Cika Electrónica

Tutorial: CTU-010
Título: **Introducción al entorno de desarrollo de Holtek: HT-IDE3000**
Autor: Sergio R. Caprile, Senior Engineer

Revisiones	Fecha	Comentarios
0	09/02/07	
1	24/02/07	Ampliación

El presente tutorial se orienta a introducir el entorno de desarrollo de Holtek a ingenieros y developers familiarizados con otras tecnologías. A tal fin comentaremos las características principales y las diferencias fundamentales que se encuentran entre éste y otros más comunes. Dado que se trata de un tutorial destinado a facilitar un inicio rápido, se recomienda la lectura de la guía del usuario de la herramienta para conocer todas las opciones posibles.

La descripción y pantallas están basadas en la versión 6.6 de la herramienta.

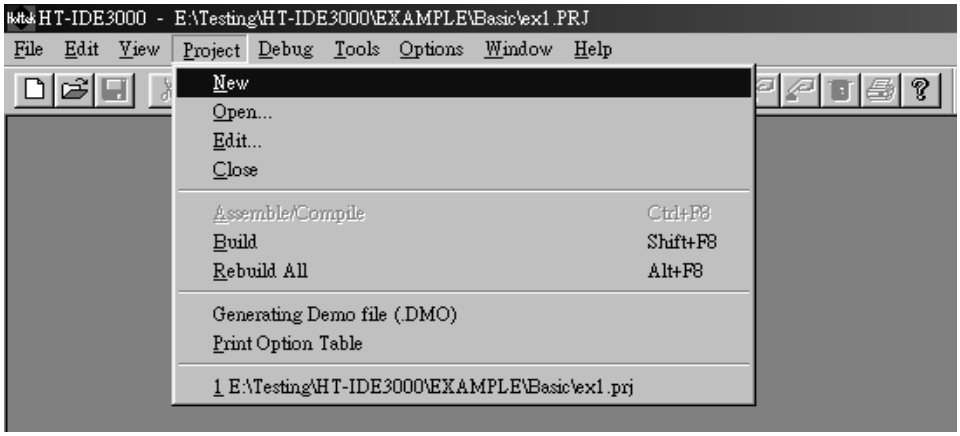
Índice de contenido

Trabajando con proyectos

La forma más cómoda de trabajar, particularmente si se aprovecha el código provisto en las notas de aplicación de Cika, es armando proyectos e incluyendo allí los archivos que se van a ensamblar o compilar.

Crear un nuevo proyecto

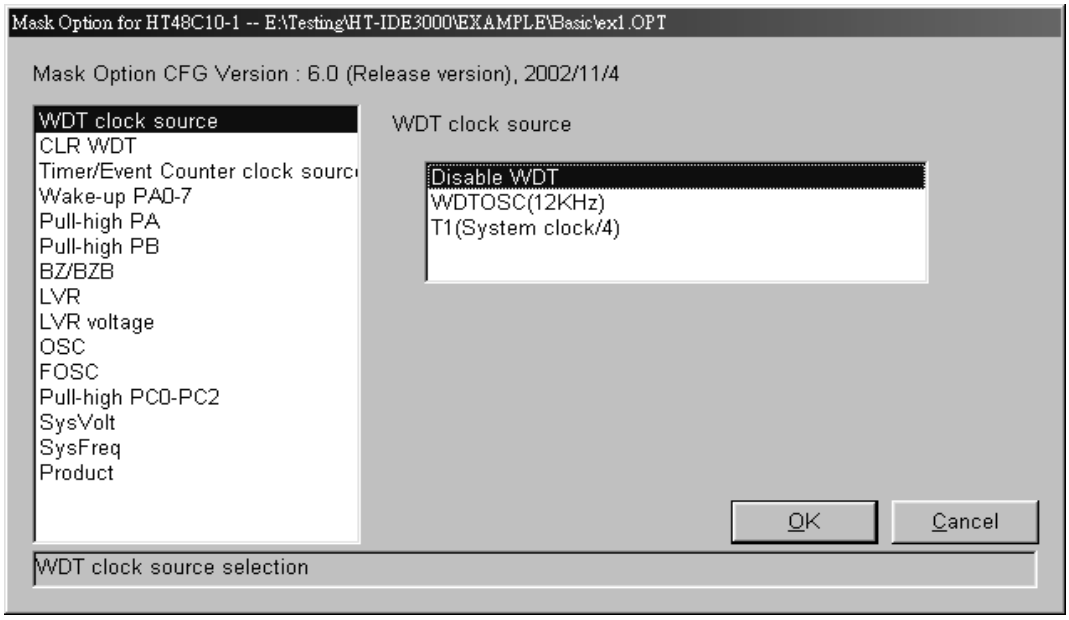
Lo hacemos desde el menú *Project*:



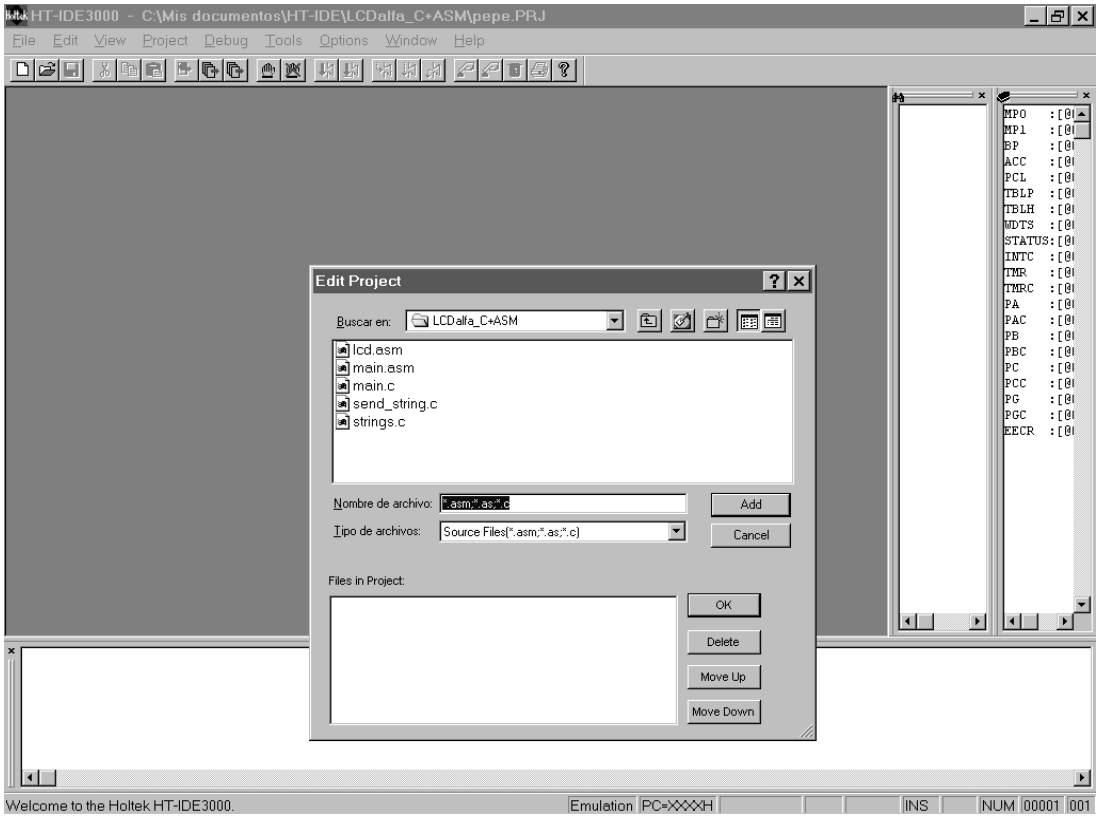
A continuación se nos presenta un requester donde podemos seleccionar el nombre del proyecto y la CPU a utilizar. A continuación otro requester nos permitirá ingresar directorios y definiciones para que el compilador pueda encontrar libraries y demás opciones.

Paso siguiente, se nos presentará una ventana similar a la que se observa a continuación (dependiendo de la CPU seleccionada), y en la que deberemos elegir las opciones que el dispositivo y el emulador nos presentan. Por ejemplo la tensión y frecuencia de trabajo, habilitación o no del watchdog, configuración de los pines PB0/PB1 como I/O o buzzer, pull-ups, etc.

Si más adelante, en el transcurso del desarrollo, necesitamos modificar alguna de estas opciones, podemos obtener esta ventana de configuración en el menú *Tools -> Configuration Options*.



Una vez hecho esto, podemos agregar al proyecto archivos que ya tengamos, o podemos empezar a escribir nuevos archivos, utilizando programación modular. En todos los casos, para que el HT-IDE sepa cuáles son los archivos que debe ensamblar o compilar para este proyecto, debemos agregarlos utizando la opción *Project -> Edit*:



Código fuente

Archivos Assembler

En un archivo assembler podemos incluir los archivos que tienen definiciones de los registros del procesador, o cualquier otro que necesitemos con definiciones o lo que fuere. Esto lo hacemos mediante la directiva *include*:

```
include ht48e30.inc
```

Luego declaramos las subrutinas y variables que serán accedidas desde otros módulos, es decir:

```
public delay
```

Las variables y rutinas que se definen en otros archivos las declaramos como *extern*, dado que son externas a este módulo, y el agregado *near* define identifica a las rutinas (direcciones de programa), mientras que *byte* identifica a las variables (direcciones de datos).

```
extern rutina:near
extern varname:byte
```

Luego, declaramos el área de variables en RAM y reservamos espacio para ellas:

```
dsec    .section 'data'

tmp     db ?
```

El signo *?* luego de la directiva *db* indica que se trata de un byte y no se lo inicializa a ningún valor.

Si necesitamos un espacio de varios bytes, lo declaramos de la siguiente forma:

```
dtmp    db 2 dup(?)
```

En el código, accedemos a cada byte como si se tratara de un array en C:

```
        mov a,dtmp[0]
        mov dtmp[1],a
```

Si necesitamos definir símbolos que después nos servirán para hacer más legible y modificable el proyecto, utilizamos la directiva *EQU*. Para asignar un nombre genérico a una posición de memoria de datos, sin por ello reservar un espacio, utilizamos la directiva *LABEL*; especificando *BYTE* a continuación:

```
TRW_DATA_LEN           equ     24
TRW_ADDRESS_LEN        equ     5

trwvars         .section 'data'
datas           db      ?
txbuffer        LABEL BYTE
address         db      TRW_ADDRESS_LEN DUP(?)
messge          db      TRW_DATA_LEN DUP(?)
```

A continuación, declaramos el área de código y escribimos las rutinas correspondientes. De ser necesario, se puede volver a definir otro tipo de área diferente en cualquier parte del archivo. Al no definir dirección, el área es relocalizable, y el linker decide qué direcciones utilizar. De todos modos, debe existir un area definida en la posición 0, que generalmente puede incluir el programa de inicialización y los vectores de interrupción:

```
scode           .section 'code'

send_cmd:       mov tmp,a
                call busy_chk
                ...

reset           .section at 0 'CODE'
                ORG 0
                jmp start
                ORG 08h
                jmp T0HND
                ORG 0Ch
start
```

Aunque en muchos casos es preferible separar las áreas para permitir alojar los vectores de interrupción en otros archivos:

```
reset           .section at 0 'CODE'
                jmp start

intT            .section at 8 'CODE'
                jmp T0HND

code            .section 'CODE'
```

Por último, si utilizamos la instrucción tabrdl, dado que ésta lee desde la última página[1] de memoria de programa (256 bytes):

```
tablas          .section at LASTPAGE 'code'

tabla1:         dw      67,105,107,97,32,69,108,101,99,116,114,111,110,105,99,97,022h
```

Al referirnos a los SFRs, registros, labels, y variables lo hacemos de forma indistinta en mayúsculas o minúsculas, el assembler no es case-sensitive. Las etiquetas en el código deben terminarse en ':'

Archivos C

Al trabajar en C utilizamos un subset del standard ANSI, y no necesitamos consideraciones especiales, más allá de las relacionadas con los nombres de los registros internos y las limitaciones propias del compilador.

```
#include        <HT48E30.h>

main()
{
        _pa=0;
        _pac=0;                                 // PA = salidas
```

Como podemos observar, desde C nos referimos a los SFR anteponiendo el caracter '_' al nombre, y **en minúsculas**. El compilador es case-sensitive. También están definidos como bits los flags de estado y los pines de I/O, por lo que para operar sobre los ports de I/O podemos operar de la forma tradicional:

```
        _pa|=(1<<2);                            // set bit 2
```

o bien:

```
        _pa2=1;                                 // set bit 2
```

En ambos casos, el compilador genera el mismo código:

```
                set pa.2
```

Para definir un vector de interrupción en C, ingresamos:

```
#pragma vector foo @ 8

void foo(void)
{
        // código del handler en C
}
```

donde *8* es el vector (offset), diferente para cada periférico. La función debe estar luego (no necesariamente inmediatamente después) del *#pragma vector*.

Entre las limitaciones del compilador, no es posible utilizar punteros constantes en memoria de programa, los punteros siempre son a espacios en RAM. Tampoco es posible que una función devuelva una estructura.
El compilador no soporta coma flotante y los enteros son de 8-bits, mientras que los enteros largos son de 16-bits.

C y assembler combinados

Para poder llamar rutinas assembler desde C, debemos respetar ciertas convenciones:

1 No es que la memoria esté paginada, sino que la nomenclatura de Holtek se refiere de esa forma a los bloques o segmentos de 256 bytes.

- Las rutinas assembler que deban ser llamadas desde C, además de estar declaradas como públicas, deberán tener el caracter '_' delante del nombre, por ejemplo: *_rutina*
- Los parámetros que deban pasarse se pasarán en una posición de RAM, la cual deberá tener el mismo nombre que la rutina (sin el '_'), seguida del número de parámetro, comenzando desde 0, por ejemplo: *rutina0*, *rutina1*. Varias rutinas pueden compartir la misma posición de memoria como parámetro, si esto no es mutuamente excluyente en su función.
- En C, llamamos a esta rutina por su nombre (sin el '_'), pero todo en mayúsculas, por ejemplo: *RUTINA (par1,par2);*

en assembler:

```
dsec    .section 'data'

send_string_RAM0 label byte
send_cmd0       label byte
send_char0      db ?

scode   .section        'code'

_send_cmd:
                mov a,send_cmd0
```

en C:

```
        SEND_CMD(0x3A);
```

También es posible llamar a funciones en C desde assembler, pero no desarrollaremos este tema aquí, dado que es menos común.

Para poder compartir variables declaradas en C, debemos respetar las siguientes convenciones:

- Si una variable global declarada en C deberá ser accedida desde assembler, la misma deberá declararse todo en mayúsculas, pues el linker trabaja en mayúsculas, por ejemplo: *unsigned char VARNAME;*
- Para acceder desde assembler a una variable global declarada en C, la declaramos como externa y nos referimos a ella anteponiéndole el caracter '_', por ejemplo: *mov a, _varname*. No es necesario usar mayúsculas pues el assembler no diferencia.

en C:

```
unsigned char VARNAME;
```

en assembler:

```
extern _varname:byte

                mov a,_varname
```

De igual modo, para poder compartir variables declaradas en assembler, debemos respetar las siguientes convenciones:

- Si una variable global declarada en assembler deberá ser accedida desde C, la misma deberá declararse anteponiéndole el caracter '_', por ejemplo: *_varname db ?*. No es necesario usar mayúsculas pues el assembler no diferencia, pero es buan práctica para no confundirse.
- Para acceder desde C a una variable global declarada en assembler, la declaramos como externa y nos referimos a ella con el nombre en mayúsculas, pues el linker trabaja en mayúsculas, por ejemplo: *VARNAME=0;*

en C:

```
extern unsigned char VARNAME;

        VARNAME=0;
```

en assembler:

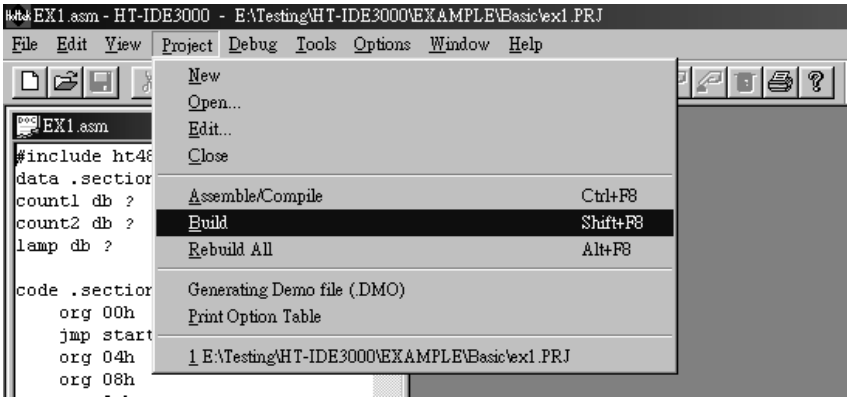
```
_varname db ?

                mov a,_varname
```

Finalmente, **si tenemos un interrupt handler en assembler, deberemos definir su vector en assembler**; la definición de vectores en C es sólo para funciones en C; el compilador no genera el vector si no encuentra la función en el mismo archivo, a continuación del *#pragma vector*.

Ensamblado/Compilado

Para que HT-IDE transforme nuestro proyecto en código, seleccionaremos la opción deseada en el menú de proyecto. La opción *Build* ensambla/compila los archivos con modificaciones, mientras que *Rebuild All* ensambla/compila todos los archivos. Si se realizan modificaciones a archivos incluidos, deberá recompilarse todo pues HT-IDE sólo detecta cambios en los archivos ingresados en su lista.



El código generado puede simularse dentro de HT-IDE si así está seleccionado, o puede emularse en circuito mediante el hardware HT-ICE. Según el micro seleccionado, se genera un archivo de tipo OTP o MTP que puede enviarse al correspondiente programador para grabarlo en el micro.

Debugging

Seleccionando *Options -> Debug*, veremos una ventana donde podemos seleccionar las diferentes opciones de debugging, entre ellas *Simulation* corresponde al uso del simulador que incorpora HT-IDE, mientras que *Emulation* corresponde al uso del hardware HT-ICE. La emulación incorpora además una función de *trace* que permite listar las líneas de código por las que pasa la ejecución.

Watches

Seleccionando la ventana correspondiente: *Window -> Watch*, podemos observar los valores de algunas variables. Para agregar una variable a la lista, deberemos escribir el nombre del archivo en el que se la declara, seguido de un signo de admiración, y a continuación, y siempre sin espacios, el nombre de la variable. Al presionar enter, observaremos que la misma fue añadida a la ventana. Por ejemplo, tipeando: *stimers.asm!CS_TIMERS[3]<ENTER>*, agregamos el cuarto elemento del array *CS_TIMERS* definido en el archivo *stimers.asm*.