

PRÁCTICA 6

Índice

1. Objetivos.
2. Introducción.
3. Desarrollo de la práctica.
4. Conclusión.

1. Objetivos

- FASE 1: Realizar un ratón USB cuya interfaz de usuario sean dos ejes del acelerómetro de la placa y el botón USER de la placa, modificar en el mismo diversos aspectos para entender su funcionamiento: cambiar descriptores, cambiar los datos que se mandan en el report.
- FASE 2: Cambiar el proyecto anterior para que sea un dispositivo de clase HID Custom. Probar su funcionamiento con el código del PC que se suministra en la práctica.
- FASE 3: Dispositivo HID Custom de salida, realizar un sistema que mande desde el PC caracteres a la placa y estos se muestren en el LCD. Controlar los LEDs de la placa desde el PC.
- FASE 4: Dispositivo HID Custom de salida, realizar un sistema que mande desde el PC la posición deseada del servo y éste se mueva a dicha posición (y si es posible mostrar la posición en el LCD).
- FASE 5: Dispositivo Custom de entrada, mandar los datos del acelerómetro desde la placa al PC y que se muestren por pantalla.
- FASE 6: Dispositivo Custom de entrada, mandar los datos de temperatura y humedad desde la placa al PC.

2. Introducción

Los puertos serie son particularmente importantes en los microcontroladores, permiten con un número escaso de líneas una conectividad muy variada. Encontramos tres tipos de puerto serie:

- Aquellos que fundamentalmente se utilizan para ampliar las capacidades del microcontrolador, normalmente conectando periféricos o memoria a éste, por ejemplo, los puertos SPI e I2C.

- Los que se utilizan sobre todo para conectar el microcontrolador a un computador de propósito general, en este caso el microcontrolador suele ser periférico de este último. Puertos de este tipo son el RS232 y el USB.

- Los que se utilizan para hacer redes de microcontroladores, por ejemplo CAN.

No cabe duda de que esta clasificación no es cerrada, ya que podemos conectar microcontroladores entre sí para formar redes mediante cualquiera de estos mecanismos serie y a su vez conectar estos a un computador de propósito general, todo depende desde qué punto de vista se analicen los sistemas.

En esta práctica se pretende ilustrar como se puede conectar un microcontrolador como un periférico de un PC mediante USB. Con este tipo de conexión obtenemos por una parte una expansión hardware relativamente fácil en los PC actuales y, por otra, las ventajas de las aplicaciones distribuidas en cuanto a flexibilidad. En el caso de que queramos controlar elementos externos mediante un PC podemos recurrir a una conexión por el puerto paralelo/lpt (antiguo de impresora), bus PCI, PCIe, ISA, RS232 o USB. Hay otras posibilidades algo menos clásicas pero muy habituales: wifi, bluetooth, ethernet, pcmcia, infrarrojos, ide/sata, tarjeta de sonido, etc. Obsérvese que entre el puerto paralelo/lpt y el RS232 (serie asíncrono) hay una clara diferencia. Por el puerto paralelo/lpt del PC se podría “atacar” a los dispositivos de entrada/salida directamente, con esto queremos indicar que podríamos colocar “interruptores y LEDs” y controlar su estado desde el PC de forma simple, en realidad la vieja conexión de impresora se parece mucho a los GPIOs de los microcontroladores. Sin embargo, el puerto serie asíncrono no tiene esa facilidad, necesitaría una transformación para llevar a cabo dicho control, casi estamos obligados a colocar un hardware en medio. Este mismo problema, pero a mayor escala, lo podemos encontrar en otras conexiones muy habituales, como son: USB, ethernet, wifi... necesitamos un elemento “inteligente” que interprete y transforme las señales serie (más o menos complejas dependiendo del protocolo), normalmente ese elemento “inteligente” es un microcontrolador.

Caben destacar algunas diferencias de esta práctica con respecto a la anterior (la de USB Audio):

- En esta práctica se va a utilizar el entorno de desarrollo local STM32CubeIDE. La metodología que vamos a emplear, en este caso, es la de generar un proyecto base con la configuración oportuna a partir de las librerías que suministra la propia herramienta de configuración. Os recuerdo que con MBED se utilizó en la práctica anterior una serie de “proyectos plantilla”, a partir de los cuales realizamos “nuestros proyectos” introduciendo los cambios oportunos, la dificultad de esta metodología estriba en descubrir qué hace y dónde lo hace el “proyecto plantilla”, es un problema de análisis. En el caso de generar un proyecto base a partir de la herramienta de configuración tenemos más información, pues suele venir mejor documentado (otra cosa es que entendamos la documentación).

- En la práctica anterior se ilustraba cómo funcionaban los dispositivos USB de Audio, sin entrar en otras utilidades que no fuera la de transferir o manipular el sonido. Aquí se

va a hacer lo mismo con los USB de clase HID, pero además se pretende mostrar cómo puede realizarse una interface USB que sirva para cualquier periférico propio que se pretenda construir. Es decir, se aprovecha la versatilidad de la clase HID para construir dispositivos que podría no considerarse de “interface humana”. En general estos dispositivos USB de clase HID que son “otra cosa” se denominan dispositivos

HID Custom. Un ejemplo de este tipo de dispositivos lo podemos ver en los robots que se encuentran en el laboratorio en frente del de prácticas, en ese laboratorio se diseñan robots de análisis hospitalario que usan como interface con el PC una conexión USB de clase HID tipo Custom.

En definitiva, el objetivo académico de esta práctica no es sólo mostrar el funcionamiento de los dispositivos USB HID, si no que además el alumno aprenda a usar ese tipo de interface para poder crear una conexión entre el PC y el microcontrolador, que sirva para manejar cualquier tipo de periférico que se pretenda diseñar con dicho microcontrolador. En resumen, dos objetivos académicos:

- Ilustrar el funcionamiento de USB HID.
- Usar la interface USB HID como conexión general entre el PC y el microcontrolador.

Esta práctica trata sobre una tecnología muy popular que encontramos en todos los equipos informáticos actuales, el USB, pero sorprendentemente es muy poco conocido el mecanismo que permite a nivel de desarrollador hacer aplicaciones que usen dicha tecnología. Desde luego que en internet se encuentra muy buena información sobre todo lo que estamos viendo en esta asignatura. Pero la excepción es el caso del desarrollo con USB, lo que viene en la web casi siempre es escaso y confuso, lo mismo pasa con las especificaciones USB publicadas (a pesar de su abultado volumen), por ello aconsejo que quienes quieran profundizar en este tema acudan a unos de los mejores libros sobre la materia: USB Complete. The Developer´s guide. Jack Axelson (2015).

Desarrollo de dispositivos USB

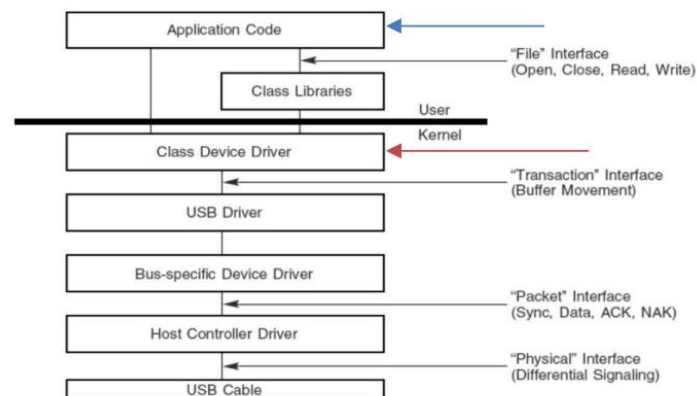
Nos vamos a centrar en el uso del microcontrolador como dispositivo (Device), no vamos a tratar al microcontrolador como Host, por otra parte es la situación más normal, pues casi siempre el Host es una máquina tipo PC, Tablet... con cualquier sistema operativo. Vamos a trabajar con el caso más sencillo, un periférico con una sola interface. A pesar de todo, realizar un dispositivo USB implica, en la mayoría de los casos, tres ámbitos de programación diferentes:

1 - El proyecto para un microcontrolador (u otro hardware, podría ser una FPGA). Esta parte del microcontrolador es la que más estudiaremos. Nos referimos por tanto al firmware que debe ejecutar el microcontrolador para ser el Device USB que pretendemos desarrollar.

2 - Para el PC necesitaríamos el driver de clase correspondiente, que se ubicaría en el Kernel del sistema operativo en cuestión. En el caso del Windows estamos hablando del módulo señalado con la flecha roja en la siguiente figura (la cual representa los niveles

dentro de un PC con sistema Windows para acceder a los dispositivos USB). Sabemos que ese driver puede que tengamos que desarrollarlo nosotros o que ya esté en el sistema operativo, dependiendo de la clase a la que pertenezca el Device que hemos programado en el punto 1.-. Por ejemplo, si hiciéramos un ratón no necesitamos el driver de clase, pues el ratón pertenece a una clase estándar llamada HID y prácticamente todos los sistemas operativos incluye el driver para dichos dispositivos. Ya hemos tratado el concepto de “clase de dispositivo”, y también hemos clasificado los dispositivos USB en dos tipos: aquellos en los que es necesario que proporcionemos el driver de clase y en los que no es necesario, pues ya los tiene incluido el sistema operativo. Programar un driver de clase propio no es difícil, sólo hay que tener un poco más de conocimiento y contar con las herramientas adecuadas. Pero programar en modo Kernel tiene más implicaciones que hacerlo en modo usuario (la primera de todas es que un error de programación resulta en la caída completa del sistema operativo... pantalla azul y volcado de memoria en Windows). Esto que hemos explicado es prácticamente igual independientemente de si el PC tiene Windows o Linux. Además, en Windows tenemos el problema de la certificación del driver, si construimos uno y no lo certifica Microsoft generamos desconfianza en el cliente. Esta parte de la programación no la vamos a tratar apenas por falta de tiempo y medios.

3 - Programa de aplicación o librería para el PC que permita conectarse con el Device mediante el driver del apartado anterior (2.-). Es el señalado con la flecha azul en la siguiente figura. Por ejemplo, en la práctica anterior, en la fase correspondiente al USB MIDI, el piano VMPK era precisamente la aplicación a la que nos referimos en este dibujo.



En resumen, tenemos la implicación de tres programas diferentes para realizar un diseño con USB: el firmware del microcontrolador, el driver en el PC y la aplicación en el PC.

El USB desde el punto de vista del microcontrolador

Ya sea partiendo de un proyecto ejemplo o con código generado, como es nuestro caso, para desarrollar el firmware que maneje el USB en el microcontrolador se precisa conocer e identificar dos puntos claves dentro del código del proyecto base: las funciones que escriben/leen de los buffers del USB (para nosotros equivalentes a

EndPoint, en adelante EP) y los descriptores. Ambos aspectos son muy importantes, pues un fallo en los descriptores implica el rechazo del Host a reconocer el Device, y por tanto imposibilita la conexión, y el exceder los límites de los buffers indicado por los descriptores puede suponer una caída del sistema operativo del Host (pantalla azul y volcado de memoria en Windows).

Otro aspecto importante que lleva a bastante confusión al desarrollar el firmware de USB es la denominación de los EP, si son salida o entrada, en general siempre se le denomina con respecto a lo que hace en el Host, un EP1IN será el endpoint 1 y es entrada para el Host (por tanto, salida para el microcontrolador). Lo que significa que el microcontrolador escribirá en el EP1IN... justo lo contrario se podría razonar para un EP1OUT. Esto es una excepción y sólo por su denominación según el convenio USB. Recordar que, como norma general en los periféricos, ya sea en el Host o en el microcontrolador, en las salidas se escribe y de las entradas se lee.

Las funciones que escriben y leen de los endpoints (buffers) pueden diferir dependiendo del microcontrolador, del sistema de desarrollo o del ejemplo que elijamos, pero todas se caracterizan por disponer de un control de flujo, de forma que podemos saber cuándo llega dato o el dato que se ha mandado ya ha sido recogido por el PC, en esas funciones se le suele pasar como parámetros: el EP de destino, un puntero a un array y el número de bytes que se pretende leer o escribir. En el array señalado por el puntero se mandan los datos o se recogen los datos leídos.

Ya hemos mencionado que los descriptores se ubican dentro del microcontrolador, y son unas estructuras de datos, compuestas por números, con información sobre el periférico que es requerida por el Host para saber cómo configurar el dispositivo. Los descriptores suelen venir representados en el programa del microcontrolador por una o varias variables/constantes tipo array, en esa ristra de números está la clave de que el host reconozca al periférico o no. Cuando introducimos en el PC una “llave de memoria USB”, el PC sabe que es eso, y no un ratón, porque el mismo dispositivo que acabamos de conectar se lo cuenta con sus descriptores.

Los descriptores se dividen en dos tipos:

a) Los descriptores estándar, los tienen todos los dispositivos USB, son cinco:

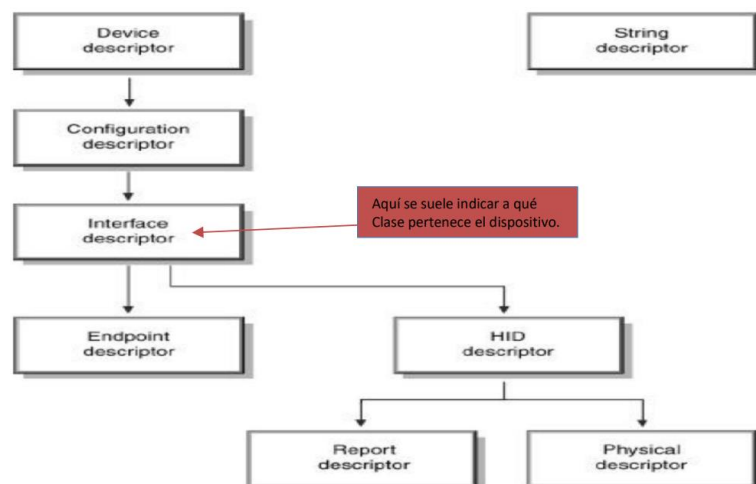
- Descriptor Device, describe al dispositivo completo.
- Descriptor Configuration, describe una configuración junto a los descriptores de interface y endpoint. Por tanto, una configuración incluye los descriptores de Configuration, Interfaces y Endpoints
- Descriptores Interface, la interface precisamente es la que se asocia al driver del kernel del host, en el descriptor de interface se indica a qué clase pertenece el dispositivo y, por tanto, qué tipo de driver en el Host le corresponde.
- Descriptores Endpoint, describe a los EP, excepto el EP0, que no precisa de descriptor y que siempre tiene que existir por defecto. Si tenemos un EP1IN y otro EP1OUT, cada uno tendrá su propio descriptor de EP, por eso se dice en muchas

ocasiones que como máximo en el USB hay 32 EP, 16 tipo salida y 16 tipo entrada. Mientras más EP tenga un dispositivo USB más necesidad de recursos hardware precisa el microcontrolador (buffers de doble puerto...).

- Descriptores de String, son los gran olvidados, su única misión es informar a los usuarios, contienen descripciones del dispositivo en lenguaje natural (humano), los veremos un poco sólo.

b) Los descriptores propios de cada clase, estos descriptores dependen de la clase del dispositivo en cuestión, por ejemplo, en los dispositivos de clase HID hay dos descriptores obligatorios y uno opcional: HID y Report Descriptor son obligatorios, y el Physical Descriptor es opcional.

Los descriptores forman una estructura jerarquizada como se muestra en la figura siguiente, dependiendo los inferiores de los que están por encima. Por ejemplo, en el descriptor de interface se dice cuántos descriptores de EP hay debajo suya. Otro ejemplo, en el descriptor de interface se indica la clase de dispositivo y, por tanto, si fuera de clase HID debe haber descriptor de HID, como el caso de la figura siguiente; el descriptor de HID se dice que cuelga del descriptor de Interface.



Conviene tener “a mano” las transparencias dónde se explican los descriptores para ir comparando las tablas que allí encontramos con los arrays que vamos a ver en el código del microcontrolador en esta práctica. Por ahora vamos a destacar sólo dos campos de toda la información que proporcionan: 1 - La clase de dispositivo (siguiente imagen) y 2 - los ID-Vendor e ID-Product.



Class	Usage	Description	Examples, or exception
00h	Device	Unspecified ^[47]	Device class is unspecified, interface descriptors are used to determine needed drivers
01h	Interface	Audio	Speaker, microphone, sound card, MIDI
02h	Both	Communications and CDC Control	Modem, Ethernet adapter, Wi-Fi adapter, RS-232 serial adapter. Used together with class 0Ah (CDC-Data, below)
03h	Interface	Human interface device (HID)	Keyboard, mouse, joystick
05h	Interface	Physical Interface Device (PID)	Force feedback joystick
06h	Interface	Image (PTP/MTP)	Webcam, scanner
07h	Interface	Printer	Laser printer, inkjet printer, CNC machine
08h	Interface	Mass storage (MSC or UMS)	USB flash drive, memory card reader, digital audio player, digital camera, external drive
09h	Device	USB hub	Full bandwidth hub
0Ah	Interface	CDC-Data	Used together with class 02h (Communications and CDC Control, above)
0Bh	Interface	Smart Card	USB smart card reader
0Dh	Interface	Content security	Fingerprint reader
0Eh	Interface	Video	Webcam
0Fh	Interface	Personal healthcare device class (PHDC)	Pulse monitor (watch)
10h	Interface	Audio/Video (AV)	Webcam, TV
11h	Device	Billboard	Describes USB-C alternate modes supported by device
DCh	Both	Diagnostic Device	USB compliance testing device
E0h	Interface	Wireless Controller	Bluetooth adapter, Microsoft RNDIS
EFh	Both	Miscellaneous	ActiveSync device
FEh	Interface	Application-specific	IrDA Bridge, Test & Measurement Class (USBTMC), ^[48] USB DFU (Device Firmware Upgrade) ^[49]
FFh	Both	Vendor-specific	Indicates that a device needs vendor-specific drivers

1 -La clase de dispositivo (tabla imagen anterior), este dato es clave para que nos funcione todo correctamente, hay clases que se indican en el descriptor de Device, pero la mayoría de las clases se indican en un campo del descriptor de Interface (ver columna flecha roja imagen anterior). Cada clase tiene un número asociado, por ejemplo, la clase HID es 3, y la clase 0xff es la específica de vendedor (esa es la clase que si se la ponemos a nuestro microcontrolador precisa que le hagamos el driver al sistema operativo).

2 - El ID de vendedor y el ID de producto (VID-PID), estos dos números son fundamentales para asociar el dispositivo con el driver del SO y con la aplicación del PC. En el caso de la clase específica del vendedor esos dos números están presentes en la instalación del driver, cuando el microcontrolador se los proporciona al PC, el sistema operativo pedirá el driver para ese dispositivo, y los números del driver deben coincidir con los ID Vendor – ID Product que haya indicado el descriptor de Device del microcontrolador, si no coincidieran el SO rechaza el driver y nos sacaría el mensaje “este driver no es adecuado para este dispositivo”. Como es natural en los dispositivos de clase standard el driver que se asocia no depende de los ID Vendor – ID Product, depende sólo de la clase, por lo que la importancia de esta pareja de identificadores es menor en los dispositivos de clase estándar. Pero también juegan su papel, por ejemplo, si tenemos varios dispositivos HID conectados, y por tanto varios drivers que dan servicio a cada uno de ellos, nos serviría para distinguirlos entre sí, cosa que haremos en esta práctica de forma manual. Otro aspecto fundamental de esa pareja de identificadores en los dispositivos de clase estándar es el papel que juega en el registro de Windows. Pongamos otro ejemplo: imaginemos que hago un firmware que es un

ratón, su descriptor de Device tendrá unos ID determinados, cuando “enchufemos” ese ratón al PC queda en el registro de ese PC que ese dispositivo, con esos ID Vendor-ID Product, es un ratón. Imaginemos que cambiamos algo en su descriptor y lo convertimos en un joystick, y lo volvemos a enchufar, en ese PC ha quedado registrado que esos IDs son los de un ratón, y considera que ese cambio de comportamiento se debe a un error, por lo que a pesar de que lo hemos cambiado de ratón a joystick el PC lo considerará como ratón, pero raro. Esto ocurre así en Windows, pero no en todos los sistemas operativos (en Android no). Por tanto, si queremos cambiar su “personalidad”, y usarlo en el mismo PC, hay que cambiar además el valor de los ID Vendor-ID Product que hemos comentado, basta con cambiar una cifra (si lo fuéramos a usar en otro PC no importaría, pues desde el principio se registraría como joystick). Esta pareja de ID es en principio única para cada periférico, y los fabricantes de dispositivos (ratones, llaves USB, robots...) deben pagar por registrarlas a su nombre como si de la IP de las redes se tratara. Claro que para hacer pruebas en nuestro PC podemos poner la que queramos, de todas formas, el ID Vendor registrado por STMicroelectronics para pruebas con sus microcontroladores es 0x0483 (1155), y el de Microchip es 0x04D8 (estos ejemplos son para pruebas no para ventas de productos). A partir de ahora le llamaremos VID y PID.

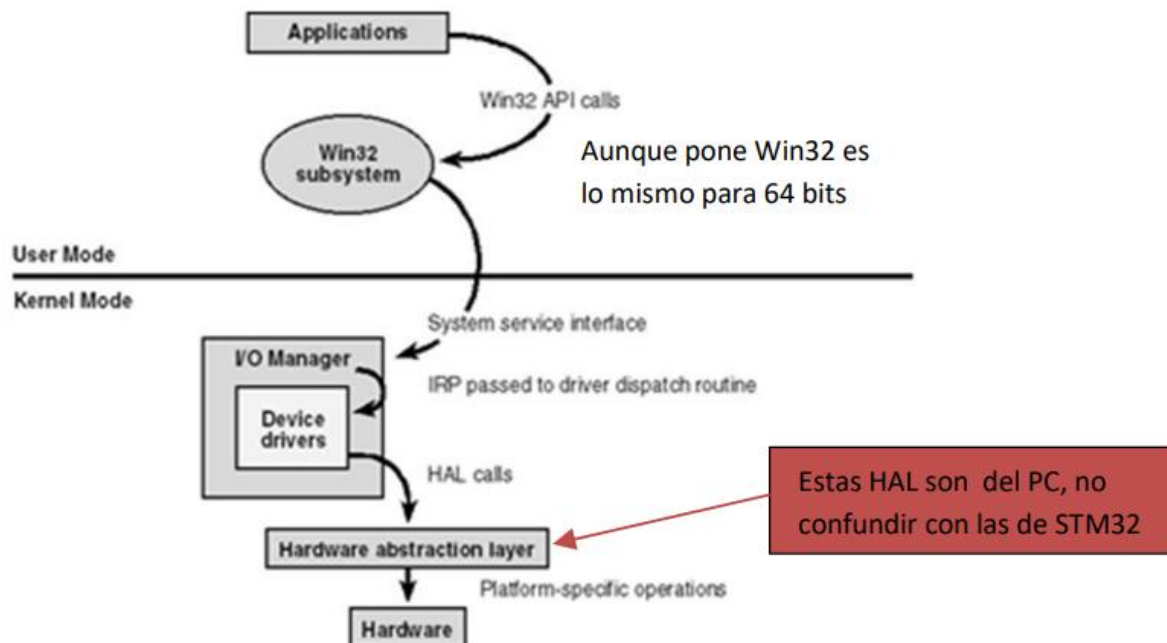
El USB desde el punto de vista del PC

A la hora de acceder a los puertos y dispositivos del PC bajo Windows se pueden utilizar diferentes mecanismos. No es objetivo de esta práctica explicar cómo se desarrollan drivers bajo Windows, sólo se va a mostrar aquellos aspectos básicos que nos ayuden a realizar la práctica.

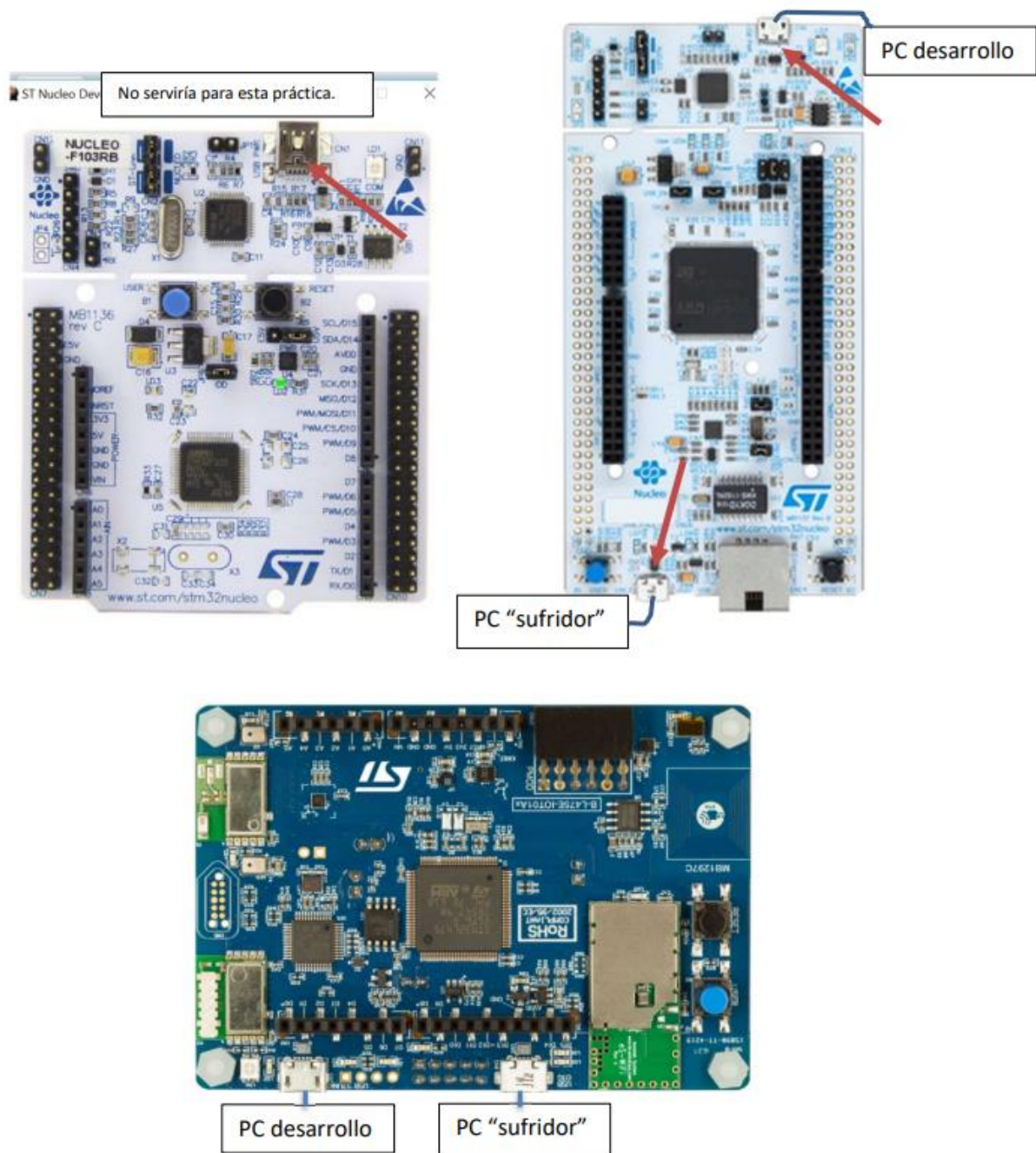
El sistema operativo en las máquinas de propósito general se puede considerar dividido en múltiples niveles, sin embargo, para nuestros objetivos basta tener en cuenta sólo dos: kernel y usuario. Sólo el núcleo del sistema (kernel) puede realizar las operaciones de entrada/salida, en él se debe localizar el manejador (driver) que permita a las aplicaciones utilizar los dispositivos o puertos. Las aplicaciones se pueden comunicar con los drivers (modo Kernel) de diferentes maneras, pero hay una que es básica y se utiliza prácticamente en todos los sistemas operativos (incluido UNIX, norma ANSI), es mediante las funciones de la API para el acceso a ficheros: CreateFile, ReadFile y WriteFile. En esta práctica se va a utilizar dicho mecanismo. Como ya se ha indicado antes, en el caso de los dispositivos HID, el driver está ya en el kernel del Windows, basta utilizar estas funciones de acceso a ficheros, como se muestra en el programa ejemplo que se acompaña a esta práctica (HID_HOST_PC), para poder hacer transferencias entre el microcontrolador y el Host.

En resumen, en nuestro caso, el driver de clase será “interpelado” desde la aplicación mediante las llamadas a la API de manejo de ficheros (CreateFile, ReadFile, etc.). Una forma simple de ver un driver es como un conjunto de subrutinas o funciones (parecido a una dll), cada vez que la aplicación “llama” al driver realmente se lanza una (o varias) subrutina de éste. Lo que realmente ocurre es que se construye una estructura denominada IRP (I/O Request Packet) que se le pasa a la función correspondiente del driver, funciona como una llamada a una subrutina, pero con las debidas protecciones

entre el modo Kernel y Usuario. En la siguiente figura se muestra esquemáticamente el proceso de llamada al driver y cómo éste es el que accede al hardware que corresponda a través del HAL (en nuestro caso USB Driver según la primera figura de este boletín de prácticas).



Estas prácticas de USB se han hecho siempre en el laboratorio de SETR1 con dos ordenadores, uno de ellos servía para desarrollar el firmware del microcontrolador, en él se editaba el código, y a él estaba conectado mediante USB el depurador del microcontrolador. El otro ordenador era el de pruebas ("sufridor o sparring") que tenía conectado el cable USB del microcontrolador al PC, de forma que cuando se ejecuta un código molesto, como el ratón circular o las pantallas azules de caída del sistema operativo, era el PC "sufridor" quien lo padecía. Como es natural esta práctica sólo se podría realizar en cualquier placa real con dos conectores USB, el de depuración y el del microcontrolador. Las placas de microcontrolador con un cable USB tienen sólo el de depuración/grabación, con lo cual el microcontrolador no puede hacer las veces de dispositivo USB.



Los REPORT en los dispositivos de clase HID

A los datos que transmite los dispositivos de clase HID se les llaman Report y el descriptor que muestra cómo son esos datos sería Report Descriptor, fijaros en la necesidad de indicar cómo van los datos y qué tipo de datos se transfieren entre el Host y el dispositivo HID, pues hay una gran variedad de este tipo de dispositivos y cada cual puede funcionar de forma diferente. Vamos ahora a analizar por encima un descriptor de report, en la siguiente imagen se muestra un ejemplo.

Es necesario hacer notar que los comentarios que vienen en este descriptor, como en los vistos anteriormente, son meramente informativos para el programador, no tienen

consecuencia ninguna sobre el funcionamiento, lo importante son los números. Pero estos comentarios son especialmente útiles para enterarnos someramente qué nos quiere contar ese descriptor de Report.

Cuando analizamos un descriptor de Report lo primero es fijarse a donde apuntan el par de flechas verdes, en este caso nos está diciendo que es un HID genérico, de los que todos conocemos: ratones, joystick, gamepad, simuladores de juegos, teclados, etc. , pero además el elemento genérico es un ratón. Las flechas azules indican la conexión con el Host, nos dicen que son Input las tres, es decir entradas para el Host (salidas para el microcontrolador), las tres Inputs son:

- a) 3 botones,
- b) otros 5 bits de relleno para formar junto con los tres botones el primer byte,
- c) y 2 bytes dedesplazamiento X-Y.

Observad que las flechas naranjas nos señalan precisamente qué elementos son: Buttons, X e Y. Por último, las flechas rojas señalan los campos que informan sobre el número y tamaño de datos: 3 botones de tamaño 1 bit, un relleno de 5 bits y dos desplazamientos X-Y de tamaño 8 bits. Sin señalar con flecha quedan otras indicaciones del report descriptor: como que ese desplazamiento es con signo y puede ser de -127 a 127, o que los botones valen 0 ó 1.

```

46  const struct(uint8_t report[HID_RPT01_SIZE]);hid_rpt01=
47  {
48      {
49          0x05, 0x01, /* Usage Page (Generic Desktop) */
50          0x09, 0x02, /* Usage (Mouse) */
51          0xA1, 0x01, /* Collection (Application) */
52          0x09, 0x01, /* Usage (Pointer) */
53          0xA1, 0x00, /* Collection (Physical) */
54          0x05, 0x09, /* Usage Page (Buttons) */
55          0x19, 0x01, /* Usage Minimum (01) */
56          0x29, 0x03, /* Usage Maximum (03) */
57          0x15, 0x00, /* Logical Minimum (0) */
58          0x25, 0x01, /* Logical Maximum (1) */
59          0x95, 0x03, /* Report Count (3) */
60          0x75, 0x01, /* Report Size (1) */
61          0x81, 0x02, /* Input (Data, Variable, Absolute) */
62          0x95, 0x01, /* Report Count (1) */
63          0x75, 0x05, /* Report Size (5) */
64          0x81, 0x01, /* Input (Constant) */
65          0x05, 0x01, /* Usage Page (Generic Desktop) */
66          0x09, 0x30, /* Usage (X) */
67          0x09, 0x31, /* Usage (Y) */
68          0x15, 0x81, /* Logical Minimum (-127) */
69          0x25, 0x7F, /* Logical Maximum (127) */
70          0x75, 0x08, /* Report Size (8) */
71          0x95, 0x02, /* Report Count (2) */
72          0x81, 0x06, /* Input (Data, Variable, Relative) */
73          0xC0, 0xC0 /* End Collection,End Collection */
74      }
75  };

```

Por tanto, este descriptor de report está diciendo que el report tiene la siguiente estructura:

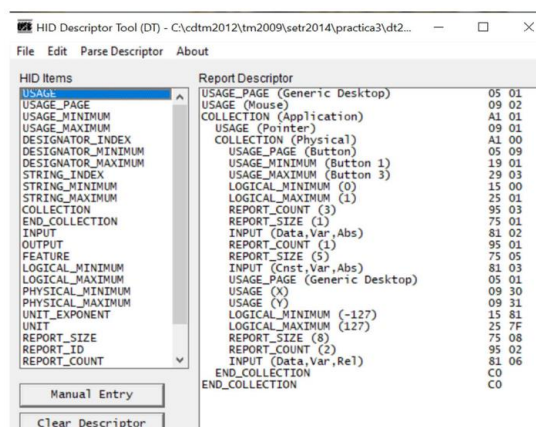
Byte 0	Byte 1			Byte 2	Byte 3
Report ID	0	1	2	Desplazamiento Eje X	Desplazamiento Eje Y

Hemos añadido un Report ID, que no vamos a tratar aquí, en nuestro caso siempre vale cero, y nos vamos a olvidar de él hasta que lo leamos en el PC. Pero observad que este ratón no manda los datos de la misma forma que el descrito en las transparencias sobre

USB (alrededor de la transparencia número 63), aquel manda primero el desplazamiento X, después el desplazamiento Y y por último los botones, y éste, sin embargo manda primero los botones, después el desplazamiento X y por último el desplazamiento Y... por eso la necesidad de describir los Report... para dar flexibilidad a los dispositivos HID.

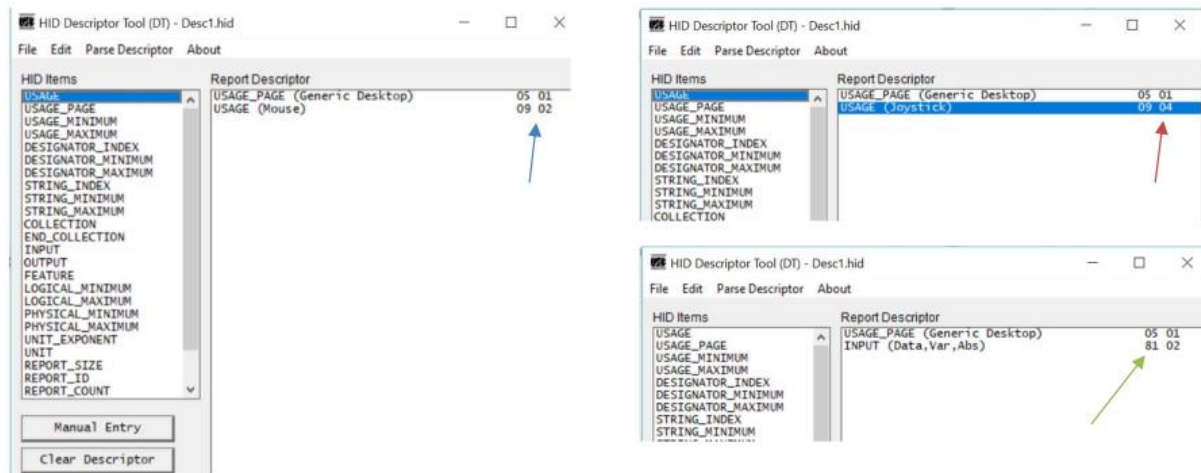
Pero la complejidad de los report descriptor es tremenda, con ellos se podría describir los elementos de control de un avión completo, y el PC los tendría en cuenta como tales... a propósito, ¿podría hacerse pasar una plaquita de estas por un simulador completo de elementos tan complejos como aviones, coches, tanques, etc? Pues claro que sí, observad que la mayoría de dispositivos HID constan de una serie de elementos que sirve de interface con el humano: botones, potenciómetros, cámaras... y otros muchos sensores, más un sistema que organice los datos recolectados y los mande al Host con la estructura adecuada. Evidentemente dicho sistema es el microcontrolador, que además previamente también se encarga de describir qué son esos datos y para qué sirven. Un ejemplo es el ratón que hacemos en la primera fase de esta práctica, en realidad no tiene nada de ratón, pero el Host se lo cree porque se lo cuenta el report descriptor y los datos que manda son coherentes con ser un ratón.

El problema de los descriptores de report es su complejidad, pero consciente de ello la propia organización encargada de la estandarización del USB tiene una aplicación gratuita para generar el report descriptor, no es fácil de manejar, pero permite solucionar muchos problemas a los diseñadores que pretenden hacer dispositivos HID complejos, hay más herramientas para hacer otros tipos de descriptores, la mayoría gratuitas y bastante mediocres. Para hacer la práctica no se precisa la herramienta que genera los report descriptor, pero si estáis interesados se puede descargar de este enlace: <https://www.usb.org/document-library/hid-descriptor-tool> , aparece la descarga en la esquina superior derecha de la web.



En las siguientes imágenes se muestra cómo cambian los valores del descriptor de report si permutamos mouse por joystick, o si ponemos que la salida en vez de relativa

es absoluta (pasamos de ratón a tableta digitalizadora).



Material necesario para esta práctica

Junto al boletín de esta práctica se incluye un programa para el PC Host. Este último es un proyecto para Visual Studio 2019, también funciona con versiones anteriores que ya tengáis instaladas, si os da algún problema vuestra versión antigua de Visual Studio mandadme un email e intentamos solucionarlo. Por regla general si la versión de Visual Studio que tenéis instalada es muy vieja va a requerir instalar el WDK o DDK para compilar este programa, pero eso sólo pasa en algunos casos. El programa del PC se ha hecho lo más reducido posible, en un solo fichero y con las mínimas llamadas al sistema, todo ello con objeto de que nos centremos en los aspectos más básicos que requiere una aplicación de PC para leer de un driver USB. No es por tanto un espectacular programa con muchas “ventanitas”, es una aplicación de consola simple.

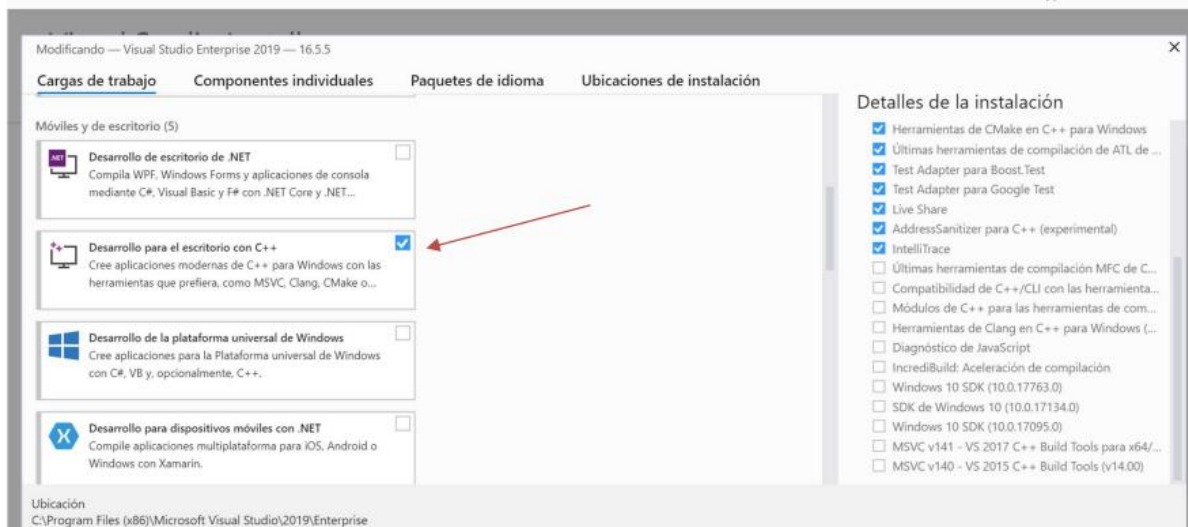
La práctica se realizará íntegra sobre un Pc con Windows, y necesitaremos:

- Las transparencias sobre USB y resto de documentación incluida en la práctica.
- El STM32CubeIDE.
- El hardware: placa microcontrolador, placa display, servo y cables.
- El código del programa ejemplo de aplicación para el PC que se incluye en esta práctica.
- Las librerías del LCD y recordar el código realizado en las prácticas 2 y 4.
- Microsoft Visual Studio para editar y ejecutar dicho programa de aplicación del PC. Si no tenéis instalado ninguna versión de Visual Studio, podéis descargaros de la siguiente página web el Visual Studio Enterprise/Professional 2019, que es la versión que yo he probado:

<https://docs.microsoft.com/es-es/windows-hardware/drivers/download-the-wdk>

Cuando lo instaléis sólo es necesario seleccionar en su instalador la opción “Desarrollo para escritorio con C++” (como se muestra en la figura siguiente, nada más), la

instalación entonces ocupa unos 7 Gb; claro que si queréis otras herramientas podéis seleccionar todo lo que os quepa en vuestros discos.



3. Desarrollo de la práctica

- En la fase 1, desarrollaremos un ratón USB utilizando dos ejes del acelerómetro de la placa como interfaz de usuario. Para empezar, crearemos un nuevo proyecto STM32 seleccionando la placa y siguiendo los mismos pasos que en prácticas anteriores.

Para que el proyecto incluya las librerías de USB de clase HID, primero debemos configurar la conexión USB. Seguiremos esta ruta: “Connectivity” -> “USB_OTG_FS” -> “Mode” -> “Device_Only”. En la misma pestaña de “USB_OTG_FS” vamos a “NVIC Settings” -> “USB OTG FS global interrupt” -> “Enable” -> ✓.

Luego seleccionamos “Middleware” -> “USB_Device” -> “Class for FS IP” -> “Custom Human Interface Device Class (HID)”, y en la pestaña de “USB_Device” -> “Parameter Settings” -> “USBD_CUSTOM_HID_REPORT...” lo ajustamos a 50, y en la misma pestaña, en “Parameter Settings” -> “USBD_CUSTOMHID...” lo ajustamos a 4.

Una vez completados estos pasos, generamos el código. Después de generarlo, podemos consultar una guía resumida en las prácticas del código generado.

A continuación, iremos a “usbd_custom_hid_if.c” y modificaremos el código dentro de la función “CUSTOM_HID_ReportDesc_FS[USBD_CUSTOM_HID_REPORT_DESC_SIZE]” dejándolo así:

```
0x05, 0x01, // USAGE_PAGE (Generic Desktop)
```

```
0x09, 0x02, // USAGE (Mouse)
```

```
0xa1, 0x01, // COLLECTION (Application)
```

```
0x09, 0x01, // USAGE (Pointer)
```

```
0xa1, 0x00, // COLLECTION (Physical)
```

```
0x05, 0x09, // USAGE_PAGE (Button)
```

```

0x19, 0x01, // USAGE_MINIMUM (Button 1)
0x29, 0x03, // USAGE_MAXIMUM (Button 3)
0x15, 0x00, // LOGICAL_MINIMUM (0)
0x25, 0x01, // LOGICAL_MAXIMUM (1)
0x95, 0x03, // REPORT_COUNT (3)
0x75, 0x01, // REPORT_SIZE (1)
0x81, 0x02, // INPUT (Data,Var,Abs)
0x95, 0x01, // REPORT_COUNT (1)
0x75, 0x05, // REPORT_SIZE (5)
0x81, 0x03, // INPUT (Cnst,Var,Abs)
0x05, 0x01, // USAGE_PAGE (Generic Desktop)
0x09, 0x30, // USAGE (X)
0x09, 0x31, // USAGE (Y)
0x15, 0x81, // LOGICAL_MINIMUM (-127)
0x25, 0x7f, // LOGICAL_MAXIMUM (127)
0x75, 0x08, // REPORT_SIZE (8)
0x95, 0x02, // REPORT_COUNT (2)
0x81, 0x06, // INPUT (Data,Var,Rel)
0xc0, // END_COLLECTION
0xc0 // END_COLLECTION

```

Luego, vamos al “main.c” y añadimos en el main() el siguiente código en las distintas secciones:

```

/* USER CODE BEGIN 1 */:

```

```

int i; //declaramos las variables que vamos a usar en nuestra función
uint8_t dato[3]; // esta es la variable donde vamos a mandar el report

```

```

/* USER CODE BEGIN 2 */:

```

```

HAL_Delay(5000); //retraso para esperar a que el USB se configure
dato[0]=0; //estos son los tres botones, ninguno pulsado

```

En while(1):

```

for (i=0;i<50;i++){

    dato[1]=4; //cada 100ms desplazamos el ratón 4 posiciones
    dato[2]=4;
    USB_CUSTOM_HID_SendReport_FS(dato, 4);
    HAL_Delay(100);

}

for (i=0;i<50;i++){

    dato[1]=-4;
    dato[2]=-4;
    USB_CUSTOM_HID_SendReport_FS(dato, 4);

```

```
    HAL_Delay(100);  
}
```

Una vez hecho todo esto, al debuguear veremos que el ratón comienza a moverse en diagonal. Aquí se presenta un análisis de los datos enviados en el siguiente descriptor de informe:

- (0x05, 0x01): Página de uso (USAGE_PAGE). Indica la categoría a la que pertenecen los siguientes usos. En este caso, se establece en "Generic Desktop".
- (0x09, 0x02): Uso (USAGE). Especifica el tipo de dispositivo en uso. Aquí se establece en "Mouse".
- (0xA1, 0x01): Colección (COLLECTION). Define el grupo de uso, en este caso, "Application".
- (0x09, 0x01): Uso (USAGE). Especifica el tipo de uso dentro de la colección, establecido aquí como "Pointer".
- (0xA1, 0x00): Colección (COLLECTION). Define una colección física de usos, en este caso, "Physical".
- (0x05, 0x09): Página de uso (USAGE_PAGE). Indica la categoría de los siguientes usos. Aquí se establece en "Button".
- (0x19, 0x01) y (0x29, 0x03): Uso mínimo (USAGE_MINIMUM) y uso máximo (USAGE_MAXIMUM). Especifica el rango para los botones, establecido entre "Button 1" y "Button 3".
- (0x15, 0x00) y (0x25, 0x01): Valor lógico mínimo (LOGICAL_MINIMUM) y valor lógico máximo (LOGICAL_MAXIMUM). Define los valores mínimo y máximo para los datos lógicos, establecidos en 0 y 1 respectivamente.
- (0x95, 0x03), (0x75, 0x01), (0x95, 0x01) y (0x75, 0x05): Recuento de informes (REPORT_COUNT) y tamaño de informe (REPORT_SIZE). Especifica el número y tamaño de los informes en bits, con 3 informes de 1 bit cada uno y 1 informe de 5 bits.
- (0x81, 0x02) y (0x81, 0x01): Entrada (INPUT). Define la información de entrada para los botones, como datos variables absolutos y datos constantes de matriz absolutos respectivamente.
- (0x05, 0x01): Se repite la página de uso (USAGE_PAGE) para los siguientes usos, establecida en "Generic Desktop".
- (0x09, 0x30), (0x09, 0x31) y (0x09, 0x38): Especifica los usos para los ejes X, Y y Wheel (Rueda).

- (0x15, 0x81) y (0x25, 0x7F): Valor lógico mínimo (LOGICAL_MINIMUM) y valor lógico máximo (LOGICAL_MAXIMUM) establecidos en -127 y 127 respectivamente.
- (0x75, 0x08) y (0x95, 0x03): Tamaño de informe (REPORT_SIZE) y recuento de informes (REPORT_COUNT) establecidos en 8 bits y 3 informes respectivamente.
- (0x81, 0x06): Entrada (INPUT) definida como datos variables relativos.
- (0xC0): Termina la colección física de usos.

▪ En la fase 2, haremos una copia de la práctica anterior. Primero, vamos a “usbd_custom_hid_if.c” y modificamos el código dentro de la función “CUSTOM_HID_ReportDesc_FS[USBD_CUSTOM_HID_REPORT_DESC_SIZE]”:

```
__ALIGN_BEGIN static uint8_t
CUSTOM_HID_ReportDesc_FS[USBD_CUSTOM_HID_REPORT_DESC_SIZE]
__ALIGN_END =
{
    /* USER CODE BEGIN 0 */
        0x06, 0x00, 0xFF, // Usage Page = 0xFF00 (Vendor Defined Page 1)
        0x09, 0x01, // Usage (Vendor Usage 1)
        0xA1, 0x01, // Collection (Application)
        0x19, 0x01, // Usage Minimum
        0x29, 0x06, // Usage Maximum //6 input usages total (0x01 to 0x06)
        0x15, 0x00, // Logical Minimum (data bytes in the report may have
minimum value = 0x00)
        0x26, 0xFF, 0x00, // Logical Maximum (data bytes in the report may have
maximum value = 0x00FF)
        0x75, 0x08, //Report Size: 8-bit field size
        0x95, 0x06, //Report Count: 6 of 8-bit fields (the next time the parser hits
an "Input", "Output")
        0x81, 0x00, //Input (Data, Array, Abs): Instantiates input packet fields
based on the above report size, count, logical min/max, and usage.
        0x19, 0x01, // Usage Minimum
        0x29, 0x06, // Usage Maximum //6 output usages total (0x01 to 6)
        0x91, 0x00, //Output (Data, Array, Abs): Instantiates output packet fields.
Uses same report size and count as "Input" fields, since nothing new/different was
specified to the parser since the "Input" item.
        0xC0 // End Collection
    /* USER CODE END 0 */
};
```

Si contamos los bytes, este nuevo array tiene una longitud de 29 bytes, por lo que debemos cambiar el valor de 50 o 74 de la longitud en la fase 1 a 29, si seguimos los mismos pasos que realizamos previamente. Además, hay que ajustar el valor del buffer a 6, ya sea en el archivo correspondiente o en el generador de código.

Después de hacer estos cambios, vamos a “main.c” y añadimos las siguientes líneas de código dentro del while(1):

```
for (i=0;i<50;i++)
{

    dato[1]=4; //cada 100ms desplazamos el ratón 4 posiciones
    dato[2]=4;
    USBD_CUSTOM_HID_SendReport_FS(dato,6);
    HAL_Delay(100);

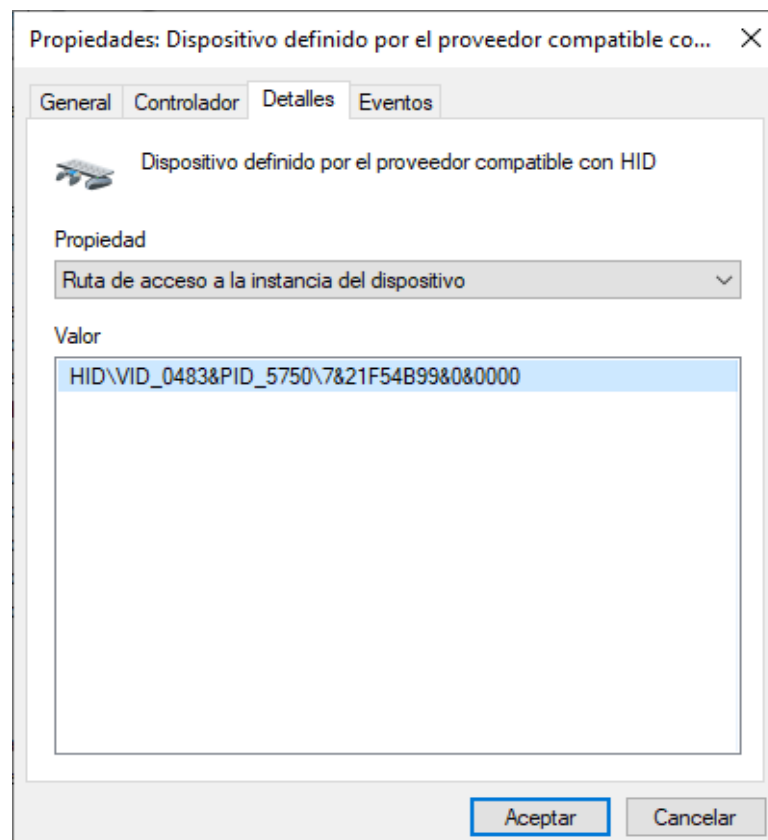
}

for (i=0;i<50;i++)
{

    dato[1]=-4;
    dato[2]=-4;
    USBD_CUSTOM_HID_SendReport_FS(dato,6);
    HAL_Delay(100);

}
```

Para saber si el dispositivo está funcionando vamos a ir al “Administrador de dispositivos” de nuestro ordenador y vamos a abrir la pestaña “Dispositivos de interfaz de usuario (HID)”. Aquí nos salen varios, donde al seleccionar el nuestro en “Ruta de acceso a la instancia del dispositivo” nos saldrá lo siguiente



Vamos a modificar el proyecto proporcionado junto a la práctica en Visual Studio para leer los datos enviados. Para ello, abrimos el proyecto y en la función CreateFile definimos como primer parámetro la dirección asignada por Windows a nuestro dispositivo, añadiendo los caracteres tras la almohadilla que vimos en la captura anterior.

```
if(resultado==0)

printf("No conseguido camino\n");

printf("%s\n", infocamino->DevicePath); //meramente informativo

//abrimos el dispositivo

handledisposi=CreateFile("\\\\?\\hid\\vid_0483&pid_5750#7621F54B9960&0000#{4D1E55B2-F16F-11CF-88CB-001111000030}",

    GENERIC_READ|GENERIC_WRITE,

    FILE_SHARE_READ|FILE_SHARE_WRITE,

    NULL,

    OPEN_EXISTING,

    0,

    NULL);
```

```
if(handledisposi==INVALID_HANDLE_VALUE)
```

```
    printf("Error abriendo dispositivo\n");
```

Al ejecutar el programa, observamos que el resultado de la escritura no es correcto: “resultado” es cero y se han escrito cero bytes. Esto se debe a que en el while debemos modificar la función WriteFile, ya que enviamos 6 bytes, pero hay que recordar que siempre se añade el IDReport, por lo que debemos cambiar el 6 por 7:

```
while(1)

{

    ReadFile(handledisposi, datos, 7, &bytesleidos, NULL);

    printf("datos=  %d,%d,%d,%d,  %d\n",  datos[0],  datos[1],  datos[2],  datos[3],

bytesleidos);


    //datos[1]= getch();

    //datos[0]=0;
```

```

//resultado = WriteFile(handledisposi, datos, 7, &bytesleidos, NULL);

//printf("datos=  %d,%d,%d,%d,  %d\n",  datos[0],  datos[1],  datos[2],  resultado,
bytesleidos);

}

```

```

return 0;

```

Sin embargo, ahora el programa se quedará bloqueado al ejecutar. Para solucionar esto, volvemos al IDE de la placa y abrimos “usbd_custom_hid_if.c”, donde modificamos la función “CUSTOM_HID_OutEvent_FS” de la siguiente manera:

```

static int8_t CUSTOM_HID_OutEvent_FS(uint8_t event_idx, uint8_t state)
{
    /* USER CODE BEGIN 6 */
    UNUSED(event_idx);
    UNUSED(state);
    miflag = 1;

    /* Start next USB packet transfer once data processing is completed */
    //USBDCUSTOMHID_ReceivePacket(&hUsbDeviceFS);

    return (USBD_OK);
    /* USER CODE END 6 */
}

```

La variable “miflag” ya ha sido declarada previamente en el mismo archivo.

Finalmente, en “main.c” modificamos el “main()” justo antes de while(1) y dentro del mismo, de la siguiente manera:

```

USBD_LL_PrepareReceive(&hUsbDeviceFS, 1, dato, 6);
/* USER CODE END 2 */

/* Infinite loop */
/* USER CODE BEGIN WHILE */
while (1)
{
    if(miflag==1)
    {

        USBD_LL_PrepareReceive(&hUsbDeviceFS, 1, dato, 6);
        miflag = 0;

    }

    for (i=0;i<50;i++)
    {

```

```

        dato[1]=4; //cada 100ms desplazamos el ratón 4 posiciones
        dato[2]=4;
        USBD_CUSTOM_HID_SendReport_FS(dato,6);
        HAL_Delay(100);

    }

    for (i=0;i<50;i++)
    {

        dato[1]=-4;
        dato[2]=-4;
        USBD_CUSTOM_HID_SendReport_FS(dato,6);
        HAL_Delay(100);

    }
    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */
}

```

Para que la función USBD_LL_PrepareReceive funcione correctamente, necesitamos declarar las siguientes variables globales como externas dentro del archivo main.c en la sección “*/* USER CODE BEGIN PV */*”:

```

extern USBD_HandleTypeDef hUsbDeviceFS;
extern uint8_t miflag;

```

Por último, vamos a “usb_customhid.c” y en la función “USB_CUSTOM_HID_ReceivePacket” tenemos que comentar la llamada a la función “USB_LL_PrepareReceive” tal que así:

```

uint8_t USB_CUSTOM_HID_ReceivePacket(USB_HandleTypeDef *pdev)
{
    USB_CUSTOM_HID_HandleTypeDef *hhid;

    if (pdev->pClassData == NULL)
    {
        return (uint8_t)USB_FAIL;
    }

    hhid = (USB_CUSTOM_HID_HandleTypeDef *)pdev->pClassData;

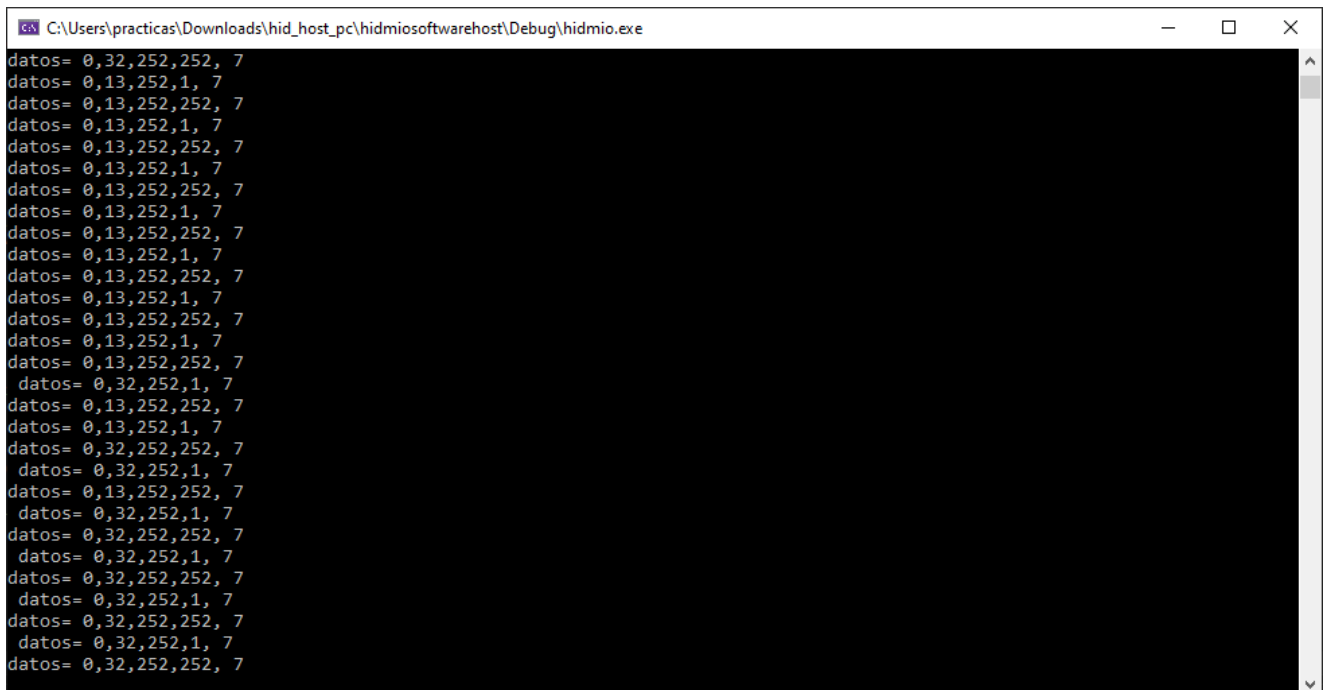
    /* Resume USB Out process */
    //(void)USB_LL_PrepareReceive(pdev, CUSTOM_HID_EPOUT_ADDR, //hhid-
    >Report_buf,USB_CUSTOMHID_OUTREPORT_BUF_SIZE);

    return (uint8_t)USB_OK;
}

```

```
}
```

Después de hacer todos estos cambios, al ejecutar el programa en Visual Studio, veremos que ahora funciona correctamente, mostrando lo siguiente en pantalla:



```
C:\Users\practicar\Downloads\hid_host_pc\hidmiosoftwarehost\Debug\hidmio.exe
datos= 0,32,252,252, 7
datos= 0,13,252,1, 7
datos= 0,13,252,252, 7
datos= 0,13,252,1, 7
datos= 0,13,252,252, 7
datos= 0,13,252,1, 7
datos= 0,13,252,252, 7
datos= 0,13,252,1, 7
datos= 0,13,252,252, 7
datos= 0,13,252,1, 7
datos= 0,13,252,252, 7
datos= 0,13,252,1, 7
datos= 0,13,252,252, 7
datos= 0,13,252,1, 7
datos= 0,13,252,252, 7
datos= 0,32,252,1, 7
datos= 0,13,252,252, 7
datos= 0,13,252,1, 7
datos= 0,32,252,252, 7
datos= 0,32,252,1, 7
datos= 0,13,252,252, 7
datos= 0,32,252,1, 7
datos= 0,32,252,252, 7
datos= 0,32,252,1, 7
datos= 0,32,252,252, 7
datos= 0,32,252,1, 7
datos= 0,32,252,252, 7
datos= 0,32,252,1, 7
datos= 0,32,252,252, 7
datos= 0,32,252,1, 7
datos= 0,32,252,252, 7
```

Realizaremos algunas modificaciones adicionales para habilitar la escritura de datos. Añadiremos el siguiente código al bucle while(1), de esta manera:

```
while(1)
{
    ReadFile(handledisposi, datos, 7, &bytesleidos, NULL);

    printf("datos=  %d,%d,%d,%d,  %d\n",  datos[0],  datos[1],  datos[2],  datos[3],
    bytesleidos);

    datos[1] = getche();

    datos[0] = 0;

    resultado = WriteFile(handledisposi, datos, 7, &bytesleidos, NULL);
```

```

    printf("datos= %d,%d,%d,%d, %d\n", datos[0], datos[1], datos[2], resultado,
bytesleidos);
}

```

```
return 0;
```

Con esta configuración, al ejecutar el programa, podremos enviar datos también. Si, por ejemplo, enviamos una “x”, se reflejará de esta manera en el Visual Studio:

```

C:\Users\practicar\Downloads\hid_host_pc\hidmiosoftwarehost\Debug\hidmio.exe
\\?\hid#vid_046d&pid_c077#6&1fae8c5d&0&0000#{4d1e55b2-f16f-11cf-88cb-001111000030}
datos= 0,0,252,252, 7
xdatos= 0,120,252,1, 7
datos= 0,0,252,252, 7
xdatos= 0,120,252,1, 7
datos= 0,120,252,252, 7
x

```

El valor que la placa recibe y que podemos ver debugueando en el IDE es el siguiente:

Name	Type	Value
(x)= i	int	50
▼ dato	uint8_t [3]	0x20017ff0
(x)= dato[0]	uint8_t	120 'x'
(x)= dato[1]	uint8_t	252 'ü'
(x)= dato[2]	uint8_t	252 'ü'

Para la fase 3, vamos a crear un dispositivo HID Custom de salida que controle el LCD y los LEDs desde el programa en ejecución en el ordenador. Para lograrlo, crearemos un nuevo proyecto y, tras configurar el .ioc de la misma manera que en la fase anterior, escribiremos el siguiente código en el “main.c” dentro de la función “main()”:

```

int main(void)
{
    /* USER CODE BEGIN 1 */

```

```

uint8_t dato[6]; //Declaramos las variables de nuestra función

/* USER CODE END 1 */

/* MCUConfiguration----- */

/* Reset of all peripherals, Initializes the Flash interface and the Systick. */

HAL_Init();

/* USER CODE BEGIN Init */

/* USER CODE END Init */

/* Configure the system clock */

SystemClock_Config();

/* USER CODE BEGIN SysInit */

/* USER CODE END Sys Init */

/* Initialize all configured peripherals */

MX_GPIO_Init();
MX_DFSDM1_Init();
MX_I2C2_Init();
MX_QUADSPI_Init();
MX_SPI3_Init();
MX_USART1_UART_Init();
MX_USART3_UART_Init();
MX_USB_DEVICE_Init();

/* USER CODE BEGIN 2 */

HAL_Delay(5000); //Retraso para esperar a que el USB se configure
dato[0] = 0; //Estos son los tres botones. Ninguno pulsado

USB_DLL_PrepareReceive(&hUsbDeviceFS, 1, dato, 6);
HAL_GPIO_WritePin(GPIOA, LED_LCD_Pin, GPIO_PIN_SET);

lcd_reset();
lcd_display_settings(1,0,0);
lcd_clear();

char str[5];
uint8_t datoVacio[6] = {0, 0, 0, 0, 0, 0};

```



```

/* USER CODE END 2 */

/* Infinite loop */

/* USER CODE BEGIN WHILE */

while(1)
{
    //Lo encendemos con la "A" y lo apagamos con la "X"
    if (miflag == 1)
    {

        USBD_LL_PrepareReceive(&hUsbDeviceFS, 1, dato, 6);
        miflag = 0;

        if (dato[0] == 97)
        {

            HAL_GPIO_WritePin(GPIOC, amarillo_Pin, GPIO_PIN_SET);

        }
        else if (dato[0] == 120)
        {

            HAL_GPIO_WritePin(GPIOC, amarillo_Pin,
GPIO_PIN_RESET);
        }

        str[0] = (char)dato[0];

        moveToXY(0, 0);
        lcd_print(str);

    }

    USBD_CUSTOM_HID_SendReport_FS(datoVacio, 6);

    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */

}

/* USER CODE END 3 */

}

```

Una vez hecho esto, podremos ejecutar el código y verificar que todo funciona correctamente. Si pulsamos la tecla “A”, se encenderá el LED, y si pulsamos la tecla “X”, se apagará.

- Para la fase 4, desarrollaremos un dispositivo HID Custom de salida. Desde el ordenador, enviaremos la posición deseada del servo y este se moverá a esa posición. Para lograrlo, crearemos un nuevo proyecto y, tras configurar el .ioc igual que en la fase anterior, escribiremos el siguiente código en el “main.c” dentro de la función “main()”:

int main(void)

```
{
    /* USER CODE BEGIN 1 */

    uint8_t dato[6]; //Declaramos las variables de nuestra función

    /* USER CODE END 1 */

    /* MCUConfiguration----- */

    /* Reset of all peripherals, Initializes the Flash interface and the Systick. */

    HAL_Init();

    /* USER CODE BEGIN Init */

    /* USER CODE END Init */

    /* Configure the system clock */

    SystemClock_Config();

    /* USER CODE BEGIN SysInit */

    /* USER CODE END Sys Init */

    /* Initialize all configured peripherals */

    MX_GPIO_Init();
    MX_DFSDM1_Init();
    MX_I2C2_Init();
    MX_QUADSPI_Init();
    MX_SPI3_Init();
    MX_USART1_UART_Init();
    MX_USART3_UART_Init();
    MX_USB_DEVICE_Init();
    MX_TIM3_Init();

    /* USER CODE BEGIN 2 */
```

```

HAL_Delay(5000); //Retraso para esperar a que el USB se configure
dato[0] = 0; //Estos son los tres botones. Ninguno pulsado

USBD_LL_PrepareReceive(&hUsbDeviceFS, 1, dato, 6);

uint8_t datoVacio[6] = {0, 100, 100, 100, 100, 0};
HAL_TIM_PWM_Start(&htim3, TIM_CHANNEL_4);

/* USER CODE END 2 */

/* Infinite loop */

/* USER CODE BEGIN WHILE */

while(1)
{
    Uint16_t servo;

    if (miflag == 1)
    {
        USBD_LL_PrepareReceive(&hUsbDeviceFS, 1, dato, 6);
        miflag = 0;
        setServoPos((float) dato[0]);
    }

    USBD_CUSTOM_HID_SendReport_FS(datoVacio, 6);

    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */
}

/* USER CODE END 3 */
}

```

La función “setServoPos” (usada en la práctica 4) es:

```

void setServoPos(float ang) {
    float th = ((ang * 200) / 18);
    __HAL_TIM_SET_COMPARE(&htim3, TIM_CHANNEL_4, th);
}

```

```
}
```

Una vez completado esto, podremos ejecutar el código y verificar que todo funciona correctamente.

- En la fase 5, vamos a desarrollar un dispositivo Custom de entrada, donde enviaremos los datos del acelerómetro desde la placa al ordenador para que se muestren en pantalla. Aunque esencialmente realizaremos lo mismo que en la fase 1, la diferencia radica en que esta vez enviaremos el valor completo de la aceleración en lugar de un desplazamiento del ratón. Comenzaremos creando un nuevo proyecto y, después de configurar el .ioc de la misma manera que en la fase anterior, escribiremos el siguiente código en el “main.c” dentro de la función “main()”:

```
int main(void)
```

```
{
/* USER CODE BEGIN 1 */

    uint8_t dato[6];

/* USER CODE END 1 */

/* MCU Configuration-----*/

/* Reset of all peripherals, Initializes the Flash interface and the Systick. */
HAL_Init();

/* USER CODE BEGIN Init */

/* USER CODE END Init */

/* Configure the system clock */
SystemClock_Config();

/* USER CODE BEGIN SysInit */

/* USER CODE END Sys Init */

/* Initialize all configured peripherals */

MX_GPIO_Init();
MX_DFSDM1_Init();
MX_I2C2_Init();
MX_QUADSPI_Init();
MX_SPI3_Init();
MX_USART1_UART_Init();
MX_USART3_UART_Init();
```

```

MX_USB_DEVICE_Init();

/* USER CODE BEGIN 2 */

HAL_Delay(5000); //Retraso para esperar a que el USB se configure
dato[0] = 0; //Estos son los tres botones. Ninguno pulsado

USB_D_LL_PrepareReceive(&hUsbDeviceFS, 1, dato, 6);

uint8_t datoVacio[6] = {0, 0, 0, 0, 0, 0};

initAccelerometer();
int16_t valorAccel;

/* USER CODE END 2 */

/* Infinite loop */

/* USER CODE BEGIN WHILE */

while(1)
{
    valorAccel = readAccel(0);

    dato[0] = valorAccel;
    dato[1] = valorAccel >> 8;

    USB_D_CUSTOM_HID_SendReport_FS(datoVacio, 6);

    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */

}

/* USER CODE END 3 */
}

```

Además, utilizaremos las funciones “initAccelerometer” y “readAccel”, que ya hemos utilizado en la Práctica 4:

```

void initAccelerometer(void){

    uint8_t buffer[1];
    buffer[0] = 0x40;
    HAL_I2C_Mem_Write(&hi2c2, 0xD4, 0x10, I2C_MEMADD_SIZE_8BIT, buffer, 1,
1000);

```

```

}

int16_t readAccel(uint8_t axis){

    uint8_t buffer[2];
    int16_t accel;
    uint8_t address = 0x28 + (2 * axis);
    HAL_I2C_Mem_Read(&hi2c2, 0xD4, address, I2C_MEMADD_SIZE_8BIT, buffer,
2, 1000);
    accel = ((int16_t) (buffer[1] << 8) | buffer[0]) * 0.061;
    return accel;

}

```

- En la fase 6, implementaremos un dispositivo Custom de entrada, donde enviaremos los datos de temperatura y humedad desde la placa al ordenador. Para esta fase, utilizaremos las librerías del HTS221 que se adjuntan en la práctica, las cuales importaremos a nuestro proyecto. Siguiendo las directrices proporcionadas en la práctica y utilizando nuestro conocimiento previo, escribiremos el siguiente código en el “main()”:

```

int main(void)

{
    /* USER CODE BEGIN 1 */

        uint8_t dato[6];

    /* USER CODE END 1 */

    /* MCU Configuration-----*/

    /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
    HAL_Init();

    /* USER CODE BEGIN Init */

    /* USER CODE END Init */

    /* Configure the system clock */
    SystemClock_Config();

    /* USER CODE BEGIN SysInit */

        /* USER CODE END Sys Init */

```

```
/* Initialize all configured peripherals */
```

```
MX_GPIO_Init();  
MX_DFSDM1_Init();  
MX_I2C2_Init();  
MX_QUADSPI_Init();  
MX_SPI3_Init();  
MX_USART1_UART_Init();  
MX_USART3_UART_Init();  
MX_USB_DEVICE_Init();
```

```
/* USER CODE BEGIN 2 */
```

```
HAL_Delay(5000); //Retraso para esperar a que el USB se configure  
dato[0] = 0; //Estos son los tres botones. Ninguno pulsado
```

```
USBD_LL_PrepareReceive(&hUsbDeviceFS, 1, dato, 6);
```

```
uint8_t datoVacio[6] = {0, 0, 0, 0, 0, 0};
```

```
HTS221_Init();  
int16_t valorTemp = 0;  
int16_t valorHum = 0;
```

```
/* USER CODE END 2 */
```

```
/* Infinite loop */
```

```
/* USER CODE BEGIN WHILE */
```

```
while(1)  
{
```

```
    HTS221_Get_Temperature(&valorTemp);  
    dato[0] = valorTemp;  
    dato[1] = valorTemp >> 8;  
    HTS221_Get_Humidity(&valorHum);  
    dato[2] = valorHum;  
    dato[3] = valorHum >> 8;
```

```
    USBD_CUSTOM_HID_SendReport_FS(datoVacio, 6);
```

```
/* USER CODE END WHILE */
```

```
/* USER CODE BEGIN 3 */
```

```
}
```

```
/* USER CODE END 3 */
```

```
}
```

4. Conclusión

He conseguido superar de manera satisfactoria todas las fases, algunas con mayor dificultad que otras, siguiendo las explicaciones proporcionadas en el documento de la práctica.

Personalmente, considero que esta práctica ha sido la más desafiante hasta ahora, especialmente debido a las últimas fases no guiadas sumado a algunos “obstáculos” por el camino donde no se detectaban bien “los dispositivos”. Sin embargo, al recurrir a lo que ya habíamos hecho en prácticas anteriores, se ha conseguido completar. También creo que ha sido una experiencia muy útil, ya que he aprendido a establecer una conexión entre la placa y el ordenador mediante la implementación de distintos usos.