Tutorial: CTU-016
Título: **Holtek HT32F USB device**
Autor: Sergio R. Caprile, Senior R&D Engineer

Revisiones	Fecha	Comentarios
0	29/04/13	

El presente tutorial tiene por objeto introducir la biblioteca de funciones para soporte de dispositivos USB en base a micros de la familia HT32F1755/65 de Holtek.

Índice de contenido

Descripción del periférico

La familia HT32F17x5 incorpora un periférico USB device compatible con la especificación USB 2.0 hasta Full Speed (12 MHz). El mismo soporta hasta 8 endpoints, uno de los cuales es el obligatorio y conocido Control Endpoint bidireccional. Cuatro de los restantes pueden configurarse para transacciones isócronas, bulk, o de interrupciones, mientras que los otros tres no soportan modo isócrono.

Internamente el periférico posee una FIFO de 1024 bytes para realizar todas las transacciones por hardware, de modo que el programa principal sólo sea interrumpido cuando la información ya está disponible.

Descripción de la biblioteca de funciones

El fabricante provee junto con el tradicional "driver", una library para soportar el acceso a este periférico y poder implementar dispositivos de diferentes clases de un modo relativamente simple. El código original está escrito para funcionar de manera conjunta con dicho driver, quien a su vez emplea la forma no-standard de referirse a los registros del micro sin el prefijo del fabricante, lo cual es inconsistente con los include files provistos por Keil y obliga al desarrollador a tener que incluir dicho driver. Para evitar esto y mantener las bondades de CMSIS, hemos podido remover dichas dependencias del código y proveemos una library consistente con todos los ejemplos anteriores, sin necesidad de utilizar el "driver".

La estructura de esta biblioteca de funciones es relativamente simple. Dada la naturaleza asíncrona de ambos mundos, USB y aplicación, la comunicación entre ambos se realiza mediante funciones callback. Las mismas deben ser inicializadas junto con el hardware, para lo cual disponemos de una serie de funciones a tal fin. Nuestra tarea será diagramar la estructura de los descriptores, proveer las funciones que requiera la clase del dispositivo a implementar, y pasar la información pertinente a la función de inicialización:

```
USBDCore_Init(USBDCore_TypeDef *pCore);
```

Más adelante, en el loop principal, cederemos tiempo de ejecución a las funciones de esta biblioteca llamando a la función

```
USBDCore_MainRoutine(USBDCore_TypeDef *pCore);
```

Esta función llamará oportunamente a la función de soporte de las características de la clase, la cual habremos escrito para funcionar como callback y habremos pasado sus datos en el proceso de inicialización. Esta función se ocupa también de los asuntos relacionados con el manejo de la energía, es decir, entrar y salir del modo de bajo consumo cuando el host USB así lo requiere; aunque en esencia todo lo que hace es identificar el pedido y llamar a la función callback correspondiente.

Las interrupciones generadas por el periférico USB deben ser procesadas por:

```
USBDCore_IRQHandler(USBDCore_TypeDef *pCore);
```

Dicha función a su vez llamará a otras funciones para procesar las transferencias del tipo de interrupción que la clase de dispositivo pudiera requerir en cada uno de sus endpoints. Dado que nosotros somos dueños de las interrupciones, deberemos entonces proveer un handler con el nombre esperado, que llame a la función anterior:

```
USB_IRQHandler()
```

Transacciones

Recordemos que la nomenclatura en USB está orientada a ser leída desde el host, por lo que un OUT es un ingreso de información a nuestro dispositivo e IN una salida de nuestra aplicación hacia el host USB. Cada endpoint se asocia a un tipo de transacción, y un modo de trabajo (isócrono, bulk, o de interrupciones) que determina la frecuencia de polling que el host le asignará.

OUT

Ante esta condición, el periférico genera una interrupción y el handler correspondiente al endpoint que tiene datos disponibles será llamado. Dentro de dicho handler, el cual escribiremos como parte del soporte de la clase que implementamos, emplearemos la función siguiente para obtener los datos:

```
USBDCore_EPTReadOUTData(EPTn, *pTo, len);    // #endpoint, dónde ponerlos, cuántos bytes max
```

Esta función resetea un flag de NAK interno, que se setea automáticamente al recibirse los datos. De este modo, cuando el host interroga para ver si puede enviarnos más datos, el periférico contesta NAK (no) sin nuestra intervención. Una vez retirados los datos mediante un llamado a dicha función, se resetea el flag y el host recibirá un ACK por respuesta, pudiendo enviarnos más datos si así lo desea. La función devuelve la cantidad de bytes recibidos.

IN

Para enviar datos por un endpoint de tipo IN, nuestra aplicación llamará a la función

```
USBDCore_EPTWriteINData(EPTn, *pFrom, len);  // #endpoint, dónde están, cuántos bytes
```

Cuando la transacción haya terminado, es decir, el host haya recibido los datos y el buffer esté libre para nuevas transferencias, el periférico generará una interrupción y la biblioteca de funciones llamará al handler correspondiente a dicho endpoint, el cual habremos escrito como parte del soporte de la clase que implementamos.

Control transfers

Son las transacciones realizadas en el endpoint 0, que es bidireccional. Una determinada clase puede requerir utilizarlas para determinado tipo de transacciones, caso contrario son manejadas por la biblioteca de funciones.
Las control transfers constan de tres etapas: setup, data, y status. La etapa de datos puede no estar presente. La biblioteca de funciones, como insinuamos, maneja la etapa de setup, identificando el pedido (request) y llamando a la función pertinente para procesarlo. Muchas clases tienen class specific requests, los cuales serán atendidos por una función callback. Dicha función informa a la library lo que debe saber para continuar o finalizar la transacción mediante una estructura.
El motivo del pedido, es decir, lo que se nos pide, se encuentra en un miembro de la estructura que recibimos como parámetro:

```
pDev->Request
```

el formato del mismo depende de la clase en cuestión.
Indicamos el destino o fuente de los datos, y su longitud, que generalmente es parte del pedido:

```
pDev->Transfer.pData = (uc8*) somewhere;
pDev->Transfer.sByteLength = pDev->Request.wLength;
```

Para leer datos enviados desde el host (SET):

```
pDev->Transfer.Action = USB_ACTION_DATAOUT;
```

Para enviar datos al host:

```
pDev->Transfer.Action = USB_ACTION_DATAIN;
```

Si la fase de datos no se debe realizar, se indica mediante un ZLP (zero length packet), es decir, se indica longitud a transmitir cero:

```
pDev->Transfer.pData = 0;
pDev->Transfer.sByteLength = 0;
```

Al salir de la función se vuelve a la library, que continúa el proceso del flujo del endpoint 0.
El procesamiento del endpoint 0 por parte de la biblioteca de funciones se realiza también por interrupciones, por lo que nuestra función para procesar class specific requests también se ejecutará dentro del interrupt handler como los handlers de cada endpoint que utilicemos.

Suspend

Hemos visto que en el loop principal cedemos tiempo de ejecución a

```
USBDCore_MainRoutine(USBDCore_TypeDef *pCore);
```

Cuando el host USB indica a nuestro dispositivo que debe ingresar en bajo consumo (suspend), el periférico genera una interrupción y la biblioteca de funciones lo coloca inmediatamente en ese estado. Luego, cuando llamemos a esta función, la misma llamará a la función callback específica que hayamos designado para procesar este pedido en nuestra aplicación.

Desarrollo de aplicaciones en C

Veremos a continuación un ejemplo concreto, a nivel arquitectónico; el desarrollo particular se encuentra en la nota de aplicación correspondiente: CAN-102.

Entorno de software

Esta biblioteca de funciones emplea la mala práctica de redefinir los tipos con nombres reducidos, por lo cual para evitar tener que realizar más malabares hemos decidido respirar hondo y aceptarlos. Por ello, nos encontraremos con variables de tipo u8 y u32 para uint8_t y uint32_t, respectivamente, que necesitaremos para nuestros buffers. Además, para optimizar las transferencias de memoria la library requiere que muchas variables y estructuras estén alineadas en words, lo cual Hoitek indica mediante __ALIGN4. En general, nos resulta más claro llamarla __ALIGNto32, y preferimos utilizar la siguiente macro:

```
#if defined (__CC_ARM)
  #define __ALIGNto32 __align(4)
#elif defined (__ICCARM__)
  #define __ALIGNto32 _Pragma("data_alignment = 4")
#elif defined (__GNUC__)
  #define __ALIGNto32  __attribute__((aligned(4)))
#endif
```

Por defecto, los compiladores mantienen la alineación a 32-bits dentro de las estructuras. Para evitar esto, y dado que la forma de indicarlo depende también de cada compilador, Holtek utiliza __PACKED_H y __PACKED_F sobre la definición de la estructura. En general, nos resulta más claro llamarlas __PACKED_pre y __PACKED_post, y preferimos utilizar las siguientes macros:

```
#if defined (__GNUC__)
  #define __PACKED_pre
  #define __PACKED_post __attribute__ ((packed))
#elif defined (__ICCARM__) || (__CC_ARM)
  #define __PACKED_pre __packed
  #define __PACKED_post
#endif
```

Entorno de hardware

La placa de desarrollo de HT32F1755/65 requiere que coloquemos un pin en alto para habilitar el pull-up de USB y así señalizar al host que estamos en el bus:

```
void HT32F175xDVB_USBConnect(void)
{
        HT_CKCU->APBCCR0 |= (1<<14)+(1<<16);  // 14=AFIOEN, 16=PAEN, enable APB clocks for AFIO and
GPIOA
        HT_AFIO->GPACFGR &= ~(0x03<<8);       // PA4 -> AF0 (PA4)
        HT_GPIOA->DIRCR |= 0x0010;            // PA4 defined as Output
        HT_GPIOA->DOUTR |= (1<<4);            // PA4=1
}
```

El periférico USB device requiere un clock de 48MHz, la secuencia de habilitación del clock APB del mismo y la configuración del clock del sistema sugerida es la siguiente:

```
        HT_CKCU->APBCCR1 |= (1<<14);          // 14=USBEN, APB clock para USB (registros/RAM)
        HT_CKCU->GCFGR &= ~(0x03<<22);        // clear USB clock prescaler bits
        // ver PLL y AHB clock en system_.c,el periférico USB requiere  48MHz
        HT_CKCU->GCFGR |= (0x02<<22);         // set USB clock to PLL/3 (144/3 = 48MHz)
```

Ejemplo: Puerto de comunicaciones virtual (VCOM)

Corresponde a una implementación de la clase CDC (Communications Device Class), con el modelo abstracto de control (Abstract Control Model). Esto se encuentra documentado en la especificación de la clase CDC y la extensión para PSTN. La operación de este tipo de dispositivos requiere de un driver, el cual (para Windows) ha sido provisto por Holtek.
Se trata entonces de una interfaz mediante la cual podemos recibir y enviar datos a y desde un host, el cual nos ve como un puerto serie. Esto requiere de dos endpoints, uno para cada sentido de comunicación, empleando transacciones bulk.
A fin de proveer el estado de las señales de interfaz e iniciar y terminar comunicaciones (Management Element Notifications), la clase especifica un endpoint adicional con transacciones por interrupciones. El tipo de señalización

depende de la subclase en particular. En PSTN, podemos controlar DSR y DCD. Notemos que CTS no existe, el dispositivo debe contestar NAK al endpoint OUT del host cuando no puede procesar la data que éste le envía.

Inicialización

Holtek nos provee ht32_usbd_core.h, que contiene todo lo necesario para utilizar el periférico. Al implementar la clase desarrollaremos un archivo con los descriptores y otro con las funciones de la clase.

```
#include "ht32_usbd_core.h"
#include "ht32_usbd_cdcclass.h"
#include "ht32_usbd_cdcdescriptor.h"

__ALIGN4 USBDCore_TypeDef gUSBCoreVCP;
USBD_Driver_TypeDef gUSBDriver;

void Suspend(u32 uPara);

void init_USBstuff()
{
        gUSBCoreVCP.pDriver = (u32 *)&gUSBDriver;

        // Configura la función callback para USB suspend
        gUSBCoreVCP.Power.CallBack_Suspend.func  = Suspend;
        //gUSBCore.Power.CallBack_Suspend.uPara = NULL;

        // Inicializa el puntero al descriptor definido en ht32_usbd_cdcdescriptor.c
        USBDCDCDesc_Init(&gUSBCoreVCP.Device.Desc);

        // Inicializa el puntero a la clase definida en ht32_usbd_cdcclass.c
        USBDCDCClass_Init(&gUSBCoreVCP.Class);

        // Inicializa el Hardware
        USBDCore_Init(&gUSBCoreVCP);
        // Habilita interrupciones de periférico USB
        NVIC_EnableIRQ(USB_IRQn);
        // Habilita pull-up en USB D+ ("here I am")
        HT32F175xDVB_USBConnect();
        // wait for enumeration
}
```

Programa principal

Inicializado el hardware, llamamos periódicamente a la función que realiza el mantenimiento del estado del periférico:

```
main()
{

        HT_CKCU->APBCCR1 |= (1<<14);          // 14=USBEN, APB clock para USB (registros/RAM)
        HT_CKCU->GCFGR &= ~(0x03<<22);        // clear USB clock prescaler bits
        // ver PLL y AHB clock en system_.c,el periférico USB requiere  48MHz
        HT_CKCU->GCFGR |= (0x02<<22);         // set USB clock to PLL/3 (144/3 = 48MHz)

        init_USBstuff();                      // init USB stuff

        while(1){
                USBDCore_MainRoutine(&gUSBCoreVCP);
                ...
        }
}
```

Bajo consumo

Un evento USB suspend event generará una llamada a la función que indicamos en el proceso de inicialización: Suspend(), cuando cedamos control a USBDCore_MainRoutine().

Interrupciones

Nuestro handler del periférico (vía el NVIC) deriva las interrupciones a la library: USBDCore_IRQHandler(), donde son procesadas y derivadas al handler del endpoint correspondiente, que escribiremos según los requerimientos de la clase.

```
void USB_IRQHandler(void)
```

```
{
__ALIGN4 extern USBDCore_TypeDef gUSBCoreVCP;

        USBDCore_IRQHandler(&gUSBCoreVCP);
}
```

Aplicación

Cuando la aplicación requiera transmitir o esté lista para recibir llamará a las funciones correspondientes, que desarrollaremos al implementar la clase.

```
        CDCclass_receive((u32 *)buffer)) // !=-1 -> mensaje de n bytes (debe caber en el buffer)
        CDCclass_send((u32 *)buffer, bytes) // != -1 -> OK); (no debe exceder buffer del endpoint)
```

Implementación de la clase

Descriptores y configuración de endpoints

El juego de descriptores es tal vez la pieza más importante, sin embargo no es código ni algo fuertemente dependiente de esta biblioteca de funciones (sino de la clase y el dispositivo). Lo hemos dejado relegado a sus archivos respectivos, sin mayores comentarios, que requieren de un estudio de la especificación USB y la clase en particular.
En síntesis, armamos el descriptor e inicializamos la estructura que la library requiere colocando las direcciones de cada uno de los descriptores en la misma

```
void USBDCDCDesc_Init(USBDCore_Desc_TypeDef *pDesc)
{
  pDesc->pDeviceDesc = dirección del Device Descriptor;
  pDesc->pConfnDesc = dirección del Config Descriptor;
  pDesc->ppStringDesc = dirección del String Descriptor;
  pDesc->uStringDescNumber = cantidad de éstos;
}
```

Aquí elegiremos los endpoints que vamos a usar. En nuestro caso elegimos y configuramos el 1 como salida de datos (bulk IN), el 2 como salida por interrupciones (interrupt IN) y el 3 como entrada de datos (bulk OUT). Todos tienen un tamaño de buffer de 64 bytes (16 el de interrupciones), que serán manejados por el periférico de forma autónoma, son buffers internos configurables. Lo que ingresemos aquí es lo que el host sabrá de nosotros, la configuración de buffers la hacemos en un archivo que será incluido por la biblioteca de funciones al compilarla: ht32f175x_275x_usbdconf.h.
El hecho de que el endpoint 2 soporte transferencias de tipo interrupt sólo significa que el host USB interrogará a este endpoint más seguido.

Inicialización

La función de inicialización de la clase la hemos llamado en el proceso de inicialización (sin saber lo que hacíamos). Aquí preparamos las funciones callback para servir al programa principal de la clase, llamado desde el loop principal cuando llamamos a USBDCore_MainRoutine().

```
void USBDCDCClass_Init(USBDCore_Class_TypeDef *pClass)
{
        pClass->CallBack_MainRoutine.func = USBDClass_MainRoutine;
        pClass->CallBack_MainRoutine.uPara = (u32)0;

        pClass->CallBack_ClassRequest = USBDClass_Request;
        pClass->CallBack_EPTn[1] = USBDClass_Endpoint1;
        pClass->CallBack_EPTn[2] = USBDClass_Endpoint2;
        pClass->CallBack_EPTn[3] = USBDClass_Endpoint3;
}
```

Envío de datos al host

Vimos en el programa principal que llamamos a la función CDCclass_send(). También vimos que disponemos de una función: USBDCore_EPTWriteINData(), para mandar datos por el endpoint 1. Cuando estos datos hayan sido enviados, es decir, el host USB haya interrogado a nuestro periférico por el endpoint 1 y los datos hayan sido entregados, se producirá una interrupción, que cederá el control del procesador a USB_IRQHandler(), que llamará a USBDCore_IRQHandler(), quien a su vez llamará a la función callback que procesa el endpoint 1.

Aquí podemos, por ejemplo, hacer que la función llamada por el programa principal ingrese datos a una cola (y avanzar directamente si ésta está vacía), tomando luego los datos de la cola (hasta vaciarla) con cada interrupción:

```
CDCclass_send(u32 *buffer, int len)
{
        if(cola llena)
                oops
        coloca datos en cola
        if(cola vacía)
                 USBDClass_Endpoint1(USBD_EPT1);
}

static void USBDClass_Endpoint1(USBD_EPTn_Enum EPTn)
{
        if(cola no vacía) {
                toma datos de cola
                USBDCore_EPTWriteINData(USBD_EPT1, buffer, len);
        }
}
```

o bien algo simple como hemos hecho en la nota de aplicación: levantamos un flag para indicar que no podemos procesar más datos (buffer lleno), el cual reseteamos ante la interrupción:

```
CDCclass_send(u32 *buffer, int len)
{
        if (flag)
                oops
        set flag
        USBDCore_EPTWriteINData(USBD_EPT1, buffer, len);
}

static void USBDClass_Endpoint1(USBD_EPTn_Enum EPTn)
{
        reset flag
}
```

Recepción de datos enviados por el host

Cuando el host USB quiere enviarnos datos, interroga al endpoint 3 y si nuestro periférico le dice que está libre le entrega los datos. El periférico entonces genera una interrupción, que cederá el control del procesador a USB_IRQHandler(), que llamará a USBDCore_IRQHandler(), quien a su vez llamará a la función callback que procesa el endpoint 3, donde decidimos qué hacer.
Recordemos que para obtener los datos del endpoint, disponemos de una función: USBDCore_EPTReadOUTData(), que resetea el flag de NAK seteado automáticamente al recibirse los datos, para evitar que el host sobreescriba el buffer. Esto hace las veces de CTS, como comentamos.
Aquí es donde podríamos por ejemplo ingresar los datos a una cola, y la aplicación, más tarde, en el programa principal, llamará (cuando esté lista para recibir datos) a la función CDCclass_receive(), que retira los datos de la cola:

```
static void USBDClass_Endpoint3(USBD_EPTn_Enum EPTn)
{
        if(cola llena)
                necesito volver a revisar cuando la aplicación retire un elemento de la cola
        USBDCore_EPTReadOUTData(USBD_EPT3, buffer, _EP3LEN);        // tamaño buffer >= _EP3LEN
        coloca datos en cola
}

CDCclass_receive(u32 *buffer)
{
        if(cola no vacía)
                toma datos de cola
        else
                no hay datos
}
```

o bien algo simple como hemos hecho en la nota de aplicación: levantamos un flag para indicar que hay datos disponibles. La aplicación, más tarde, en el programa principal, llamará (cuando esté lista para recibir datos) a la función CDCclass_receive(), que revisará dicho flag y a su vez llamará a la función que copia los datos del buffer y resetea el flag de NAK:

```
static void USBDClass_Endpoint3(USBD_EPTn_Enum EPTn)
```

```
{
        set flag
}

CDCclass_receive(u32 *buffer)
{
        if (flag){
                reset flag
                USBDCore_EPTReadOUTData(USBD_EPT3, buffer, _EP3LEN);    // tamaño buffer >= _EP3LEN
        }
        else
                no hay datos
}
```

Recepción del estado de las señales de interfaz

Si bien esto puede no ser necesario en algunas implementaciones, es un buen ejemplo del manejo de los class specific requests.
Cuando el host USB deba informarnos de una modificación como ésta, se valdrá de un comando específico dentro del tráfico habitual del endpoint 0, que mayormente nos es ajeno y tal vez hasta desconocido. Vimos en la función de inicialización de la clase que configurábamos una función para atender estos menesteres (sin saberlo, claro):

```
        pClass->CallBack_ClassRequest = USBDClass_Request;
```

Dicha función recibirá el control del procesador cuando una interrupción que procesa el endpoint 0 determine que se trata de un pedido para la clase, es decir, class specific requests. Dentro de éstos, que se hallan especificados y detallados en la documentación de la misma, tenemos el que nos interesa, que tiene por función informarnos del estado de DTR y RTS para que, por ejemplo, en el caso que seamos un puerto serie real, las informemos al exterior.
Por ejemplo, mostrando sólo lo pertinente:

```
static void USBDClass_Request(USBDCore_Device_TypeDef *pDev)
{
        u8 USBCmd = *((u8 *)(&(pDev->Request.bRequest)));
        u16 len = *((u16 *)(&(pDev->Request.wLength)));

        switch (USBCmd)        {
                case CDCCLASS_REQ_SET_CONTROL_LINE_STATE:
                        if(len == 0)
                                USBDClass_SetControlLineState(pDev);
                        break;
        }
}
```

Dentro de la función USBDClass_SetControlLineState() vamos a procesar el mensaje y contestar, de la forma que vimos al principio de este tutorial:

```
static void USBDClass_SetControlLineState(USBDCore_Device_TypeDef *pDev)
{
        mysignals = pDev->Request.wValueL;    // obtiene estado de las señales de interfaz
        pDev->Transfer.pData = 0;
        pDev->Transfer.sByteLength =0;
        pDev->Transfer.Action = USB_ACTION_DATAOUT;
}
```

En este caso, como vemos, los datos a recibir son pocos y viajan dentro del request, por lo que indicamos a la library que no existe fase de datos.

Envío del estado de las señales de interfaz

Si bien esto puede no ser necesario en algunas implementaciones, es un buen ejemplo del uso de los control transfers y de la finalidad de la función principal de la clase.
La lectura del estado de las señales propiamente dichas la hacemos en la función principal de la clase, llamada por la función principal de la library cuando le cedemos tiempo de ejecución (en el loop principal). Aquí, si observamos un cambio, lo reportamos al host llamando a la función correspondiente tal cual como cuando enviamos datos.

```
static void USBDClass_MainRoutine(u32 uPara)
{
        if (hay un cambio en alguna señal) {
                prepara header acorde al formato que requiere y documenta la clase + subclase
                USBDCore_EPTWriteINData(USBD_EPT2, (u32 *)myheader, mybytes);
        }
```

```
}
```

Enviados dichos datos el periférico genera una interrupción que mediante el mecanismo indicado encuentra su camino hasta la función correspondiente, la cual como no necesitamos que haga nada, dejamos vacía.