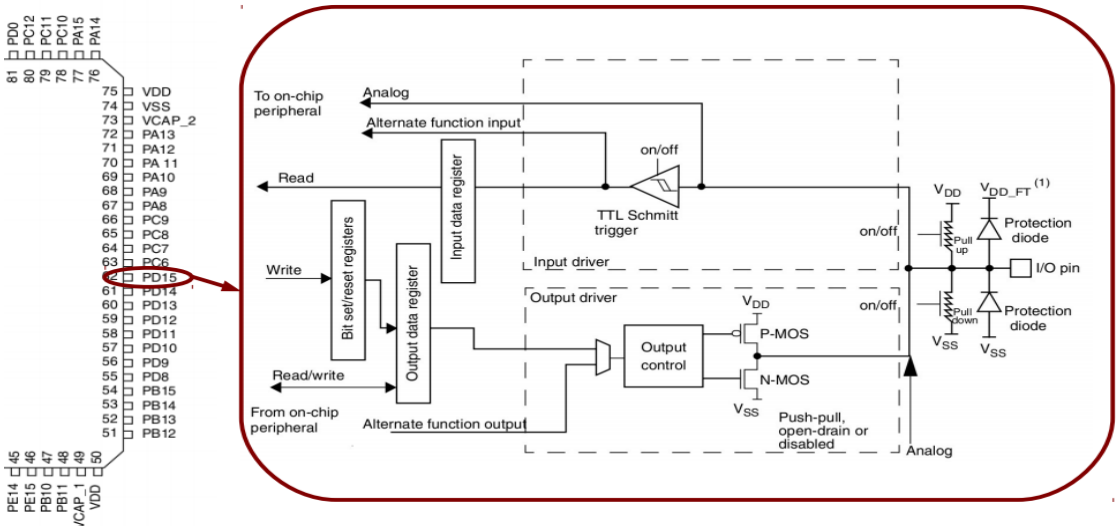


A continuación las características técnicas:

Tipo	Valores
Intervalos de medición	
Intervalo de saturación de Sp02	De 1% a 100%
Intervalo de Frecuencia del pulso	De 20 a 250 latidos por minuto (lpm)
Intervalo de perfusión	De 0,03% a 20%
Exactitud de las medidas	
Exactitud de frecuencia de pulso	De 20 a 250 latidos por minuto (lpm) +/- 3 dígitos
Sp02 exactitud de saturación	Del 70% al 100% de +/- 2 a +/- 3 dígitos
Intervalo de Funcionamiento y disipación	
Longitud de onda de luz roja	Aproximadamente 660 nm
Longitud de onda de luz infrarroja	Aproximadamente 900 nm
Potencia de salida óptica	Menos de 15 mW
Disipación de Alimentación	52,5 mW

Debido a la conexión interna de los leds rojo e infrarrojo dentro del sensor, éstos deben encenderse alternativamente, para que el fotodiodo pueda captar las señales individuales

emitidas por uno y otro. Para excitar los led cuando queramos, utilizaremos dos salidas del microcontrolador, como hay que alternar la excitación de los leds, cuando uno este excitado el otro deberá estar conectado a masa para, ello configuramos las salidas en modo *Push-Pull*.



Se ha decidió utilizar los pines PD9 y PD10, a continuación se muestra la configuración de estos pines en el programa del microcontrolador.

```
/* GPIOD Periph clock enable */
  RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOD, ENABLE);
  /* Configure PD9 y PD10 in output pushpull mode */
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9|GPIO_Pin_10 ;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;
  GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;
  GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;
  GPIO_Init(GPIOD, &GPIO_InitStructure);
```

Con las siguientes instrucciones se activará la salida digital, la cual excitar un led con 3.3V y 20mA y la otra se podrá a masa.

```
GPIO_SetBits(GPIOD, GPIO_Pin_10);
GPIO_ResetBits(GPIOD, GPIO_Pin_9);
```

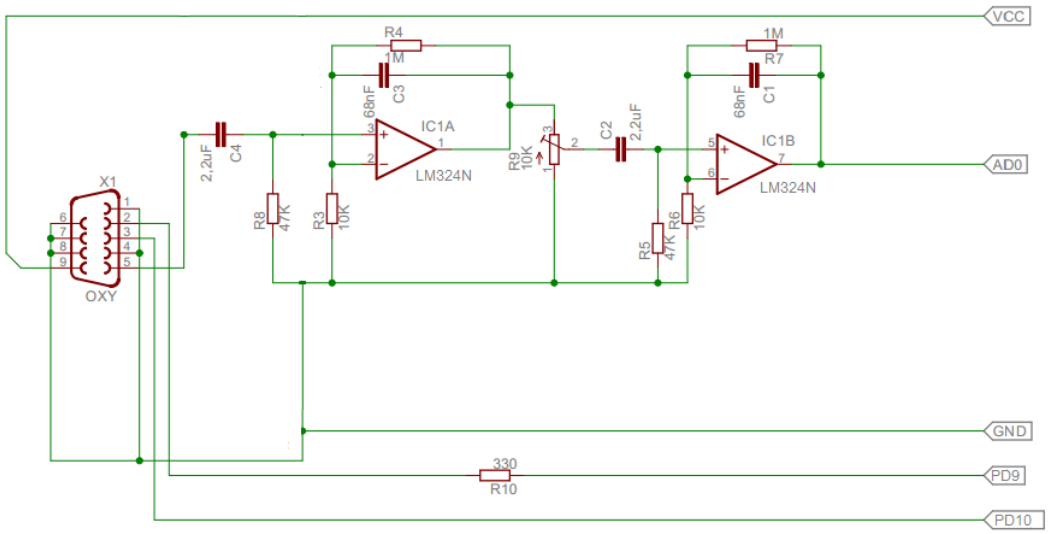
4.5.2 FILTRADO Y AMPLIFICACIÓN

La señal proporcionada por el foto-transistor es muy pequeña por lo que debemos filtrarla y amplificarla. En el proceso de filtrado se quitan las frecuencias que no nos interesan y que pueden interferir en la medición, por ejemplo, la frecuencia generada por la red eléctrica. En el proceso de amplificado la señal se aumenta a los niveles en los que el microcontrolador pueda trabajar con la señal.

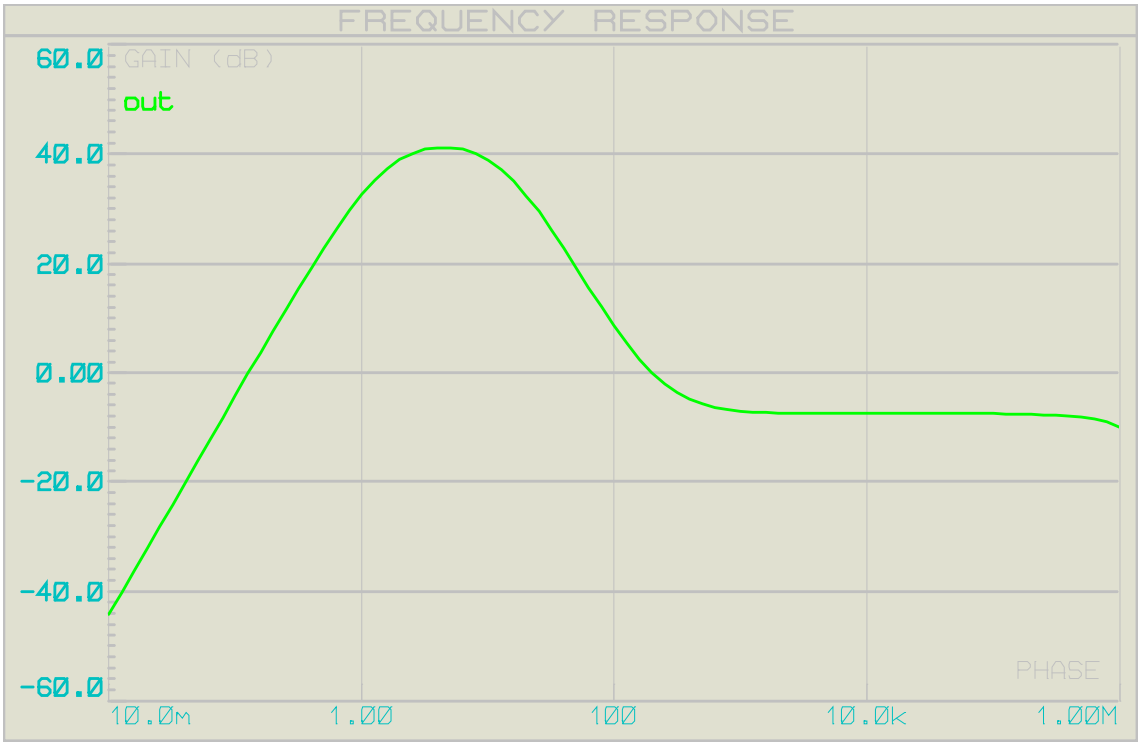
El corazón de este circuito es un amplificador operacional de STMicroelectronics, concretamente el modelo LM324N. Este es un amplificador asimétrico debido a su alimentación que puede ser de 5V o de 3.3V y por lo tanto no necesitaremos una fuente de alimentación externa, ya que la placa del microcontrolador se alimenta con 5V.

Teniendo en cuenta un rango de pulsaciones por minuto razonable, que pueden oscilar entre 30 - 300, se define la correspondencia a un ancho de banda de 0,5 Hz a 5 Hz, lo cual se convierte en el respaldo teórico para determinar las frecuencias de corte del filtro a diseñar. En esta etapa cabe resaltar, que las mayores fuentes de ruido e interferencia para este modelo están relacionadas con las que afectan directamente la medición hecha por el fotodiodo, así que se debe asegurar una buena ubicación del sensor dentro del sistema, que lo proteja de fuentes externas directas, que puedan alterar el buen desempeño del esquema propuesto.

El diseño del filtro se ha hecho a través de un programa online de diseño de filtros. Obteniendo el siguiente resultado:



Como vemos es un filtro pasivo pasabanda de segundo orden, se ha colocado un potenciómetro entre la dos etapas para ajustar la ganancia. En la siguiente figura podemos observar la simulación del filtro (respuesta en frecuencia) mediante el programa PROTEUS.



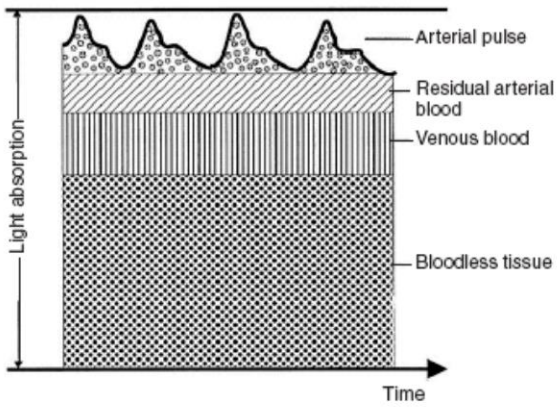
La salida del circuito se conecta a una de las entradas analógicas del microcontrolador y se muestrea a una frecuencia de 10 milisegundos.

4.5.3 CALCULOS SOFTWARE

En el apartado 3.4 se describió teóricamente el diseño de un pulsioxímetro, como ya se comentó en el apartado de introducción finalmente se decidió implementar solo un pulsómetro, por lo tanto en este apartado nos centraremos únicamente en los pasas a seguir para implementar un pulsómetro.

Ya que solo queremos obtener frecuencia cardiaca, con utilizar solo un led en la sonda nos bastará, el led utilizado será el rojo ya que es el unico visible, este LED emitir luz a través de un lecho capilar pulsátil, por ejemplo un dedo. Y el fotodetector captara la luz roja que atraviesa al dedo.

A la salida de la etapa de amplificación y filtrado obtendremos una señal como la de la figura siguiente.



Light Absorption by Tissue Type[3]

En la figura podemos diferenciar una componente pulsátil o AC, que corresponde a la pulsación de la sangre arterial, y una base o componente DC, que representa la absorción del lecho tisular, incluidos la sangre venosa, capilares sanguíneos y la sangre arterial no pulsátil.

Para determina la frecuencia cardiaca (como se explica en el apartado 3.4.4) bastará con muestrear la señal con el microcontrolador y realizar una FFT para obtener el espectro en frecuencia de la señal muestreada, de ahí podremos obtener la frecuencia cardiaca, que corresponderá a la frecuencia de más amplitud del espectro.

Muestreo

La frecuencia a la cual muestreamos tiene que ser mayor que la frecuencia dada por el teorema de Nyquist. Este teorema fundamental indica que si la frecuencia más alta contenida en una señal analógica $Xa(t)$, es $Fmax = B$, y la señal se muestrea a una tasa $Fs > 2Fmax \equiv 2B$, entonces $Xa(t)$ se puede recuperar totalmente a partir de sus muestras. La frecuencia que nos interesa detectar viene dada por el tiempo crítico. Si deseamos detectar el pulso, la frecuencia que tiene la señal es la generada por el pulso humano, siendo este de 40 ppm a 250 ppm en sus casos más extremos, lo que nos da un 1 latido cada 240 milisegundos (en el peor caso), es decir, una frecuencia de 4,17 Hz. Por lo tanto, debemos tomar muestras como máximo cada 120 milisegundos.

Para muestrear con STM32F407 se utiliza el conversor analógico digital ADC3 y DMA para transferir datos convertidos de forma continua y rápida de ADC3 a la memoria. A continuación se muestra el código de configuración.

```
/* ADC3 configuration*****************************************/
  /*  - Enable peripheral clocks */
  /*  - DMA2_Stream0 channel2 configuration */
  /*  - Configure ADC Channel12 pin as analog input */
  /*  - Configure ADC3 Channel12*/
void ADC3_CH12_DMA_Config(void)
{
  ADC_InitTypeDef       ADC_InitStructure;
  ADC_CommonInitTypeDef ADC_CommonInitStructure;
  DMA_InitTypeDef       DMA_InitStructure;
  GPIO_InitTypeDef      GPIO_InitStructure;

  /* Enable ADC3, DMA2 and GPIO clocks****************************/
  RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_DMA2 | RCC_AHB1Periph_GPIOC,
      ENABLE);
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC3, ENABLE);

  /* DMA2 Stream0 channel0 configuration**************************/
  DMA_InitStructure.DMA_Channel = DMA_Channel_2;
  DMA_InitStructure.DMA_PeripheralBaseAddr =
      (uint32_t)ADC3_DR_ADDRESS;
  DMA_InitStructure.DMA_Memory0BaseAddr =
      (uint32_t)&ADC3ConvertedValue;
  DMA_InitStructure.DMA_DIR = DMA_DIR_PeripheralToMemory;
  DMA_InitStructure.DMA_BufferSize = 1;
  DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Disable;
  DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Disable;
  DMA_InitStructure.DMA_PeripheralDataSize =
      DMA_PeripheralDataSize_HalfWord;
  DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_HalfWord;
  DMA_InitStructure.DMA_Mode = DMA_Mode_Circular;
  DMA_InitStructure.DMA_Priority = DMA_Priority_High;
  DMA_InitStructure.DMA_FIFOMode = DMA_FIFOMode_Disable;
  DMA_InitStructure.DMA_FIFOThreshold = DMA_FIFOThreshold_HalfFull;
  DMA_InitStructure.DMA_MemoryBurst = DMA_MemoryBurst_Single;
  DMA_InitStructure.DMA_PeripheralBurst = DMA_PeripheralBurst_Single;
  DMA_Init(DMA2_Stream0, &DMA_InitStructure);
  DMA_Cmd(DMA2_Stream0, ENABLE);

  /* Configure ADC3 Channel12 pin as analog input***************/
```

```
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_2;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AN;
  GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL ;
  GPIO_Init(GPIOC, &GPIO_InitStructure);

  /* ADC Common Init **********************************************/
  ADC_CommonInitStructure.ADC_Mode = ADC_Mode_Independent;
  ADC_CommonInitStructure.ADC_Prescaler = ADC_Prescaler_Div2;
  ADC_CommonInitStructure.ADC_DMAAccessMode =
      ADC_DMAAccessMode_Disabled;
  ADC_CommonInitStructure.ADC_TwoSamplingDelay =
      ADC_TwoSamplingDelay_5Cycles;
  ADC_CommonInit(&ADC_CommonInitStructure);

  /* ADC3 Init ****************************************************/
  ADC_InitStructure.ADC_Resolution = ADC_Resolution_12b;
  ADC_InitStructure.ADC_ScanConvMode = DISABLE;
  ADC_InitStructure.ADC_ContinuousConvMode = ENABLE;
  ADC_InitStructure.ADC_ExternalTrigConvEdge =
      ADC_ExternalTrigConvEdge_None;
  ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right;
  ADC_InitStructure.ADC_NbrOfConversion = 1;
  ADC_Init(ADC3, &ADC_InitStructure);

  /* ADC3 regular channel12 configuration*********************/
  ADC_RegularChannelConfig(ADC3, ADC_Channel_12, 1,
      ADC_SampleTime_3Cycles);
 /* Enable DMA request after last transfer (Single-ADC mode) */
  ADC_DMARequestAfterLastTransferCmd(ADC3, ENABLE);
  /* Enable ADC3 DMA */
  ADC_DMACmd(ADC3, ENABLE);
  /* Enable ADC3 */
  ADC_Cmd(ADC3, ENABLE);
}
```

● El ADC3 está configurado para convertir del pin PC2 continuamente.

● Cada vez que se produce un final de conversión, el DMA transfiere los datos convertidos del registro DR de ADC3 DR hacia la variable `ADC3ConvertedValue` en modo circular.

Para pasar a voltaje real el valor ADC que se guarda en la variable `ADC3ConvertedValue` hay que tener en cuenta que el valor máximo de `ADC3ConvertedValue` es 0xFFF y el mínimo 0. Sabemos que el mayor voltaje que puede detectar el microcontrolador es 3.3V y el mínimo 0. Por lo tanto para obtener el valor en voltios hay que hacer la siguiente operación

```
ADCVoltageValue = ADC3ConvertedValue *3300/0xFFF
```

● El reloj del sistema es igual a 144 MHz, el reloj APB2 es igual a 72 MHz y el reloj ADC es igual a APB2 / 2.

● Dado que el reloj ADC3 es 36 MHz y el tiempo de muestreo se establece en 3 ciclos, el tiempo de conversión de datos de 12 bits es de 12 ciclos, por lo que el tiempo de conversión total es(12 + 3) / 36 = 0.41 us.

Como podemos observar este tiempo es despreciable frente al tiempo de muestreo que vamos a utilizar para muestrear la señal que será de entre 10 ms. Por lo que asumiremos que cada vez que queramos almacenar una muestra no habrá ningún tiempo de retardo.

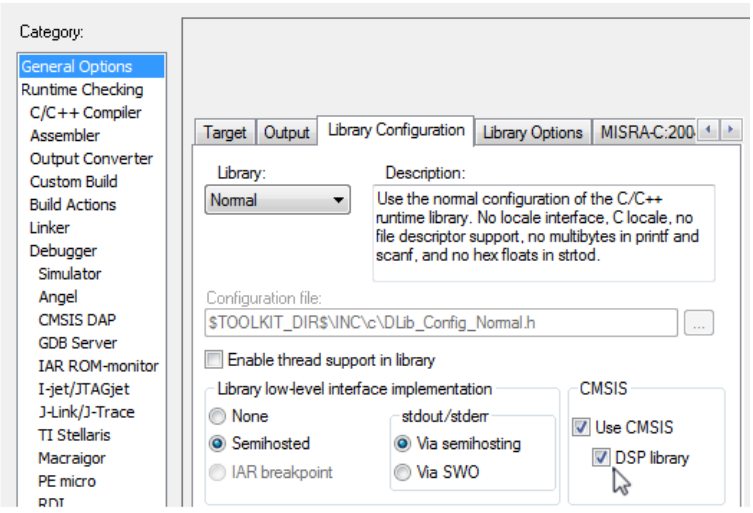
Cálculo de la FFT y obtención de frecuencia cardiaca

Los procesadores ARM Cortex-M3 / M4 proporciona instrucciones para el procesamiento de señales, por ejemplo SIMD (*Single Instrucción Multi Data*). Especialmente Cortex-M4 (que es el que usa la placa Discovery STM32F407) está diseñado para aplicaciones DSP y soporta instrucciones avanzadas SIMD, MAC (Multiplicar y acumular). Además, los dispositivos Cortex-M4f tienen FPU (unidad de coma flotante) para el manejo de cálculos de coma flotante.

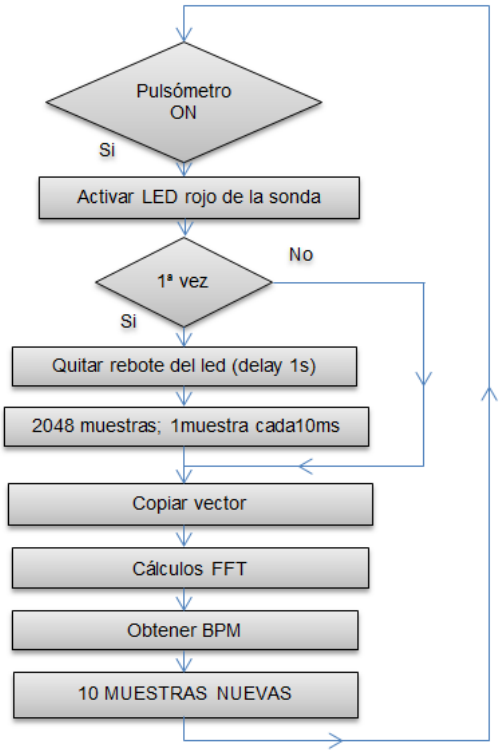
Hay varias maneras de utilizar estas instrucciones, por ejemplo usando rutinas de ensamblador o funciones intrínsecas, pero uno de los enfoques más prácticos es utilizar la librería CMSIS DSP. La librería CMSIS DSP está diseñada para los procesadores Cortex-M y proporciona funciones de procesamiento de señal digital optimizada (14).

El compilador que se ha utilizado para el desarrollo del proyecto *IAR Embedded Workbench* proporciona una librería CMSIS-DSP precompliada y su código fuente. A continuación vamos a ver cómo utilizar la librería CMSIS-DSP junto con *IAR Embedded Workbench* para ARM.

En *IAR Embedded Workbench* para ARM, hay que habilitar el uso de la librería CMSIS-DSP. Para ello, hay que establecer la opción de librería CMSIS-DSP en *General Options>Library*. Esto establecerá el PATH para el procesador C e importará la biblioteca precompilada CMSIS.



Para realizar la FFT se utilizado como punto de partida el ejemplo de FFT que proporciona ST en sus librerías y se ha modificado para adaptarlo al proyecto en cuestión. A continuación se muestra el diagrama de flujo que describe los cálculos para obtener la FFT de la señal muestreada.



Cuando el *Flag* que controla la activación del pulsómetro este a "1" el programa principal ejecutará periódicamente el código necesario para obtener la frecuencia cardiaca. La primera vez que se ejecute se tomarán 2048 muestras, 1 cada 10ms. El primer resultado de frecuencia cardiaca tardará más de 2048*10ms (20,48s) en ser obtenido. Para que no tarde tanto en obtener los siguientes valores de frecuencia cardiaca, en el siguiente cálculo se sustituirán las 10 muestras más viejas por otras 10 nuevas y el valor de frecuencia cardiaca se irá actualizando cada 100ms.

Los pasos para obtener la frecuencia cardiaca son:

- Configurar periféricos (DMA, GPIO, ADC)
- Esperar 1s.

Se ha comprobado que el fotodetector tarda aproximadamente 1 segundo en estabilizarse y empezar a tomar medidas correctas.

- Obtener 2048 muestras cada 10ms en la primera vuelta del bucle

```
for( int n=0; n<2048; n++){
      testInput [n] = ADC3ConvertedValue *3300/0xFFF;  //PC2
      while(TM_Time-tiempo2<1); //delay 10ms
}
```

- Calcular FFT

```
/* Initialize the CFFT/CIFFT module */
status= arm_cfft_radix4_init_f32(&S, fftSize, ifftFlag, doBitReverse);
/* Process the data through the CFFT/CIFFT module */
arm_cfft_radix4_f32(&S, testInput);
```

```
/* Process the data through the Complex Magnitude Module for
                    calculating the magnitude at each bin */
arm_cmplx_mag_f32(testInput, testOutput, fftSize);
/* Calculates maxValue and returns corresponding BIN value */
arm_max_f32(testOutput, fftSize, &maxValue, &testIndex);
```

Después de realizar estas instrucciones tendremos en el vector *tesOutput* el resultado de la FFT y *testIndex* apuntara al elemento de mayor valor.

- Obtención de la frecuencia cardiaca.

Como se ha dicho en el punto anterior, al finalizar los cálculos de la FFT tendremos un vector (*testOutput*) de 1024 elemento (ya que los otros 1024 son replicas en la parte negativa del eje). El valor máximo la FFT depende del número de muestras que se toma. Si se muestrea una señal con solo componente DC todo lo que se obtiene es un resultado en el vector *testOuptup[0]* (el valor será el número de muestras que se han tomado, en nuestro caso 2048) el resto del vector serán ceros.

Un parámetro a tener en cuenta es la resolución en frecuencia, la resolución viene determinada por:

$$\Delta f = \frac{\text{fs}}{\text{N}} = \frac{1}{\Delta t \cdot N}$$

Si tomamos una muestra cada 10ms la frecuencia de muestreo será de 100Hz y teniendo en cuenta que N (número de muestras) = 2048.

$$\Delta f = \frac{100}{2048} = 0.04883\ Hz$$

Esto quiere decir que cada elemento de *testOutput[n]* estará separado del anterior 0.04883 Hz, esto es:

FFT sample	Frecuencia	Descripción
Output[0]	0Hz	El primer parámetro es siempre la componente DC de la señal
Output[1]	0.04883Hz	Amplitud de la frecuencia 0.04883Hz de la señal
Output[n]	n * 0.04883Hz	Resultado del n-ésimo valor de frecuencia
Output[N – 1] = Output[1023]	1023 * Resolution = 1023 * ~0.049 = ~50Hz	La máxima frecuencia es una unidad de resolución menos que la mitad de la frecuencia de muestreo: 100 / 2 – 0.04883 = ~50Hz

Teniendo en cuenta todo lo anterior y sabiendo que el valor más alto de vector *testOutput* se almacena en la variable *testIndex*, la frecuencia cardiaca se obtendrá como sigue:

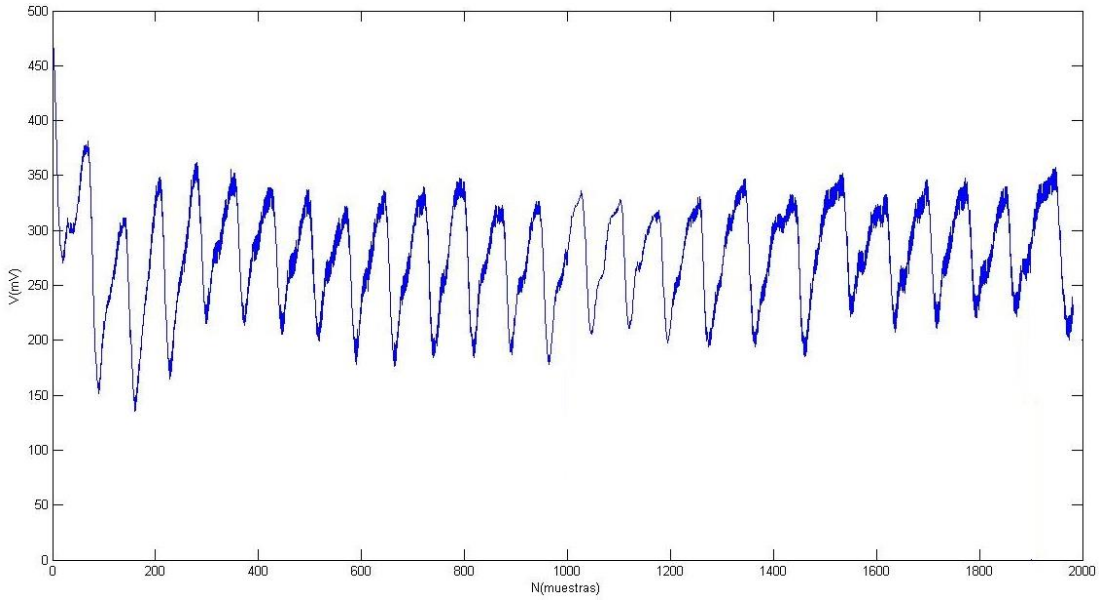
```
ppm = (100.0f/2048.0f)*( testIndex)*60.0f;
```

- El último paso es sustituir las 10 muestras más antiguas por 10 más nuevas para volver a calcular la FFT e ir actualizando el valor de la frecuencia cardiaca. Esto se repetirá hasta que se desactive la obtención de la frecuencia cardiaca.

4.5.4 Prueba pulsioxímetro

- Señal muestreada: *testInput[n]*

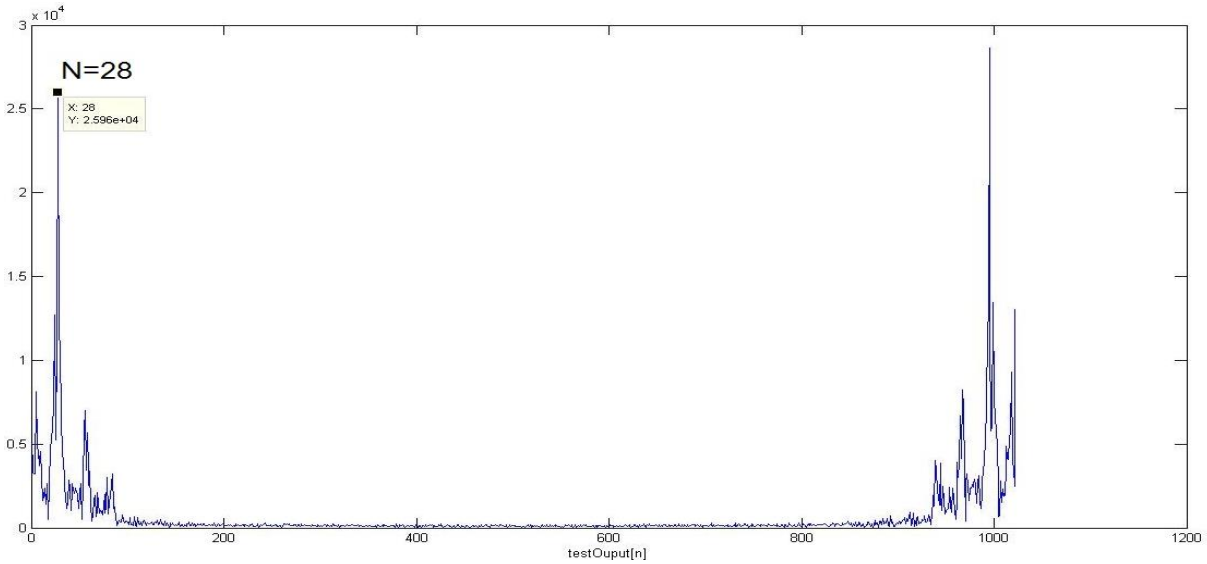
En la siguiente imagen podemos observar la señal que recibe el fotodetector de la sonda de pulsioxímetro después de ser filtrada, amplificada y muestreada. Está señal muestreada se almacena en el vector *testInput[n]*.



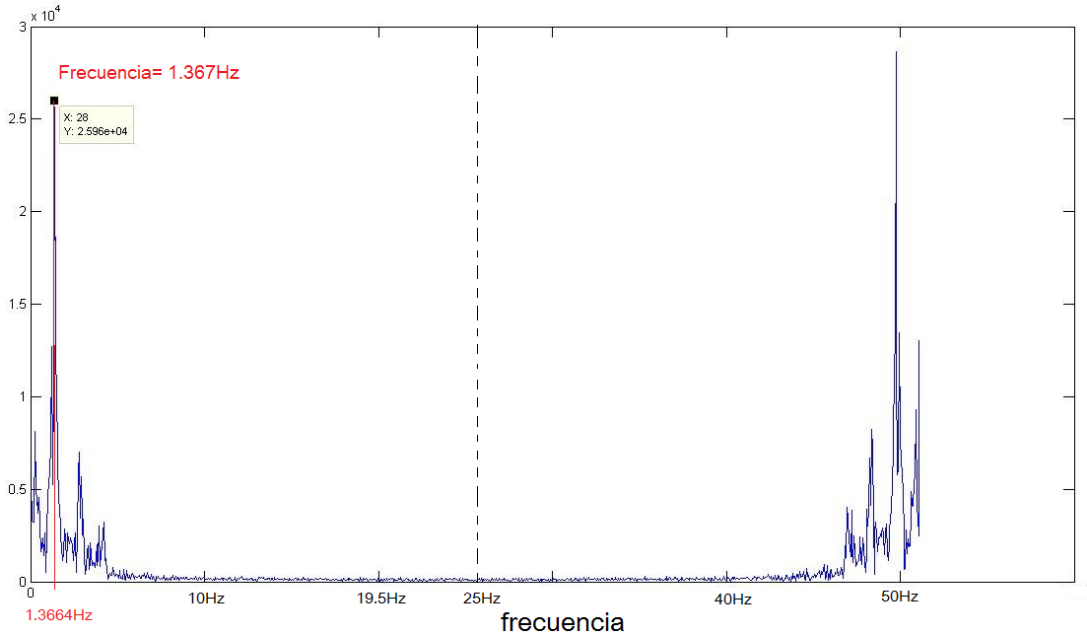
- Resultado FFT: *testOutput[n]*

Una vez que se realiza una FFT de la señal *testInput[]* de 2048 elementos se guarda el resultado en el vector *testOuput[]*, este vector es de 1024 muestras ya que las otras 1024 corresponden al eje imaginario.

Si representamos este vector en Matlab podemos observar un espectro de frecuencias simétrico.



Para ser un espectro de frecuencias el eje de abscisas debería representar frecuencias. Para ello solo hay que multiplicar el número de muestra N por la resolución en frecuencia→ f=N*Δf=N*0.04883Hz



Como podemos observar en la imagen anterior la frecuencia predominante es 1.367 Hz. Para obtener pulsaciones por minuto multiplicamos 1.367 Hz * 60 y obtenemos 82 pulsaciones por minuto.

Se ha realizado varias pruebas comparando los resultados de pulsómetro del proyecto con un pulsómetro comercial y los resultados eran parecidos tanto en exactitud como en la velocidad de obtención.

5 CONCLUSIONES

5.1 Conclusiones Técnicas

- **Qué iba a hacer**

El objetivo principal del proyecto era construir un dispositivo flexible que funcionara como un USB HID, midiendo el tiempo en que se producían diferentes eventos en el dispositivo. Este dispositivo se implementará sobre una placa Discovery STM32F407D.

Los tiempos medidos de los eventos que se producen (pulsar/soltar pulsador, inicio/fin estimulo, etc.) se pretende almacenarlos en la memoria Flash del microcontrolador.

Para poder utilizar los datos recogidos por el dispositivo se pretendía añadir capacidad de comunicación serie. Sobre este sistema de comunicación, el dispositivo puede recibir varias órdenes (*SetKey* como mínimo) que lo configuran y recogen los datos.

Para poder tratar los datos es necesario sincronizar relojes entre el PC donde se estudiarán los datos y el microcontrolador. Para realizarlo se usará en enlace de comunicación serie.

Opcionalmente, se pretendía implementar un pulsioxímetro para disponer de más información a la hora de realizar análisis sobre las acciones del usuario del dispositivo. De esta forma se añadiría otra mejora a los dispositivos ya existentes.

- **Qué he hecho**

Se ha implementado un teclado USB HID en la placa Discovery STM32F40D con dos pulsadores, los cuales se pueden configurar para que al pulsarlos se envíe cualquiera de los caracteres que podemos encontrar en un teclado estándar.

Para configurar los pulsadores y poder utilizar los datos recogidos por el dispositivo se ha implementado en el microcontrolador una consola de control que se comunica mediante puerto serie con un PC. Desde el PC, trabajando con cualquier programa que disponga de funciones de puerto serie, se puede pedir al dispositivo que realice varias órdenes: configurar teclas, obtener el valor actual de las teclas, obtener tabla que muestra el tiempo que tardó en producirse cada evento (pulsar/soltar pulsador, inicio/fin estimulo, activar/desactivar pulsómetro, guardar/leer/borrar datos de la memoria Flash y sincronizar relojes).

Ya que uno de los objetivos de este proyecto era mejorar los dispositivos USB HID para análisis psicológicos ya existentes (3), se creyó conveniente implementar un pulsómetro para que el especialista pudiese controlar la frecuencia cardiaca del sujeto bajo test en cualquier instante de tiempo.

- **Lo qué no he hecho ha sido por…**

Finalmente se decidió medir únicamente la pulsación ya que se carecía de los medios necesarios para verificar si la medida del oxígeno en sangre era correcta, no obstante, en la memoria se indican tanto la teoría como la implementación que se debería usar en el microcontrolador para la obtención de esta medida. De hecho, una vez realizada la FFT de la señal de entrada lo único que hay que hacer para obtener el porcentaje de oxígeno es sangre es aplicar dos simples fórmulas que dependen de 4 parámetros que se obtienen al realizar la FFT. Pero era imposible verificar las medidas ya que no se disponía de los medios necesarios.

- **Qué más se podría hacer**

Se podría haber optimizado más el pulsómetro para obtener el primer resultado más rápido añadiendo ceros en el vector donde se almacenan las muestras de la señal de entrada. Si en lugar de tomar 2048 muestras se toman 1500 (por ejemplo) y lo demás se rellena con ceros, se tardaría menos tiempo en muestrear la señal y el resultado de la FFT sería prácticamente el mismo.

Para optimizar costes y obtener un producto final más comercial se podría haber realizar un circuito impreso añadiendo a este los elementos necesarios y un encapsulado. También se podría haber realizado un software informático que trabajase con datos recibidos por el puerto serie y que sirviese de interfaz entre el dispositivo y el especialista.

El motivo por el que no se han realizado estas mejoras es porque este proyecto es un proyecto de final de carrera en el que el director determinó que el objetivo principal era realizar un dispositivo USB HID con *data-logger* al que se le podría añadir un pulsioxímetro como suplemento.

5.2 Conclusiones Personales

El objetivo personal principal a la hora de realizar el Proyecto Final de Carrera era poner a prueba el máximo de conceptos aprendidos a lo largo de toda la Ingeniería en Telecomunicaciones con especialidad en electrónica. La implementación de un dispositivo USB HID con *data-logger* en un microcontrolador ha servido para repasar y asentar conceptos aprendidos en asignaturas como Sistemas Electrónicos Avanzados, ACSO I y II, Electrónica Digital, Programación y Sistemas Electrónicos Digitales.

La realización de un pulsioxímetro se planteó al inicio del proyecto como un objetivo opcional, finalmente se decidió incluirlo porque aparte de ser una aportación interesante para proyecto, el obtener la frecuencia cardiaca mediante FFT sirvió para repasar y asentar conceptos de asignaturas de la parte de comunicaciones dentro de la Ingeniería en Telecomunicaciones como: Tratamiento Digital de la Señal, Laboratorio de TDS, Teoría de la Comunicación, etc...

Para acondicionar la señal procedente de la sonda de pulsioxímetro a los niveles del microcontrolador se ha tenido que diseñar un filtro activo con ganancia empleando conocimiento adquiridos en gran parte de las asignaturas relacionadas con la electrónica en la titulación como Componentes Electrónicos, Electrónica Analógica, Instrumentación Electrónica, Subsistemas Electrónicos de Comunicación.

6 Bibliografía

1. Why you should be worried about millisecond timing! [En línea] http://www.blackboxtoolkit.com/why.html.

2. ¿Sabías que Exactitud no es lo mismo que Precisión? [En línea] http://www.e-medida.es/documentos/Numero-1/exactitud-no-es-lo-mismo-que-precision.htm.

3. The Black Box ToolKit. [En línea] http://www.blackboxtoolkit.com/urp.html.

4. Universal Serial Bus (USB) Device Class Definition for Human Interface Devices (HID). [En línea] 27 de 07 de 01. http://www.usb.org/developers/hidpage/HID1_11.pdf.

5. User manual. STM32Fxx USB On-The-Go host and device library. [En línea] March de 2012. http://www.st.com/st-web-ui/static/active/jp/resource/technical/document/user_manual/CD00289278.pdf.

6. Universal Serial Bus Revision 2.0 specification. [En línea] http://www.usb.org/developers/docs/usb20_docs/#usb20spec.

7. **Dr. Luis Díaz Soto.** *Comparación de la saturación arterial de oxigeno por oximetría de pulso y gasometría arterial, Rev Cub Med Int Emerg* . 2003.

8. **Luis G. Meza Contreras, Luis Enrique Llamosa R., Silvia Patricia Ceballos.** *Diseño de procedimientos para la calibración de pulsioxímetros.* s.l. : Scientia, Diciembre 2007.

9. **López-Herranz, G. Patricia.** Oxiometría de pulso: A la vanguardia en la monitorización no invasiva de la oxigenación. [En línea] Jul.-Sep. de 2003. http://www.respira.com.mx/docs/f1279559061-0.pdf.

10. **Joseph Bailey, Michael Fecteau, Noah L. Pendleton.** WIRELESS PULSE OXIMETER. [En línea] https://www.wpi.edu/Pubs/E-project/Available/E-project-042408-101301/unrestricted/WPO_MQP-Final_04242008.pdf.

11. Pulse Oximeter Fundamentals and Design. [En línea] http://www.freescale.com/files/32bit/doc/app_note/AN4327.pdf?tid=AMdlDR.

12. http://www.usb.org/developers/hidpage/Hut1_12v2.pdf. [En línea] 28 de 10 de 2004. http://www.usb.org/developers/hidpage/Hut1_12v2.pdf.

13. [En línea] http://www.frankshospitalworkshop.com/equipment/documents/pulse_oximeter/service_manuals/Nellcor%20DS-100A.jpg.

14. Improve performance of digital signal processing with IAR Embedded Workbench for ARM. [En línea] https://www.iar.com/Support/resources/articles/using-iar-embedded-workbench-for-arm-and-the-cmsis-dsp-library/.

7 Apéndice

7. 1.Programa principal *main.c*

```
/**
 *    IAR project for USB HID Device
 *
 *    @author     Juan Domingo Jiménez
*    @email        juajimje@upv.es
 */
/* Include core modules */
#include "stm32f4xx.h"
#include "stm32f4_discovery.h"
#include "defines.h"
#include "tm_stm32f4_usb_hid_device.h"
 #include "tm_stm32f4_delay.h"
#include "tm_stm32f4_disco.h"
#include <stdio.h>
#include <stdlib.h>

__IO uint8_t DemoEnterCondition = 0x00;
__IO uint8_t UserButtonPressed = 0x00;

extern uint8_t tecla1;
extern uint8_t tecla2;

int echo=1;

/*Time*/
extern __IO uint32_t TM_Time; //incrementa en 1 cada ms
__IO uint32_t ms_decimales,cs_decimales, VALo,Tiempo_Actual=0;
//variables para obtener el tiempo en decimas de ms

/* Vectores*/
#define MAX 256
uint32_t __IO Tiempos[MAX];
char event_type[MAX][4];
uint32_t event_count[MAX];
uint32_t next_slot=0;
int idx_time=0, idx_event=0; //indices
uint32_t np1i=0, np1f=0, np2i=0,np2f=0, nei=0,nef=0;  //contadores de
eventos de cada tipo
```

```
char st1,st2,st3,st4,st5,st6;

//******* function prototypes USART ******************
void UART_Initialize(void);                          //**
void GPIOInitialize(void);                           //**
void NVICInitialize(void);                           //**
void USART_puts(   volatile char *s);                //**
                                                     //**
USART_InitTypeDef USART_InitStructure;               //**
GPIO_InitTypeDef GPIO_InitStructure;                 //**
extern NVIC_InitTypeDef NVIC_InitStructure;          //**
//******************************************************

//*****************************PULSOMETRO*****************************
#include "arm_math.h"
#define TEST_LENGTH_SAMPLES 2048
static float32_t testOutput[TEST_LENGTH_SAMPLES/2];
 uint32_t fftSize = 1024,ifftFlag = 0, doBitReverse = 1, testIndex =
0; ;
float32_t testInput_f32_10khz[2048],Input_f32_10khz[2048];
#define ADC3_DR_ADDRESS     ((uint32_t)0x4001224C)
__IO uint16_t ADC3ConvertedValue = 0;
__IO uint32_t VR = 0, VIR=0; //voltage
uint32_t tiempo1=0, tiempo2=0,tiempo3=0, ti=0, tf=0 ,Tcalculos=0;
uint8_t flag1=0;
float ppm=0;
void ADC3_CH12_DMA_Config(void);

extern int activar_p;
//*****************************************************************
uint32_t dms(void);

int main(void) {

/*Initialize LEDs and User_Button on STM32F4-Discovery */
    STM_EVAL_PBInit(BUTTON_USER, BUTTON_MODE_EXTI);
    STM_EVAL_LEDInit(LED4); STM_EVAL_LEDInit(LED3);
STM_EVAL_LEDInit(LED5); STM_EVAL_LEDInit(LED6);
```

```
  /* ADC3 configuration
*****************************************************/
  /*  - Enable peripheral clocks
*/
  /*  - DMA2_Stream0 channel2 configuration
*/
  /*  - Configure ADC Channel12 pin as analog input
*/
  /*  - Configure ADC3 Channel12
*/
  ADC3_CH12_DMA_Config();
  /* Start ADC3 Software Conversion
*/
  ADC_SoftwareStartConv(ADC3);
  /*************************************************************/
  /* GPIOD Periph clock enable */
  RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOD, ENABLE);
  /* Configure PD12, PD13, PD14 and PD15 in output pushpull mode */
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9|GPIO_Pin_10 ;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;
  GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;
  GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;
  GPIO_Init(GPIOD, &GPIO_InitStructure);
/*************************************************************/
//********************** USART1 **********************
    GPIOInitialize();                                 //**
    UART_Initialize();                                //**
    NVICInitialize();                                 //**
//****************************************************


    uint8_t already1 = 0, already2=0,flag = 0; //flags


    TM_USB_HIDDEVICE_Keyboard_t Keyboard;  /* Set struct */
    SystemInit();   /* Initialize system */
    TM_DISCO_LedInit(); /* Initialize leds */
    TM_DISCO_ButtonInit(); /* Initialize button */


    /*Inicializamos boton externo*/
    TM_GPIO_Init(GPIOD, GPIO_PIN_14, TM_GPIO_Mode_IN,
TM_GPIO_OType_PP, TM_DISCO_BUTTON_PULL, TM_GPIO_Speed_Low);
    /*Inicializamos entrada estimulo*/
    TM_GPIO_Init(GPIOE, GPIO_PIN_7, TM_GPIO_Mode_IN, TM_GPIO_OType_PP,
TM_DISCO_BUTTON_PULL, TM_GPIO_Speed_Low);
```

```
    TM_DELAY_Init();  /* Initialize delay */
    TM_USB_HIDDEVICE_Init();   /* Initialize USB HID Device */
    TM_USB_HIDDEVICE_KeyboardStructInit(&Keyboard); /* Set default
values for keyboard struct */

    USART_puts("introduce orden \n"); //Frase inicial en consola

    while (1) {
//==================================WHILE(1)====================== /
        /* If we are connected and drivers are OK */
        if (TM_USB_HIDDEVICE_GetStatus() ==
TM_USB_HIDDEVICE_Status_Connected) {
            /* Turn on green LED */
            TM_DISCO_LedOn(LED_GREEN);

/*Boton1*/ if ((TM_GPIO_GetInputPinValue(GPIOD, GPIO_PIN_14)!=0) &&
already1 == 0) { // Button1 on

                already1 = 1;
                Keyboard.Key1 = tecla1;
                TM_USB_HIDDEVICE_KeyboardSend(&Keyboard);// Send
keyboard report

                np1i++;//contador evento pulsar boton1
                Tiempos[next_slot]=dms();
                event_type[next_slot][0]='P';
                event_type[next_slot][1]='1';
                event_type[next_slot][2]='I';
                event_count[next_slot]=np1i; next_slot++;
                Delay(20000); // 0.1s

            } else if (!TM_GPIO_GetInputPinValue(GPIOD, GPIO_PIN_14)
&& already1 == 1) { // Button1 release
                already1 = 0;
               /* Release all buttons*/
                Keyboard.L_GUI = TM_USB_HIDDEVICE_Button_Released;
// No button
                Keyboard.Key1 = 0x00;
// No key
                TM_USB_HIDDEVICE_KeyboardSend(&Keyboard); // Send
keyboard report

                np1f++;
                Tiempos[next_slot]=dms();
                event_type[next_slot][0]='P';
```

```
                event_type[next_slot][1]='1';
                event_type[next_slot][2]='F';
                event_count[next_slot]=np1f; next_slot++;
            }


/*Boton2*/ if (TM_DISCO_ButtonPressed() && already2 == 0) { //Button2
on
                already2 = 1;
                Keyboard.Key1 = tecla2;
                TM_USB_HIDDEVICE_KeyboardSend(&Keyboard); // Send
keyboard report

                np2i++;
                Tiempos[next_slot]=dms();
                event_type[next_slot][0]='P';
                event_type[next_slot][1]='2';
                event_type[next_slot][2]='I';
                event_count[next_slot]=np2i; next_slot++;
                Delay(20000); //0.1s

            } else if (!TM_DISCO_ButtonPressed() && already2 == 1) {
// Button2 release
                already2 = 0;
                /* Release all buttons*/
                Keyboard.L_GUI = TM_USB_HIDDEVICE_Button_Released;
// No button
                Keyboard.Key1 = 0x00;
// No key
                TM_USB_HIDDEVICE_KeyboardSend(&Keyboard); // Send
keyboard report

                np2f++;
                Tiempos[next_slot]=dms();
                event_type[next_slot][0]='P';
                event_type[next_slot][1]='2';
                event_type[next_slot][2]='F';
                event_count[next_slot]=np2f; next_slot++;
            }
/*Estimul*/if (TM_GPIO_GetInputPinValue(GPIOE, GPIO_PIN_7)!=0) { //
estimulo activo
                if(flag==0){
                  TM_DISCO_LedOn(LED_ORANGE);
                  nei++;
```

```
                    Tiempos[next_slot]=dms();
                    event_type[next_slot][0]='E';
                    event_type[next_slot][1]='I';
                    event_count[next_slot]=nei; next_slot++;
                    Delay(5000); //25ms
                    flag=1;
                  }
              }
              else if ((TM_GPIO_GetInputPinValue(GPIOE,
GPIO_PIN_7)==0)&& flag == 1){

                    nef++;
                    Tiempos[next_slot]=dms();
                    event_type[next_slot][0]='E';
                    event_type[next_slot][1]='F';
                    event_count[next_slot]=nef; next_slot++;
                    TM_DISCO_LedOff(LED_ORANGE);
                    Delay(5000);//25ms
                    flag=0;
              }
              } else {
                    /* Turn off green LED */
                    TM_DISCO_LedOff(LED_GREEN);
              }
/*pulsomet*/if(activar_p==1){  /*******************PULSOMETRO*******/
                     GPIO_SetBits(GPIOD, GPIO_Pin_9);
                    if(flag1==0){
                      tiempo1=TM_Time;  //solo la primera vez
                      while(TM_Time-tiempo1<100);  //quitar rebote
sensor
                      //  while(TM_Time-tiempo1<3) /* convert the ADC
value (from 0 to 0xFFF --4095 decimal) to a voltage value (from 0V to
3.3V)*/
                      for( int n=0; n<2048; n++){
                          testInput_f32_10khz[n] = ADC3ConvertedValue
*3300/0xFFF;  //PC2
                          tiempo2=TM_Time;
                          while(TM_Time-tiempo2<1);
                      }
                      flag1=1;
                    }
                      ti=dms();//Tiempo inicio calculos
                      for(int j=0;j<2038;j++){ //Guardamos todas las
muestras menos las 10 primeras antes de machacar el vector
```

```
                    Input_f32_10khz[j]=testInput_f32_10khz[j+10];
                  }
                //  /* Disable ADC3 DMA */  ADC_DMACmd(ADC3,
DISABLE);
                //  /* Disable ADC3 */      ADC_Cmd(ADC3, DISABLE);
                  arm_status status;
                  arm_cfft_radix4_instance_f32 S;
                  float32_t maxValue;
                  status = ARM_MATH_SUCCESS;
                  /* Initialize the CFFT/CIFFT module */
                  status = arm_cfft_radix4_init_f32(&S, fftSize,
ifftFlag, doBitReverse);
                  /* Process the data through the CFFT/CIFFT module
*/
                  arm_cfft_radix4_f32(&S, testInput_f32_10khz);
                  /* Process the data through the Complex Magnitude
Module for
                  calculating the magnitude at each bin */
                  arm_cmplx_mag_f32(testInput_f32_10khz, testOutput,
fftSize);
                  testOutput[0]=0;
                  /* Calculates maxValue and returns corresponding
BIN value */
                  arm_max_f32(testOutput, fftSize, &maxValue,
&testIndex);

                  ppm = (100.0f/2048.0f)*(1024.0f-testIndex)*60.0f;


                  for(int q=0;q<2048;q++){ //recuperamos vector
entero
                    testInput_f32_10khz[q]=Input_f32_10khz[q];
                  }
                 // /* Enable ADC3 DMA */  ADC_DMACmd(ADC3, ENABLE);
                 // /* Enable ADC3 */      ADC_Cmd(ADC3, ENABLE);
                  tf=dms();
                  Tcalculos= ti -tf;
                  for( int k=2038; k<2048; k++){   //añadimos 10
muestras actuales
                    testInput_f32_10khz[k] = ADC3ConvertedValue
*3300/0xFFF;  //PC2
                    tiempo2=TM_Time;
                    while(TM_Time-tiempo2<1);
                  }
              }
```

```
    } //End while(1)
} //End Main

/*Funcion que retorna decimas de milisegundo*/
uint32_t dms(void)
{
       VALo=SysTick->VAL;  //Valor actual del SysTick
       cs_decimales=(uint32_t)(VALo*99/168000);    //entre 0 y 99
decimas de s^-2
       ms_decimales=(TM_Time*100)+(cs_decimales);
       return ms_decimales;
}

//***********************************************FUNCIONES UART1*/
void UART_Initialize(void)
{
 /* Enable peripheral clock for USART1 */
 RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1, ENABLE);

/* USART1 configured as follow:
 * BaudRate 9600 baud
 * Word Length 8 Bits
 * 1 Stop Bit
 * No parity
 * Hardware flow control disabled
 * Receive and transmit enabled
 */
 USART_InitStructure.USART_BaudRate = 9600;
 USART_InitStructure.USART_WordLength = USART_WordLength_8b;
 USART_InitStructure.USART_StopBits = USART_StopBits_1;
 USART_InitStructure.USART_Parity = USART_Parity_No;
 USART_InitStructure.USART_HardwareFlowControl =
USART_HardwareFlowControl_None;
 USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx;

USART_Init(USART1, &USART_InitStructure); // USART configuration
 USART_Cmd(USART1, ENABLE); // Enable USART
}

void GPIOInitialize(void)
{
 RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB, ENABLE); //Enable clock
for GPIOB
```

```
/* USART1 Tx on PB6 | Rx on PB7 */
 GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6 | GPIO_Pin_7;
 GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;
 GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;
 GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
 GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;

GPIO_Init(GPIOB, &GPIO_InitStructure);
 GPIO_PinAFConfig(GPIOB, GPIO_PinSource6, GPIO_AF_USART1);//Connect
PB6 to USART1_Tx
 GPIO_PinAFConfig(GPIOB, GPIO_PinSource7, GPIO_AF_USART1);//Connect
PB7 to USART1_Rx
}

void NVICInitialize(void)
{

 NVIC_InitStructure.NVIC_IRQChannel = USART1_IRQn;
 NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1;
 NVIC_InitStructure.NVIC_IRQChannelSubPriority = 1;
 NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;

NVIC_Init(&NVIC_InitStructure);
 USART_ITConfig(USART1, USART_IT_RXNE, ENABLE);    //habilita las
interrupciones USART

}
void USART_puts(volatile char *s){
  if(echo!=48){  // El 0 de la consola se detecta como 48
    // wait until data register is empty
    while(*s){
      while( !(USART1->SR & 0x00000040) );
      USART_SendData(USART1, *s);
      *s++;
    }
  }
  return;
} //*****************************FIN FUNCIONES UART1****************

/************************PULSOMETRO*********************************/
void ADC3_CH12_DMA_Config(void)
{
```

```
  ADC_InitTypeDef       ADC_InitStructure;
  ADC_CommonInitTypeDef ADC_CommonInitStructure;
  DMA_InitTypeDef       DMA_InitStructure;
  GPIO_InitTypeDef      GPIO_InitStructure;

  /* Enable ADC3, DMA2 and GPIO clocks ****************************/
  RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_DMA2 | RCC_AHB1Periph_GPIOC,
ENABLE);
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC3, ENABLE);

  /* DMA2 Stream0 channel0 configuration**************************/
  DMA_InitStructure.DMA_Channel = DMA_Channel_2;
  DMA_InitStructure.DMA_PeripheralBaseAddr =
(uint32_t)ADC3_DR_ADDRESS;
  DMA_InitStructure.DMA_Memory0BaseAddr =
(uint32_t)&ADC3ConvertedValue;
  DMA_InitStructure.DMA_DIR = DMA_DIR_PeripheralToMemory;
  DMA_InitStructure.DMA_BufferSize = 1;
  DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Disable;
  DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Disable;
  DMA_InitStructure.DMA_PeripheralDataSize =
DMA_PeripheralDataSize_HalfWord;
  DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_HalfWord;
  DMA_InitStructure.DMA_Mode = DMA_Mode_Circular;
  DMA_InitStructure.DMA_Priority = DMA_Priority_High;
  DMA_InitStructure.DMA_FIFOMode = DMA_FIFOMode_Disable;
  DMA_InitStructure.DMA_FIFOThreshold = DMA_FIFOThreshold_HalfFull;
  DMA_InitStructure.DMA_MemoryBurst = DMA_MemoryBurst_Single;
  DMA_InitStructure.DMA_PeripheralBurst = DMA_PeripheralBurst_Single;
  DMA_Init(DMA2_Stream0, &DMA_InitStructure);
  DMA_Cmd(DMA2_Stream0, ENABLE);

  /* Configure ADC3 Channel12 pin as analog input ******************/
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_2;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AN;
  GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL ;
  GPIO_Init(GPIOC, &GPIO_InitStructure);

  /* ADC Common Init **********************************************/
  ADC_CommonInitStructure.ADC_Mode = ADC_Mode_Independent;
  ADC_CommonInitStructure.ADC_Prescaler = ADC_Prescaler_Div2;
  ADC_CommonInitStructure.ADC_DMAAccessMode =
ADC_DMAAccessMode_Disabled;
```

```
  ADC_CommonInitStructure.ADC_TwoSamplingDelay =
ADC_TwoSamplingDelay_5Cycles;
  ADC_CommonInit(&ADC_CommonInitStructure);

  /* ADC3 Init ****************************************************/
  ADC_InitStructure.ADC_Resolution = ADC_Resolution_12b;
  ADC_InitStructure.ADC_ScanConvMode = DISABLE;
  ADC_InitStructure.ADC_ContinuousConvMode = ENABLE;
  ADC_InitStructure.ADC_ExternalTrigConvEdge =
ADC_ExternalTrigConvEdge_None;
  ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right;
  ADC_InitStructure.ADC_NbrOfConversion = 1;
  ADC_Init(ADC3, &ADC_InitStructure);

  /* ADC3 regular channel12 configuration *************************/
  ADC_RegularChannelConfig(ADC3, ADC_Channel_12, 1,
ADC_SampleTime_3Cycles);

 /* Enable DMA request after last transfer (Single-ADC mode) */
  ADC_DMARequestAfterLastTransferCmd(ADC3, ENABLE);

  /* Enable ADC3 DMA */
  ADC_DMACmd(ADC3, ENABLE);

  /* Enable ADC3 */
  ADC_Cmd(ADC3, ENABLE);
}
/**************************************************************/
```

7.2 Gestor de interrupciones *stm32f4xx_it.c*

```
/******************************************************************
  * @file    stm32f4xx_it.c
  * @author  Juan Domingo Jiménez Jerez
  * @date    14-Mayo-2015
  * @brief   Main Interrupt Service Routines.
  *          This file provides all exceptions handler and peripherals
interrupt
  *          service routine.

******************************************************************/
/* Includes ------------------------------------------------------*/
#include "stm32f4xx_it.h"
//#include "main.h"
```

```
#include "usb_core.h"
#include "usbd_core.h"
#include "stm32f4_discovery.h"
#include "usbd_hid_core.h"
#include "stm32f4xx_conf.h"
#include "usb_bsp.h"
#include "tm_stm32f4_delay.h"
#include "core_cm4.h"
//#include "tm_stm32f4_delay.c"  //contiene systick
#include <stdio.h>
#include <stdlib.h>

/* Private typedef -----------------------------------------------------*/
/* Private define ------------------------------------------------------*/
#define CURSOR_STEP     7

extern uint8_t Buffer[6];
/* Private macro -------------------------------------------------------*/
/* Private variables ---------------------------------------------------*/
extern __IO uint8_t DemoEnterCondition;
extern __IO uint8_t UserButtonPressed;

uint32_t TiempoA;
/*-------------------------------------------------------------------*/

extern uint32_t __IO Tiempos[];
extern char event_type[][4];
extern uint32_t next_slot;
extern uint32_t event_count[];

/* Private function prototypes -----------------------------------------*/
extern USB_OTG_CORE_HANDLE          USB_OTG_dev;
extern uint32_t USBD_OTG_ISR_Handler (USB_OTG_CORE_HANDLE *pdev);

void ejecuta_orden(volatile char *linea);
void codificador_hex1(volatile char *linea_hex1, int j);
void codificador_hex2(volatile char *linea_hex2, int jj);
void codificador1(char letra1);
void codificador2(char letra2);

extern uint32_t dms(void);
```

```
//extern void USART_puts(USART_TypeDef *USARTx, volatile int8_t
tecla); //usart


uint8_t tecla1 = 0;
uint8_t tecla2 = 0;


uint8_t ajuste=0; //ajuste de tiempo


 char ultimo_valor_tecla1[3];
 char ultimo_valor_tecla2[3];
 extern int echo;//habilitado por defecto
 char strHEX1[2];
 char strHEX2[2];

/***********************************PULSOMETRO*********************/
 int activar_p=0;
 extern float ppm;
//***********************************FLASH************************
#define FLASH_USER_START_ADDR   ADDR_FLASH_SECTOR_2   /* Start @ of
user Flash area*/         //**
#define FLASH_USER_END_ADDR     ADDR_FLASH_SECTOR_5   /* End @ of user
Flash area*/           //**
/* Base address of the Flash sectors*/
//**
#define ADDR_FLASH_SECTOR_0     ((uint32_t)0x08000000) /* Base @ of
Sector 0, 16 Kbytes*/     //**
#define ADDR_FLASH_SECTOR_1     ((uint32_t)0x08004000) /* Base @ of
Sector 1, 16 Kbytes*/     //**
#define ADDR_FLASH_SECTOR_2     ((uint32_t)0x08008000) /* Base @ of
Sector 2, 16 Kbytes*/     //**
#define ADDR_FLASH_SECTOR_3     ((uint32_t)0x0800C000) /* Base @ of
Sector 3, 16 Kbytes*/     //**
#define ADDR_FLASH_SECTOR_4     ((uint32_t)0x08010000) /* Base @ of
Sector 4, 64 Kbytes*/     //**
#define ADDR_FLASH_SECTOR_5     ((uint32_t)0x08020000) /* Base @ of
Sector 5, 128 Kbytes*/    //**
#define ADDR_FLASH_SECTOR_6     ((uint32_t)0x08040000) /* Base @ of
Sector 6, 128 Kbytes*/    //**
#define ADDR_FLASH_SECTOR_7     ((uint32_t)0x08060000) /* Base @ of
Sector 7, 128 Kbytes*/    //**
#define ADDR_FLASH_SECTOR_8     ((uint32_t)0x08080000) /* Base @ of
Sector 8, 128 Kbytes*/    //**
#define ADDR_FLASH_SECTOR_9     ((uint32_t)0x080A0000) /* Base @ of
Sector 9, 128 Kbytes*/    //**
#define ADDR_FLASH_SECTOR_10    ((uint32_t)0x080C0000) /* Base @ of
Sector 10, 128 Kbytes*/   //**
```

```
#define ADDR_FLASH_SECTOR_11    ((uint32_t)0x080E0000) /* Base @ of
Sector 11, 128 Kbytes*/   //**

//**
uint32_t StartSector = 0, EndSector = 0, Address = 0, i = 0 ;
//**
uint32_t GetSector(uint32_t Address);
//**
//*************************************************************
/*************************************************************/
/*            Cortex-M3 Processor Exceptions Handlers
*/
/*************************************************************/


/**
  * @brief   This function handles NMI exception.
  * @param  None
  * @retval None
  */
void NMI_Handler(void)
{
}


/**
  * @brief  This function handles Hard Fault exception.
  * @param  None
  * @retval None
  */
void HardFault_Handler(void)
{
  /* Go to infinite loop when Hard Fault exception occurs */
  while (1)
  {
  }
}


/**
  * @brief  This function handles Memory Manage exception.
  * @param  None
  * @retval None
  */
void MemManage_Handler(void)
{
```

```
  /* Go to infinite loop when Memory Manage exception occurs */
  while (1)
  {
  }
}

/**
  * @brief  This function handles Bus Fault exception.
  * @param  None
  * @retval None
  */
void BusFault_Handler(void)
{
  /* Go to infinite loop when Bus Fault exception occurs */
  while (1)
  {
  }
}

/**
  * @brief  This function handles Usage Fault exception.
  * @param  None
  * @retval None
  */
void UsageFault_Handler(void)
{
  /* Go to infinite loop when Usage Fault exception occurs */
  while (1)
  {
  }
}

/**
  * @brief  This function handles SVCall exception.
  * @param  None
  * @retval None
  */
void SVC_Handler(void)
{
}

/**
```

```
  * @brief  This function handles Debug Monitor exception.
  * @param  None
  * @retval None
  */
void DebugMon_Handler(void)
{
}

/**
  * @brief  This function handles PendSVC exception.
  * @param  None
  * @retval None
  */
void PendSV_Handler(void)
{
}

#define MAX_WORDLEN 10
volatile char received_str[MAX_WORDLEN + 1];
extern void USART_puts(   volatile char *s);

void USART1_IRQHandler(void){   /// USART2 Interrupt request handler
for ALL (Tx/Rx) USART2 interrupts

  if( USART_GetITStatus(USART1, USART_IT_RXNE)){  // Check the
Interrupt status to ensure the Rx interrupt was triggered, not Tx
      static int cnt = 0;
    // Get the byte that was transferred
    char ch = USART1->DR;

          // Check for "Enter" key, or Maximum characters
       if((ch != '\n') && (cnt < MAX_WORDLEN)){
          received_str[cnt++] = ch;
        }
        else{
          received_str[cnt] = '\0';
          cnt = 0;
          USART_puts(received_str);
        ejecuta_orden(received_str);

        }
  }
```

```
}


void ejecuta_orden(volatile char *linea)
{
  int i=0;
  if(linea[0]=='s'){   //orden setkey
    while(linea[i]!=' ') i++;
    if(linea[i-1]=='1'){  //letra antes del espacio ¿tecla1?
      if((linea[i+1]=='0')&&(linea[i+2]=='x'))
codificador_hex1(linea,i); //hexadecimal?
      else { codificador1(linea[i+1]);}  //enviamos la siguiente letra
despues del espacio

    }else if(linea[i-1]=='2'){ //¿tecla2?
      if((linea[i+1]=='0')&&(linea[i+2]=='x'))
codificador_hex2(linea,i); //hexadecimal
      else {codificador2(linea[i+1]);}  //enviamos la siguiente letra
despues del espacio
    }else{
      USART_puts("orden setkey incorrecta\n introduzca setkeyX donde X
es 1 o 2 \n");
    }
    return;
  }
  if(linea[0]=='g') { //orden getkey
    USART_puts("el valor de la tecla 1 es: ");
    USART_puts(ultimo_valor_tecla1);
    USART_puts("el valor de la tecla 2 es: ");
    USART_puts(ultimo_valor_tecla2);
    USART_puts("\n");
  }
  if(linea[0]=='e') { //orden echo
    while(linea[i]!=' ') i++;
    int echo_anterior=echo;
    echo=49; //para que siempre muestre por pantalla si esta
deshabilitado o habilitado; luego se cambia al correspondiente
    if(linea[i+1]==48){ USART_puts("echo deshabilitado \n"); // el 0
de la consola aquí es 48
        echo=linea[i+1];  }
    else if(linea[i+1]==49){ USART_puts("echo habilitado \n");// El 1
de la consola es 49
        echo=linea[i+1];   }
    else {USART_puts("echo incorrecto \n");
        echo=echo_anterior;}
```

```
  }
    if(linea[0]=='h') { //orden set time  COJE HORA Y ENVIA ACK, el
ajuste lo hace el otro programa
    volatile char nueva_linea[MAX_WORDLEN + 1]="vacia";
    while(linea[i]!=' ') i++;
    for (int f=i+1;f<=MAX_WORDLEN; f++){
      nueva_linea[f-i-1]=linea[f];  // almacena todo lo que hay
despues del espacio (hora en 0.1ms)
    }
    uint32_t hora;
    char str11[15];
    sprintf(str11, "%s", nueva_linea);
    hora = atoi(str11);  // ahora hora es un entero
    /*Enviar ACK*/
    USART_puts("ACK");
    return;
  }
  if(linea[0]=='t'){   //tabla de tiempos 'time table'

     TiempoA=dms();
     char str2[15];
     sprintf(str2, "%d", TiempoA);
     USART_puts(str2);
     USART_puts(" ms/10 \n");

     char str3[15],str4[15],str5[15];
     for(int i=0;i<next_slot;i++){

       sprintf(str3, "%d", Tiempos[i]);  //Pasamos de int32 a string
'int to string';
       sprintf(str5, "%d", event_count[i]);  //Pasamos de int32 a
string;
       USART_puts(str3);
       strcpy(str4, "ms/10 -- ");
       strcat(str4, event_type[i]);
       strcat(str4, str5);
       strcat(str4, "\n");
       USART_puts(str4);
     }

  }
```

```
  if(linea[0]=='w'){ //Guardar datos en FLASH 'write flash'
    STM_EVAL_LEDOn(LED3); STM_EVAL_LEDOn(LED4); STM_EVAL_LEDOn(LED5);
STM_EVAL_LEDOn(LED6);
     Address = FLASH_USER_START_ADDR;
     FLASH_Unlock();  // Unlock the Flash to enable the flash control
register access *************
        /* Erase the user Flash area
                (area defined by FLASH_USER_START_ADDR and
FLASH_USER_END_ADDR) ***********/


      FLASH_ClearFlag(FLASH_FLAG_EOP | FLASH_FLAG_OPERR |
FLASH_FLAG_WRPERR |  /* Clear pending flags (if any) */
                  FLASH_FLAG_PGAERR |
FLASH_FLAG_PGPERR|FLASH_FLAG_PGSERR);


       StartSector = GetSector(FLASH_USER_START_ADDR);  /* Get the
number of the start and end sectors */
       EndSector = GetSector(FLASH_USER_END_ADDR);


      for (i = StartSector; i < EndSector; i += 8)
      {


    // Device voltage range supposed to be [2.7V to 3.6V], the
operation will be done by word
            if (FLASH_EraseSector(i, VoltageRange_3) !=
FLASH_COMPLETE)
            {
              // Error occurred while sector erase. User can add here
some code to deal with this error
              while (1)
              {
              }
             }


      }
       Address = FLASH_USER_START_ADDR;
      uint8_t sstrtoint;
      uint32_t add;
      FLASH_ProgramWord(Address, next_slot); // Guarda Primero el
Ultimo slot
       Address = Address + 4;
       for(int i=0;i<next_slot;i++){
         FLASH_ProgramWord(Address, Tiempos[i]);
          Address = Address + 4;
         FLASH_ProgramWord(Address, event_count[i]);
```

```
          Address = Address + 4;

        // sstrtoint=strtol(event_type[i], NULL, 36);

         //sstrtoint = strtol(event_type[i][0], NULL, 16);
         sstrtoint= (int)event_type[i][0];
         FLASH_ProgramByte(Address, sstrtoint);
          Address = Address + 1;

        sstrtoint= (int)event_type[i][1];
         FLASH_ProgramByte(Address, sstrtoint);
          Address = Address + 1;

         sstrtoint= (int)event_type[i][2];
         FLASH_ProgramByte(Address, sstrtoint);
          Address = Address + 1;

          sstrtoint= (int)event_type[i][3];
         FLASH_ProgramByte(Address, sstrtoint);
          Address = Address + 1;

          add=sizeof(event_type[i]);
       }

      FLASH_Lock();  // Lock the Flash to disable the flash control
register access (recommended to protect the FLASH memory against
possible unwanted operation) ********
      STM_EVAL_LEDOff(LED3); STM_EVAL_LEDOff(LED4);
STM_EVAL_LEDOn(LED5); STM_EVAL_LEDOff(LED6);
  }
  if(linea[0]=='c'){ //Borrar datos en FLASH 'clear flash'


  }
  if(linea[0]=='r'){ //Leer datos en FLASH 'read flash'
      Address = FLASH_USER_START_ADDR;
      next_slot = *(__IO uint32_t*)Address;  //primero leemos
next_slot
      Address = Address + 4;
      //char str90[15];
      //int int90;
     // uint8_t int80;
      for(int i=0;i<next_slot;i++){
          Tiempos[i] = *(__IO uint32_t*)Address;
```

```
           Address = Address + 4;
          event_count[i] = *(__IO uint32_t*)Address;
           Address = Address + 4;
          event_type[i][0] = *(char*)Address;
          //sprintf(event_type[i], "%d", int80);  //Pasamos de int a
string;
          //event_type[i]= str90;
           Address = Address + 1;
           event_type[i][1] = *(char*)Address;
           Address = Address + 1;
           event_type[i][2] = *(char*)Address;
           Address = Address + 1;
           event_type[i][3] = *(char*)Address;
           Address = Address + 1;
      }

            char str3[15],str4[15],str5[15];
     for(int i=0;i<next_slot;i++){

       sprintf(str3, "%d", Tiempos[i]);  //Pasamos de int32 a string
'int to string';
       sprintf(str5, "%d", event_count[i]);  //Pasamos de int32 a
string;
       USART_puts(str3);
       strcpy(str4, "ms/10 -- ");
       strcat(str4, event_type[i]);
       strcat(str4, str5);
       strcat(str4, "\n");
       USART_puts(str4);
     }

  }
  if(linea[0]=='a'){ //activar pulsometro
   activar_p=1;
  }
    if(linea[0]=='p'){ //pulso
      if(activar_p==0){
        USART_puts("Pulsometro inactivo, activar con orden 'a' \n");
      }
      else{
        char str3[15];
        sprintf(str3, "%f", ppm);  //Pasamos de float32 a string 'int
to string';
```

```
        strcat(str3,"ppm");
        strcat(str3, "\n");
        USART_puts(str3);
      }
      }

   return;
}    /******************* CODIFICADORES ******************/
void codificador_hex1(volatile char *linea_hex1, int j){
  strHEX1[0]=NULL; strHEX1[1]=NULL; strHEX2[0]=NULL;
  strHEX1[0]=linea_hex1[j+3];
  strHEX2[0]=linea_hex1[j+4];
      strcat(strHEX1, strHEX2);
      //t = atoi(l);
      tecla1=strtol(strHEX1, NULL, 16);
}

void codificador_hex2(volatile char *linea_hex2, int jj){
  strHEX1[0]=NULL; strHEX1[1]=NULL; strHEX2[0]=NULL;
  strHEX1[0]=linea_hex2[jj+3];
  strHEX2[0]=linea_hex2[jj+4];
      strcat(strHEX1, strHEX2);
      //t = atoi(l);
      tecla2=strtol(strHEX1, NULL, 16);
}
void codificador1(char letra1){
  ultimo_valor_tecla1[0]=letra1;
  ultimo_valor_tecla1[1]='\n';

  if(letra1=='a')tecla1=0x04;
  if(letra1=='b')tecla1=0x05;
  if(letra1=='c')tecla1=0x06;
  if(letra1=='d')tecla1=0x07;
  if(letra1=='e')tecla1=0x08;
  if(letra1=='f')tecla1=0x09;
  if(letra1=='g')tecla1=0x0A;
  if(letra1=='h')tecla1=0x0B;
  if(letra1=='i')tecla1=0x0C;
  if(letra1=='j')tecla1=0x0D;
  if(letra1=='k')tecla1=0x0E;
  if(letra1=='l')tecla1=0x0F;
  if(letra1=='m')tecla1=0x10;
```

```
  if(letra1=='m')tecla1=0x11;
  if(letra1=='o')tecla1=0x12;
  if(letra1=='p')tecla1=0x13;
  if(letra1=='q')tecla1=0x14;
  if(letra1=='r')tecla1=0x15;
  if(letra1=='s')tecla1=0x16;
  if(letra1=='t')tecla1=0x17;
  if(letra1=='u')tecla1=0x18;
  if(letra1=='v')tecla1=0x19;
  if(letra1=='w')tecla1=0x1A;
  if(letra1=='x')tecla1=0x1B;
  if(letra1=='y')tecla1=0x1C;
  if(letra1=='z')tecla1=0x1D;
  if(letra1=='1')tecla1=0x1E;
  if(letra1=='2')tecla1=0x1F;
  if(letra1=='3')tecla1=0x20;
  if(letra1=='4')tecla1=0x21;
  if(letra1=='5')tecla1=0x22;
  if(letra1=='6')tecla1=0x23;
  if(letra1=='7')tecla1=0x24;
  if(letra1=='8')tecla1=0x25;
  if(letra1=='9')tecla1=0x26;
  if(letra1=='0')tecla1=0x27;
  if(letra1==' ')tecla1=0x2C;

  return;
}
void codificador2(char letra2){
  ultimo_valor_tecla2[0]=letra2;
  ultimo_valor_tecla2[1]='\n';

  if(letra2=='a')tecla2=0x04;
  if(letra2=='b')tecla2=0x05;
  if(letra2=='c')tecla2=0x06;
  if(letra2=='d')tecla2=0x07;
  if(letra2=='e')tecla2=0x08;
  if(letra2=='f')tecla2=0x09;
  if(letra2=='g')tecla2=0x0A;
  if(letra2=='h')tecla2=0x0B;
  if(letra2=='i')tecla2=0x0C;
  if(letra2=='j')tecla2=0x0D;
  if(letra2=='k')tecla2=0x0E;
```

```
  if(letra2=='l')tecla2=0x0F;
  if(letra2=='m')tecla2=0x10;
  if(letra2=='m')tecla2=0x11;
  if(letra2=='o')tecla2=0x12;
  if(letra2=='p')tecla2=0x13;
  if(letra2=='q')tecla2=0x14;
  if(letra2=='r')tecla2=0x15;
  if(letra2=='s')tecla2=0x16;
  if(letra2=='t')tecla2=0x17;
  if(letra2=='u')tecla2=0x18;
  if(letra2=='v')tecla2=0x19;
  if(letra2=='w')tecla2=0x1A;
  if(letra2=='x')tecla2=0x1B;
  if(letra2=='y')tecla2=0x1C;
  if(letra2=='z')tecla2=0x1D;
  if(letra2=='1')tecla2=0x1E;
  if(letra2=='2')tecla2=0x1F;
  if(letra2=='3')tecla2=0x20;
  if(letra2=='4')tecla2=0x21;
  if(letra2=='5')tecla2=0x22;
  if(letra2=='6')tecla2=0x23;
  if(letra2=='7')tecla2=0x24;
  if(letra2=='8')tecla2=0x25;
  if(letra2=='9')tecla2=0x26;
  if(letra2=='0')tecla2=0x27;
  if(letra2==' ')tecla2=0x2C;
 // else{tecla2=letra2;}
  return;
}

void EXTI0_IRQHandler(void)
{
  UserButtonPressed = 0x01;

  /* Clear the EXTI line pending bit */
  EXTI_ClearITPendingBit(USER_BUTTON_EXTI_LINE);
}
uint32_t GetSector(uint32_t Address)
//**********flash**************************************************
{
  uint32_t sector = 0;
```

```
  if((Address < ADDR_FLASH_SECTOR_1) && (Address >=
ADDR_FLASH_SECTOR_0))
  {
    sector = FLASH_Sector_0;
  }
  else if((Address < ADDR_FLASH_SECTOR_2) && (Address >=
ADDR_FLASH_SECTOR_1))
  {
    sector = FLASH_Sector_1;
  }
  else if((Address < ADDR_FLASH_SECTOR_3) && (Address >=
ADDR_FLASH_SECTOR_2))
  {
    sector = FLASH_Sector_2;
  }
  else if((Address < ADDR_FLASH_SECTOR_4) && (Address >=
ADDR_FLASH_SECTOR_3))
  {
    sector = FLASH_Sector_3;
  }
  else if((Address < ADDR_FLASH_SECTOR_5) && (Address >=
ADDR_FLASH_SECTOR_4))
  {
    sector = FLASH_Sector_4;
  }
  else if((Address < ADDR_FLASH_SECTOR_6) && (Address >=
ADDR_FLASH_SECTOR_5))
  {
    sector = FLASH_Sector_5;
  }
  else if((Address < ADDR_FLASH_SECTOR_7) && (Address >=
ADDR_FLASH_SECTOR_6))
  {
    sector = FLASH_Sector_6;
  }
  else if((Address < ADDR_FLASH_SECTOR_8) && (Address >=
ADDR_FLASH_SECTOR_7))
  {
    sector = FLASH_Sector_7;
  }
  else if((Address < ADDR_FLASH_SECTOR_9) && (Address >=
ADDR_FLASH_SECTOR_8))
  {
    sector = FLASH_Sector_8;
  }
```

```
  else if((Address < ADDR_FLASH_SECTOR_10) && (Address >=
ADDR_FLASH_SECTOR_9))
  {
    sector = FLASH_Sector_9;
  }
  else if((Address < ADDR_FLASH_SECTOR_11) && (Address >=
ADDR_FLASH_SECTOR_10))
  {
    sector = FLASH_Sector_10;
  }
  else/*(Address < FLASH_END_ADDR) && (Address >=
ADDR_FLASH_SECTOR_11))*/
  {
    sector = FLASH_Sector_11;
  }

  return sector;
}
```