Cika Electrónica

Tutorial: CTU-012
Título: **XBee Programable**
Autor: Sergio R. Caprile, Senior Engineer

Revisiones	Fecha	Comentarios
0	16/02/11	
1	17/06/11	Agrega algunos periféricos del micro de aplicación

El presente tutorial tiene por objeto presentar la forma de utilizar el XBee PRO ZB Programable. Para información sobre la utilización de otros módulos XBee ZB, se sugieren los CTC-058 al 062.

Índice de contenido

Descripción del módulo

Los módulos XBee-PRO ZB Programables incluyen un procesador MC9S08QE32 (core HCS08) de Freescale además del EM250 de Ember. Este procesador adicional permite que el usuario pueda cargar en él el firmware de su agrado, dedicándolo a la tarea que necesite realizar. Trae cargado de fábrica un bootloader que permite actualizar el firmware del EM-250 (el XBee-PRO ZB en sí) y cargar una aplicación en el HCS08, mediante XMODEM, utilizando (por ejemplo) X-CTU.

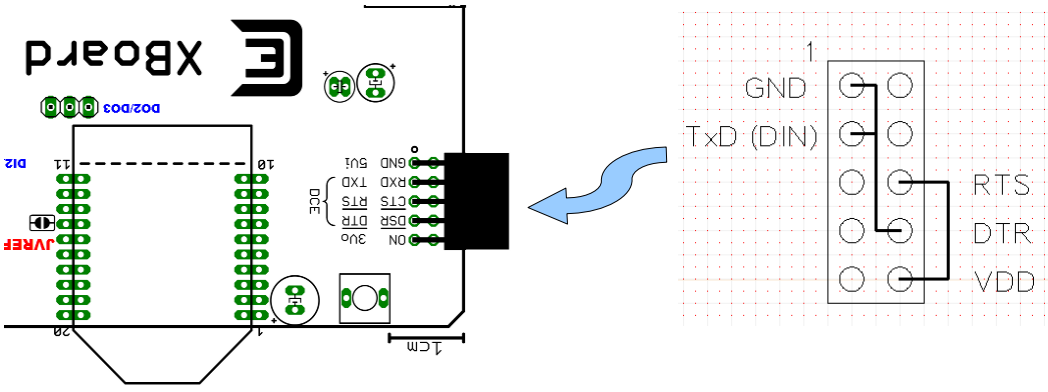
Utilización del bootloader

El bootloader que viene pre-cargado en el módulo es quien inicia el procesador, verifica la presencia de una aplicación, y la ejecuta si corresponde. Cuando el módulo está corriendo el bootloader (no tiene ninguna aplicación cargada o ésta devolvió el control al bootloader), enviando un retorno de carro (tecla <ENTER>) por su puerto serie a 9600bps, o bien desde otro XBee ZB, obtendremos el menú del bootloader:

```
B-Bypass Mode
F-Update App
T-Timeout
V-BL Version
A-App Version
R-Reset
>
```

Bypass Mode: El bootloader realiza una conexión virtual entre el puerto serie y el EM250, comportándose como un XBee PRO ZB hasta el siguiente reset.
Update App: Comienza el handshake del protocolo XMODEM en espera del envío de la aplicación.
App Version: Devuelve el texto que la aplicación eligió para identificarse.
Reset: Reinicia al procesador, si existe una aplicación cargada la ejecuta, sino permanece en el bootloader.

En el caso que la aplicación cargada no prevea la devolución del control del procesador al bootloader, es posible evitar el inicio de la misma mediante una combinación de señales en algunos de los pines de I/O al momento de aplicar alimentación al módulo: $\overline{RTS}=1, \overline{DTR}=0, DIN=0$.
En la placa XBoard, esto corresponde a insertar en el conector serie una tira de pines con el siguiente conexionado:



Dichos pines de I/O ya pueden volver a su estado normal (retirar el conector en la XBoard), pero el módulo no debe perder la alimentación pues de reiniciarse lo haría de forma normal.

Configuración y actualización de firmware del EM250

El módulo debe estar corriendo el bootloader, es decir, no debe tener una aplicación cargada, ésta debe tener una forma de escapar al bootloader, o se lo inicia mediante la combinación de pines de I/O.
En X-CTU, se selecciona la solapa *Terminal* para poder ingresar un <ENTER> y observar el menú del bootloader. En dicho menú, se selecciona la opción *B*, a partir de este instante el EM250 es accesible desde X-CTU como en cualquier otro módulo, con lo cual podemos realizar el upgrade de firmware y configurarlo como cualquier módulo.

Carga o actualización del firmware de aplicación del MC9S08QE32

El módulo debe estar corriendo el bootloader, es decir, no debe tener una aplicación cargada, ésta debe tener una forma de escapar al bootloader, o se lo inicia mediante la combinación de pines de I/O.

En X-CTU o cualquier programa de comunicaciones, seleccionamos la opción *F*. El bootloader inicia el handshake de XMODEM (veremos una serie de letras *C*) y debemos iniciar el envío del firmware mediante este protocolo, utilizando paquetes de 64 bytes. Si utilizamos X-CTU (recomendado), la opción *XModem* se encuentra en el menú cuando está seleccionada la solapa *Terminal* (que necesitamos para ingresar al menú del bootloader).

Terminada la carga, podemos consultar la versión de la aplicación mediante la opción *A*, y correrla mediante la opción *R*.

Desarrollo de aplicaciones en C

A continuación realizamos una breve descripción de los recursos de que disponemos para el desarrollo de aplicaciones en C.

Manejo de I/O del procesador

El entorno de desarrollo define los bits individuales como *bitfields* dentro de una *union*; es decir, es posible operar directamente sobre un bit en una sentencia C. La relación entre los pines de I/O del módulo XBee y los del procesador se realiza incluyendo el archivo *pin_mapping.h*.

```
#include "derivative.h"
#include "pin_mapping.h"
```

Los puertos de I/O tienen (al menos) un registro para controlar el estado, otro para seleccionar el sentido de comunicación, y otro para habilitar la resistencia interna de pull-up. La nomenclatura de este archivo utiliza *_D* para el registro de dirección (entrada/salida) y *_PE* para habilitación del pull-up (**P**ull-up **E**nable). Por ejemplo: operamos sobre $DIO8/\overline{DTR}$:

```
IO_DIO8_DTR=0;                // bajo
```

seleccionamos si $DIO7/\overline{CTS}$ es entrada o salida:

```
IO_DIO7_CTS_HOST_D=0;         // entrada
```

habilitamos el resistor de pull-up:

```
IO_DIO7_CTS_HOST_PE=1;        // habilitado
```

Comunicación con un host por puerto serie

Dicha comunicación utiliza la UART1 del MC9S08QE32. El módulo del procesador se denomina SCI1 (Serial Communications Interface). La inicialización la podemos realizar mediante una herramienta del entorno de desarrollo. Dado que se utiliza un clock interno del procesador, el fabricante del módulo sugiere limitar la velocidad de operación a 9600bps

Modo polled

Para una comunicación polled (sin interrupciones), podemos:
revisar si se recibió un dato:

```
if ( SCI1S1_RDRF )
```

leer el dato recibido:

```
dato = SCI1D;
```

revisar si es posible enviar un dato:

```
if ( SCI1S1_TDRE )
```

enviar un dato:

```
SCI1D = dato;
```

Mediante interrupciones

Si deseamos utilizar interrupciones, deberemos escribir los handlers correspondientes. Si utilizamos la herramienta de inicialización que se provee en el entorno de desarrollo, los handlers son definidos en el archivo MCU_init.c que se genera:

```
__interrupt void isrVsci1tx(void)
{
  /* Write your interrupt code here ... */

}
/* end of isrVsci1tx */


__interrupt void isrVsci1rx(void)
{
  /* Write your interrupt code here ... */

}
/* end of isrVsci1rx */
```

El interrupt handler para interrupciones de recepción, debe leer el registro de estado *SCI1S1* a modo de reconocimiento de la interrupción.

Comunicación con el EM250

Dicha comunicación utiliza la SCI2 del MC9S08QE32. Nuevamente, la inicialización la podemos realizar mediante una herramienta del entorno de desarrollo. En esta UART veremos lo que hayamos grabado y configurado en el EM250, es decir, es como si tuviéramos conectado un XBee PRO ZB. Por defecto, el firmware grabado es API y es lo que el bootloader espera. Es posible operar sobre un firmware AT si se prescinde del acceso remoto al bootloader o se lo modifica. La velocidad de operación es la que se configure en el EM250, y por defecto es de 9600bps. Se recomienda no excederla, y si se desea que el bootloader pueda comunicarse con el EM250, no cambiarla.

Firmware AT

Cargado el firmware AT, operaremos sobre la SCI2 de igual modo que como hiciéramos sobre la SCI1. Los registros son: *SCI2S1* y *SCI2D*.

Firmware API

Este es el firmware por defecto. Si bien tanto el bootloader como la aplicación de ejemplo que distribuye el fabricante tienen implementaciones de API, sugerimos la utilización de la implementación que se describe en CAN-089. Para ello, deberemos proveer dos funciones que realicen la interfaz entre las funciones de API y la SCI. Éstas son *TX()* y *RX()*, cuyas características se describen en CAN-089.

Modo polled

El siguiente listado corresponde a una implementación mínima:

```
#include "apiframe.h"
#include "apicommon.h"
#include "apizb.h"

int TX(unsigned char data)
{
        if (SCI2S1_TDRE)
                SCI2D = data;
        else
                return(-1);
        return(1);
}

int RX()
{
        if (SCI2S1_RDRF)
                return SCI2D;
        return(-1);
}
```

Mediante interrupciones

En caso de utilizarse interrupciones, el interrupt handler de recepción deberá colocar los datos en un buffer, del que serán extraídos por *RX()*; mientras que *TX()* colocará los datos en un buffer del que serán extraídos por el interrupt handler de transmisión.

Comunicación con el bootloader

La comunicación entre la aplicación y el bootloader se realiza mediante variables compartidas situadas en un área especial de RAM:

```
#include "common.h"
#include "sharedRAM.h"
```

La aplicación tiene la opción de devolver el control al bootloader si lo desea, indicando una causa de reset en una variable compartida. Dichas causas se definen en el archivo *common.h*:

APP_CAUSE_NOTHING: reset sin causa aparente. Valores entre 0 y 255 son ignorados por el bootloader y pueden ser utilizados por la aplicación.
APP_CAUSE_FIRMWARE_UPDATE: solicita al bootloader inicie un proceso de firmware update
APP_CAUSE_BYPASS_MODE: instruye al bootloader a que se coloque en modo bypass
APP_CAUSE_BOOTLOADER_MENU: instruye al bootloader a que no inicie la aplicación.

La cesión de control al bootloader se realiza dejando expirar el watchdog timer. Por ejemplo:

```
	for(;;){
		__RESET_WATCHDOG();
		if( quiero ) {
			DisableInterrupts;
			AppResetCause = APP_CAUSE_BOOTLOADER_MENU;
			for(;;);                                  // espera timeout de WDT
		}
		...
	}
```

Datos de la aplicación

El texto que indica el nombre y versión de la aplicación se indica al bootloader mediante variables compartidas, de la siguiente forma:

```
#pragma CONST_SEG APPLICATION_VERSION
static const uint8 version[] = "miprograma 1.0";

#pragma CONST_SEG DEFAULT
static const unsigned long pAppVersion @0x0000F1BC = (unsigned long)version;
```

Zonas de memoria y vectores de interrupciones

Para que la aplicación utilice las zonas de memoria que le corresponden, debemos indicárselo al linker del entorno de desarrollo. El archivo que realiza esta función se encuentra en el directorio *prm* y su nombre por defecto es *Project.prm*. Es posible definir otro archivo mientras se lo indiquemos al IDE.

Al final de dicho archivo encontramos la definición de los vectores de inicialización e interrupciones:

```
//VECTOR 0 _Startup     /* Reset vector: this is the default entry point for an application. */

	VECTOR ADDRESS 0x0000F1FE  _Startup    /*VECTOR ADDRESS 0x0000F1EA  vReset     */
```

A continuación, se agregarán entradas para los vectores de interrupciones que se utilicen, si corresponde (por ejemplo una SCI).

Herramientas de desarrollo

En esta sección veremos las opciones de desarrollo de que disponemos.

IDE

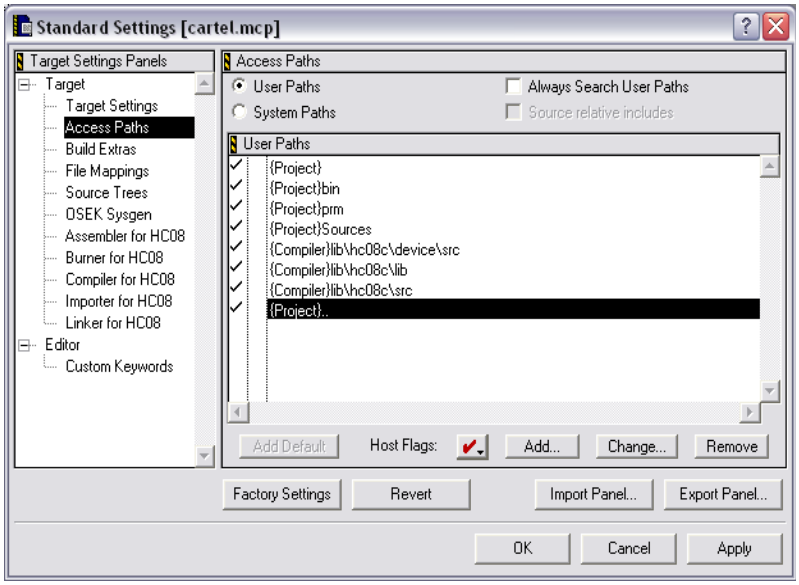
El entorno de desarrollo empleado es el Freescale Codewarrior for Microcontrollers 6.3 SE. Si bien existen versiones más recientes basadas en Eclipse, la información provista por Digi y todo lo realizado en este documento emplean la versión 6.3

La creación de un nuevo proyecto se realiza mediante una wizard que guía al usuario en los pasos a seguir. entre ellos están la elección de la herramienta de programación y depuración, y la inicialización del chip.

Los archivos de código fuente se ubican en el directorio *Sources* del proyecto.

Include files

Los archivos *.h* que compartimos entre varios proyectos, como por ejemplo los mencionados en este documento: *pin_mapping.h*, *common.h* y *sharedRAM.h*, pueden ubicarse a nuestro gusto. Para que el compilador los encuentre, debemos agregar el directorio a la lista de *Access paths*. La forma más simple es por lo menos agregar uno de los archivos en cuestión al proyecto, en la sección *Includes*, para que el IDE agregue automáticamente la ruta a la lista:

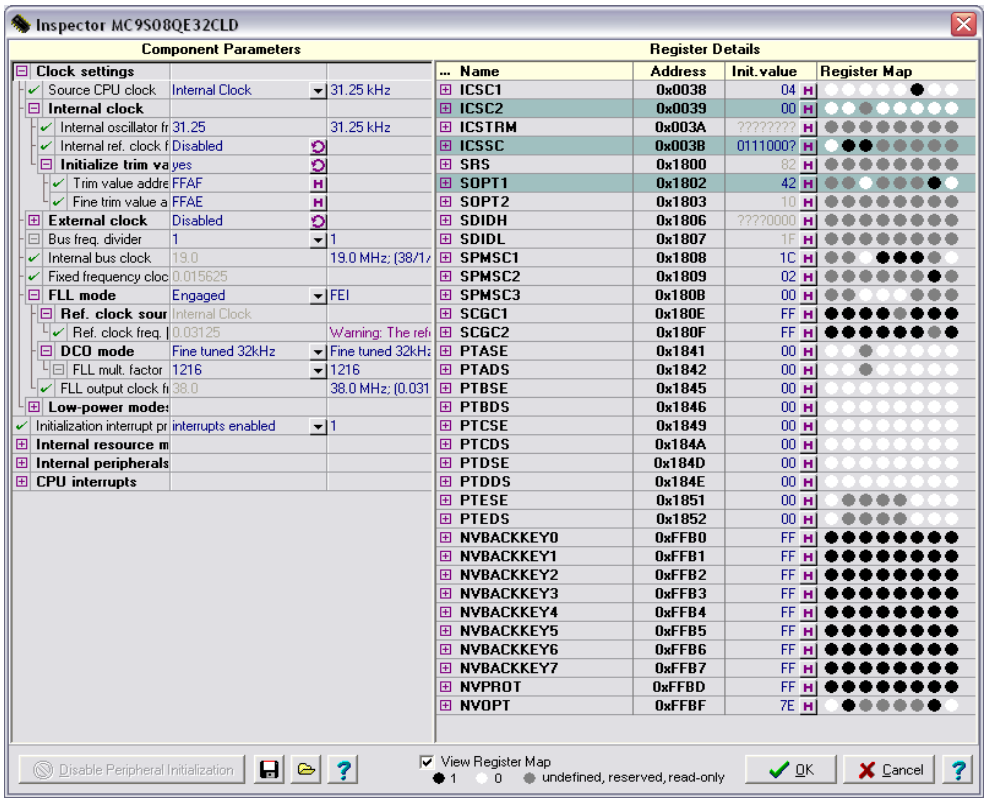


Processor Expert

Esta es la herramienta encargada de realizar la inicialización del chip. De forma gráfica se accede a los registros de control del micro y sus periféricos, y esta herramienta genera el código correspondiente. El IDE lo incluye automáticamente entre los fuentes disponibles. Si bien el micro es inicializado por el bootloader, la aplicación dispone de total control sobre éste y, excepto algunos registros que sólo pueden escribirse en el arranque, puede configurarlo a su agrado.

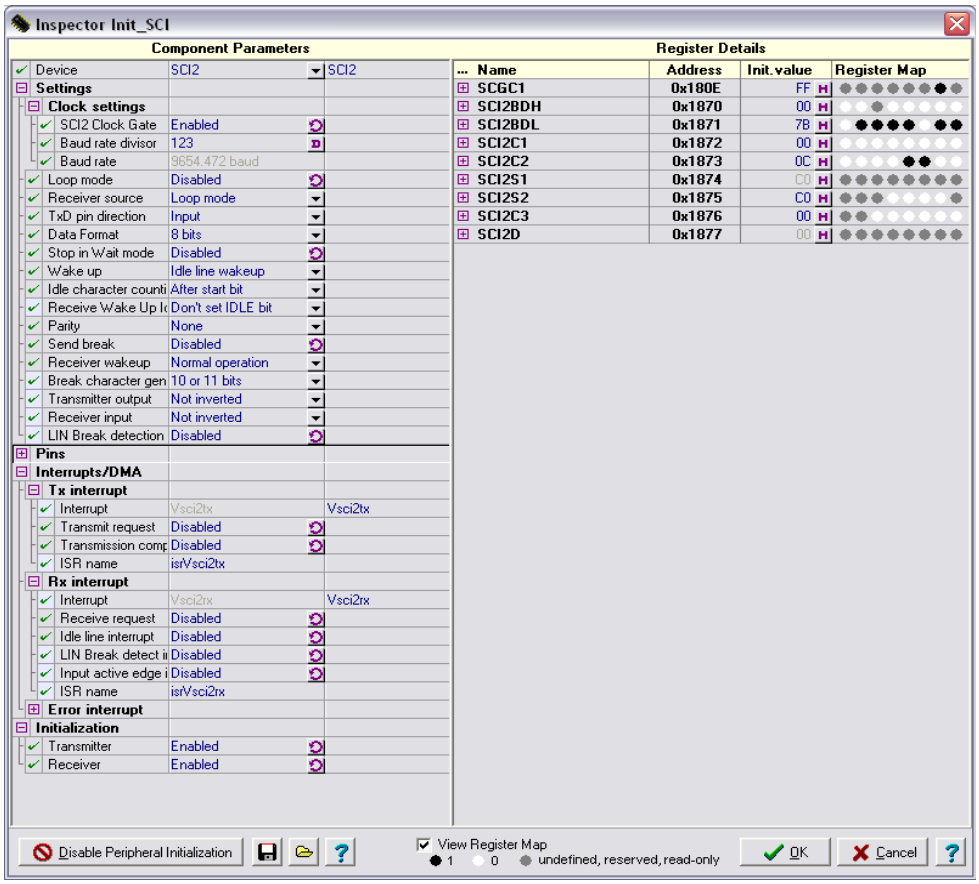
Clock

La captura siguiente muestra la configuración de clock para máxima velocidad:



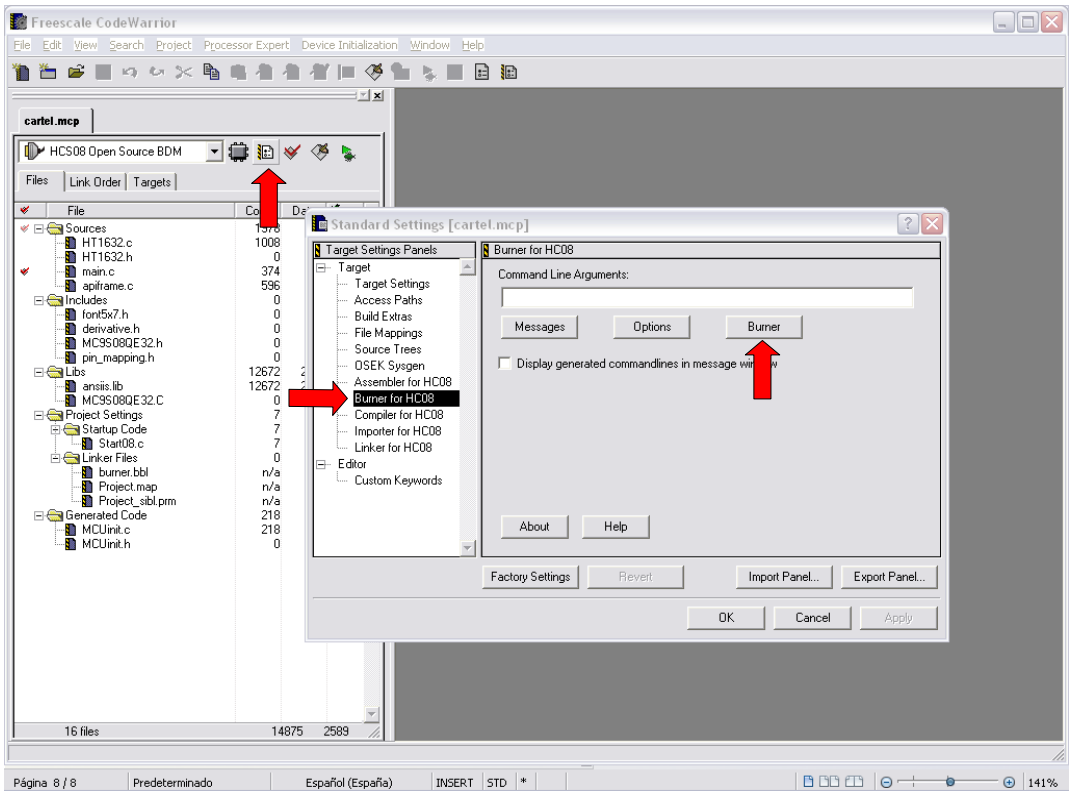
SCI2

La captura siguiente muestra la configuración de la SCI2 para operación en modo polled:

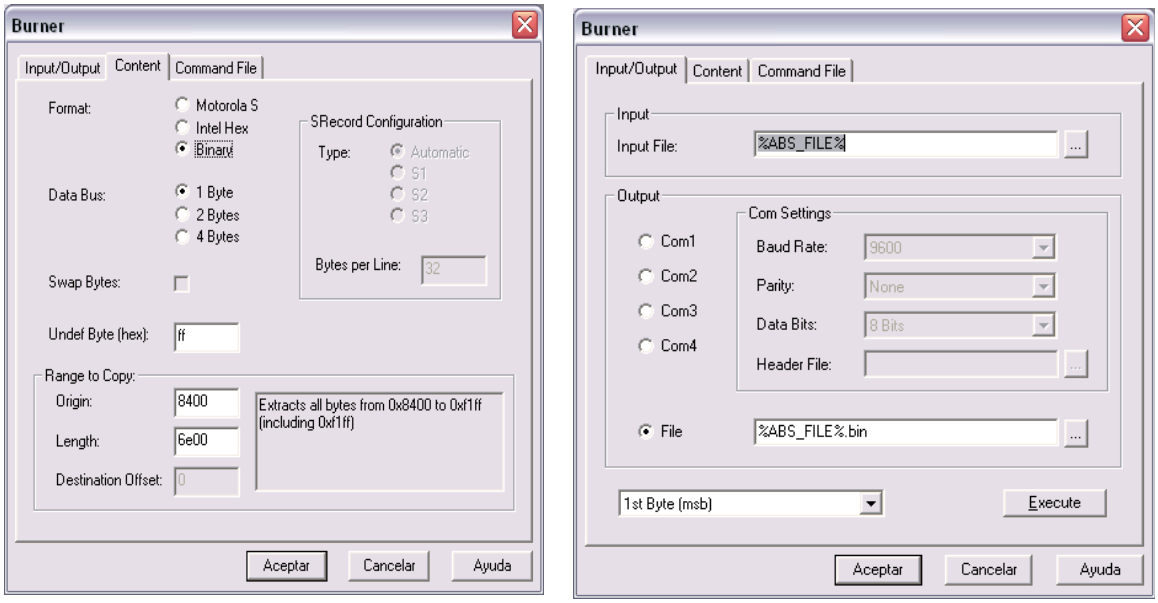


Generación del código

Una vez depurada la aplicación, generamos el código a ser cargado mediante XMODEM utilizando el *Burner*. Accedemos a él de una forma un poco compleja, como podemos ver en la captura siguiente:



Una vez allí, la instruimos para que genere binario por la totalidad del espacio de memoria disponible, ya que esto es lo que espera el bootloader:



El archivo así generado tendrá el nombre *Project.abs.bin*, y se encontrará en el directorio *bin* del proyecto. Éste es el archivo que debemos utilizar para realizar la carga de la aplicación por XMODEM en el producto final.

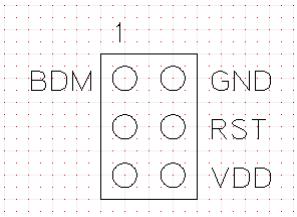
Programador-Debugger

Sin duda una herramienta de este tipo acelera el proceso de desarrollo y depuración del código. No obstante, si se desea conservar el bootloader, no es posible utilizarla para la grabación final de la aplicación, a menos que se incluya todo el código (bootloader y aplicación) en un proyecto.

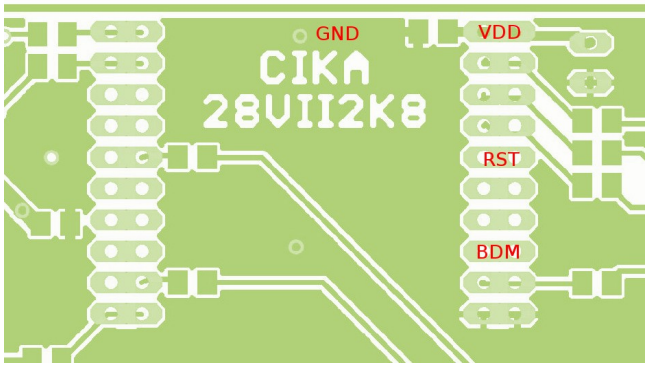
La conexión de esta herramienta con el módulo se realiza mediante cuatro pines:

señal	*pin*
BDM	8
RESET	5
VDD	1
GND	10

Estas herramientas esperan encontrarse con un conector de pines de paso .1”, cuyo pinout es el siguiente:



En la placa XBoard existe un conector de este tipo. Para adecuar una placa XBoard más antigua a un XBee Programable, podemos conectarnos en los siguientes puntos:



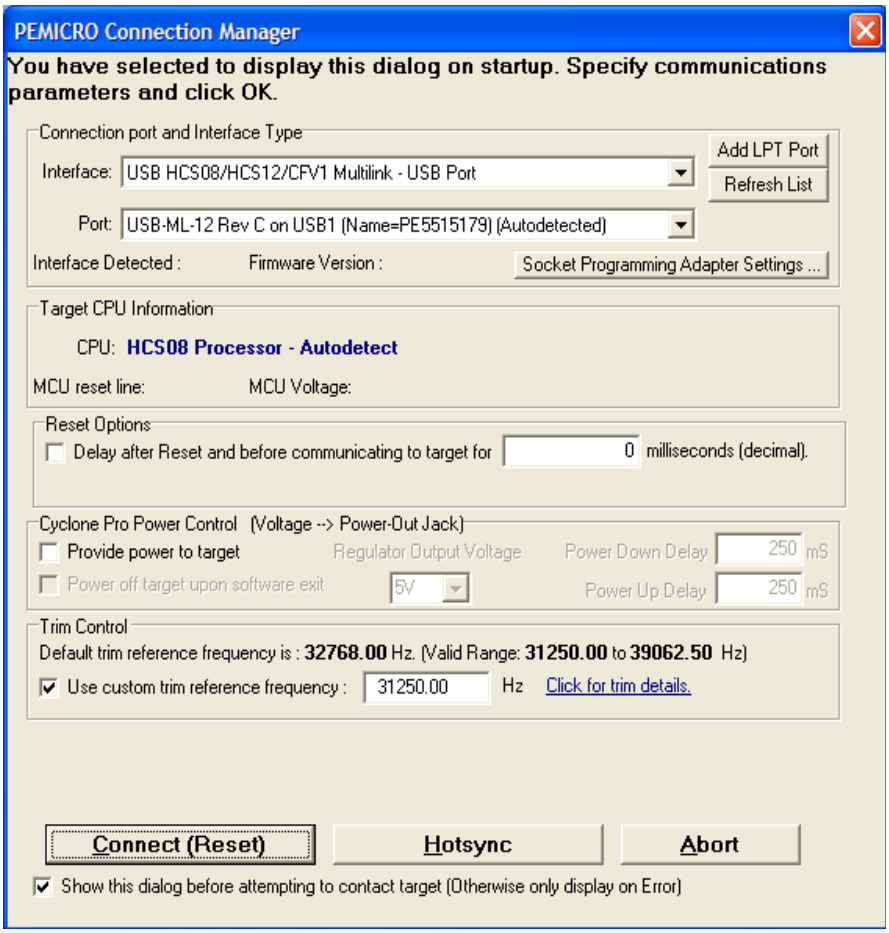
Es conveniente trabajar primero con la aplicación en sí, grabándola directamente con el programador/debugger. Cuando ya tengamos la aplicación depurada compilamos para poder generar el ejecutable que subiremos por XMODEM al módulo. Esto requiere de disponer de dos versiones del archivo de definiciones para el linker, una que mapee los vectores para que la aplicación tenga el control, y otra que respete los lineamientos del bootloader. El primer caso es generado automáticamente por el entorno de desarrollo, y el segundo caso lo hemos analizado unos apartados atrás.

Multilink (P&E Microcomputer Systems)

Esta es la herramienta sugerida por Digi. Es un producto de costo elevado y buenas prestaciones.

Lo único que debemos tener en cuenta al utilizar esta herramienta es el seteo correspondiente a la referencia interna del clock de la CPU, en 31250 Hz.

La captura siguiente muestra esta situación:



Alternativas Open Source

Existen muchas alternativas basadas en código y hardware abiertos. Por ejemplo hemos ensayado el WTUSBDML de Witztronics. El circuito esquemático y el código pueden obtenerse en la Internet.

Estas herramientas, al menos en Code Warrior V6.3, presentan el inconveniente de no preservar la calibración de la referencia del oscilador interno del microcontrolador.

FreesBee

Cika ha desarrollado una herramienta basada en código y hardware abiertos, denominada FreesBee. Esta herramienta es de muy bajo costo y se la describe en su manual correspondiente. En el mismo encontraremos además los pasos y estrategias a seguir para desbloquear el bootloader y regenerar el valor de calibración de la referencia del oscilador interno del microcontrolador.

Ejemplos

Un ejemplo concreto, que además contiene los include files mencionados, se encuentra en la Nota de Aplicación CAN-092.

De él debemos extraer:

- en el directorio Sign
  - los include files para definición de los pines: *pin_mapping.h*
  - y comunicación con el bootloader: *common.h* y *sharedRAM.h*
- en el directorio Sign\CartelBeeAPI\prm:
  - los archivos de definición del linker, para trabajar con: *Program_sibl.prm*
  - y sin el bootloader: *Program_nobl.prm*

Apéndice: Periféricos del MC9S08QE32

Si bien la literatura de Freescale es amplia, presentamos aquí un pequeño resumen para una rápida utilización de los periféricos del micro.

ADC

El conversor A/D del micro de aplicación es bastante poderoso, pudiendo operar del reloj del sistema o de uno propio. En el primer caso, debe tenerse en cuenta el divisor a utilizar para respetar la frecuencia máxima de operación.

Los pines a utilizar como entradas analógicas se configuran en una pareja de registros de control, *APCTL1* y *APCTL2*; mientras que la señal a convertir se selecciona en el registro de control *ADCSC1*.

La resolución del conversor se selecciona mediante el registro *ADCCFG*, que además permite configurar el tiempo de sampling y el clock de conversión. Existe además una lógica de comparación que permite detectar (sin intervención de la CPU) si la señal convertida se encuentra dentro de un rango específico configurable.

El siguiente listado muestra una configuración sugerida, en la cual operamos el ADC por polling con una resolución de 12-bits:

```
ADCCFG_ADLPC=0;               // clock speed = rápido (standard, no "low power")
//ADCCFG_ADIV                 // no usamos divisor
ADCCFG_ADLSMP=0;              // short sample time
ADCCFG_MODE=1;                // 12-bits
ADCCFG_ADICLK=3;              // asynchronous ADC clock
ADCSC2_ACFE=0;                // disable compare logic
ADCSC2_ADTRG=0;               // software trigger
APCTL1=APCTL1_ADPC0_MASK+APCTL1_ADPC3_MASK;  // pines usados como entradas analógicas
APCTL2=0;                     // demás pines: general purpose I/O
```

La conversión se inicia al seleccionar la entrada (escribiendo *ADCSC1*), y el valor convertido está disponible cuando el flag *COCO* (COnversion COmplete) en el registro *ADCSC1* es colocado en estado alto:

```
/*
ADCSC1_AIEN=0;                // no interrupts
ADCSC1_ADCO=0;                // single conversion
ADCSC1_ADCH=0;                // channel 0
*/
ADCSC1=0;
while(!ADCSC1_COCO);
```

El tiempo de conversión, con la configuración sugerida, es de algo menos de 15us.[1]

El canal seleccionado corresponde al siguiente mapa:

pin XBee	nombre	canal ADC
20	IO_DIO0_ADC0_COMMISSIONING	0
19	IO_DIO1_ADC1	3
18	IO_DIO2_ADC2	6
17	IO_DIO3_ADC3	9
11	IO_DIO4_ADC4	7

El valor de la magnitud analógica se recupera conociendo el valor de la tensión de referencia. En nuestro caso, proveemos 2,75V por el pin correspondiente:

```
/*
Vref = 2.75V
span = 2^12 -1 = 4095
V = Vref * N / span = 2.75V * N / (2^12 - 1)
*/
#define VREF   2.75
#define SPAN   4095

        adval=ADCR;           // lee ADC
        v=(VREF*adval)/SPAN;  // obtiene magnitud medida
```

1 basado en un clock asincrónico de aproximadamente 3,3 MHz, con 23 ciclos de clock para conversión (7us), 5us de jitter de inicio y 5 bus clocks adicionales.

Nótese que el registro ADCR es en realidad una concatenación de los registros ADCRH y ADCRL; el compilador realiza una lectura de 16-bits que los accede en el orden correcto.

Existe además la posibilidad de disparar automáticamente el ADC mediante una señal configurable, tema que no desarrollaremos aquí.

I²C

El controlador I²C permite operar como slave o como master, en un ambiente multi-master. Veremos aquí cómo utilizarlo como único master para acceder a uno o más slaves en el bus, en modo polled.

Inicialización del controlador:

```
IICF=0x14;                  // rate = bus clock / 80 (ver hoja de datos)
IICC2=0;
IICC1_IICEN=1;              // enable module
IICC1_MST=0;                // slave mode (?)
IICS_SRW=0;                 // receive mode (?)
```

Antes de iniciar cualquier nueva operación, debemos chequear si la última se ha completado:

```
while(IICS_BUSY);
```

Chequeamos cada transacción mediante el flag IICS_TCF. Este flag se resetea al comenzar la transacción, por lo que debemos esperar antes de revisarlo por primera vez:

```
void wait(void) {
uint8_t i;

        for(i=0;i<32;i++);
        while(!IICS_TCF);
}
```

Escritura de un dato en un slave:

```
        while(IICS_BUSY);
        IICC1_TX=1;                 // transmit mode
        IICC1_MST=1;                // master mode (generate START)
        IICD=(0x1C<<1)+0;           // address + WR
        wait();
        err=0;
        if(!IICS_RXAK){
                IICD=0x02;          // dato a escribir en el slave
                wait();
        } else err=1;               // device not present
        IICC1_MST=0;                // slave mode (generate STOP)
```

Lectura de un dato en un slave:

```
        while(IICS_BUSY);
        IICC1_TX=1;                 // transmit mode
        IICC1_MST=1;                // master mode (generate START)
        IICD=(0x1C<<1)+0;           // address + WR
        wait();
        err=1;
        if(!IICS_RXAK){
                IICD=0x06;                  // dirección o registro en el slave
                wait();
                if(!IICS_RXAK){
                        IICC1_RSTA=1;       // repeated START
                        IICD=(0x1C<<1)+1;   // address + RD
                        wait();
                        if(!IICS_RXAK){
                                IICC1_TXAK=1;       // send ACK
                                IICC1_TX=0;         // receive mode
                                x=IICD;             // inicia receive sequence
                                wait();
                                IICC1_MST=0;        // slave mode (generate STOP)
                                x=IICD;             // obtiene el dato que leyó
                                err=0
                        }
                }
        } // else device not present
        if(err)
```

```
        IICC1_MST=0;          // slave mode (generate STOP)
```

SPI

El controlador SPI permite operar como slave o como master. Veremos aquí cómo utilizarlo como master para acceder a uno o más slaves en el bus, en modo polled.

El pin $\overline{SS}$ puede ser comandado directamente por el controlador. Sin embargo, por generalidad lo controlaremos manualmente:

```
IO_DIO3_ADC3_D=1;             // SS as output
IO_DIO3_ADC3=1;               // SS inactive
```

Inicialización del controlador:

```
SPIBR=0x02;                   // rate = bus clk / 8
SPIC1=0;
SPIC2=0;
SPIC1_MSTR=1;                 // master mode
SPIC1_CPOL=0;                 // mode 0
SPIC1_CPHA=0;
SPIC1_SPE=1;                  // enable module
```

Para escribir datos, podemos chequear el flag SPIS_SPTEF. Sin embargo, este flag indica que el registro está listo para aceptar un nuevo byte, no que terminó de enviarlo. El final de una transacción se detecta mediante el flag SPIS_SPRF, que indica que hay un byte disponible para ser leído.

Escritura y lectura de datos multi-byte en un slave:

```
IO_DIO3_ADC3=0;               // SS active
SPID=0x02;                    // dato 1 a escribir
while(!SPIS_SPRF);
x=SPID;                       // lectura, clear SPRF flag
SPID=0x85;                    // dato 2 a escribir
while(!SPIS_SPRF);
IO_DIO3_ADC3=1;               // SS inactive
x=SPID;                       // lectura, clear SPRF flag
```

RTC

El micro posee un módulo que le permite generar interrupciones como una base de tiempo. Entre las opciones de clocking, disponemos de un oscilador de muy bajo consumo de 1KHz (30%). Además de un prescaler, posee un contador de módulo variable.

La inicialización del módulo RTC para obtener interrupciones cada 100ms es la siguiente:

```
RTCMOD=0;
RTCSC = 0x1D;  // interrupt every 100ms, internal 1-KHz osc
```

La rutina de interrupciones deberá resetear el flag de interrupt:

```
__interrupt void Vrtc_isr (void)
{
        RTCSC_RTIF = 1;
        ...
}
```

No deberemos olvidarnos de la correspondiente entrada para el vector número 24 en *Program.prm*:

sin el bootloader

```
VECTOR 24 Vrtc_isr
```

con el bootloader

```
    VECTOR ADDRESS 0x0000F1CE  Vrtc_isr
```

PWM (TPM)

El micro cuenta con tres módulos timer con capacidad de generación de PWM, además de captura y comparación. El listado siguiente muestra la inicialización de uno de los canales para obtener una señal de ciclo de trabajo 1/16 a la frecuencia de bus dividida por 256 (resolución de 8-bits), prescalada 32 veces:

```
TPM1MOD = 0x00FF; // Modulo value: 256
TPM1C1SC = 0x28; // PWM mode, clears output on channel value match
TPM1SC = 0x0D; // Overflow interrupt disabled, edge-aligned, busclk/32
TPM1C1V = 0x0010; // Channel value. duty = 1/16. freq = bus/(32*256)
```