Revisión	Fecha	Comentario	Autor
0	25/10/2007		Ulises Bigliati

1. Objetivos:

Familiarizarnos con las herramientas de desarrollo *NET+OS 7.x* de Digi aproximándonos a su RTOS (real time operative system) basado en el kernel *ThreadX* de Express Logic .

Realizar la implementación básica del protocolo *1-Wire* para la lectura de los dispositivos de identificación *DS1990A* correspondientes a la familia *Id Only* de la línea de productos *iButton* de Dallas Semiconductors.

Fuera del alcance de esta nota queda toda explicación referente a la instalación y descripción de los diferentes componentes presentes en el kit de desarrollo de Digi (*Jump Start Kit*) que incluye todos los elementos necesarios para desarrollar sobre la plataforma propuesta. Puede encontrar información al respecto en el siguiente link: http://www.digi.com/products/embeddedsolutions/devkits/connectcore7udevkit.jsp

2. Introducción

Como plataforma de hardware empleamos el módulo *ConnectCore 7U* que presenta un microprocesador de 32 bits ARM7 TDMI @ 55Mhz, una memoria Flash de 2MB y 16MB de RAM e interfaz Ethernet entre otras características. Una descripción completa de éste módulo puede hallarse en el siguiente link: http://www.digi.com/products/embeddedsolutions/connectcore7u.jsp

El código del programa que proporciona la funcionalidad buscada está escrito en lenguaje ANSI C, y será desarrollado en forma modular dentro del entorno de desarrollo integrado Digi ESP que incorpora todas las herramientas de software NET+OS.

Como dispositivo de identificación se utilizaron los iButtons DS1990A, que gracias a una simple interfaz de conexión con la placa de desarrollo del módulo son leídos mediante la implementación por software del protocolo *1-Wire*. Puede encontrarse información detallada sobre los iButtons y sobre este protocolo en el siguiente link: http://www.maxim-ic.com/products/ibutton/ . Recomendamos la *App Note 937 Book of iButton Standards.*

Cabe mencionar que el propósito de la elección de éstas herramientas de desarrollo es puramente didáctico ya que las prestaciones que este módulo es capaz de proporcionar exceden ampliamente a los modestos requerimientos de la aplicación propuesta.

Nota: Si bien en esta ocasión trabajamos con el módulo CC7U, el mismo desarrollo será válido para ser aplicado a cualquier otro módulo de la línea embedded de Digi (por ejemplo CCW9C), ya que el entorno de desarrollo nos proporciona la abstracción necesaria para que la portabilidad sea posible.

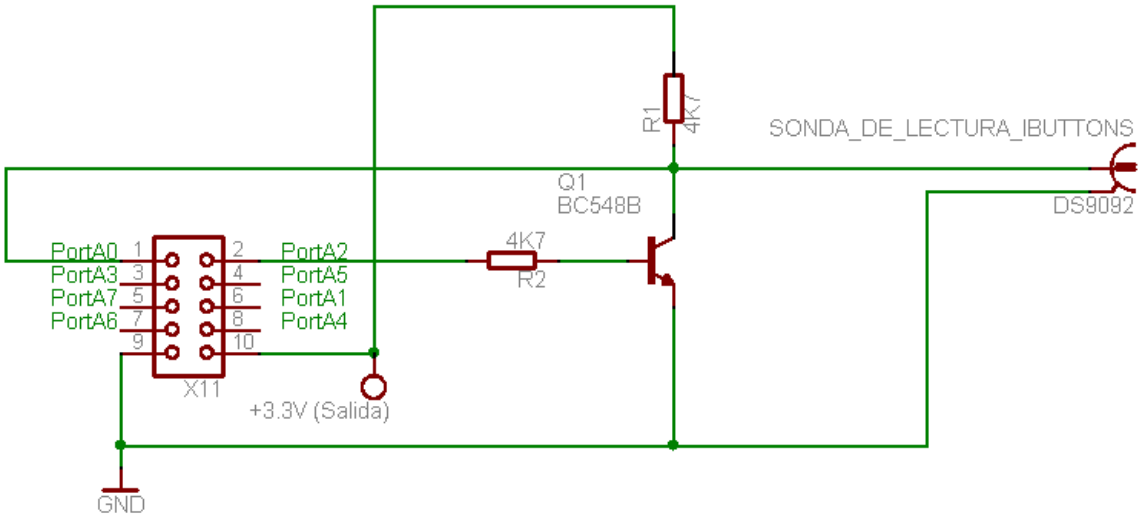
3. Implementación

Para el diseño de esta demostración necesitamos disponer de los siguientes componentes:

1. Placa de desarrollo *UNCBAS* para el módulo Digi *CC7U*. (incluida en el *Jump Start kit*)
2. El módulo Digi *ConnectCore 7U*. (incluido en el *Jump Start kit*)
3. El entorno de desarrollo integrado *Digi ESP* con *NET+OS 7.x* (incluido en el *Jump Start kit*)
4. Dispositivos de identificación *iButtons DS1990* (Producto de Dallas Semiconductors)
5. Hardware adicional para construir el bus 1-Wire:
    a. Sonda de lectura para *iButtons DS9092* (Producto de Dallas Semiconductors)
    b. Sencillo circuito de interfaz descrito a continuación. (sección 2.1)
6. Módulos de código escritos en ANSI C detallados en la sección correspondiente.

3.1. Hardware

Lo primero que debemos hacer es construir el sencillo circuito que se muestra a continuación.
El uso del transistor es necesario debido a que no contamos en nuestro microprocesador con puertos del tipo *Colector Abierto*, ya que si así fuera bastaría con un solo pin de entrada/salida y una resistencia de pull-up para conectar la sonda de lectura.
Como los puertos de nuestro controlador no son *Colector Abierto* debemos utilizar un pin como entrada y un segundo pin como salida para poder actuar sobre el bus *1-Wire*. En este caso utilizamos el *PORTA0* y *PORTA2* respectivamente.



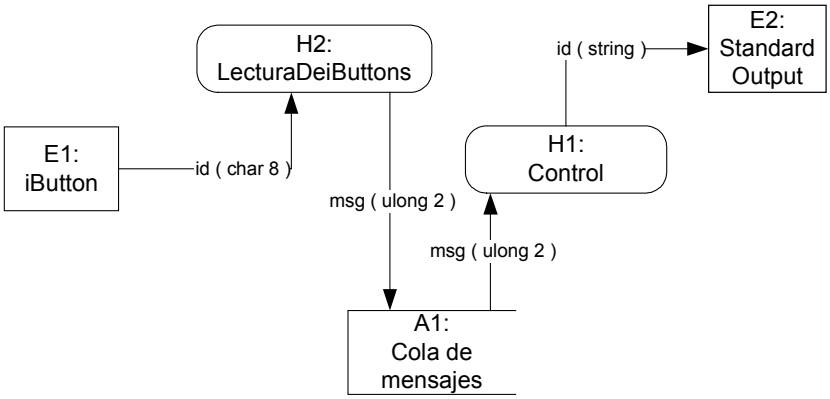
El conector X11 está presente en la placa de desarrollo del módulo Digi y como puede verse nos facilita el acceso a todos los pines de entrada/salida de uso general *(GPIO)* del *Port A* del microprocesador. Dicho conector también nos provee en sus pines 9 y 10 los contactos de masa y alimentación de 3.3V respectivamente.

3.2. El software

Entre las herramientas de desarrollo que provee *NET+OS* contamos con *ThreadX*, un sistema operativo de tiempo real del tipo *preemptive multitasking* basado en threads, que dispone de objetos como *colas*, *timers de aplicación*, *semáforos*, *flags de eventos*, etc., para la sincronización y comunicación entre procesos. Este sistema operativo está incluido en los *kits de desarrollo de Digi para NET+OS* y es además, libre de regalías. Escapa al alcance de esta nota la descripción de la arquitectura y funcionalidades del núcleo. Pero como una aproximación utilizaremos algunos de los mecanismos que están disponibles para el programador, en este caso *Colas de Mensajes* y *threads*.

3.2.1. DFD (diagrama de flujo de datos)

El esquema simplificado del funcionamiento del software puede apreciarse en el siguiente diagrama de flujo de datos (DFD):



3.2.2. Especificación de procesos del DFD:

3.2.2.1. Almacenamientos temporales de datos

Código: A1.
Nombre: Cola de mensajes.
Tipo: Objeto del sistema operativo(*TX_QUEUE*).
Objeto de implementación: myQueue en módulo *root.c*.
Función: que permite la intercomunicación entre procesos mediante el paso de mensajes.
Descripción: cola de intercomunicación de 128 bytes que contiene 16 mensajes de 2 words de 32-bit c/u (8 bytes)

3.2.2.2. Entidades Externas

Código: E1.
Nombre: iButton.
Tipo: Dispositivo de Hardware.
Objeto de implementación: *iButton DS1990A*
Función: Medio de identificación utilizado para registrarse en el equipo.
Descripción: Consistente en un número de serie de 6 bytes, precedido por un byte que identifica la familia del dispositivo y seguido por 1byte de CRC.

Código: E2.
Nombre: Standard Output.
Tipo: Salida por pantalla.
Objeto de implementación: Consola de comunicación serial del entorno de desarrollo.
Función: Desplegar los mensajes de lectura de *iButtons* provenientes de la aplicación.
Descripción: Comunicación serial desde la placa de desarrollo a la PC .

3.2.2.3. Procesos

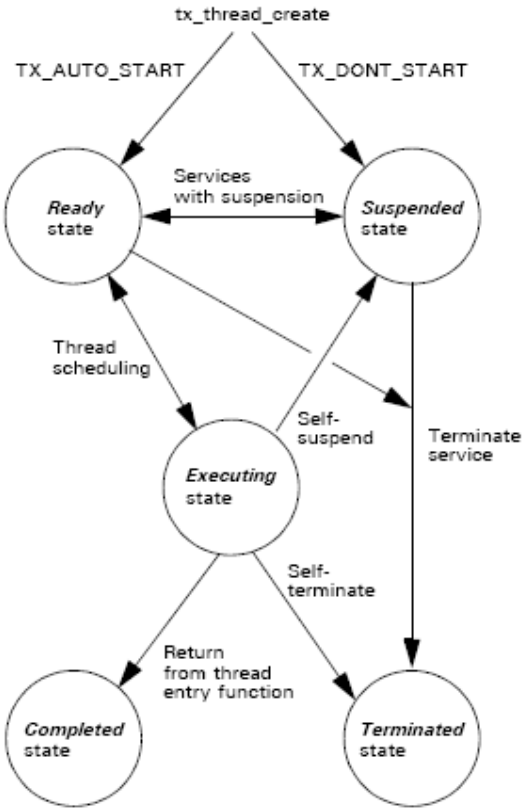
Código: H1.
Nombre: Control
Tipo: Thread (*TX_THREAD*).
Objeto de implementación: *pivot_thread* en módulo *root.c*
Función: Actúa como objeto de control, y coordinador entre tareas del sistema.
Descripción: verifica la existencia de mensajes en cola y los gestiona si estos existen.

Código: H2.
Nombre: LecturaDeiButtons.
Tipo: Thread (*TX_THREAD*).
Objeto de implementación: "*ib_thread*" en módulo *root.c*
Función: Tarea de lectura de *iButton*.
Descripción: verifica en forma recurrente la presencia del medio de identificación en el puerto de entrada del dispositivo.

3.2.3. Procesos subyacentes del Sistema Operativo

Además de los procesos de nuestra aplicación, por supuesto existen procesos de scheduling del S.O. que son transparentes al usuario y que le dan consistencia al mecanismo basado en threads.
Estos procesos conforme a la característica *preemptive* del S.O. administran el tiempo de procesador entre los diferentes procesos de nuestra aplicación alterando sus estados (*Ready, Suspended, Executing, Completed y Terminated*) según sea conveniente, conforme a un esquema de *Round Robin*, controlado por prioridades, y política FIFO, y adicionalmente por la metodología de *Time-Slicing.*

3.2.3.1. Esquema de transición de estados de los threads



3.2.4. El código

Finalmente llegamos a la descripción de nuestro código (el cual está disponible para ser utilizado en los archivos que acompañan la nota). La aplicación está físicamente implementada en tres archivos:

Módulo	Descripción
`root.c`	Es el módulo principal de una aplicación desarrollada sobre NET+OS. Dentro de este módulo se encuentra la función `applicationStart()` que es el punto de inicio del código del usuario.
`iButton.h`	Encabezado conteniendo las constantes y declaraciones de funciones relativas a la lectura de *iButtons.*
`iButton.c`	Modulo que contiene las definiciones de las funciones declaradas en header, mas el código de mas bajo nivel para el acceso al bus *1-Wire*.

A continuación examinaremos funcionalmente el código contenido en los archivos mencionados.

3.2.4.1. Modulo `root.c`

Un breve comentario sobre cada library que es incluida en este, el módulo principal:

Estas libraries no requieren presentación:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

Funciones del S.O (Administración de threads, colas, etc.)

```
#include <tx_api.h>
```

Capa de abstracción del hardware.

```
#include "bsp_api.h"
```

Define que servicios adicionales estarán presentes para nuestra aplicación (Servidor WEB, FTP, etc.)

```
#include <appservices.h>
```

Define parámetros de comportamiento globales para nuestra aplicación (Parámetros de red, de inicialización, etc.).

```
#include "appconf.h"
```

Define constantes para utilización de puertos de I/O

```
#include "gpio.h"
```

A continuación incluimos el header del módulo que contiene todo el código necesario para leer los *iButtons* mediante el bus *1-Wire*:

```
#include "ibutton.h"
```

Declaramos los threads que vamos a usar, ya habíamos definido estos bloques en el DFD (Sección 3.2.1.) cada thread constituirá una tarea autónoma e independiente:

```
TX_THREAD ib_thread ;
TX_THREAD pivot_thread;
```

Se declaran las funciones que serán el punto de entrada para cada tarea. Mediante estas funciones el sistema operativo y también la aplicación pueden manipular a los threads y es en el cuerpo de estas funciones donde escribimos el código que define el comportamiento de la tarea que está representada por el thread en cuestión.

```
void ib_thread_entry();        //Implementa el proceso H2 del DFD (Sección 3.2.1.)
void pivot_thread_entry();     //Implementa el proceso H1 del DFD (Sección 3.2.1.)
```

Declaramos la cola de mensajes que servirá de objeto de intercomunicación entre tareas de acuerdo a lo definido en el DFD.

```
TX_QUEUE my_queue;        // Implementa el almacén de datos A1 del DFD (Sección 3.2.1.)
```

En este punto tenemos declarados los objetos principales de la aplicación en correspondencia con el esquema planteado en la sección 3.2.1.

La función applicationStart() es el punto de entrada de la aplicación del usuario. Esta función será llamada luego de que el kernel ThreadX, los drivers de dispositivo NET+OS, y el stack TCP/IP estén en funcionamiento.
Esta rutina es responsable de iniciar la aplicación del usuario, es aquí donde se deben crear todos los threads y demás recursos que nuestra aplicación necesite.

```
void applicationStart (void){
```

Inicializa los servicios de sistema para la aplicación.

```
    initAppServices();
```

A continuación utilizamos las funciones de configuración para GPIO. Debemos especificar los pines que conformarán el bus 1-Wire. Necesitamos un pin de entrada y un pin de salida. Para hacer referencia a los pines utilizamos las constantes que figuran en *gpio.h*.
En este caso vamos a utilizar el PORTA0 para entrada y el PORTA2 para salida.

```
    //iButton IN
    NAconfigureGPIOpin( BSP_GPIO_PIN_A0, NA_GPIO_INPUT_STATE,1);

    //iButton OUT
    NAconfigureGPIOpin( BSP_GPIO_PIN_A2, NA_GPIO_OUTPUT_STATE,0);
```

Ahora creamos e inicializamos la cola de mensajes que ya habíamos declarado. Para esto NET+OS nos provee la función `tx_queue_create()`. A la cual debemos pasarle los siguientes parámetros:

1. La referencia a la cola que vamos a utilizar (*my_queue*)
2. Un string de caracteres con el nombre simbólico de la cola ("*my_queue_name*")
3. El tamaño del mensaje, siendo la unidad de medida el *unsigned long*, en este caso *TX_2_ULONG*.
4. Un puntero a la zona de memoria reservada para la cola, obtenemos esto con *malloc(128)* (espacio total para 128 bytes)
5. Por último le indicamos el tamaño total de la cola en bytes, en este caso 128.

Como nuestro mensaje mide 2 words de 32-bit , tenemos una longitud de mensaje de 8 bytes, que es lo que necesitamos para guardar la lectura de un *iButton*. En 128 bytes podríamos contener entonces como máximo 16 mensajes.

```
unsigned int status;
status = tx_queue_create( &my_queue,"my_queue_name",TX_2_ULONG,
                          malloc(128),128);
if (status==TX_SUCCESS)
      printf("Cola de mensajes creada con éxito\n");
```

Ahora nos falta crear los threads que van a implementar físicamente nuestra representación lógica de las tareas que definimos en el esquema de la sección 3.2.1.
Genéricamente la llamada a la función de creación de un thread esta constituida de la siguiente manera:

```
tx_thread_create (
                        thread_ptr,
                        name_ptr,
                        entry_function,
                        entry_input,
                        stack_start,
                        stack_size,
                        priority,
                        preempt_threshold,
                        time_slice,
                        auto_start
                );
```

Donde :

Name	Description
thread_ptr	Puntero a la ubicación del thread declarado en el encabezado.
name_ptr	Puntero a un nombre simbólico del thread.
entry_function	Es la función inicial para la ejecución del thread. Cuando un thread retorna de esta función, es colocado en estado *Completo* y su ejecución se suspende indefinidamente.
entry_input	Un valor de 32-bits que es pasado a la función de entrada del thread cuando es ejecutado por primera vez. Puede ser usado libremente como parametrización de alguna función del thread.
stack_start	Puntero a la dirección de inicio del área de memoria que va a contener la pila para este thread.
stack_size	Número de bytes para la pila. El área de pila para un thread debe ser lo suficientemente grande para albergar las variables locales y para poder manejar el peor caso de anidamiento de llamadas a función. Digi recomienda un mínimo espacio de pila de 8kbytes.
priority	Valor de prioridad del thread entre 0 y 31, donde 0 representa la prioridad mas alta.
preempt_threshold	Nivel de prioridad a partir del cual el thread cede el control por *preemption*. Solo a threads con valores de prioridad mas alta que este nivel se les permite "ganarle el control" a este thread. Este valor deber ser menor o igual que la prioridad especificada. Típicamente, se utiliza el mismo valor de que el especificado para la prioridad del thread.
time_slice	Número de timer-ticks durante los cuales que éste thread tiene permitido ejecutarse sin verificar si hay algún otro thread de la misma prioridad listo para ejecutarse. Threads en estado *Listo* con prioridades iguales o menores que el valor especificado en *preemption threshold* están con posibilidades de ejecutarse cuando el time-slice expira. El rango va desde 1 hasta 0xFFFFFFFF. Un valor de *TX_NO_TIME_SLICE* (un valor de 0) deshabilita el *time-slicing* para este thread.
auto_start	Especifica si el thread creado se iniciará automáticamente o si por el contrario va a permanecer en estado *Suspendido*: • *TX_AUTO_START* • *TX_DONT_START*

Creamos el thread para la lectura de *iButtons:*

```
status = tx_thread_create (     &ib_thread,
                                "IButton Thread",
                                ib_thread_entry,
```

```
                                    0,
                                    malloc (8192),
                                    8192,
                                    APP_DEFAULT_API_PRIORITY,
                                    APP_DEFAULT_API_PRIORITY,
                                    1 ,
                                    TX_AUTO_START);

      if (status==TX_SUCCESS)
            printf("Hilo de lectura de iButtons creado con éxito\n");
```

Y creamos el thread que oficiará como proceso de control :

```
    status = tx_thread_create (     &pivot_thread,
                                    "Pivot Thread",
                                    pivot_thread_entry,
                                    0,
                                    malloc (8192) ,
                                    8192,
                                    APP_DEFAULT_API_PRIORITY,
                                    APP_DEFAULT_API_PRIORITY,
                                    1,
                                    TX_AUTO_START);
      if (status==TX_SUCCESS)
            printf("Hilo de control creado con éxito\n");

}
```

Acá finaliza el cuerpo de la función *applicationStart()* y comienza la definición de los threads y del resto de las funciones que fueran necesarias.
El código correspondiente a la función de entrada del thread para la lectura de los iButtons es el siguiente:

```
      void ib_thread_entry (){
            Vector para guardar localmente la lectura:
            unsigned char ns[8];
            Vector para armar el mensaje para la cola:
            unsigned long msg[2];
            int i;
            A partir de acá el thread entra en un loop infinito de interrogación del bus 1-Wire, que solo es
            interrumpido por el S.O.:
            while(1){
                  for(i=0;i<8;i++)
                        ns[i]=0;
                  Si ibReset() detecta un iButton en el bus…
                  if (ibReset()==IB_READY){
                        entonces llamamos a ibRead(), y si la lectura estuvo ok...
                        if ( ibRead(ns)==IB_LECTURA_OK ){
                              entonces encolamos el mensaje y seguimos con lo nuestro...
                              memcpy(msg, ns, 8);
                              tx_queue_send(&my_queue, msg, TX_NO_WAIT);
                              (Lo que sigue es solo un retardo para evitar múltiples lecturas dentro de lo
                              que queda del time_slicing del thread)
                              tx_thread_sleep(50);
```

```
                }
                else{
                        Si la lectura reportó un error, probablemente de CRC, podemos hacer algo
                        acá o simplemente ignorar.
                }
            }
        }
        return;
}
```

Punto de entrada del thread de control de la aplicación. En este caso solo revisa la cola buscando mensajes, si encuentra uno, lo muestra por el standard output y continúa con su labor.

```
void pivot_thread_entry (){
        unsigned long msg[2];
        unsigned int status;
        char ns[8];
        int i;
```

Acá también entramos en un loop infinito, que solamente es interrumpido por la planificación del S.O.

```
        while(1){
```

En este loop consultamos recurrentemente la cola buscando mensajes de lectura de iButtons:
A la función que sigue le pasamos como argumentos un puntero con la dirección de la cola, un puntero al vector donde guardamos localmente el mensaje y un último argumento llamado *wait_option* que define el comportamiento del thread en esta operación según se describe a continuación:

wait_option	Define cómo actuar si la cola está vacía según las siguientes opciones: *TX_NO_WAIT* (0x00000000) Retorna inmediatamente sin importar si hubo éxito o no en la búsqueda de mensajes. *TX_WAIT_FOREVER* (0xFFFFFFFF) provoca que el thread llamador se suspenda indefinidamente hasta que llegue algún mensaje. timeout_value (0x00000001-0xFFFFFFFE) Especifica el máximo número de timer-ticks de suspensión mientras se espera un mensaje.

```
status=tx_queue_receive(&my_queue, msg, TX_NO_WAIT );
```

Si status= TX_SUCCESS es porque encontramos una lectura en cola...

```
if (status==TX_SUCCESS){
```

Entonces hay que extraerlo:

```
        memcpy(ns,msg, 8);
```

Y luego se muestra el Id por el standard output con un formato conveniente y se continúa dentro del loop.

```
        printf("Ibutton n° ");
        for(i=0;i<8;i++)
                printf("%02X",ns[i]);
```

```
                        printf("\n");
                }
        }
        return;
}
```

3.2.4.2. Modulo `iButton.h`

El módulo contiene los prototipos de la funciones y constantes necesarias para leer los Id´s de los iButtons Dallas Semiconductors (en este caso los 1990A).

```
#ifndef IBUTTON_H_
#define IBUTTON_H_
```

Constantes utilizadas como valor de retorno de las funciones declaradas en este módulo:

```
#define IB_LECTURA_OK         1
#define IB_LECTURA_ERROR      0
#define IB_READY              1
#define IB_NOT_READY          0
```

Se definen los pines que se utilizarán para IB_OUT e IB_IN . Las constantes BSP_GPIO_PIN_Xn están definidas en el archivo gpio.h que fue comentado mas arriba.
En este caso tomamos la lectura por defecto en el puerto A:
A0=Entrada A2=Salida . (Pines 1 y 2 respectivamente en el conector X11 de la placa UNC20.)

```
#define IB_OUT                      BSP_GPIO_PIN_A2
#define IB_IN                       BSP_GPIO_PIN_A0
```

La función *ibReset()* genera el ciclo de reset del bus *1-Wire* e indica si hay un *iButton* listo para ser leído en el bus. Retorna *IB_READY* (valor 1) si hay respuesta de un *iButton* o *IB_NOT_READY* (valor 0) si no hay i*Buttons* para leer.

```
unsigned int ibReset(void);
```

La función *ibRead()* debe llamarse a continuación de *ibReset()*, solo si ésta última devolvió *IB_READY* por haber detectado un iButton en el bus.
Devolverá *IB_LECTURA_ERROR* (valor 0) si falló la lectura o *IB_LECTURA_OK* (valor 1) si se obtuvo una lectura válida, y en este caso, el número de Id se encontrará como un string de ocho caracteres en la dirección indicada por el puntero a carácter que debe pasarse como argumento.

```
unsigned int ibRead(unsigned char * ns);


#endif /*IBUTTON_H_*/
```

3.2.4.3. Modulo `iButton.c`

Dentro de este módulo se encuentran las definiciones de las funciones necesarias para leer el Id de los iButtons de Dallas Semiconductors. Los detalles sobre el protocolo que estamos implementando pueden profundizarse en la documentación provista por Dallas Semiconductors.

Las funciones que nos provee NET+OS para escribir y leer pines individuales son respectivamente las que se comentan a continuación:

a) *int NAsetGPIOpin(unsigned int channel, unsigned int state);*
b) *int NAgetGPIOpin(unsigned int channel, unsigned int * state);*

chanel es una constante definida en *gpio.h* de la *forma BSP_GPIO_PIN_Xn* donde X=Puerto (A,C) y n=pin (0,1..7)

En el caso a), *state* representa el estado lógico a escribir. En el caso b) se trata de un puntero a entero donde la función depositará el valor leído en el pin indicado por *chanel*.

La primer función que vamos a describir es *ibReset()* que genera el ciclo de reset (Fig.1) sobre el bus 1-Wire y retorna *IB_READY(1)* si el *presense pulse* es detectado o *IB_NOT_READY (0)* en caso contrario.

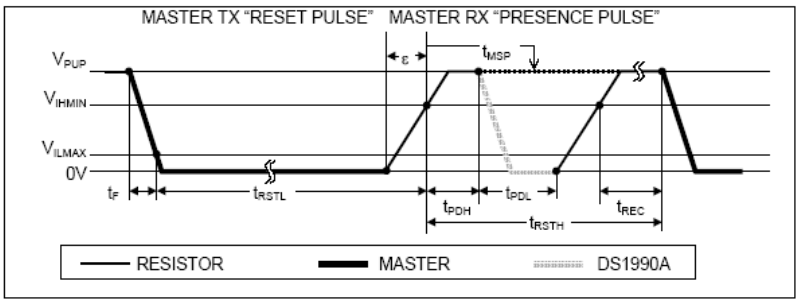


Figura1

```
unsigned int ibReset(void){
      unsigned int result,i;
```

Bajamos la línea para generar el pulso de reset (notar que lo hacemos con un "1" ya que la salida está conectada a la base de un transistor)

```
      NAsetGPIOpin(IB_OUT, 1);
      usDelay(450);                 //  ~= 488 us
```

Luego liberamos el bus...

```
            NAsetGPIOpin(IB_OUT, 0);
            usDelay(60);                    //  ~= 70 us
```

Y esperamos la respuesta del iButton :

```
            for(i=0;i<200;i++){
               NAgetGPIOpin(IB_IN, &result);
               if (!result)
                        goto tpdl_ok;
            }
```

Si el bucle finaliza sin respuesta, entonces no hay ningún iButton en el bus y retornamos sin novedad:

```
            return IB_NOT_READY;
```

Si llegamos acá es porque algún iButton respondió al reset, a continuación esperamos el *recovey pulse* y retornamos con la indicación adecuada según se detecte o no la señal esperada.

```
      tpdl_ok:
            for(i=0;i<200;i++){
                  NAgetGPIOpin(IB_IN, &result);
                  if (result){
                        usDelay(450);
                        return  IB_READY;
                  }
            }
            return IB_NOT_READY;
}
```

Hasta acá sabemos que hay un dispositivo inicializado en el bus esperando ser leído, entonces ahora la función *ibRead()* estará encargada de esta tarea, para esto necesitará la asistencia de otras funciones de mas bajo nivel que definiremos en su momento, a saber:

IbByteWrite():
Escribe ocho bits en el bus, necesitamos escribir un comando para que el dispositivo nos devuelva su número de serie, para nuestro caso el comando es el 0x33.

IbByteRead():
Lee ocho bits del bus y lo devuelve en un char.

crccheck():
Realiza el cálculo de CRC de la lectura obtenida y devuelve el valor de CRC en un char.

La función *ibRead()* debe ser llamada a continuación de un ciclo de reset exitoso (*ibReset=IB_READY*) y se le debe pasar como parámetro un puntero a caracter, a partir del cual depositará los 8 bytes de una lectura válida del *iButton* (Ver Figura2)

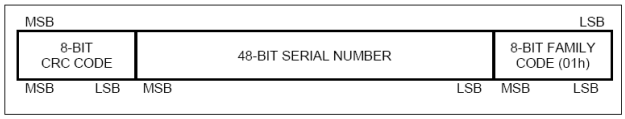


Figura 2

Previamente a devolver el número de serie obtenido del dispositivo de identificación, esta función invoca a crccheck(), para comparar el valor devuelto por ésta con el valor de CRC leído. Si dicha comparación falla se descarta la lectura y retorna*: IB_LECTURA_ERROR*, caso contrario retorna *IB_LECTURA_OK*.

```
      unsigned int ibRead(unsigned char * ns){
            unsigned int result,i;
            unsigned char ibSerial[8];
```

```
            ibByteWrite(0x33);->
                  for(i=0;i<8;i++)
                        ibSerial[i]=ibByteRead();

            if (crccheck(ibSerial)==ibSerial[7]){
                  memcpy(ns, ibSerial, 8);
                  result= IB_LECTURA_OK;
            }
            else{
                  result= IB_LECTURA_ERROR;
            }
            return result;
      }
```

Finalmente tenemos las funciones de mas bajo nivel, comenzando por *ibByteRead()* que recolecta ocho bits individuales para devolver un byte completo, e *ibByteWrite()* que escribe ocho bits consecutivos correspondientes a un comando. Ambas se sirven de las funciones de bajo nivel ibBitRead() e ibBitWrite respectivamente.

```
unsigned char ibByteRead( void){
   uchar i;
   uchar result = 0;

   for (i = 0; i < 8; i++){
         result |= ibBitRead() << i;
   }
   return result;
}

void ibByteWrite( unsigned char command){
   uchar i;
      unsigned int bit=0;

   for (i = 0; i < 8; i++){
          bit= (command >>i)& 1;
     ibBitWrite(bit) ;
   }
}
```

La función ibBitRead() obtiene un bit desde el bus *1-Wire* generando los *time-slots* (Figura 3) en la forma adecuada para respetar los tiempos que define el protocolo.

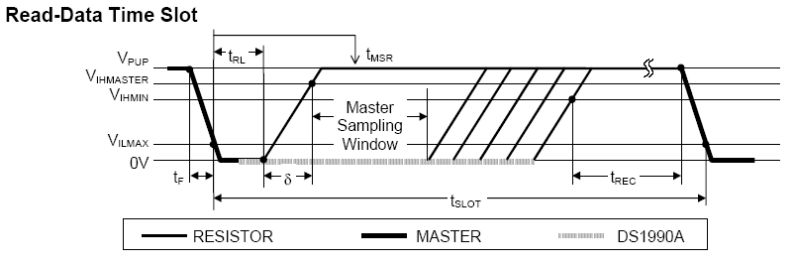


Figura 3

```
 unsigned int ibBitRead(void){
   unsigned int result;
```

```
        NAsetGPIOpin(IB_OUT, 1);        //Inicio time slot, Bajo la línea.
              //usDelay(0);             //Este retardo debe ajustarse a la velocidad del hardware
        NAsetGPIOpin(IB_OUT, 0);        //Suelto la línea liberando el bus para poder leer.
              //usDelay(0);             //Este retardo debe ajustarse a la velocidad del hardware
        NAgetGPIOpin(IB_IN, &result); //Supongo tener un dato valido

        usDelay(30);                    //Completo el time-slot
        return result;                  //Retorna el bit leido
}
```

La función ibBitWrite() escribe un bit pasado como parámetro en el bus *1-Wire* ajustándose a los tiempos definidos en el protocolo para generar los time-slots correctamente (Figura 4).

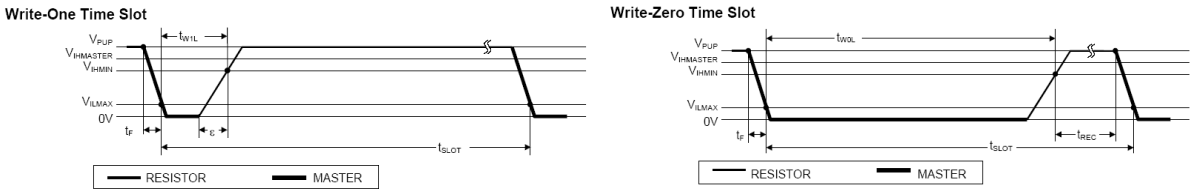


Figura 4

```
void ibBitWrite( unsigned int sendbit){
   if (sendbit){                        //ESCRIBE un UNO
      NAsetGPIOpin(IB_OUT, 1) ;         //Inicia el time-slot
          usDelay(1);
      NAsetGPIOpin(IB_OUT, 0) ;         //Escribe un 1
      usDelay(50)             ;         //Completa el time-slot
   }
   else{                                //ESCRIBE un CERO
         NAsetGPIOpin(IB_OUT, 1);       //Inicia el time-slot
         usDelay(60)           ;        //Completa el time-slot (el bus ya estaba en cero)
         NAsetGPIOpin(IB_OUT, 0);       //Sale dejando el bus libre
   }
}
```

A continuación aprovechamos la rutina de cálculo de CRC que nos facilita el fabricante. Dicha rutina simplifica el algoritmo de cálculo del CRC valiéndose del siguiente vector estático:

```
static uchar dscrc_table[] = {
        0, 94,188,226, 97, 63,221,131,194,156,126, 32,163,253, 31, 65,
      157,195, 33,127,252,162, 64, 30, 95,  1,227,189, 62, 96,130,220,
       35,125,159,193, 66, 28,254,160,225,191, 93,  3,128,222, 60, 98,
      190,224,  2, 92,223,129, 99, 61,124, 34,192,158, 29, 67,161,255,
       70, 24,250,164, 39,121,155,197,132,218, 56,102,229,187, 89,  7,
      219,133,103, 57,186,228,  6, 88, 25, 71,165,251,120, 38,196,154,
      101, 59,217,135,  4, 90,184,230,167,249, 27, 69,198,152,122, 36,
      248,166, 68, 26,153,199, 37,123, 58,100,134,216, 91,  5,231,185,
      140,210, 48,110,237,179, 81, 15, 78, 16,242,172, 47,113,147,205,
       17, 79,173,243,112, 46,204,146,211,141,111, 49,178,236, 14, 80,
      175,241, 19, 77,206,144,114, 44,109, 51,209,143, 12, 82,176,238,
```

```
 50,108,142,208, 83, 13,239,177,240,174, 76, 18,145,207, 45,115,
202,148,118, 40,171,245, 23, 73,  8, 86,180,234,105, 55,213,139,
 87,  9,235,181, 54,104,138,212,149,203, 41,119,244,170, 72, 22,
233,183, 85, 11,136,214, 52,106, 43,117,151,201, 74, 20,246,168,
116, 42,200,150, 21, 75,169,247,182,232, 10, 84,215,137,107, 53};
```

La función calcula el CRC de la cadena de caracteres señalada por el puntero pasado como argumento y devuelve un tipo char conteniendo el valor del CRC calculado.

```
unsigned char crccheck(char *serie){
      unsigned char crc8=0;
      unsigned char i;

      for(i=0;i<7;i++){
            crc8 = dscrc_table[crc8 ^ serie[i]];
      }
      return crc8;
}
```

La última función del modulo es simplemente una rutina de retardo utilizada para cumplir con los tiempos exigidos por el protocolo.

```
void usDelay (unsigned int us){
      unsigned int i;
      for (i=0;i<us;i++);
}
```