Nota de Aplicación: CAN-102
Título: **HT32F USB device: COM virtual**
Autor: Sergio R. Caprile, Senior R&D Engineer

Revisiones	Fecha	Comentarios
0	17/05/13	

La presente nota de aplicación detalla parte de lo explicado en el tutorial CTU-016. Se sugiere enfáticamente la lectura del mismo dado que contiene detalles que no abordamos aquí.

El código de esta nota se basa en material entregado por el fabricante, modificado por claridad y adaptado a la forma de trabajo que consideramos más apropiada.

Puerto de comunicaciones virtual (VCOM)

Por este nombre nos referimos a una implementación en el entorno USB 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(Public Switched Telephone Network), destinada a "modernizar" modems y demás sistemas similares.

Mediante ésta, una aplicación en un entorno con un host USB puede escribir y leer datos sobre el dispositivo que la implementa, viéndolo desde dicha aplicación como un puerto serie tradicional.

La operación de este tipo de dispositivos requiere de un driver, el cual (para Windows) ha sido provisto por Holtek y lo encontramos en el archivo HT32_VCP.inf.[1]

El ejemplo que proponemos es algo muy simple que realiza un loop entre entrada y salida, por lo que veremos un eco a los caracteres que ingresemos en cualquier programa capaz de abrir el puerto de comunicaciones que proveeremos al correr este programa en nuestro HT32F1755_65, conectado a una PC.

Implementación de la clase CDC

Configuración de endpoints

La clase CDC define el empleo de transacciones bulk.

Host	Sentido	Dispositivo
Bulk OUT	→	Recibe un mensaje y lo mantiene en un buffer del periférico
Bulk IN	←	Transmite el contenido de un buffer del periférico, si previamente se colocó algo allí

A esta operación agregamos además un endpoint con transacciones por interrupciones para proveer el estado de las señales de interfaz e iniciar y terminar comunicaciones (Management Element Notifications). Esto 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.

Dado que no disponemos de las señales de interfaz en realidad, las simulamos en memoria y haremos también un loop entre algunas de ellas.

La distribución de endpoints entonces, es la siguiente:

1 Se asume que el lector conoce el sistema operativo de su computadora lo suficiente como para poder instalar el driver mencionado.

Endpoint	Tipo de transferencia	Tamaño de buffer
0	Control	64 bytes
1	Bulk IN	64 bytes
2	Interrupt IN	16 bytes
3	Bulk OUT	64 bytes

La configuración de endpoints y buffers la hacemos en un archivo que será incluido por la biblioteca de funciones al compilarla: ht32f175x_275x_usbdconf.h.
Deberemos cargar también parte de la información en los descriptores

Descriptores

ver tutorial

Inicialización

ver tutorial

Envío de datos al host

Vimos en el programa principal que llamamos a la función CDCclass_send(), La misma llama a USBDCore_EPTWriteINData() para mandar datos por el endpoint 1. Inmediatamente levantamos un flag para indicar que no podemos recibir más datos (buffer lleno):

```
CDCclass_send(u32 *buffer, int len)
{
        if (txbusy)
                return(-1);
        txbusy = 1;
        USBDCore_EPTWriteINData(USBD_EPT1, buffer, len);
        return(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, donde reseteamos nuestro flag de ocupado:

```
static void USBDClass_Endpoint1(USBD_EPTn_Enum EPTn)
{
        txbusy = 0;
}
```

En el caso particular de la clase CDC, la vida real es algo más complicada, situación que dejamos para el apéndice.

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. Dado que el buffer que en ese momento contiene los datos no será sobreescrito hasta que llamemos a la función de lectura que a su vez resetea el flag de NAK, como vimos (el host recibe NAK cuando interroga a este endpoint y su buffer está lleno porque aún no lo hemos vaciado, esto oficia a la vez de señal de CTS) seteamos un flag que indique que hay datos disponibles:

```
static void USBDClass_Endpoint3(USBD_EPTn_Enum EPTn)
{
        rxready = 1;
}
```

La aplicación, más tarde, llamará 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:

```
CDCclass_receive(u32 *buffer)
{
	if (rxready){
		rxready = 0;
		return(USBDCore_EPTReadOUTData(USBD_EPT3, buffer, _EP3LEN));
	}
	return(-1);
}
```

Recepción del estado de las señales de interfaz y otros menesteres de la clase

Cuando el host USB deba informarnos de una modificación en las señales de interfaz, se valdrá de un comando específico dentro del tráfico habitual del endpoint 0, 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.
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;
		case CDCCLASS_REQ_SET_LINE_CODING:
			USBDClass_SetLineCoding(pDev);
			break;
		case CDCCLASS_REQ_GET_LINE_CODING:
			USBDClass_GetLineCoding(pDev);
			break;
		// others not supported
		default:
			break;
	}
}
```

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

```
static void USBDClass_SetControlLineState(USBDCore_Device_TypeDef *pDev)
{
	mysignals = pDev->Request.wValueL;    // obtiene señales, almacena para loop
	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.
Las funciones USBDClass_SetLineCoding() y USBDClass_GetLineCoding() operan sobre una estructura que contiene la información de velocidad de transmisión y largo de palabra. Esto es irrelevante en nuestro ejemplo pero útil en el caso de existir un puerto serie real que deba ser configurado (si por ejemplo usamos la UART del micro para comunicar al host USB con el exterior a través nuestro).

```
typedef struct _VCP_LINE_CODING
{
  u32 dwDTERate;      //Bit rate;
  u8  bCharFormat;    //Stop bits:
                      //0 = 1   Stop bit
                      //1 = 1.5 Stop bit
                      //2 = 2   Stop bit
  u8  bParityType;    //parity:
```

```
                        //0 = None
                        //1 = Odd
                        //2 = Even
                        //3 = Mark
                        //4 = Space
  u8  bDataBits;      //Number of data bits (7,8,9)
}USBDCDCClass_LINE_CODING;

static void USBDClass_SetLineCoding(USBDCore_Device_TypeDef *pDev)
{
        pDev->Transfer.pData = (uc8*)&USBDCDCClassLineCoding;
        pDev->Transfer.sByteLength =
                (sizeof(USBDCDCClassLineCoding) > pDev->Request.wLength) ?
                        (pDev->Request.wLength) : (sizeof(USBDCDCClassLineCoding));
        pDev->Transfer.Action = USB_ACTION_DATAOUT;
        return;
}

static void USBDClass_GetLineCoding(USBDCore_Device_TypeDef *pDev)
{
        pDev->Transfer.pData = (uc8*)&USBDCDCClassLineCoding;
        pDev->Transfer.sByteLength =
                (sizeof(USBDCDCClassLineCoding) > pDev->Request.wLength) ?
                        (pDev->Request.wLength) : (sizeof(USBDCDCClassLineCoding));
        pDev->Transfer.Action = USB_ACTION_DATAIN;
        return;
}
```

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.
Dentro de las Management Element Notifications, implementamos SERIAL_STATE para la subclase PSTN, que simplemente envía el estado de las señales de interfaz por un endpoint de interrupciones, con un header:

```
typedef __PACKED_H struct
{
        u8 bmRequestType;
        u8 bNotification;
        u16 wValue;
        u16 wIndex;
        u16 wLength;
        u16 Data[1];                                    // definir buffer si aplica
} __PACKED_F CDCclass_Notification_TypeDef;

#define CDCCLASS_NOTHEADER_LEN (offsetof(CDCclass_Notification_TypeDef,Data))
```

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)
{
static CDCclass_Notification_TypeDef n;

        if (mysignals != myoldsignals) {      // detecta cambios
                myoldsignals=mysignals;
                n.bmRequestType = REQ_DIR_01_D2H | REQ_TYPE_01_CLS | REQ_REC_01_INF; // 0xA1
                n.bNotification = CDCCLASS_NOT_PSTN_SERIAL_STATE;
                n.wValue = 0;
                n.wIndex = 0;
                n.wLength = 2;
                n.Data[0] = (u16) mysignals;  // insertar señales reales aquí
                USBDCore_EPTWriteINData(USBD_EPT2, (u32 *)&n, CDCCLASS_NOTHEADER_LEN +
                        n.wLength);           // y envía
        }
}
```

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:

```
static void USBDClass_Endpoint2(USBD_EPTn_Enum EPTn)
{
}
```

Programa principal

Inicializado el hardware, llamamos periódicamente a la función que realiza el mantenimiento del estado del periférico, y eventualmente cuando la aplicación requiera transmitir o esté lista para recibir llamará a las funciones correspondientes, que desarrollamos al implementar la clase. Aquí realizamos un loop, el cual implementamos copiando el contenido del buffer de entrada (inbuff) al de salida (outbuff).

```
main()
{
__ALIGN4 static u8 inbuff[64];       // tamaño acorde al endpoint buffer
__ALIGN4 static u8 outbuff[64];
int ret, state=0;

        HT_CKCU->APBCCR1 |= (1<<14);          // 14=USBEN, APB clock para USB (registros/RAM)
        HT_CKCU->GCFGR &= ~(0x03<<22);        // clear USB clock prescaler bits
        HT_CKCU->GCFGR |= (0x02<<22);         // set USB clock to PLL/3 (144/3 = 48MHz)

        init_USBstuff();                      // init USB stuff

        while(1){
                USBDCore_MainRoutine(&gUSBCoreVCP);
                switch(state){
                case 0:         // idle
                        if((ret=CDCclass_receive((u32 *)inbuff))!=-1){
                                // recibí un mensaje!
                                memcpy(outbuff,inbuff,ret);   // loopback
                                state = 1;
                        }
                        break;
                case 1:         // debo enviar un mensaje (de longitud <= al tamaño del buffer)
                        if(CDCclass_send((u32 *)outbuff, ret) != -1)
                                state = 0;
                        break;
                }
}
```

Apéndice 1: Un COM virtual en la vida real

En el ejemplo que desarrollamos en esta nota, la información vuelve al host, no es tomada del exterior, con lo cual nos evitamos un pequeño problema que hace una treintena de años motivó algunas recomendaciones de la por entonces CCITT (hoy ITU-T), y no tan atrás un capítulo de un libro relacionado con la cría de conejos, o algo por el estilo..

Resulta que en la vida real tendremos bytes ingresando por el puerto serie, un medio secuencial cuya unidad es el carácter; que debemos enviar por otro medio cuya unidad es el paquete[1]. Cada vez que recibimos uno, deberemos decidir si esperamos a ver si vienen más, o los mandamos ya. En un caso, maximizamos el aprovechamiento del ancho de banda del bus y por ende la velocidad de transmisión; en el otro, minimizamos la latencia. Puede aprenderse sobre estos menesteres estudiando las recomendaciones X.28 y X.3 de la ITU-T, preguntando a un venerable anciano qué es un PAD (Packet Assembler-Disassembler), o analizando alguna implementación “puerto serie a socket” de las que hay por ahí. A rasgos generales, la operación a realizar es:

- Recibo un caracter
  - Lo coloco en el buffer
  - ¿Es “especial”? (corresponde a un grupo determinado)
    - Envío el contenido del buffer
  - ¿Se llenó el buffer ?
    - Envío el contenido del buffer

1 en realidad el término correcto es transacción, pero dejamos este detalle para el otro apéndice.

- No pasa nada, ¿expiró el tiempo prefijado desde el último caracter recibido?
  - Envío el contenido del buffer

Mientras la generación de los datos sea controlada por el dispositivo, no tendremos este inconveniente y podemos usar tranquilamente el esquema propuesto (Sin embargo, tenemos además otro problema, que veremos en el apéndice siguiente).

Apéndice 2: Por qué esta nota de aplicación no funciona (64-byte blue's)

Bueno, para ser justos... sí funciona, aunque no del todo bien.
Si probamos de enviar de a un caracter por vez, lo vemos bien. Si armamos un grupo de caracteres y lo enviamos, lo vemos volver. Funciona con grupos de a 2, 3, 20, 63, 80... uy, no funciona con grupos de 64 o múltiplos... pero cuándo mando otra cantidad recibo todo, hasta lo que no recibí antes, ¿qué pasa?
En USB, los datos se pasan por transacciones de cantidades específicas. Según la especificación, una transacción finaliza cuando se transfiere la cantidad especificada de datos, o se indica con un paquete de longitud menor al tamaño de buffer.
La clase CDC es un caso particular, en el que la recepción del host no conoce la cantidad de datos que va a recibir. La única forma de terminar una transacción o transferencia es enviando un paquete de longitud menor al tamaño de buffer. Esto significa que si el dispositivo USB reporta un tamaño de buffer de 64 bytes y envía esta cantidad de datos, los mismos pasan al host, pero se quedan esperando en algún buffer intermedio a que se produzca la finalización de la transferencia, para ser entregados a la aplicación que escucha a ese endpoint. Esta es la razón por la cual no vemos los mensajes cuya longitud no corresponde a múltiplos de este valor, hasta tanto un mensaje más corto indique la finalización de la transacción, o se llegara al tamaño de transferencia especificado, valor el cual pocos mortales parecen conocer.
El número 64 es arbitrario, el "problema" aparece cada vez que se intenta enviar un conjunto de bytes múltiplo del tamaño de buffer. Aquí, 64, es el número más común.

Entonces, en la vida real, nuestra aplicación en el micro estará utilizando un puerto serie virtual implementado mediante la clase CDC para comunicarse con una aplicación en una computadora. Dicha aplicación puede básicamente ser de uno de estos dos tipos: un COM virtual propiamente dicho, que toma caracteres del exterior; o un intercambio de mensajes.
El primer caso está cubierto por lo analizado en el apéndice anterior, con la salvedad de excluir a los múltiplos del tamaño de buffer de endpoint para nuestro tamaño de buffer.
El segundo caso entonces implica que la aplicación sabe cuántos bytes debe enviar al host, y bien puede encargarse de enviar un ZLP (zero-lenght packet, mensaje de longitud cero que indica fin de transacción)[1] cuando la longitud del mensaje iguala a un múltiplo del tamaño del buffer. Resuelto el problema.
¿Por qué no modificar la clase para resolver esto? Porque es más complicado de lo que parece, y como ya hemos explicado, es más simple resolverlo a nivel de aplicación en la vida real.
¿Complicado?¿Por qué no enviar siempre un ZLP luego de los 64 bytes? Porque algunas aplicaciones remolonas en la computadora se ponen tan contentas cuando reciben un mensaje que olvidan por algunos milisegundos de iniciar una nueva transacción, con lo cual si enviamos 64 + ZLP + 64 + ZLP, los últimos 64 bytes se quedan durmiendo en un buffer perdido hasta tanto llegue algo nuevo.[2]
Como hemos visto, es más largo y complicado explicar esto que todo lo anterior, razón por la cual hemos elegido este camino. Se invita a los lectores disidentes a modificar el código de la clase a su antojo, si así los motiva su disidencia.

1 Incluso también la aplicación en computadora podría saber cuántos bytes recibirá y pedir una transacción de esa longitud, evitando todo el problema.
2 Hemos comprobado esto por ejemplo con X-CTU 5.2.7.5 (Digi) en Win XP SP2; no así con Docklight 1.9 que parecía obtener siempre todo inmediatamente.