Tutorial: CTU-015
Título: **Holtek HT32F125x**
Autor: Sergio R. Caprile, Senior R&D Engineer

Revisiones	Fecha	Comentarios
0	23/03/12	
1	17/04/13	Agrega HT32F1755/65

El presente tutorial tiene por objeto introducir al core de 32-bits ARM Cortex-M3 y presentar la forma de utilizar los micros de las familias HT32F125x y HT32F1755/65 de Holtek.

Índice de contenido

Descripción del micro....3
ARM Cortex-M3....3
Interrupciones: NVIC....3
Timer: SysTick....4
CMSIS....4
Buses internos: AHB y APB....4
Revisiones....4
HT32F125x....4
HT32F1755/65....5
Cuadro comparativo....5
Herramientas de desarrollo....6
Device driver, firmware library....6
IDEs....6
Keil....6
CMSIS....6
IAR....6
CMSIS....7
CooCox....7
CMSIS....7
Programador-Debugger....7
e-Link32....7
Keil....7
IAR....8
Coocox....8
Alternativas Open Source....8
Desarrollo de aplicaciones en C....8
Entorno....8
Inicio y CMSIS....8
Keil....8
IAR....8
CooCox....9
Tipos y registros internos....9
Vectores de interrupción....9
SysTick....10
Manejo de I/O....10
Bit banding....10
Timing....11
Clock y funciones alternativas (AFIO)....11
Compartiendo los pines con otros periféricos....11
ADC....11
Inicialización....11

Detección de fin de conversión y lectura....12
Modo polled....12
Mediante interrupciones....12
Conversión iniciada por un timer....13
Timers GPTM....13
Control de una salida....13
PWM....14
USART....14
Modo polled....14
Mediante interrupciones....15
Amplificador operacional y comparador....15
Watchdog....15
FMC: escritura en flash....16

Descripción del micro

La familia HT32F125x utiliza un core ARM Cortex-M3, al cual agrega una serie de interesantes periféricos. Describimos a ambos en sendos apartados.

ARM Cortex-M3

El core Cortex-M3 emplea la arquitectura ARMv7-M, más reciente que los tradicionales cores ARM7 y ARM9, que empleaban ARMv4 y ARMv5, respectivamente. Funcionalmente guardan muchas similitudes, y comparten gran cantidad de instrucciones, aunque evidentemente ARMv7-M ha sido diseñada más para el ambiente de microcontroladores, y por consiguiente difiere de las anteriores en áreas como manejo de interrupciones y demás excepciones, niveles de ejecución, y manejo de memoria.

Sin entrar demasiado en detalles, el set de instrucciones es un subset de Thumb-2, en el cual existen instrucciones de 16-bits y de 32-bits. El micro lee 32-bits cada ciclo, lo cual puede corresponder a 1 ó 2 instrucciones. La lectura y ejecución forman dos de las tres etapas de un pipeline que sumado al empleo de arquitectura Harvard le permite superponer tareas y lograr un score de 1,25 DMIPS[1] por MHz de clock. Entre las instrucciones se encuentran multiplicaciones de 32x32=32 en un ciclo y divisiones 32/32=32; ambas con hardware ad-hoc. Existen también instrucciones multi-ciclo de multiplicación y división larga, y de movimiento de múltiples registros a y desde memoria. La arquitectura Harvard es esencialmente load-store, por lo que todas las operaciones se realizan sobre los registros, y estos se mueven hacia y desde la memoria con un conjunto ortogonal de modos de direccionamiento. Una característica interesante es la existencia de un registro denominado LR (Link Register, R14), el cual, soportado por instrucciones BL (Branch with Link), permite hacer saltos a subrutinas sin tener que guardar el PC en el stack, dado que se lo guarda en este registro BL. El retorno de la subrutina consiste en copiar el contenido de LR al PC.

El Cortex-M3 presenta un mapa unificado de memoria, lo que le permite ocultar su estructura Harvard a los ojos del programador (y fundamentalmente del compilador), evitando inconvenientes de asignación de espacios y ubicación de constantes comúnmente encontrados en otros microcontroladores que la utilizan. Internamente, código y datos utilizan buses separados y pueden superponer sus accesos a flash y RAM, pero como las direcciones asignadas en el mapa de memoria son además diferentes, es posible ver a ambos como en un mismo espacio.

Dentro de este mapa unificado, existen áreas especiales denominadas bit band, las cuales mapean una dirección a un bit en otra área; es decir, un espacio de 32MB accede bit a bit a los mismos bits que se encuentran en 1MB, accedidos por bytes (8-bits), half words (16-bits), o words (32-bits). Esto permite realizar operaciones booleanas y manejo de bits en memoria como el histórico MCS-51 (8051), y acceder a I/O bit a bit sin necesidad de utilizar instrucciones read-modify-write ni requerir de periféricos especiales.

La definición del mapa de memoria por parte del proveedor del core (ARM), permite que todos los fabricantes que lo utilizan ubiquen tanto memoria como periféricos en las mismas direcciones (en general). Esto, sumado a la existencia de periféricos comunes como el NVIC (Nested Vectored Interrupt Controller) y el timer SysTick, y el software de abstracción CMSIS (Cortex Microcontroller Software Interface Standard), logran que el usuario final pueda portar su aplicación de un fabricante a otro sin mayores inconvenientes.

Interrupciones: NVIC

A diferencia de ARM7 y ARM9, no existen IRQ y FIQ; y las interrupciones son separadas y anidadas gracias al NVIC. El Cortex-M3 posee una interrupción no enmascarable (NMI), la cual puede ocurrir en cualquier momento, inclusive al arrancar el sistema. Por tal razón, la primera palabra del vector de interrupciones es el valor a cargar en el Stack Pointer (R13, SP), que se realiza por hardware al momento de reset. A continuación le siguen el vector de reset, que indica la posición donde se encuentra la primera palabra de código, y así le siguen las demás interrupciones. El vector de interrupciones se ubica inicialmente en la posición 0x000000, pero puede relocalizarse mediante un registro en el NVIC.

Cada interrupción tiene un nivel de prioridad, y sólo aquéllas de mayor prioridad pueden interrumpir a otra que esté siendo atendida. Existen como máximo 255 niveles de prioridad, y 240 interrupciones, que dependen de la creatividad del fabricante del micro.

El proceso de atención de una interrupción demora unos 12 ciclos de clock, durante los cuales se salvan los registros R0 a R3 y R12 en el stack, junto con obviamente el PC (R15) y el LR(R14), simultáneamente con la búsqueda del vector de interrupciones, dado que ambas ocurren en espacios separados, datos y código respectivamente.

---

1 DMIPS: Dhrystone MIPS (Million Instructions Per Second). 1DMIPS/MHz: cantidad de iteraciones del algoritmo por segundo, a 1MHz, normalizado a la performance de una VAX VMS 11/1750 (1757). Refleja mayormente la performance en cálculos con enteros, y permite, en cierto modo, comparar micros entre sí.

Timer: SysTick

El SysTick es un timer de 24-bits contenido en el NVIC. De muy simple utilización, permite contar con una base de tiempo portable.

CMSIS[2]

Con este nombre denominamos a un conjunto de archivos que conforman un nivel de abstracción que permite acceder a micros de diferentes fabricantes de una forma común; siempre que utilicen el core Cortex-M3[3], por supuesto. ARM provee los archivos necesarios para acceder al NVIC y para poder utilizar las características distintivas del micro con diversos compiladores. Así, existen funciones intrínsecas que son mapeadas mediante macros al compilador utilizado, de modo que el usuario pueda también cambiar de compilador cuando lo desee. De momento Keil, IAR y GNU son los entornos soportados.

Además, cada fabricante provee un set de archivos que permite acceder a los periféricos del micro de modo uniforme mediante esctructuras en C. Así, para escribir en el primer puerto de I/O de un Toshiba escribimos:

```
TOS_PA->DATA = byte;
```

, en un Fujitsu:

```
FM3_GPIO->PDOR1 = byte;
```

y en Holtek

```
HT_GPIOA->DOUTR = byte;
```

Buses internos: AHB y APB

Sin ánimo de entrar demasiado en detalles, la comunicación entre el core y la memoria y periféricos se realiza mediante el bus AHB[4] (Advanced High-performance Bus), el cual puede incluir un bridge hacia un bus APB (Advanced Peripheral Bus). Más allá de sus nombres, se trata de arquitecturas de bus normalizadas con una determinada cantidad de líneas paralelo y un timing particular. Cada transacción emplea una cierta cantidad de clocks y en algunos casos es posible realizar pipelining. Así, el acceso a la flash se realiza a la velocidad del bus AHB y puede tener wait-states. El acceso a los I/O generalmente se realiza mediante una comunicación AHB-APB, por lo que la operación transcurre en paralelo con el funcionamiento del micro. Sin embargo, operaciones sucesivas sobre I/O requieren que el micro espere a que la anterior se complete. El timing de I/O generado depende entonces de la velocidad de clocking de AHB y APB del micro en particular, más que de las instrucciones empleadas y la velocidad del core.

Revisiones

Existen al momento tres revisiones mayores de la arquitectura Cortex-M3: r0, r1, y r2. La diferencia más notable es que r2 incorpora el WIC (Wake-up Interrupt Controller), que permite que el core pueda entrar en deep-sleep, lo que requiere que se apague el clock principal y por ende el NVIC, y ser despertado por una interrupción y retomar la ejecución normal.

HT32F125x

Este micro de Holtek es un Cortex-M3 r2p0 sencillo, con 8, 16 ó 32 KB de flash y 2, 4 u 8 KB de RAM, en un encapsulado LQFP de 48 pines. Claramente orientado a aplicaciones donde tradicionalmente se utilizaría un micro de 8-bits de igual o mayor capacidad de flash, al cual reemplaza con mayores prestaciones y menor costo; así como también para oficiar de ingreso al mundo de los 32-bits para quienes aún manifiestan cierto temor ante esta novedad.

La flash del micro se encuentra sobre el bus AHB, y funciona sin wait-states hasta 24Mhz, con 1 wait-state hasta 48MHz y 2 wait-states hasta 72MHz, la velocidad máxima del core, provista por una fuente externa de hasta 16MHz o una interna de 8MHz, y un PLL interno.

Los 32 pines de I/O se comparten con el port SWD (Serial Wire Debugging) y los periféricos esperables como timers con capacidad PWM, conversor A/D, USART, y ports I^2C y SPI. Posee además un watchdog y un controlador de memoria flash que permite grabarla en la aplicación misma (IAP, In Application Programming). Completa el set un

2 pronunciado “ci emsis”.
3 CMSIS también soporta Cortex-M0 y Cortex-M4, al momento.
4 La implementación de Cortex-M corresponde a AHB-Lite, una versión reducida.

manejador de energía con dominio de backup, que permite escribir aplicaciones que puedan salvar su estado en un área de RAM y apagar el resto, recuperándolo más adelante.

HT32F1755/65

Esta pareja de micros de Holtek amplían la familia anterior, incorporando mayor cantidad de algunos periféricos, y otros totalmente nuevos, en encapsulados LQFP de 48, 64, y 100 pines.
Se orienta a aplicaciones algo más complejas que la otra familia, dada su mayor capacidad de memoria.

Cuadro comparativo

La presente tabla muestra las diferencias entre los periféricos (y el encapsulado) de una y otra familia. Este tutorial sólo contempla aquellos periféricos que son comunes a ambas.

	HT32F125x	HT32F1755 / 65
SRAM	1251: 2 KB	1755: 32 KB
	1252: 4 KB	1765: 64 KB
	1253: 8 KB	
FMC	1251: 8 KB	128 KB
	1252: 16 KB	
	1253: 32 KB	
PWRCU	BOD: 2.5 V	BOD: 2.6 V
GPIO	hasta 32	hasta 80
ADC	En modo poll, debe observarse el bit ADVLD para confirmar la validez de la conversión.	El dato es válido al momento de indicarse fin de conversión. Offset cancellation register Más fuentes de disparo
BFTM		32 bit compare/match up counter sin control de I/O
MCTM		Un GPTM con funciones adicionales: • Salidas complementarias • Programmable dead-time insertion • Contador de repeticiones • Break input
I2C	1 controlador, hasta 400 kHz	2 controladores, hasta 1 MHz
SPI	1 controlador, master o slave hasta 18 MHz	2 controladores, master hasta 36 MHz y slave hasta 18 MHz
USART	1 puerto	2 puertos, auto hardware flow control
SCI		ISO 7816-3
USB Device		USB 2.0 Full Speed (12 MHz) Endpoints: 1 Control, 3 Bulk / Interrupt, 4 Bulk / Interrupt / Isochronous 1024 Bytes FIFO
PDMA		12 canales

	HT32F125x	HT32F1755 / 65
Encapsulado	LQFP 48 (7mm x 7mm)	QFN 48 (7mm x 7mm) LQFP 48 (7mm x 7mm) LQFP 64 (7mm x 7mm) LQFP 64 (10mm x 10mm) LQFP 100 (14mm x 14mm)

Herramientas de desarrollo

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

Device driver, firmware library

Es común que cada fabricante provea una serie de funciones para inicializar y llevar a cabo algunas funciones con los periféricos de su micro. Esto lo realiza bajo el nombre de device driver o firmware library. Si bien en algunos podrían ser o son un ejemplo de utilización del periférico, hemos observado que en la gran mayoría de las veces complican la operatoria debiendo considerar casos genéricos o esotéricos en vez de resolver el problema puntual que el desarrollador tiene. En otros casos, hemos encontrado que sólo algunas de las funciones del periférico están cubiertas, y hasta hemos observado software loops. Además, desde el punto de vista de un usuario tradicional de microcontroladores, generalmente son funciones largas con muchos argumentos, formando un API complejo más cerca de lo esperable en una aplicación de computadora que en un sistema dedicado simple, particularmente el que se desarrollaría con un micro de 8 a 32KB de flash.

Por este motivo, los ejemplos provistos por Cika no hacen uso de este device driver o firmware library.

IDEs

El entorno de desarrollo en un Cortex-M3 no es una elección del fabricante, sino del desarrollador. Si bien existen algunos casos particulares que son todo lo contrario, siempre existe la posibilidad de recurrir a un proveedor no atado con el fabricante o incluso gratuito o hasta Open Source. A continuación describimos los más comunes.

Keil

Keil es desde hace unos años una empresa del grupo ARM. Como tal, el compilador es algo así como "el oficial". El IDE utilizado es uVision, el producto es el ARM-MDK (Microcontroller Development Kit), del que puede obtenerse en página web una versión sin costo con límite de 32KB de código generado. Dada la capacidad de la familia HT32F125x, es posible utilizar esta excelente herramienta sin costo.

El IDE uVision incluye un editor con resaltado de sintaxis y debugger. El debugger permite observar el estado de los registros de los periféricos del micro. Existe además un simulador, que permite depurar código pero no simula los periféricos.

Debe instalarse un plugin provisto por Holtek.

CMSIS

El soporte para CMSIS se encuentra por defecto, no sólo para la compilación sino que los include files que provee el MDK son CMSIS. La única diferencia en el caso de Holtek es que al momento de revisarlos, los archivos de IAR incluyen *HT_* delante de los nombres de los registros, mientras que los provistos por Holtek no. Todo el material provisto por Cika utiliza la nomenclatura de Keil, con el prefijo *HT_*.

Para utilizar el device driver de Holtek, deben utilizarse los include files de Holtek

IAR

El producto Embedded Workbench for ARM (EWARM) puede ser descargado de la página del fabricante, luego de registrarse.

Existe una versión sin costo con límite de 32KB de código generado. Dada la capacidad de la familia HT32F125x, es posible utilizar esta excelente herramienta sin costo.

El IDE incluye un editor con resaltado de sintaxis y debugger, sin embargo el debugger no permite observar el estado de los registros de los periféricos del micro. Existe además un simulador, que permite depurar código pero no simula los periféricos.

Debe instalarse un plugin provisto por Holtek

CMSIS

El soporte para incluir los archivos de CMSIS provistos se encuentra a partir de la versión 6.2, y debe activarse manualmente en las opciones del proyecto (General->Library->Use CMSIS). Esto sólo aplica a los archivos genéricos para el core, los include files de los dispositivos que provee IAR es para su propio formato, que es diferente de CMSIS. Holtek provee un paquete para utilizar CMSIS con IAR, sin embargo Cika provee uno diferente, que es totalmente compatible con el utilizado en Keil, lo que permite usar indistintamente uno u otro IDE y compilador. La diferencia fundamental entre unos y otros archivos es el prefijo *HT_*.

Para utilizar el device driver de Holtek, deben utilizarse los include files de Holtek

CooCox

Dicho nombre hace referencia a Cooperating in Cortex, un entorno de distribución gratuita basado en Eclipse, con el compilador GNU (gcc) provisto por Code Sourcery ya incluido. Afortunadamente a pesar de estar basado en Eclipse es relativamente rápido, siendo órdenes de magnitud más lento para toda acción que tanto Keil como IAR.

No posee simulador, ni forma de ver los periféricos al momento.

CMSIS

El soporte para incluir los archivos de CMSIS provistos debe activarse manualmente en las opciones del proyecto. Los include files de los dispositivos que provee CooCox responden al mismo formato que Holtek provee. Sin embargo, Cika provee uno diferente, que es totalmente compatible con el utilizado en Keil, lo que permite usar indistintamente uno u otro IDE y compilador. La diferencia fundamental en los include files es el prefijo *HT_*; sin embargo el código de startup ha sido modificado para incluir la inicialización del clock en base a los otros entornos.

Para utilizar el device driver de Holtek, deben utilizarse los include files de CooCox.

Programador-Debugger

Sin duda una herramienta de este tipo acelera el proceso de desarrollo y depuración del código.

La conexión de esta herramienta con el micro se realiza mediante los siguientes pines:

señal	***pin***
SWDIO	25
SWCLK	26
SWO	27
nRST	17
+3,3V	
GND	

Estas herramientas esperan encontrarse con un conector de pines de paso .1” ó .05”. Los micros anteriores utilizaban el conector de .1” con 20 pines, compatible JTAG. A partir de la introducción de la familia Cortex, el conector se redujo a 10 pines de paso .05”. Algunos micros poseen mbas interfaces, JTAG y SWD.

e-Link32

Esta es la herramienta oficial de Holtek. Posee solamente SWD, al igual que la familia HT32F125x. Emplea un conector de 10 pines de paso .1”.

Debe instalarse un driver provisto por Holtek para que el dispositivo funcione en un port USB.

Keil

Es soportada de forma directa, observándose en el listado de debuggers disponibles. Debe instalarse un plugin provisto por Holtek para que el MDK lo reconozca

IAR

Es soportada de forma indirecta a través del driver RDI. Debe instalarse un plugin provisto por Holtek y configurar adecuadamente el driver RDI para que EWARM lo reconozca

Coocox

Es soportada de forma directa, observándose en el listado de debuggers disponibles.

Alternativas Open Source

Existen alternativas basadas en código y hardware abiertos. Por ejemplo Colink-EX funciona bajo CooCox; sin embargo no lo hemos comprobado.

Desarrollo de aplicaciones en C

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

Entorno

El primer paso que debemos dar en un nuevo proyecto es configurar el entorno de trabajo con los archivos mínimos necesarios para el inicio del micro y soporte CMSIS. Luego podemos empezar a escribir nuestro código.

Inicio y CMSIS

El entorno CMSIS está dividido en los archivos generales para soportar particularidades del core y del compilador, y los archivos particulares para soportar el micro. Estos son:

- Generales
  - *core_cm3.c*
- Particulares
  - *system_ht32f125x.c*, contiene *SystemInit()*, rutina que se encarga de inicializar el clock
  - *startup_ht32f125x.s* o *startup_ht32f125x.c*, contiene vectores de interrupción y reset. Este último llama a *SystemInit()* y luego salta a ejecutar el código del usuario en *main()*.
  - Opcional
    - *ht32f125x_op.s*, permite configurar opciones de protección de la flash

Keil

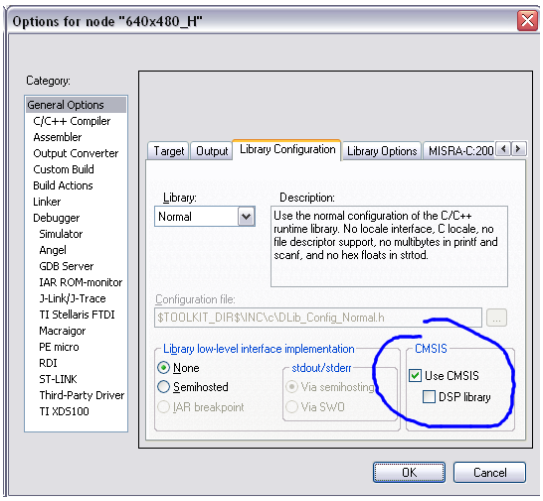
Con Keil agregamos los archivos desde donde residen. Podemos simplemente incluirlos o copiarlos a nuestro directorio de trabajo. Como regla general, es cómodo copiarlos sólo si se requiere modificarlos. Por ejemplo *system_ht32f125x.c* inicializa todos los clocks por defecto, sin embargo puede requerirse los pines compartidos con el cristal para otra aplicación.

Los archivos requeridos se encuentran en la siguiente ubicación por defecto:

- Generales
  - *C:\Keil\ARM\Startup*
- Particulares
  - *C:\Keil\ARM\Startup\Holtek\HT32F125x*

IAR

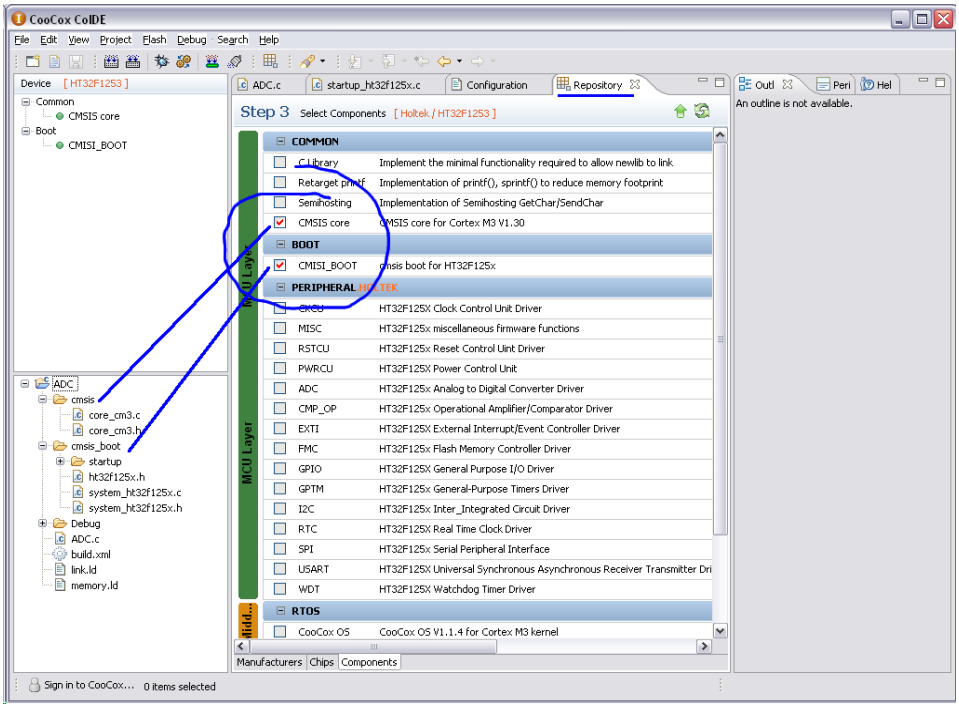
Con IAR, los archivos generales de CMSIS se incluyen mediante una opción de proyecto:



Los archivos particulares los debemos copiar a nuestro directorio de trabajo e incluirlos manualmente, por nuestra cuenta. Cika provee un paquete de soporte.

CooCox

En CooCox, indicamos en la pestaña *Repository* nuestra intención de utilizar CMSIS y éste copia a nuestro directorio de trabajo ambos sets de archivos. Si utilizamos los archivos particulares provistos por Cika, debemos copiarlos sobre éstos y no volver a jugar con estas opciones del repositorio.



Tipos y registros internos

Los tipos básicos C99 se encuentran en stdint.h; sin embargo este archivo ya es incluido por el archivo *HT32F125x.h*, que incluye el soporte para todos los registros y periféricos particulares del micro

```
#include "HT32F125x.h"
```

Vectores de interrupción

El archivo *startup_ht32f125x* incluye vectores de interrupción vacíos para todas las excepciones del micro. Si habilitamos una excepción (interrupción, por ejemplo) y no escribimos un interrupt handler, será atendida por este

vector, que quedará en loop llamando la atención del debugger. Estos handlers están definidos de forma “débil” (weak), por lo que cualquier otra definición tiene precedencia.

Para escribir un interrupt handler simplemente escribimos una función en C con el nombre correspondiente, el cual podemos tomar del archivo de startup. Dado que el micro salva automáticamente en el stack los registros R0 a R3 y R12, que son los definidos para uso por todas las funciones, más LR y PC. No es necesario salvar ningún registro manualmente, y no existe diferencia entre una función C “void normal” y un handler de interrupciones. De igual modo, el retorno de una función se produce siempre cargando el registro LR en el PC. Al momento de salvar los registros, el LR se carga con un valor especial fuera del mapa de memoria que al ser escrito luego en el PC dispara el mecanismo de retorno de excepción. De esta forma, no existen *#pragma*s ni *__interrupt*s ni ningún otro mecanismo no standard, un handler es una función más.

SysTick

El timer SysTick es sumamente fácil de utilizar, simplemente cargamos el registro de cuenta con el valor que deseamos, habilitamos su funcionamiento, y recibiremos una interrupción a cada fin de cuenta. No es necesario recargar la cuenta ni nada más. El ejemplo siguiente muestra cómo lograr una interrupción cada 1ms para mantener un esquema de software timers:

```
volatile unsigned long SysTickCnt;      // SysTick Counter

void SysTick_Setup (void)
{
  SysTick->LOAD = 72000 - 1;            // 1ms: 72000 ticks a 72MHz
  SysTick->CTRL = 0x0007;               // Habilitamos Counter y Tick Interrupt
}

void SysTick_Handler (void)
{
  SysTickCnt++;
}
```

Manejo de I/O

Los puertos de I/O tienen un registro para controlar el estado, otro para leerlo, otro para seleccionar el sentido de comunicación, y otro para habilitar la resistencia interna de pull-up. También permiten configurarse como open-drain y colocar resistencias de pull-down. Los ports son de 16-bits.
Por ejemplo:
seleccionamos el port A como salida:

```
HT_GPIOA->DIRCR = 0x00FF;                   // PA0-7 defined as Outputs
```

escribimos sobre él:

```
HT_GPIOA->DOUTR |= (1<<3);                  // bit 3 en alto
HT_GPIOA->DOUTR &= ~(1<<3);                 // bit 3 en bajo
```

seleccionamos el port A como entrada:

```
HT_GPIOA->INER |= 0xFFFF;                   // PA0-15, habilita circuitos de entrada
HT_GPIOA->DIRCR = 0x0000;                   // PA0-15 entradas
```

habilitamos el resistor de pull-up:

```
HT_GPIOA->PUR |= (1<<3);                    // pull-up en PA3
```

lo leemos:

```
if(HT_GPIOA->DINR & (1<<3))                 // lee bit 3
```

Bit banding

Estas mismas operaciones se pueden realizar utilizando el área de bitbanding. Desafortunadamente el fabricante no nos provee con una macro o set de macros, pero podemos realizarla nosotros fácilmente. Sabemos que cada bit ocupa una palabra (32 bits, 4 bytes), por lo que cada registro de I/O ocupa 32 bytes. Sabemos además que el área de bit banding para periféricos comienza en una posición fija: *PERIPH_BB_BASE*, y el área de periféricos también: *PERIPH_BASE*. El resto es simple aritmética:

```
// Bit-band alias = Bit-band base + (byte offset * 32) + (bit number * 4)
#define BITBAND_PERI(addr, bitnum) (HT_PERIPH_BB_BASE + (((uint32_t)(addr) - HT_PERIPH_BASE) << 5) + \
        ((uint32_t)(bitnum) << 2))
```

Así, podemos escribir en el bit 3 del port A de la siguiente forma:

```
(*((__IO uint32_t *)BITBAND_PERI(&HT_GPIOA->DOUTR, 3)))
```

El uso de bit banding permite economizar instrucciones, pues no es necesario realizar una operación read-modify-write. Esto, a su vez, simplifica la relación entre tareas que comparten el uso de un port.

Timing

La cantidad mínima de instrucciones para mover un pin de I/O mediante bit banding es de tres. Necesitamos un registro del micro para apuntar al registro de I/O, y otro para contener el valor a escribir; luego la operación de escritura indexada realiza la operación. Siguientes operaciones sobre el mismo port pueden hacerse con dos instrucciones, o incluso una si reutilizamos el registro que contiene el valor (0 ó 1). Las operaciones de modificación de varios bits simultáneamente requieren de una lectura del port de I/O a un registro, modificación del registro, y escritura al port de I/O.

Este proceso demora muy poco tiempo, el timing para acciones sucesivas sobre I/O viene dado por las transacciones en el bus AHB y luego el APB, sumado al tiempo del procesador para ordenar el cambio (ejecución de las instrucciones). El tiempo mínimo de cycling de un pin de I/O corresponde a la máxima velocidad de transferencia en estos buses, que corresponde a 2 clocks en cada uno, siendo el clock de ambos configurable poir el usuario, 72MHz como máximo y por defecto; sumado al tiempo de ejecución del procesador. Esto nos da un mínimo observado de aproximadamente 100ns[5].

Clock y funciones alternativas (AFIO)

Antes de poder utilizar los GPIO, debemos habilitar el APB clock a su circuitería. Esto se realiza en un registro de control de la unidad de control de clock (CKCU):

```
HT_CKCU->APBCCR0 |= (1<<17)+(1<<16);  // 17=PBEN, 16=PAEN
```

Compartiendo los pines con otros periféricos

Si deseamos que otro periférico utilice algún pin, necesitamos configurar una función alternativa en el AFIO (Alternate Function Input Output). Los registros de éste disponen de dos bits por cada pin de I/O, que permiten seleccionar una de 4 opciones de conexión. Veremos esto al utilizar cada periférico en particular. También debemos habilitar el clock del AFIO:

```
HT_CKCU->APBCCR0 |= (1<<14);  // 14=AFIOEN
```

ADC

El conversor A/D tiene una frecuencia máxima de operación de 14MHz, debe tenerse en cuenta el divisor a utilizar para respetarla. Puede realizar una o varias conversiones, donde la sucesión de canales es configurable. El inicio de conversión puede configurarse para hacerlo manual, es decir, por software. o automáticamente mediante un timer o una interrupción.

Los pines a utilizar como entradas analógicas se configuran en el AFIO.

La resolución del conversor es de 12-bits. 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.

Inicialización

Debemos habilitar los clocks de AFIO y el ADC

```
HT_CKCU->APBCCR0 |= (1<<14);          // 14=AFIOEN
HT_CKCU->APBCCR1 |= (1<<24);          // 24=ADCEN
```

Utilizaremos el primer canal, que se encuentra en PA0. Por defecto la circuitería digital de entrada se encuentra inhabilitada. Si necesitamos inhabilitarla manualmente, podemos escribir en el registro INER del GPIOA, pero recordemos que en ese caso también debemos habilitar el clock del GPIOA (al menos antes de escribir en INER, podemos inhabilitarlo después)

5 observado con osciloscopio en una rutina optimizada para escribir un display

```
HT_GPIOA->INER &= ~(1<<0);            // PA0, desconecta circuito de entrada
```

Le decimos al AFIO que nos preste el pin PA0 para el ADC:

```
HT_AFIO->GPACFGR = 0x01;              // PA0 -> AF1 (ADC)
```

Configuramos el clock del ADC

```
HT_CKCU->APBCFGR= (0x06<<16);         // ADC clock = APB/64 (72/64 = 1,125Mhz)
```

y procedemos a inicializarlo según nuestro interés:

```
// Sampling time = (ADSTn [7:0] + 1.5) A/D Converter clock cycles.
HT_ADC->STR[0] = 10;                  // 11.5 clock cycles -> ~10us
// Lista de canales a convertir, en secuencia
HT_ADC->LST[0] = 0 + (0<<8) + (0<<16) + (0<<24); // PA0
// Cantidad de Canales a convertir -1 , modo de conversión: 0 -> single, 2 -> continuo
HT_ADC->CONV = (0<<8) + 0;            // 1 canal, single
// Señal de inicio de conversión: b0=ADSW, b1=ADEXTI, b2=ADTM
HT_ADC->TCR = (1<<0);                 // software
```

Si vamos a utilizar interrupciones, debemos además configurar al ADC para generarlas al final de la conmversión, y al NVIC para aceptarlas:

```
NVIC_EnableIRQ(ADC_IRQn);             // Habilita ADC Interrupts
HT_ADC->IM |= (1<<2);                 // ADIMC, interrupción al final de conversión
```

La función de operación sobre el NVIC es parte de CMSIS.

Detección de fin de conversión y lectura

Para esto debemos observar los bits *ADIRAW* en el registro *ADCIRAW*, según el tipo de conversión. Este bit se setea al finalizar la conversión, y debe ser reseteado manualmente.

En la familia HT32F125x, el dato convertido se almacena en un registro interno y requiere de algunos ciclos de clock del ADC para pasar al registro de datos. Por este motivo, si se lee inmediatamente el registro de datos, se obtiene un dato anterior. Esto puede comprobarse observando el bit *ADVLD* de dicho registro, que se setea al escribirse el nuevo valor. Esto no ocurre si se lee este registro en el handler de interrupciones del ADC, dado que la interrupción se produce luego de almacenado el nuevo valor, o en la familia HT32F1755/65.

Modo polled

Iniciamos la conversión reseteando el bit que señala el fin de conversión y escribiendo en el registro de inicio de conversión por software_

```
HT_ADC->ICLR=(1<<2);                  // reset ADIRAWC
HT_ADC->TSR |= (1<<0);                // set ADSC para iniciar conversión
```

Detectamos el fin de conversión observando el flag de status de interrupciones (sí, aunque parezca raro)

```
HT_ADC->IRAW&(1<<2)
```

A continuación vamos a verificar que el dato convertido sea válida observando el bit 31 del registro de datos. Si no está seteado, debemos volver a leer (sólo HT32F125x):

```
HT_ADC->DR[ch]&(1U<<31))
```

El dato convertido lo tenemos en el mismo registro, en sus 12 bits menos significativos:

```
(uint16_t)HT_ADC->DR[ch]
```

Mediante interrupciones

Si deseamos utilizar interrupciones, la operatoria es muy similar, hasta más sencilla. No necesitamos resetear el flag de status al iniciar la conversión, dado que será resetado en el interrupt handler:

```
HT_ADC->TSR |= (1<<0);                // set ADSC para iniciar conversión
```

En el handler de interrupciones, sí reseteamos el flag de status y podemos leer el dato convertido directamente:

```
void ADC_IRQHandler(void)
{
uint16_t d;

        HT_ADC->ICLR = (1<<2);               // reset ADIRAWC
        d=(uint16_t)HT_ADC->DR[0];           // obtiene resultado
}
```

Conversión iniciada por un timer

En caso que deseemos realizar una conversión a instantes precisos y repetitivos, nos conviene configurar un timer para este menester (lo cual veremos en la sección correspondiente) y configurar al ADC para iniciar su conversión ante la señal provista por el timer, lo cual realizamos de la siguiente manera:

```
        // Fuente de inicio de conversión: b0=ADSW, b1=ADEXTI, b2=ADTM (más en HT32F1755/65)
        HT_ADC->TCR = (1<<2);                // Timer
        HT_ADC->TSR = (2<<16) + (1<<24);     // seleccionamos timer GPTM0, canal CH0
```

Timers GPTM

El micro posee un par de timers que pueden utilizarse en modo captura o comparación, pudiendo generar interrupciones, controlar pines de salida y generar señales PWM.

Comenzamos el proceso habilitando el clock APB para el módulo timer:

```
        HT_CKCU->APBCCR1 |= (1<<8);          // 8=GPTM0EN
```

A continuación inicializamos el timer según nuestro requerimiento. En este caso vamos a configurarlo con un prescaler por 36000 y hacerlo contar ascendente, para poder usar los registros de comparación independientemente para generar otro timing:

```
        HT_GPTM0->PSCR = 36000;              // /36000 prescaler
        HT_GPTM0->EVGR = 0x100;              // cargarlo ya
        HT_GPTM0->CRR = 0xFFFF;              // contar hasta 0xFFFF
        HT_GPTM0->CNTCFR = 0;                // cuenta ascendente
        HT_GPTM0->MDCFR = 0;                 // normal
        HT_GPTM0->CNTR = 0;                  // comienza a contar desde 0
        HT_GPTM0->CTR = (1<<0);              // arranca!
```

Luego configuramos el canal 0 en modo comparación y colocamos en su registro el número 2000, correspondiente a un tiempo de un segundo:

```
        HT_GPTM0->CH0ICFR = 0;               // CH0 -> compare
        HT_GPTM0->CH0CCR = 2000;             // valor a comparar (cuenta)
```

Si queremos que el timer nos interrumpa, lo habilitamos como vemos a continuación.

```
        HT_GPTM0->ICTR |= (1<<0);            // habilita interrupción de comparación en CH0
        NVIC_EnableIRQ(GPTM0_IRQn);          // Habilita timer Interrupts en NVIC
```

Si además queremos que este proceso se repita cada un segundo, debemos sumar 2000 a la cuenta y reescribir el registro, en el interrupt handler:

```
void GPTM0_IRQHandler(void)
{
        HT_GPTM0->INTSR = ~(1<<0);                          // Reset interrupt pending Bit
        HT_GPTM0->CH0CCR = (uint16_t)(HT_GPTM0->CH0CCR+2000);  // reload
}
```

Así, el canal 0 ya puede controlar al ADC para convertir cada 1 segundo.

Control de una salida

Si deseamos controlar una salida, debemos pedírsela prestada al AFIO y configurar el canal de acuerdo al modo que deseamos. En este caso lo configuramos para cambiar de estado ante cada comparación (toggle):

```
        HT_CKCU->APBCCR0 |= (1<<14);         // 14=AFIOEN

        HT_GPTM0->CHCTR &= ~(1<<0);          // inhabilita la salida
        HT_AFIO->GPACFGR |= (3<<6);          // PA3 -> AF3 (GPTM0_CH0), reclama el pin
        HT_GPTM0->CHPOLR &= ~(1<<0);         // activo en alto (compara => 1)
```

```
HT_GPTM0->CH0OCFR = 3;              // toggle
HT_GPTM0->CHCTR |= (1<<0);          // habilita salida
```

PWM

Si deseamos generar una señal PWM, jugaremos con el prescaler y el módulo del contador para lograr la resolución y frecuencia deseadas, por ejemplo podemos configurar el prescaler para dividir por 72 y el timer para contar hasta 1000, con lo cual obtenemos una señal de 1KHz con resolución de 1/1000. El ciclo de trabajo lo configuramos con el registro de comparación, al cual configuramos para que la salida cambie de estado ante una comparación y el overflow del contador:

```
HT_CKCU->APBCCR0 |= (1<<14);        // 14=AFIOEN
HT_CKCU->APBCCR1 |= (1<<8);         // 8=GPTM0EN

HT_GPTM0->PSCR = 72;                // /72 prescaler (1MHz)
HT_GPTM0->EVGR = 0x100;             // cargar ahora
HT_GPTM0->CRR = 1000;               // contar hasta 1000 (1KHz), resolución 1/1000
HT_GPTM0->CNTCFR = 0;               // cuenta ascendente
HT_GPTM0->MDCFR = 0;                // normal
HT_GPTM0->CNTR = 0;                 // comienza a contar desde 0
HT_GPTM0->CTR = (1<<0);             // arranca

HT_GPTM0->CH0ICFR = 0;              // configura CH0 para comparación
HT_GPTM0->CH0CCR = 250;             // ancho de pulso de 25% (250/1000)

HT_GPTM0->CHCTR &= ~(1<<0);         // inhabilita salida
HT_AFIO->GPACFGR |= (3<<6);         // PA3 -> AF3 (GPTM0_CH0)
HT_GPTM0->CHPOLR &= ~(1<<0);        // activo en alto
HT_GPTM0->CH0OCFR = 6;              // modo PWM, ancho de pulso = CCR0/CRR
HT_GPTM0->CHCTR |= (1<<0);          // habilita salida
```

USART

El micro incorpora una USART bastante completa, con FIFOs de 16 caracteres tanto en recepción como en transmisión. Veremos una forma de utilizarla en modo asincrónico, es decir, como UART, a una velocidad de operación de 9600bps.

Comenzamos habilitando los clocks de la UART y el AFIO, y reclamando los pines:

```
HT_CKCU->APBCCR0 |= (1<<14)+(1<<8);  // 14=AFIOEN, 8=UREN

HT_AFIO->GPACFGR &= ~(0xF<<16);      // PA8,9 AF: clear bits (AF0)
HT_AFIO->GPACFGR |= (0xA<<16);       // PA8,9 -> AF2 (UART)
```

Seguimos configurando el largo de palabra y el baud rate generator (clock):

```
HT_USART->LCR = 1;                  // 8-bits, no parity, 1 stop
HT_USART->MDR = 0;                  // asincrónico, LSB primero
HT_USART->DLR = 7500;               // APB (72MHz) / 7500 = 9600
HT_USART->TPR = 0;                  // no usamos funciones de timing
```

Modo polled

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

```
if((HT_USART->LSR&(1<<0)))          // RDR bit, dato disponible en FIFO (de 1 a 16)
```

leer el dato recibido:

```
        dato = HT_USART->RBR;
```

revisar si es posible enviar un dato:

```
if(!(HT_USART->LSR&(1<<5)))         // TXFEMPT bit, puedo enviar
```

enviar un dato:

```
HT_USART->TBR = dato;
```

Dado que no disponemos de un flag para saber si la FIFO está llena, no podemos escribir varios bytes por adelantado. Sí podemos esperar a leer los bytes recibidos, pues se acumulan en la FIFO

Mediante interrupciones

Utilizando interrupciones, podemos mantener una cuenta de los bytes escritos y resetearla cuando sabemos que la FIFO se vacía. De este modo podemos escribir varios bytes a la vez. En recepción, dado que tenemos una FIFO de 16-caracteres, no necesitamos un buffer circular, por lo que no realizamos modificaciones.

Inicializamos la UART para que genere interrupciones cuando la FIFO tiene 8 lugares (o más) libres, inicializamos una variable compartida, y habilitamos las interrupciones:

```
        HT_USART->FCR = (3<<6)+(3<<4);        // Rx FIFO: 14 bytes, Tx FIFO: 8 bytes
        txbytes=0;
        HT_USART->IER = (1<<1);               // interrupt en transmisión
        NVIC_EnableIRQ(USART_IRQn);           // Habilita Interrupts
```

Escribimos una función para enviar, que permite escribir hasta 8 bytes, de modo que en el peor de los casos llenamos la FIFO, sin desbordar:

```
int UART_tx(uint8_t data)
{
        if(txbytes>7)
                return(-1);                   // no hay lugar
        HT_USART->IER &= ~(1<<1);             // inhabilita TX interrupt
        HT_USART->TBR = data;                 // escribe dato en FIFO
        txbytes++;                            // mantiene la cuenta de cuántos hay en la FIFO
        HT_USART->IER |= (1<<1);              // habilita TX interrupt
        return(0);                            // OK
}
```

Luego, en el interrupt handler, sabemos que la FIFO nos avisa que hay espacio para 8 ó más caracteres, por lo que reseteamos la cuenta y sabemos que podemos volver a escribir:

```
void USART_IRQHandler(void)
{
        switch(HT_USART->IIR & 0xF){          // determina qué función interrumpe
        case 2:                               // Tx int, FIFO con 8 ó más lugares disponibles
                txbytes=0;                    // reset cuenta
                HT_USART->IER &= ~(1<<1);     // inhabilita interrupt
                break;
        }
}
```

En el caso de trabajar enviando y recibiendo mensajes, podemos aprovechar ambas FIFOs e interrupciones. No desarrollaremos este tema aquí.

Amplificador operacional y comparador

Este módulo es tal vez el más fácil de usar. Simplemente debemos habilitar el clock y pedirle los pines al AFIO:

```
        HT_CKCU->APBCCR0 |= (1<<14);          // 14=AFIOEN
        HT_CKCU->APBCCR1 |= (1<<22);          // 22=OPA0EN

        HT_AFIO->GPBCFGR &= ~(0x3F<<4);       // PB2,3,4 AF: clear bits (AF0)
        HT_AFIO->GPBCFGR |= (0x15<<4);        // PB2,3,4 -> AF1 (CN0,CP0,AOUT0)

        HT_CMP_OP0->OPAC = 1;                 // habilita como operacional.
//      HT_CMP_OP0->OPAC = 3;                 // habilita como comparador
```

En caso que se requiera, podemos minimizar el offset escribiendo en un registro un valor de compensación, determinado mediante un procedimiento previo de calibración.

```
        HT_CMP_OP0->OFVC = offset;            // load offset cancellation value
```

En el caso de utilizarlo como comparador, podemos hacer que genere interrupciones ante un cambio de estado. Como operacional, podemos utilizarlo “fuera del micro” o conectarlo a una entrada del ADC uniendo los pines “por afuera”.

Watchdog

El watchdog puede resetear la micro ante un timeout, o puede configurarse para una actitud menos drástica como por ejemplo generar una interrupción de alta prioridad que mire qué está sucediendo. Es posible configurar el reloj,

utilizando un generador interno de 32768Hz, el timing, y una ventana de seguridad de modo que deba ser reseteado dentro de un tiempo determinado, no antes ni después.

```
#define RESTART_KEY    0x5FA00000U
#define PROTECT_KEY    0x000035CAU

        HT_CKCU->GCFGR &= ~(1<<3);            // 3=WDTSRC, 0 -> internal 32768, 1-> external 32768
        HT_CKCU->APBCCR1 |= (1<<4);           // 4=WDTEN

        HT_WDT->PR = PROTECT_KEY;             // podemos escribir en el WDT
        HT_WDT->MR0 = (1<<12) + 0xFFF;        // WDT expira => interrupt, /2^12
//      HT_WDT->MR0 = (1<<13) + 0xFFF;        // WDT expira => reset, /2^12
        HT_WDT->CR = RESTART_KEY | (1<<0);    // reload counter
        HT_WDT->MR1 = (7<<12) + 0x555;        // /128 prescaler, 0x555=1/3 del total de cuentas
        HT_WDT->PR = 0;                       // ya no podemos escribir en el WDT (seguridad)
        /* 128/32768 * 2^12 = 16s de timeout y 555/FFF = 1/3 de ventana (~5s)
        resetearemos el WDT entre 11s y 16s a partir de ahora */
        NVIC_EnableIRQ(WDT_IRQn);             // habilita int
```

En el loop que queremos proteger, resetearemos el watchdog escribiendo el valor correcto en el registro correcto, en el intervalo correcto de tiempo:

```
        HT_WDT->CR = RESTART_KEY | (1<<0);
```

En el interrupt handler, podemos saber si llegamos allí porque el watchdog fue reseteado antes del tiempo previsto, o no fue reseteado:

```
void WDT_IRQHandler(void)
{
        if(HT_WDT->SR&(1<<0))
                ;// timeout sin reset del WDT
        if(HT_WDT->SR&(1<<1))
                ;// reset del WDT antes de tiempo
}
```

FMC: escritura en flash

La flash del micro está dividida en segmentos de 1KB. Cada uno de ellos puede borrarse por separado y cada palabra dentro de este segmento puede escribirse mediante un conjunto de registros del FMC (Flash Memory Controller). Un segmento especial de flash contiene los bits de configuración de protección, lo que en otras arquitecturas se conoce como option byte, option register, fuses, etcétera. Es posible utilizar el resto de este segmento para almacenar datos de calibración, de configuración, o lo que el developer considere oportuno.

Para operar sobre la flash disponemos de un registro donde escribimos la operación a realizar, otro donde indicamos la dirección de memoria, y otro donde escribimos el dato que queremos escribir. Un flag de estado nos indica cuándo ha concluido la operación. Para más información, consultar la nota de aplicación CAN-100.