Nota de Aplicación: CAN-101
Título: **HT6P20x2 Encoder para aplicaciones remotas de control**
Autor: Ing. Iván C. Sierra

Revisiones	Fecha	Comentarios
0	28/01/13	

En esta oportunidad le presentamos un nuevo encoder, el HT6P20x2. Un encoder que cuenta con una dirección interna de hasta 22bits, grabada dentro del chip en fabrica, y una longitud de datos de hasta 5bits. La longitud de la dirección interna de estos enconders es muy superior a la de los viejos enconders HT12. Por ejemplo, para el caso del encoder HT6P20B2 que cuenta con 22bits de dirección, se tiene 4.194.304 combinaciones posibles contra las 256 combinaciones del HT12 (8bits). Como se puede ver, el número de combinaciones del HT6P20x2 es muy superior a la del HT12 y por éste motivo, se reduce notablemente las probabilidades que 2 controles remotos con la misma dirección se encuentren dentro de la zona de alcance del receptor. Logrando de esta manera incrementar el nivel de seguridad.

Descripción del HT6P20x2

El HT6P20x2 es un encoder de la empresa Holtek, el cual es capaz de soportar hasta 22bits de direcciones y 5bits de datos. Puede trabajar en un amplio rango de tensión, que va desde 2V hasta 12V. Posee oscilador RC interno, lo cual incrementa la inmunidad al ruido.

A diferencia del HT12, en el cual la dirección se definía mediante el estado de 8 entradas, el HT6P20x2 no cuenta con pines para definir su dirección, la misma viene grada en el chip y no puede ser modificada. Gracias a esto, se tiene un chip de reducido tamaño con solo 8 pines.

El HT6P20x2 cuenta con hasta 5 entradas digitales denominadas (D0~D4). Todas las entradas cuenta con un resistor de pull-up interno.

Para reducir el consumo del chip, éste permanece “dormido” hasta que una o varias de sus entradas se activen, es decir, pasen de un estado lógico alto (“1”) a un estado lógico bajo (“0”). En ese momento el chip comienza el proceso interno de codificación. Como resultado de este proceso se obtiene en su salida (DOUT) un tren de pulsos conteniendo la siguiente información:

- Piloto: es un pulso cuya duración, la cual llamaremos λ (Lambda), es la que se debe utilizar al momento de la decodificación de la señal, en el receptor, para poder interpretar si el dato leído corresponde a un “0” o un “1”.
- Dirección: dirección interna del chip cuya longitud varia según el modelo del chip. Para HT6P20B2 la longitud es de 22bits, para el HT6P20D2 es de 20bits y para el HT6P20F2 es de 19bits.
- Dato: Estado actual de las entradas. La longitud del dato varia según el modelo del chip. Para HT6P20B2 la longitud es de 2bits, para el HT6P20D2 es de 4bits y para el HT6P20F2 es de 5bits.
- EndCode: Es una secuencia conformada por 4bits e indica el fin de la trama. La secuencia enviada es “0101”

El proceso de codificación se mantiene mientras el estado lógico bajo de la/s entrada/s se mantenga.

La dirección y el dato son enviados desde del bit menos significativo (LSB) al mas significativo (MSB). Por este motivo al decodificar los datos recibidos se deben invertir los bit a fin de recuperar la información correcta.

El diagrama de tiempo para determinar si se esta recibiendo un “0 ” o un “1” es el siguiente:

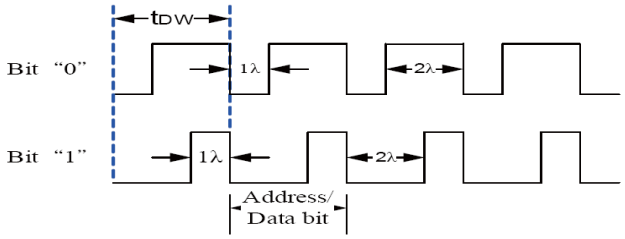


Figura 1.

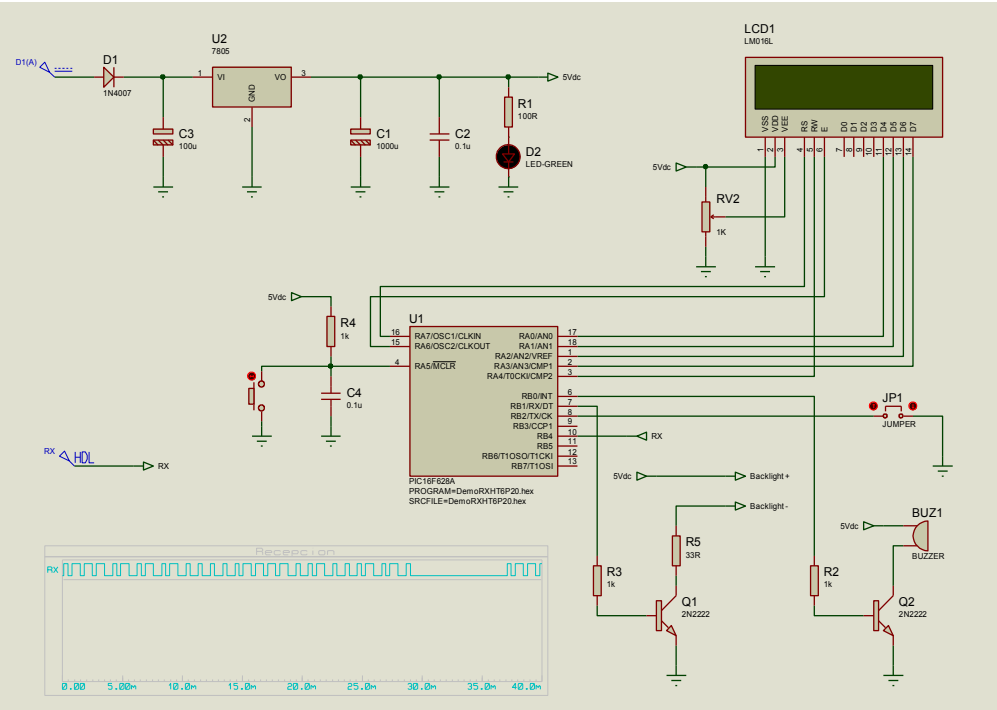
En el diagrama de la figura 1 se puede observar cual es el sentido del envió del pulso piloto y de la medición de su duración (λ), ya que cada bit de información tiene una duración de 3λ. El numero de intervalos λ en alto, de la señal, define cuando se esta recibiendo un "0" (1λ en bajo y 2λ en alto) o un "1" (2λ en bajo y 1λ en alto) por parte del transmisor.

La longitud del paquete enviado, por el encoder, es siempre la misma sin importar el modelo. Esto se debe a que si se aumenta el numero de bits de direcciones se reduce el numero de bits de datos. La siguiente tabla muestra la longitud de la dirección y datos para los 3 modelos de encoders de la familia HT6P20x2.

Encoder	Dirección	Dato	EndCode	Total de bits
HT6P20B2	22bits	2bits	4bits	28bits
HT6P20D2	20bits	4bits	4bits	28bits
HT6P20F2	19bits	5bits	4bits	28bits

Tabla 1.

Hardware de implementación para el decodificador.



El circuito del ejemplo cuenta básicamente con un microcontrolador PIC16F628A, un receptor de RWS-374 de 433,92MHz, un display alfanumérico de 16x2 con controlador compatible con el HD44780 y un control remoto RPD20402, el cual tiene como encoder al HT6P20D2.

En el diagrama no se incluyo el receptor RWS-374, sin embargo la salida de éste se conecta a RB4 (PIN10) del PIC16F628A

Software de implementación para el decodificador.

Ya explicada la forma de funcionamiento del HT6P20x2, se procede a explicar la implementación de un decodificador para el HT6P20x2 sobre un PIC16F628A, de la empresa Microchip. Se tomo esta marca microcontrolador para el ejemplo por tratarse de una marca muy conocida en el mercado local. Sin embargo, el ejemplo fue escrito en C con lo cual, con algunos cambios, se puede portar el código al microcontrolador de su agrado.

El ejemplo permite visualizar en un display, la dirección y el dato enviados desde un control remoto cuyo encoder sea el HT6P20B2 o el HT6P20D2. La selección del tipo de encoder se realiza mediante un jumper conectado a RB2. Con el jumper colocado el programa realiza la decodificación del HT6P20D2, en caso contrario, se realiza la decodificación del HT6P20B2.

La estructura de funcionamiento de la aplicación es bastante sencilla. Luego de inicializado el microcontrolador y el display, el programa se queda a la espera de la recepción de una nueva trama a decodificar. La detección y lectura de éste, se realiza mediante una interrupción por hardware por cambio de nivel en la entrada RB4 del microcontrolador, a la cual se encuentra conectada la salida del receptor RWS-374. Una vez que el microcontrolador detecta el inicio de una nueva trama, procede a decodificar la información. Para ello se mide, primeramente, el ancho de pulso λ, que luego es utilizado para determina si los siguientes pulsos recibidos corresponden a un “0” o un “1”. A medida que se van leyendo los datos recibidos, se cargan dentro de un buffer para luego procesarlos y poder mostrar la información obtenida en el display de manera legible al usuario.

La medición de los intervalos de tiempo en estado bajo y alto de la señal recibida se realizan mediante el TIMER1, el cual se incrementa cada 1uSeg al no utilizarse prescaler.

```
setup_timer_1 (T1_DIV_BY_1 | T1_INTERNAL);
```

El TIMER0 se utiliza como un timeout. Si no se produce ningún cambio en la entrada RB4 antes que el TIMER0 produzca un overflow, se considera que hay algún problema con la recepción y se comienza el ciclo de espera de una nueva trama nuevamente. El TIMER0 se configuro de manera que se produzca un overflow cada aproximadamente 1mSeg. (frecuencia del clock interno 4MHz y prescaler por 4).

```
setup_timer_0 (RTCC_DIV_4 | RTCC_INTERNAL);
```

A continuación se muestra la rutina que atiende a la interrupción por RB4.

```
void interrupt_Ext () {

   static unsigned int Lambdatime = 0;       // almacena el valor del ancho del pulso lambda
   static unsigned int PulseHighTime = 0;    // almacena el valor del ancho del pulso en alto
   static unsigned int PulseLowTime = 0;     // almacena el valor del ancho del pulso en bajo

   // maquina de estados para la lectura de los bits
   switch (RX.State) {
      case RX_IDLE:
         // se verifica que RB4 este en 1 para comenzar a medir LAMBDA
         if (input(RX_IN) == 1) {
            // habilita la interrupcion por TMR0
            set_timer0 (0x00);
            clear_interrupt (INT_TIMER0);
            enable_interrupts (INT_TIMER0);
            // inicializa el valor del tiempo en bajo, en alto y lambda
            Lambdatime = 0x0000;
            PulseHighTime = 0x0000;
            PulseLowTime = 0x0000;
```

```
      // borra el contador del TMR1
      set_timer1(0x0000);
      // setea el siguiente paso
      RX.State = RX_GET_LAMBDA;
   }
break;

case RX_GET_LAMBDA:
   // se verifica que RB4 este en 0 para terminar a medir LAMBDA
   if (input(RX_IN) == 0) {
      // resetea el TMR0
      set_timer0 (0x00);
      // carga el valor de lambda
      Lambdatime = get_timer1();
      // borra el contador del TMR1
      set_timer1 (0x0000);
      // borra los contadores de longitud de pulso
      PulseHighTime = 0x0000;
      PulseLowTime = 0x0000;
      // borra el contador de bit
      RX.CounterBits = 0x00;
      // setea el siguiente paso
      RX.State = RX_GET_PACKAGE;
   }
break;

case RX_GET_PACKAGE:
   // se verifica el estado del RB4 para ver el flanco
   if (input(RX_IN) == 1) {
      // resetea el TMR0
      set_timer0 (0x00);
      // carga el valor del pulso en bajo
      PulseLowTime = get_timer1();
      // borra el contador del TMR1
      set_timer1 (0x0000);
   }
   else {
      // resetea el TMR0
      set_timer0 (0x00);
      // carga el valor del pulso en alto
      PulseHighTime = get_timer1();
      // borra el contador del TMR1
      set_timer1 (0x0000);
   }

   // se verifica si ya se levanto un bit completo
   if ((PulseHighTime > 0) && (PulseLowTime > 0)) {
      // se analiza el resultado
      if (PulseLowTime > ((2 * Lambdatime) + LAMBDA_ERROR)){
         // deshabilita todas las interrupciones
         disable_interrupts(GLOBAL);
         // seta el resultado de la lectura
         RX.Result = RX_GET_ERROR;
      }
      else if (PulseLowTime < (Lambdatime - LAMBDA_ERROR)) {
         // deshabilita todas las interrupciones
         disable_interrupts(GLOBAL);
         // seta el resultado de la lectura
         RX.Result = RX_GET_ERROR;
      }
      else if (PulseHighTime > ((2 * Lambdatime) + LAMBDA_ERROR)) {
         // deshabilita todas las interrupciones
         disable_interrupts(GLOBAL);
         // seta el resultado de la lectura
         RX.Result = RX_GET_ERROR;
      }
      else if (PulseHighTime < (Lambdatime - LAMBDA_ERROR)) {
         // deshabilita todas las interrupciones
         disable_interrupts(GLOBAL);
         // seta el resultado de la lectura
         RX.Result = RX_GET_ERROR;
      }
      else {
         RX.CounterBits++;
         RX.Package <<= 1;
```

```
               RX.Package &= 0xFFFFFFFE;
               if (PulseLowTime > PulseHighTime)
                  RX.Package |= 1;
               // seta el resultado de la lectura
               RX.Result = RX_GET_OK;
            }
            // reseta los medidores de pulso
            PulseLowTime = 0x0000;
            PulseHighTime = 0x0000;
         }
      break;

      case RX_GET_PACKAGE_OK:
      break;

      case RX_GET_PACKAGE_ERROR:
      break;

      default:
         // deshabilita todas las interrupciones
         disable_interrupts(GLOBAL);
         // seta el resultado de la lectura
         RX.Result = RX_GET_ERROR;
      break;
   }
   // reseta el flag
   clear_interrupt(INT_RB);
}
```

nota: La constante LAMBDA_ERROR define un máximo de error tolerable en la medida del ancho del pulso. Su valor se obtuvo por prueba y se definió en 40uSeg.

La función que atiende la interrupción en RB4 solo lee los bits que van llegando y los almacena en un buffer. El procesamiento de esta información se realiza dentro de la función *main.* Allí realiza la verificación del *end code*, el parseado de los bits (dirección y datos) y presentación de la información recibida en el display. Para ello se controla el numero de bits recibidos. Al momento de llegar el contador a 28 indica que se recibió la trama completa y se procede a analizarla. Parte del código que realiza esta tarea se presenta a continuación.

```
.
.
.
// se parsean los datos
if (RX.Model == HT6P20B) {
   // se rotan los bits de la direccion para dejar el MSB primero
   long_buf_address1 = RX.Package >> (HT6P20X2_LEN_PKG - HT6P20B_LEN_ADDRESS);
   long_buf_address1 &= 0x003FFFFF;
   long_buf_address2 = 0;
   long_buf_address2 = long_buf_address2 | (long_buf_address1 & 0x00000001);
   for (i = 0; i < (HT6P20B_LEN_ADDRESS - 1); i++){
      long_buf_address2 <<= 1;
      long_buf_address1 >>= 1;
      long_buf_address2 = long_buf_address2 | (long_buf_address1 & 0x00000001);
   }
   RX.Address = long_buf_address2;

   // se rotan los bits de datos para dejar el MSB primero
   chr_buf_data1 = (char)(RX.Package >> 4);
   chr_buf_data1 &= 0x03;
   chr_buf_data2 = 0;
   chr_buf_data2 = chr_buf_data2 | (chr_buf_data1 & 0x01);
   chr_buf_data2 <<= 1;
   chr_buf_data1 >>= 1;
   chr_buf_data2 = chr_buf_data2 | (chr_buf_data1 & 0x01);
   RX.Data = chr_buf_data2;
}
else {
   // se rotan los bits de la direccion para dejar el MSB primero
   long_buf_address1 = RX.Package >> (HT6P20X2_LEN_PKG - HT6P20D_LEN_ADDRESS);
   long_buf_address1 &= 0x000FFFFF;
   long_buf_address2 = 0;
   long_buf_address2 = long_buf_address2 | (long_buf_address1 & 0x00000001);

   for (i = 0; i < (HT6P20D_LEN_ADDRESS - 1); i++){
      long_buf_address2 <<= 1;
```

```
      long_buf_address1 >>= 1;
      long_buf_address2 = long_buf_address2 | (long_buf_address1 & 0x00000001);
   }
   RX.Address = long_buf_address2;

   // se rotan los bits de datos para dejar el MSB primero
   chr_buf_data1 = (char)(RX.Package >> 4);
   chr_buf_data1 &= 0x0F;
   chr_buf_data2 = 0;
   chr_buf_data2 = chr_buf_data2 | (chr_buf_data1 & 0x01);
   for(i = 0; i < 3; i++){
      chr_buf_data2 <<= 1;
      chr_buf_data1 >>= 1;
      chr_buf_data2 = chr_buf_data2 | (chr_buf_data1 & 0x01);
   }
   RX.Data = chr_buf_data2;
}

// se convierte el dato recibido en ASCII para el display
LongToStrHex(RX.Address, RX.strAddress);
ByteToStrHex(RX.Data, RX.strData);

.
.
.

// se muestran los datos recibidos en el display
Lcd_PutC('\f');
sprintf(buffer_str_lcd, "Dir.:0x%s", RX.strAddress);
Lcd_OutStr(1, 1, buffer_str_lcd);
sprintf(buffer_str_lcd, "Dato:0x%s", RX.strData);
Lcd_OutStr(1, 2, buffer_str_lcd);
delay_ms(500);
```

nota: el compilador usado para el ejemplo es el CCS, en el cual la variable long se representa por medio de 16bits. Es por ello que se tuvo que usar la variable long long (32bits) para poder representar la dirección.