Técnicas Digitales II

Arquitectura de 32bits – ARM

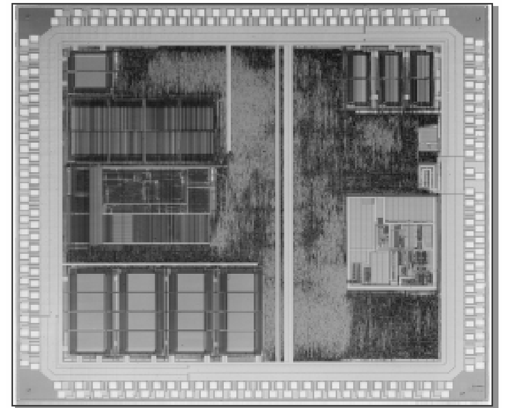
Enfoque teórico y práctico

Autor: Pablo Luis Joaquim

Índice

Objetivo....3
Historia....3
Arquitectura ARM7....4
  Introducción....4
  El pipeline de instrucciones (3 etapas)....5
  Excepciones....5
    Proceso de tratamiento de una excepción....6
    Vector de Excepciones....6
  Formatos de memoria....6
  Rendimiento, densidad de código y ciclos de operación....7
  Registros....8
  Modos de operación....8
  Registros de modo ARM....9
  Registros de modo THUMB....9
  Descripción de los Registros....10
  RISC y CISC....12
    CISC:....12
    RISC:....13
  Coprocesador....14
  Debug....14
Aplicaciones....16
Otras arquitecturas ARM....16
  ARM7TDMI-S....16
  ARM7EJ-S....16
  ARM720T....16
  Familia ARM9....16
  ARM920T y ARM922T....17
  ARM940T....17
  Familia ARM11....17
Ejemplos de código....18
  Led....18
  Serie_por_Pooling....20
  Multi_Interrupciones....22
  Serie_por_Interrupciones....25
  Tipos_de_Memoria....28
    Main.c....28
    Variables.h....31
    Seno_con_Ruido.h....32
    Dig2.m....32
  ARM y THUMB....34
    FFT....34
  Usuario&Supervisor....39
    Main.c....39
    TimerTick.c....42

Objetivo



El objetivo del presente documento es tratar de explicar de una forma rápida y sencilla los conceptos básicos de la arquitectura de 32-bits, en especial aplicada a los microcontroladores ARM7, tratando de lograr, al finalizar su lectura, que el programador/lector pueda sentarse frente a la computadora dispuesto a ingresar al mundo de las aplicaciones que hoy dominan el mundo tecnológico de 32-bits sin prejuicios, y de manera casi inmediata.

Comentaremos algo de su historia. Revisaremos la arquitectura del microprocesador ARM (Advanced Risc Machine) de ARM, Ltd.

Nombraremos algunos dispositivos que hoy en día constan con este tipo de dispositivos.

Presentaremos una serie de ejemplos aplicados a un procesador en particular, tratando de ejemplificar de forma clara y concisa los conceptos enunciados.

Historia

El diseño del ARM comenzó en 1983 como un desarrollo de la empresa Acorn Computers Ltd. Roger Wilson y Steve Furber lideraban el equipo, cuya meta era, originalmente, el desarrollo de un procesador avanzado, pero con una arquitectura similar a la del MOS 6502. La razón era que Acorn tenía una larga línea de computadoras personales basadas en dicho micro, con lo que la idea era desarrollar uno nuevo con el que los desarrolladores de aplicaciones para dichos ordenadores se sintieran cómodos.

El equipo terminó el diseño preliminar y los primeros prototipos del procesador en el año 1985, al que llamaron ARM1. La primera versión utilizada comercialmente se bautizó como ARM2 y se lanzó en el año 1986.

La arquitectura del ARM2 posee un bus de datos de 32 bits y ofrece un espacio de direcciones de 26 bits, junto con 16 registros de 32 bits. Uno de estos registros se utiliza como contador de programa, aprovechándose sus 4 bits superiores y los 2 inferiores para contener los flags de estado del procesador.

El ARM2 es probablemente el procesador de 32 bits útil más simple del mundo, ya que posee sólo 30.000 transistores. Su simplicidad se debe a que no está basado en microcódigo (sistema que suele ocupar alrededor de la cuarta parte de la la pastilla del procesador) y a que, como era común en aquella época, no incluye caché. Gracias a esto, su consumo de energía es bastante bajo, a la vez que ofrece un mejor rendimiento que un 286. Su sucesor, el ARM3, incluye una pequeña memoria caché de 4 Kb, lo que mejora los accesos repetitivos a memoria.

A finales de los años 80, Apple Computer comenzó a trabajar con Acorn en nuevas versiones del núcleo ARM.

En Acorn se dieron cuenta de que el hecho de que el fabricante de un procesador fuese también un fabricante de ordenadores podría alejar a la competencia, por lo que se decidió a crear una nueva compañía llamada Advanced RISC Machines, que sería la encargada del diseño y gestión de las nuevas generaciones de procesadores ARM. Ocurría esto en el año 1990.

Este trabajo derivó en el ARM6, presentado en 1991. Apple utilizó el ARM 610 (basado en el ARM6), como procesador básico para su innovador PDA, el Apple Newton. Por su parte, Acorn lo utilizó en 1994 como procesador principal en su RiscPC.

El núcleo mantuvo su simplicidad a pesar de los cambios: en efecto, el ARM2 tiene 30.000 transistores, mientras que el ARM6 sólo cuenta con 35.000. La idea era que el usuario final combinara el núcleo del ARM con un número

opcional de periféricos integrados y otros elementos, pudiendo crear un procesador completo a la medida de sus necesidades.

La mayor utilización de la tecnología ARM se alcanzó con el procesador ARM7TDMI, con millones de unidades en teléfonos móviles y sistemas de videojuegos portátiles.

DEC licenció el diseño, lo cual generó algo de confusión debido a que ya producía el DEC Alpha, y creó el StrongARM. Con una velocidad de reloj de 233 Mhz, este procesador consumía solo 1 watt de potencia (este consumo de energía se ha reducido en versiones más recientes). Esta tecnología pasó posteriormente a manos de Intel, como fruto de un acuerdo jurídico, que la integró en su línea de procesadores Intel i960 e hizo más árdua la competencia.

Freescale (una empresa que derivó de Motorola en el año 2004), IBM, Infineon Technologies, Texas Instruments, Nintendo, Philips, VLSI, Atmel, Sharp, Samsung y STMicroelectronics también licenciaron el diseño básico del ARM.

El diseño del ARM se ha convertido en uno de los más usados del mundo, desde discos rígidos hasta juguetes. Hoy en día, cerca del 75% de los procesadores de 32 bits poseen este chip en su núcleo.

Arquitectura ARM7

Introducción

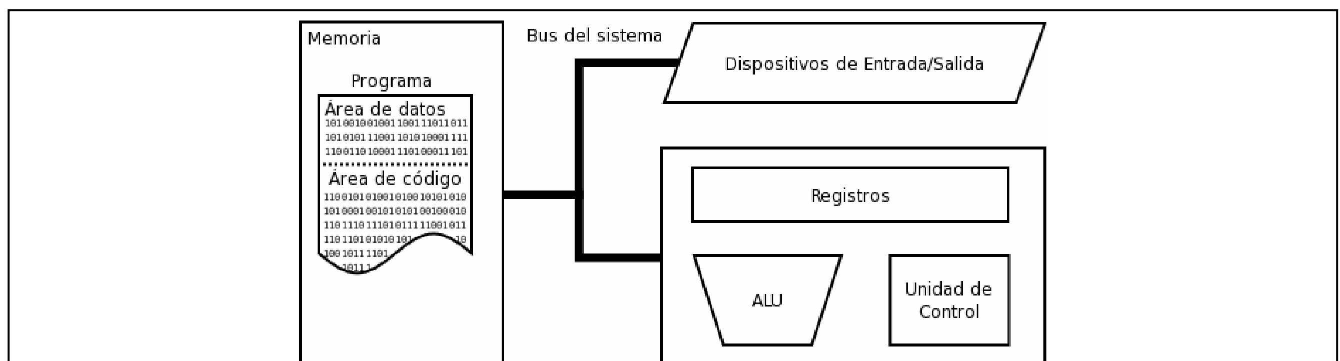
Hablaremos en particular sobre la arquitectura ARM7TDMI por ser la mas utilizada en la gran mayoría de las aplicaciones industriales de microprocesadores RISC de 32bits. Provee bajo consumo, pequeño tamaño, y alto rendimiento, tan necesario en aplicaciones portátiles y embebidas.

Existen otras familias ARM7 compatibles como la ARM7TDMI-S, ARM720T, y la ARM7EJ-S, todas ellas con sutiles diferencias entre sí.

El software desarrollado para ARM7TDMI es totalmente compatible hacia arriba, es decir con familias ARM9, ARM9E, ARM10, StrongARM y XScale.

Existen sistemas operativos embebidos para estas arquitecturas como Windows CE, Linux, Palm OS y Symbian OS; sistemas operativos en tiempo real como QNX, Wind River, VxWork y VRTX de Mentor Graphics.

El núcleo ARM7TDMI esta basado en arquitectura Von Neumann, con un bus de 32-bits de datos e instrucciones. Los datos pueden accederse de a 8, 16, o 32 bits.

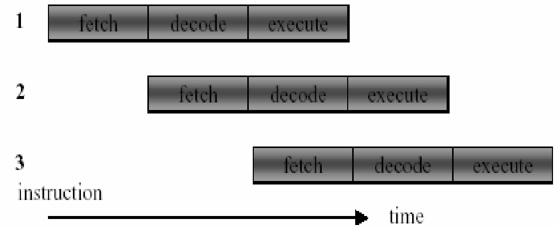


Arquitectura VonNeumann

El concepto central en la Arquitectura Von Neumann es que las instrucciones y los datos tenían que almacenarse juntos en un medio común y uniforme , en vez de separados , como hasta entonces se hacía. De esta forma , no sólo se podían procesar cálculos , sino que también las instrucciones y los datos podían leerse y escribirse bajo el control del programa. A partir de esta idea básica se sigue que un elemento en la memoria tiene una calidad ambigua con respecto a su interpretación; esta ambigüedad se resuelve , sólo temporalmente , cuando se requiere ese elemento y se ejecuta como una instrucción , o se opera como un dato. Un beneficio de esta ambigüedad es el hecho de que un dato , obtenido como resultado de algunas operaciones en la unidad aritmetico-lógica del computador , podía colocarse en la memoria como si fuera cualquier otro dato , para entonces usarlo y ejecutarlo como si fuera una instrucción.

El pipeline de instrucciones (3 etapas)

- Permite superponer en el tiempo la ejecución de varias instrucciones a la vez.
- No requiere hardware adicional. Solo se necesita lograr que todas las partes del procesador trabajen a la vez.
- Trabaja con el concepto de una línea de montaje:
  - Cada operación se descompone en partes.
  - Se ejecutan en un mismo momento diferentes partes de diferentes operaciones.
  - Cada parte se denomina etapa (stage).



- **El resultado:** Una vez entrado en regimen ejecuta a razón de una instrucción por ciclo de clock.
- Una instrucción de salto provoca el borrado del pipeline.

Excepciones

Se llaman excepciones a los eventos que se producen durante la ejecución de un programa, por ejemplo, para atender una interrupción desde un periférico. Antes de tratar de manejar una excepción, el procesador ARM7TDMI resguarda el estado actual del programa para que luego de darle atención al handler pueda regresar a su estado original.

Si dos o mas excepciones llegan simultáneamente, se atenderán de acuerdo a un orden prefijado según la siguiente tabla:

PRIORIDAD	EXCEPCIÓN
1 (Máxima Prioridad)	Reset
2	Data Abort
3	FIQ
4	IRQ
5	Prefetch Abort
6 (Mínima Prioridad)	Undefined instruction SWI

El handler de las excepciones se ejecuta siempre en modo ARM. Si una excepción ocurre en modo THUMB, el procesador pasa automáticamente a modo ARM. El regreso a modo THUMB al finalizar la ejecución del handler también es automático. Puede ejecutarse en modo THUMB, cambiándolo “manualmente”, pero es necesario cambiar a modo ARM antes de volver para finalizar correctamente.

Al hablar de excepciones nos referimos tanto a interrupciones de hardware como de software, aunque estas últimas no puedan considerarse un evento inesperado, se consideran excepciones por el tratamiento que se hace de ellas.

Lo mismo sucede con el RESET del microprocesador, su tratamiento es similar al de una excepción.

Las excepciones del ARM se dividen de la siguiente manera:

- Excepciones generadas por la ejecución de una instrucción:
  - Interrupciones de software
  - Instrucciones no definidas
  - Prefetch aborts, debido a un error en el acceso a memoria al buscar un código de operación.

- Excepciones generadas como efecto secundario en la ejecución de una instrucción:
  - Data aborts, debido a un error en el acceso a memoria al buscar o almacenar un dato.

- Excepciones generadas externamente
  - RESET
  - IRQ
  - FIQ

Proceso de tratamiento de una excepción

Todas las excepciones tienen el mismo tratamiento excepto la del RESET, nos referiremos a ella mas abajo.

Al llegar una excepción se procede de la siguiente manera:

1. Se finaliza la ejecución de la instrucción en curso
2. La siguiente instrucción a ejecutar será la correspondiente a la rutina de la excepción asociada (entramos en modo ARM).
3. Se cambia el modo de trabajo al correspondiente a la excepción asociada.
4. Se salva el contador de programa en el r14 del modo al que se ha conmutado.
5. Se salva el CPSR en el SPSR correspondiente.
6. Se desactivan las interrupciones IRQ, y la FIQ en caso en que esta sea la fuente de la excepción.
7. Se carga en el PC el vector correspondiente al modo en que se entra.

En caso de producirse un RESET se actúa de la siguiente forma:

1. Los registros r14 y SPSR toman valores impredecibles
2. Se activa el modo Supervisor
3. Se activa el modo ARM
4. Se inhabilitan las interrupciones IRQ y FIQ
5. Se carga en el PC el vector de reset

En el retorno de una excepción se debe:

1. Recuperar el contenido de los registros empleados en la rutina.
2. Restaurar el CPSR a partir del SPSR.
3. Restaurar el valor del PC a su valor antes de la ejecución de la rutina.

La dos últimas operaciones se realizan simultáneamente.

Vector de Excepciones

Excepción	Modo	Dirección del vector
Reset	SVC	0x00000000
Undefined instruction	UND	0x00000004
Software Interrupt(SWI)	SVC	0x00000008
Prefetch Abort (instruction fetch memory fault)	Abort	0x0000000C
Data Abort (data access memory fault)	Abort	0x00000010
IRQ (normal interrupt)	IRQ	0x00000018
FIQ (fast interrupt)	FIQ	0x0000001C

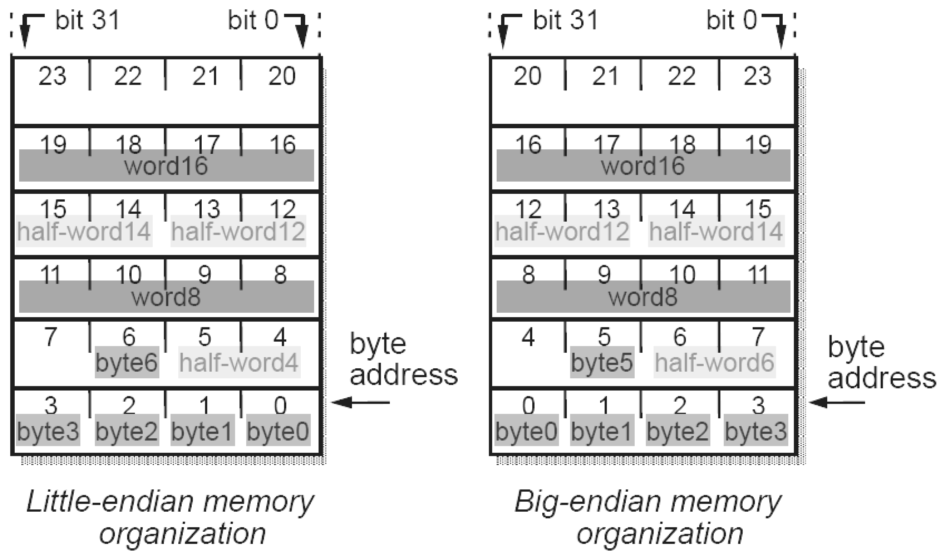
En la dirección de cada vector se pone una instrucción de salto a la dirección de la rutina de tratamiento de la excepción, excepto en las interrupciones FIQ, que al ser el último vector, puede empezar el código de forma inmediata, con el consiguiente ahorro de tiempo.

Formatos de memoria

El ARM7TDMI puede configurarse para trabajar tanto con el formato de almacenamiento de words big-endian como little-endian. El sistema Big-Endian, adoptado por motorota entre otros, consiste en representar los bytes en el orden “natural”, a diferencia del sistema little-endian adoptado por Intel, entre otros, en el que los datos se almacenan al revés.

Como ejemplo podemos decir que el número hexadecimal 0xAABBCCDD se almacenaría en formato Big-Endian de la forma: {AA, BB, CC, DD}, mientras que en el formato Intel (al que estamos acostumbrados a trabajar) para el mismo ejemplo sería {DD, CC, BB, AA}.

A estas arquitecturas que manejan ambos formatos a veces se las denomina middle-endian.



Rendimiento, densidad de código y ciclos de operación

El ARM7TDMI tiene dos modos de trabajo con un set de instrucciones para cada uno:

- ARM de 32bits
- THUMB de 16bits

Una instrucción en modo ARM es de 32bits de longitud. El modo THUMB es un modo de trabajo comprimido de 16bits. Se pueden lograr velocidades de acceso mayores a memorias de 16bits y mayor densidad de código trabajando en modo THUMB, lo que hace a esta arquitectura ideal para aplicación embedded.

Sin embargo el modo THUMB posee las siguientes limitaciones:

- El código generado en modo THUMB utiliza mas instrucciones para la misma tarea. Con lo cual el código en modo ARM es mas eficiente para maximizar el rendimiento en aplicaciones donde el tiempo de ejecución es crítico.
- El modo THUMB no posee ciertas instrucciones necesarias para el manejo de excepciones, con lo cual el micro conmuta a modo ARM para atender una excepción.

En caso de tener una memoria de código de 16bits, trabajando en modo ARM será necesario hacer dos accesos a memoria para leer el código de operación, con lo cual en estos casos es recomendable utilizar el modo THUMB.

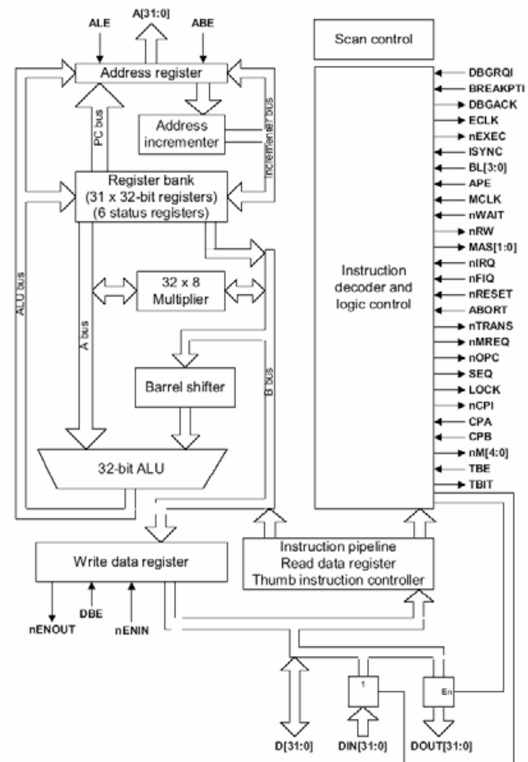
Sin embargo la RAM, en estos sistemas, suele ser de 32bits de ancho, con lo cual al ejecutar código desde la RAM en modo ARM suele ser lo más adecuado cuando el tiempo es crítico en la operación.

Ambos modos de trabajo pueden utilizarse alternativamente en el mismo programa, como veremos mas adelante.

Registros

El núcleo del ARM7TDMI consta de un bus de 32bits y una lógica de control asociada. Este bus conecta a 31 registros de 32 bits de propósitos generales, 7 regitros de 32 bits de uso dedicado, asociados a un barrel-shifter, ALU (Arithmetic Logic Unit) y un multiplicador.

- Banco de registros (Register Bank)
- Barrel Shifter: Desplaza a izquierda o derecha el dato (multiplicaciones y divisiones por 2 a alta velocidad).
- Registro de direcciones e incrementador de direcciones (Address registrer+incrementer): Almacena las direcciones de acceso a memoria, incrementa la dirección automáticamente para mejorar el rendimiento de ciclos de lectura repetitivos a memoria, tipo ciclo for.
- Registro de datos (Data Register): Almacena los datos resultados de una transferencia a memoria.
- Control y decodificador de instrucciones (Instruction decode & control).
- Controlador de instrucciones Thumb (Thumb Instruction controller): Descomprime las instrucciones de 16bits en instrucciones de 32bits con las que trabaja la arquitectura.



Modos de operación

El ARM7TDMI tiene siete modos de trabajo:

- User Mode: Es el estado normal de ejecución del programa.
- Fast Interrupt (FIQ): Interrupciones mas rápidas, poseen mayor prioridad que una interrupción normal, por lo cual pueden interrumpirlas. Ideales para transferencias de datos, procesamiento por canales de conversión AD, etc.
- Interrupt (IRQ): Para la atención de interrupción de propósito general.
- Supervisor Mode: Es un modo de trabajo protegido, esta pensado para trabajar con sistemas operativos, no puede ser interrumpida su ejecución.
- Abort Mode: Usado cuando se aborta el ciclo fetch fetch de datos o de datos o instrucciones.
- System Mode: Es un modo con privilegio para ejecución de tareas del sistema operativo.
- Undefined Mode: Entramos en este modo cuando se ejecuta una operación indefinida (por ejemplo en caso de un error en la memoria de código).

Todos los modos a excepción del User Mode son modos privilegiados. Estos se utilizan para atender interrupciones de hardware, excepciones, interrupciones de software. Cada modo privilegiado tiene asociado un SPSR (Saved Program Status Register). Este registro se utiliza para almacenar el CPSR (Current Program Status Register) de la tarea que estaba en curso antes que la excepción ocurra.

En estos modos privilegiados, hay bancos de registros específicos para cada uno. Estos vuelven a su estado original automáticamente al volver al estado previo de la excepción.

El System Mode no tiene bancos de registros, utiliza los registros de modo usuario.

El System Mode ejecuta las tareas que requieren un tratamiento privilegiado y permite invocar a toda clase de excepciones.

Registros de modo ARM

En modo ARM, son accesibles 16 registros de uso general y uno o dos registros de estado. En modo Privilegiado están disponibles registros banqueados específicamente para cada modo.

En la siguiente figura vemos que registros estan disponibles en cada modo.

El conjunto de registros de modo ARM contiene 16 registros, de r0 a r15. Además de un registros CPSR, que contiene los flags de condición y los del modo de ejecución actual. Los registros r0 a r13 son de propósito general para almacenar datos o direcciones.

Los registros r14 y r15 tienen funciones especiales: Link Register (aquí se copia el r15 cuando se ejecuta una instrucción BL (Branco with Link) y Program Counter (PC) respectivamente.

Por conveción el r13 se utiliza como el Snack Pointer (SP).

Los registros de modo ARM son los siguientes:

	System and User	FIQ	Supervisor	Abort	IRQ	Undefined
General registers	r0	r0	r0	r0	r0	r0
	r1	r1	r1	r1	r1	r1
	r2	r2	r2	r2	r2	r2
	r3	r3	r3	r3	r3	r3
	r4	r4	r4	r4	r4	r4
	r5	r5	r5	r5	r5	r5
	r6	r6	r6	r6	r6	r6
	r7	r7	r7	r7	r7	r7
	r8	r8_fiq	r8	r8	r8	r8
	r9	r9_fiq	r9	r9	r9	r9
	r10	r10_fiq	r10	r10	r10	r10
	r11	r11_fiq	r11	r11	r11	r11
	r12	r12_fiq	r12	r12	r12	r12
	r13	r13_fiq	r13_svc	r13_abt	r13_irq	r13_und
	r14	r14_fiq	r14_svc	r14_abt	r14_irq	r14_und
Program counter	r15 (PC)	r15 (PC)	r15 (PC)	r15 (PC)	r15 (PC)	r15 (PC)
Program status registers	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		SPSR_fiq	SPSR_svc	SPSR_abt	SPSR_irq	SPSR_und

ARM-state program status registers

= banked register

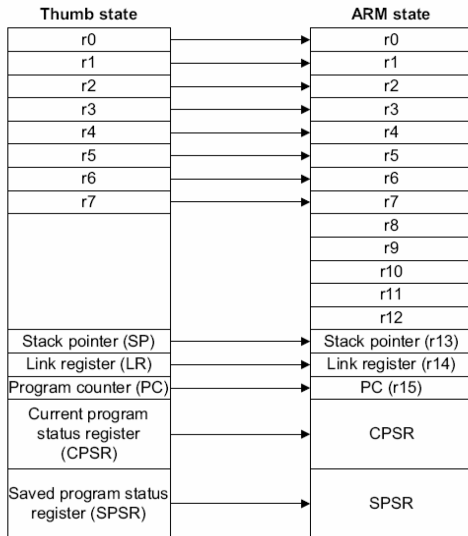
Registros de modo THUMB

El conjunto de registros de modo THUMB son un subconjunto de los registros de modo ARM. También hay registros banqueados para cada modo privilegiado.

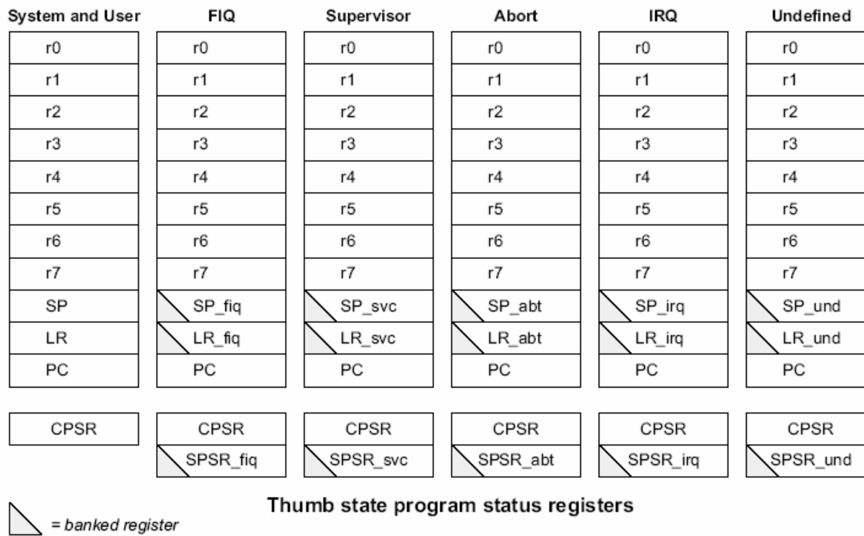
Los registros de modo ARM y THUMB se relacionan de la siguiente forma:

- Los registros de modo THUMB r0 a r7 y los de modo ARM r0 a r7 son idénticos.
- Lo mismo sucede con el CPSR y el SPSR.
- El registro SP de modo THUMB se mapea como el r13 de los registros de modo ARM.
- El registro LR de modo THUMB se mapea como el r14 de los registros de modo ARM.
- El registro PC de modo THUMB se mapea como el r15 (PC) de los registros de modo ARM.

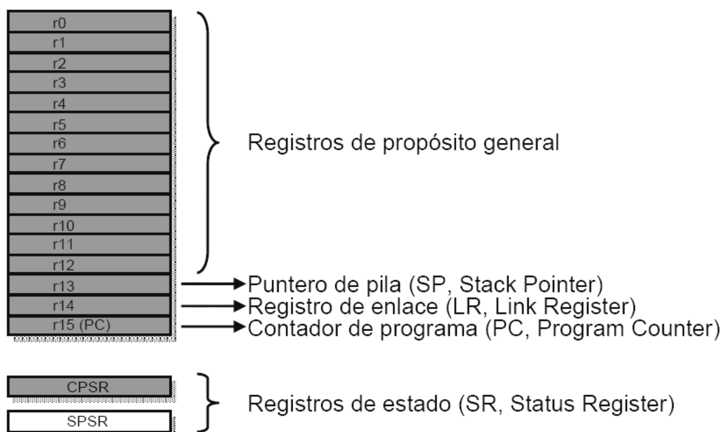
Vemos estas relaciones en el siguiente gráfico:



Los registros de modo THUMB son los siguientes:



Descripción de los Registros



- Registros de propósito general:
  - Unbanked Registers R0-R7:
    - Son comunes a todos los modos de funcionamiento.
  - Banked Registers R8-R14:
    - Se accede a distintos registros dependiendo del modo en que se encuentre.

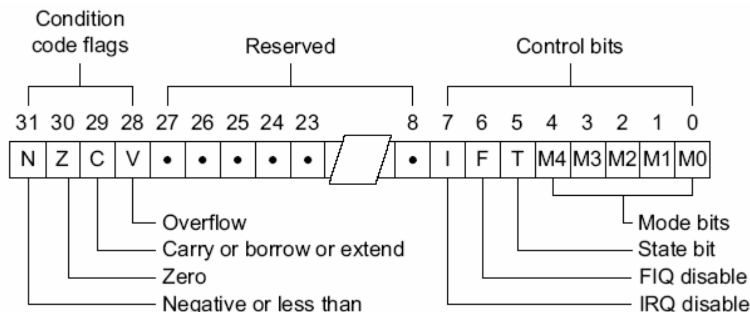
- R13 es el SP (Stack Pointer) por convención:
  - No hay instrucciones específicas excepto en modo THUMB.
  - Se emplea para el manejo de la pila.
  - Cada modo de funcionamiento tiene su propio SP que debe ser inicializado a una zona de memoria dedicada.

- R14 es el LR (Linked Register):
  - Existe uno para cada modo de funcionamiento.
  - Permite almacenar la dirección de retorno en una llamada a rutina.
  - Evita el almacenamiento del contador de programa en la pila.
  - Proporciona un retorno rápido.
  - Se emplea también para almacenar la dirección de retorno cuando se producen excepciones.

- R15 es el PC (Program Counter):
  - Contiene la dirección de la instrucción que se va a ejecutar.
  - Se puede emplear como registro de propósito general salvo determinadas excepciones.
  - La lectura del contador de programa mediante una instrucción devuelve el valor de la dirección actual+8bytes.
  - Debe tenerse en cuenta para realizar direccionamientos relativos.
  - La escritura del PC provoca un salto a la dirección guardada.
  - La dirección almacenada debe ser multiplo de 4, ya que las instrucciones son de tamaño word en modo ARM, y multiplo de 2 en modo THUMB ya que las instrucciones son de tamaño half-word.

- Registro de estado CPSR (Current Program Status Register):
  - Accesible en todos los modos de funcionamiento.
  - Contiene los flags de condición.

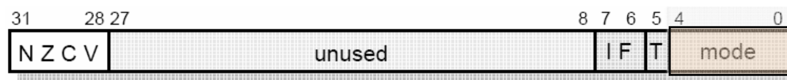
- SPSR (Saved Program Status Register):
  - Contiene una copia del CPSR cuando se entra en un modo privilegiado.
  - Tienen este registro todos los modos excepto User y System.



- Flags de condición:
  - N: Bit de signo, 0-positivo, 1-negativo
  - Z: Bit de cero, 0-resultado de la operación no dio 0, 1-resultado de la operación dio 0
  - C: Bit de Carry
  - V: Overflow
  - Q: Desbordamiento/Saturación (**solo en versiones E**).

- Bits de control:
  - I:Puesto a 1 deshabilita las interrupciones IRQ.
  - F: Puesto a 1 deshabilita las interrupciones FIQ.
  - T: Thumb, 0-Ejecuta ARM, 1-Ejecuta THUMB

- Bits de modo:
  - M0,M1,M2,M3 y M4 representan el modo de operación del micro



CPSR[4:0]	Mode	Use	Registers
10000	User	Normal user code	user
10001	FIQ	Processing fast interrupts	_fiq
10010	IRQ	Processing standard interrupts	_irq
10011	SVC	Processing software interrupts (SWIs)	_svc
10111	Abort	Processing memory faults	_abt
11011	Undef	Handling undefined instruction traps	_und
11111	System	Running privileged operating system tasks	user

RISC y CISC

CISC:

CISC es un modelo de arquitectura de computadores (del inglés Complex Instruction Set Computer, Repertorio de instrucciones complejo). Los microprocesadores CISC tienen un conjunto de instrucciones que se caracteriza por ser muy amplio y permitir operaciones complejas entre operandos situados en la memoria o en los registros internos, en contraposición a la arquitectura RISC.

Antes de 1980 el principal objetivo en el desarrollo de los repertorios de instrucciones era aumentar su complejidad de forma de reducir la del compilador.

Simples instrucciones del repertorio se descomponían en complejas secuencias de operación utilizando varios ciclos de clock.

Los procesadores se vendían por su nivel de complejidad, número de modos de direccionamiento, tipos de datos que manejaba, etc.

Esto se origino en los años 70 con el desarrollo de minicomputadoras. Estas tenían una memoria principal relativamente lenta asociada al procesador que estaba constituído por varios circuitos integrados mas simples.

El procesador era controlado por el código alojado en ROM (que eran mas veloces que la memoria principal), con lo cual tenía sentido utilizar operaciones que reunieran secuencias de microcódigo a aquellas que requerían varias instrucciones a ser seguidas por la memoria principal.

Sin embargo esta memoria ROM ocupaba una desproporcionada proporción del área del chip, dejando poco espacio para otras funcionalidades que pudieran mejorar el rendimiento del procesador.

Así la industria de los procesadores se volcó a realizar cada vez mas complejos procesadores, con mayor cantidad de transistores, siguiendo las demandas del mercado de las minicomputadoras, cada vez mas complejas. Esto dejaba a los chips basados en CISC (Repertorio de instrucciones complejo) de fines de los 70s un repertorio de instrucciones que comprometía al limitado silicio disponible.

Este tipo de arquitectura dificulta el paralelismo entre instrucciones, por lo que, en la actualidad, la mayoría de los sistemas CISC de alto rendimiento implementan un sistema que convierte dichas instrucciones complejas en varias instrucciones simples del tipo RISC, llamadas generalmente microinstrucciones.

Los CISC pertenecen a la primera corriente de construcción de procesadores, antes del desarrollo de los RISC. Ejemplos de ellos son: Motorola 68000, Zilog Z80 y toda la familia Intel x86 usada en la mayoría de ordenadores personales del planeta.

Hay que hacer notar, sin embargo que la utilización del término CISC comenzó tras la aparición de los procesadores RISC como nomenclatura despectiva por parte de los defensores/creadores de éstos últimos.

RISC:

En este escenario de repertorios de instrucciones cada vez mas complejo nacen los procesadores RISC. Este concepto fue utilizado en el desarrollo de los procesadores ARM, de hecho ARM significa “Advanced RISC microprocessor”.

En 1980 Patterson y Ditzel publican un paper titulado “The Case for the Reduced Instruction Set Computer” (“El caso de la computadora de repertorio de instrucciones reducido”) . En este trabajo exponían la visión de que la arquitectura para un procesador sinlge-chip no tenía que ser la misma que la óptima arquitectura que para un procesador multi-chip. Su argumento estaba sustentado en los resultados de un proyecto acerca del diseño de un procesador de un egresado de Berkeley, el cual incorporaba una arquitectura para un repertorio de instrucciones reducido (RISC).

Este diseño, conocido com Berkeley RISC I, era mucho mas simple que el de los procesadores CISC comerciales de aquellos tiempos, además de que requería mucho menos esfuerzo en su desarrollo; sin embargo nunca demostró un rendimiento similar.

El repertorio de instrucciones RISC I difería del de una minicomputadora CISC en varios aspectos:

- Un tamaño fijo de palabra de instrucción de 32 bits con algunos formatos; los procesadores CISC tipicamente tenían una longitud de palabra variable con muchos formatos.
- Un arquitectura de load-store(cargar-almacenar) donde las instrucciones que procesan el dato solo operan con registros y estan separadas de las que se utilizan para acceder a memoria. Los procesadores CISC tipicamente almacenan valores en memoria que serán utilizados como operandos en instrucciones de procesamiento de datos.
- Un gran banco de registros de 32 bits, todos ellos pueden ser usados con cualquier propósito, permitiendo a la arquitectura load-store trabajar eficientemente,. Los registros en un procesador CISC se iban haciendo mas grandes y la mayoría para distintos propósitos (por ejemplo los registros de datos y direcciones en el Motorola MC68000).

Estas diferencias simplificaban mucho el diseño del procesador y permitían al diseñador implementar una arquitectura cuyas características aumentaban el rendimiento del dispositivo:

- ❖ Lógica de decodificación cableada; los procesadores CISC utilizaban ROMs grandes para almacenar las complejas instrucciones.
- ❖ Ejecución en pipeline; los procesadores CISC permitían una pequeña, si es que había, solapamiento entre instrucciones consecutivas.
- ❖ Ejecución de una instrucción por ciclo de clock; los procesadores CISC utilizaban varios ciclos de clock para completar una instrucción.

Patterson y Ditzel decían que RISC ofrecía las siguientes ventajas principales:

- ❖ Pequeño tamaño: Un procesador tan simple requería menos transistores y menos silicio. Por lo cual toda una CPU cabría en un chip, y dejaría mas espacio de la pastilla para otras funcionalidades, como memoria cache, funciones para manejo de memoria, hardware para manejo de datos en punto-flotante, y mucho mas.
- ❖ Menos tiempo de desarrollo: Un procesador tan simple debía tomar menor tiempo de diseño y entonces debía tener un menor costo de diseño.
- ❖ Un mayor rendimiento: Esto era difícil de ver, en un mundo donde los rendimientos mayores se conseguían aumentando la complejidad del sistema, este se convertía en un punto difícil de asimilar. El argumento era algo así: las cosas pequeñas tienen frecuencias mas altas (los insectos mueven sus alas más rápido que los pájaros pequeños, y estos más rápido que los grandes, y así siguiendo), por lo cual un procesador mas simple debía permitir frecuencias de clock mas altas.

Estos argumentos se respaldaron en resultados experimentales y en procesadores prototipo (el Berkeley RISC II que surgió luego del RISC I).

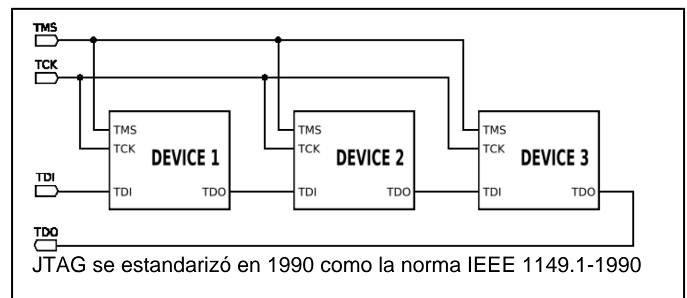
Los fabricantes de procesadores que estaban escépticos en un principio, y principalmente los mas nuevos en el mercado vieron en esta arquitectura una oportunidad de bajar sus costos de desarrollo. Estos diseños RISC comerciales, de los cuales el ARM fue el primero, mostraron que la idea funcionaba, y desde 1980 todos los procesadores de propósitos generales han utilizado conceptos de la arquitectura RISC en una mayor o menor medida.

Coprocesador

Se pueden conectar hasta 16 coprocesadores a un sistema ARM7TDMI.

Debug

El ARM7TDMI incorpora extensiones de hardware para debug (debug embebido), facilitando el desarrollo de las aplicaciones.



JTAG se estandarizó en 1990 como la norma IEEE 1149.1-1990

Es posible inspeccionar el estado interno del núcleo (variables, registros, etc.) utilizando una interfaz JTAG.

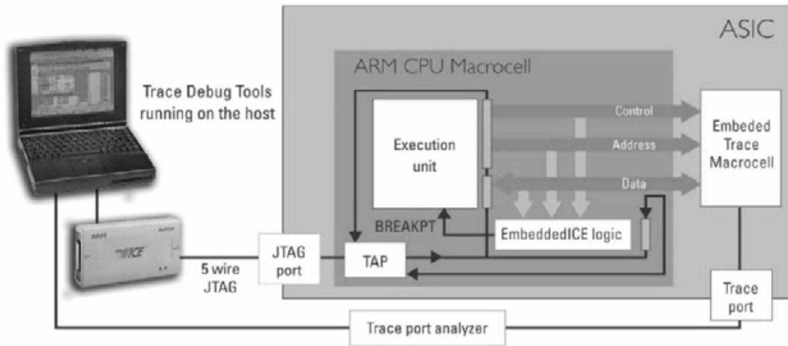
Internamente el núcleo ARM7TDMI incluye una unidad conocidad como Embedded ICE Logic. Esta se configura para evaluar el estado y actividad del núcleo del micro para ciertas instrucciones o ciertos datos específicos.

Se detendrá la ejecución cuando los valores pre-programados coincidan con los valores leídos en la unidad especificada causando un breakpoint (código), o watchpoint (datos).

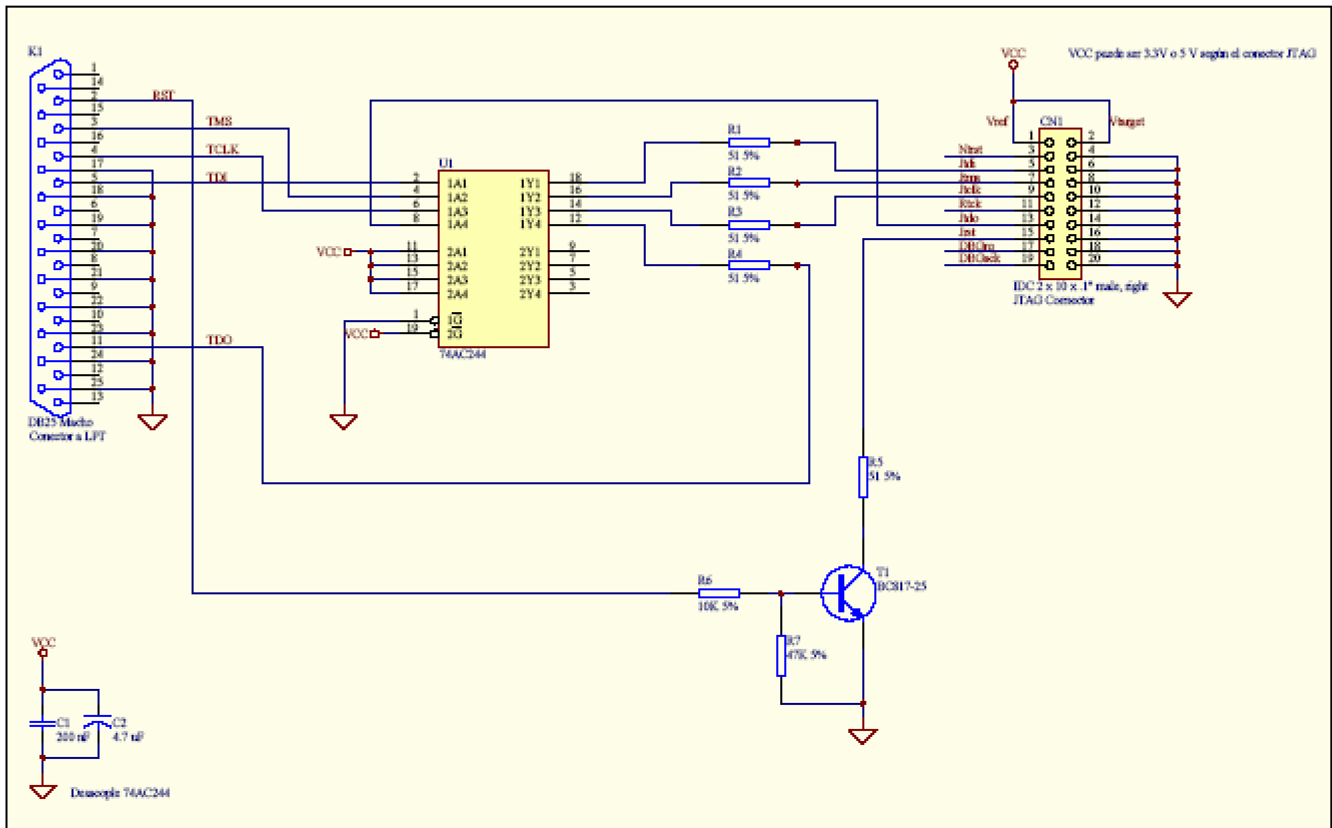
Así es posible especificar que deseamos que se detenga la ejecución del programa cuando el PC valga un determinado valor, generando de esta forma un breakpoint.

Esta configuración se realiza a través de la interfaz JTAG.

La interfaz JTAG no alcanza velocidades muy elevadas, para ello es posible utilizar una interfaz ETM (Embedded Trace Macrocell), así podemos leer la información necesaria sin detener al micro, el cual sigue trabajando a alta velocidad.



La interfaz JTAG circuitalmente es muy sencilla, solo consta de buffers y adaptadores de nivel, por lo cual cualquier interfaz JTAG debería poder analizar el port JTAG de cualquier ARM7, de la marca o línea que sea. Presentamos un circuito propuesto. Adicionalmente es necesario tener un software que permita comunicarse con el micro a través de la interfaz.



Aplicaciones

- Industriales
  - Control Preciso
  - Diagnóstico/Monitoreo
- Automotriz
  - Sensores precisos
  - Sistema de control
    - Control de apertura/cierre de ventanas
    - Sistemas de sonido de alta calidad
    - Sistemas de mejoramiento ambiental
- Instrumentación
  - Sensores inteligentes
  - Sensores de proximidad
- Comunicaciones
  - Sistemas en estaciones base
  - Transceivers

Otras arquitecturas ARM

ARM7TDMI-S

- Versión sintetizable del ARM7TDMI, con los mismos niveles de rendimiento y características en conjunto.
- Optimizado para las tendencias modernas de diseño donde portabilidad y flexibilidad son clave.
- Recorta el tiempo de entrega al mercado, reduciendo el tiempo de desarrollo a la vez que aumenta la flexibilidad en diseño.

ARM7EJ-S

- Versión sintetizable, incorpora las bondades del ARM7TDMI.
- Soporta ejecución acelerada de Java y operaciones DSP.
- Emplea tecnología ARM Jazelle (Máquina JAVA embebida).

ARM720T

- Para sistemas que requieren manejo completo de memoria virtual y espacios de ejecución protegidos.
- Memoria caché de 8K
- MMU: unidad controladora de memoria.
- Para aplicaciones de plataforma abierta como Windows CE, Linux, Palm OS y Symbian OS.
- Buffer de escritura.
- Bus de interface AMBA AHB.
- Coprocesador de interface ETM para expansión del sistema y debugueo en tiempo real.
- Coprocesador para control interno de la memoria caché y la MMU.
- Memoria externa puede soportar procesadores adicionales o canales DMA, con pérdida mínima de rendimiento.

Familia ARM9

- Es una familia constituida por los procesadores ARM920T, ARM922T Y ARM940T.
- Construida en base al procesador ARM9TDMI.
- Set de instrucciones de 16 Bits.
- El procesador es RISC de 32 Bits.
- Buffer de escritura de 8 entradas.
- Pipeline de 5 estados que alcanza 1.1 MIPS/MHz, expandible a 300 MIPS.
- Bus de interface AMBA de 32 Bits.
- MMU (Memory Management Unit) que soporta Windows CE, Symbian OS, Linux, Palm OS.

- MPU (Memory Protection Unit) soportando una amplia gama de sistemas operativos en tiempo real, incluyendo VxWORKS.

ARM920T y ARM922T

- Macroceldas basadas en el ARM9TDMI RISC de 32 Bits convenientes para una gama de aplicaciones basadas en plataforma OS, ofrecen caches para instrucciones y datos, son idénticos pero se diferencian en que uno es de 16k y el otro de 8k.
- MMU permitiendo soporte para otros sistemas operativos importantes.

ARM940T

- Comparten características de los anteriores.
- Se caracteriza por ser de 4k para caches de instrucciones y datos.
- MPU habilitada para soportar sistemas operativos en tiempo real (RTOS).

	Cache size	Mem. Cont.	Thumb	DSP	Jazelle
ARM 920T	16k	MMU	Si	No	No
ARM 922T	8k	MMU	Si	No	No
ARM 940T	4k	MPU	Si	No	No

Familia ARM11

- Arquitectura con un potente repertorio de instrucciones tipo ARMv6
- El repertorio de instrucciones Thumb reduce los requerimientos de memoria en un 35%.
- Tecnología Jazelle para ejecución eficiente de JAVA embebido.
- Extensiones de DPS embebidas.
- Extensiones SIMD (Single Instruction Multiple Data) para aplicaciones multimedia, llevando al doble el rendimiento en el procesamiento de video.
- Tecnología ARM TrustZone otorga niveles de seguridad on-chip (ARM1176JZ-S y ARM1176JZF-S cores)
- Núcleo Thumb-2, mejora el rendimiento, el uso de energía y la densidad de código (ARM1156T2-S y ARM1156T2F-S)
- Bajo consumo:
  - 0.6mW/MHz (0.13µm, 1.2V) incluyendo los controladores de cache
  - Modos de bajo consume para ahorro de energía
  - Gerenciamiento inteligente de la energía (Intelligent Energy Manager (IEM)) en forma dinámica (ARM1176JZ-S and ARM1176JZF-S)
- Procesadores de alto rendimiento:
  - Pipeline de 8 etapas, aumenta la velocidad del clock (9 etapas para el ARM1156T2(F)-S)
  - Pipelines separados para las operaciones de carga-almacenamiento y aritméticas
  - Predicción de salto (Branch Prediction) y Retorno de pila (Return Stack)
- Sistema de memoria de alto rendimiento
  - Posee cache de 4-64k
  - Se puede manejar una memoria adicional mediante canal DMA para aplicaciones multimedia
  - Sistema de memoria de alto rendimiento de 64 bits aumenta la velocidad de acceso a los datos para procesamiento de aplicaciones multimedia y aplicaciones de redes.
  - Sistema de memoria ARMv6 acelera el cambio de contexto del Sistema Operativo
- Interfaz de interrupciones vectorizadas y modo de baja latencia, aumentando la velocidad de respuesta y el rendimiento en tiempo real
- Coprocesador para Punto Flotante (ARM1136JF-S, ARM1176JZF-S y ARM1156T2F-S) para aplicaciones de control de automóviles/industrials y aceleración gráfica 3D

Ejemplos de código

Los ejemplos detallados han sido realizados utilizando una placa prototipo con el microcontrolador ARM7 ADuC7026 de la firma Analog Devices. Se ha utilizado una interfaz JTAG miDasLink RDI.

El entorno de trabajo es el uVision3 de la firma KEIL recientemente incorporada al consorcio de empresas ARM.

Led

Este programa permite prender y apagar un led con ayuda de la interfaz JTAG, de lo contrario no podremos ver el parpadeo que pasa muy rápido.

```
#include <stdio.h>
#include <ADuC7026.H>

void Configurar_GPIO(void)
{
    //Control de cada pin, selecciona la función de cada pin
    GP0CON=0x00000000;
    GP1CON=0x00000000;
    GP2CON=0x00000000;
    GP3CON=0x00000000;
    GP4CON=0x00000000;

    //Parámetros para los puertos 0,1 y 3
    GP0PAR=0x20000000;
    GP1PAR=0x00000000;
    GP3PAR=0x00222222;

    //Datos y Configuración del port, selecciona la dirección de los datos
    GP0DAT=0x00000000;//Configuro el pin P4.2 como salida
    GP1DAT=0x00000000;
    GP2DAT=0x00000000;
    GP3DAT=0x00000000;
    GP4DAT=0x04000000;

    //Setea el contenido del bit de cada puerto, 0 no afecta a la salida, 1 pone un 1 en la salida
    GP0SET=0x00000000;
    GP1SET=0x00000000;
    GP2SET=0x00000000;
    GP3SET=0x00000000;
    GP4SET=0x00000000;

    //Setea el contenido del bit de cada puerto, 0 no afecta a la salida, 1 pone un 0 en la salida
    GP0CLR=0x00000000;
    GP1CLR=0x00000000;
    GP2CLR=0x00000000;
    GP3CLR=0x00000000;
    GP4CLR=0x00000000;
}

void main(void)
{

    Configurar_GPIO();

    while(1){
       GP4SET=0x00040000;

       GP4CLR=0x00040000;

       }
}
```

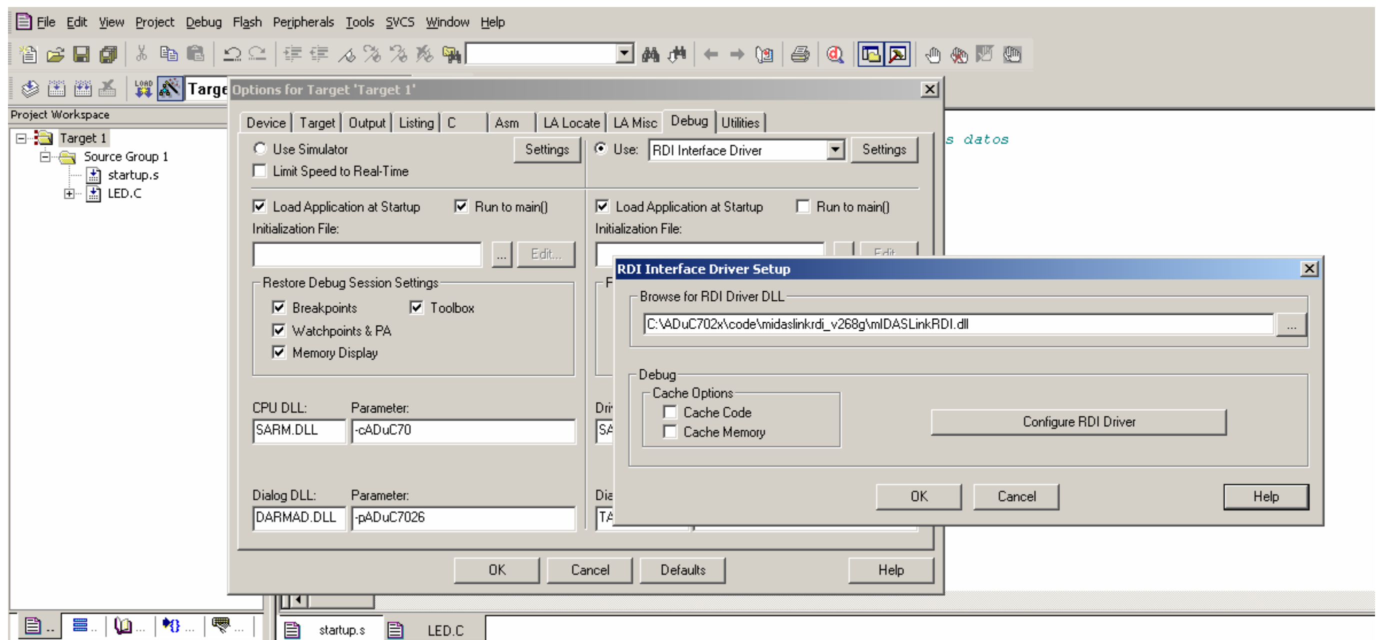
Como vemos, los registros son de 32 bits, incluso para manejar los puertos de entrada/salida.

Para setear un puerto (poner un 1 en la salida) utilizamos la función GPxSET y para limpiarlo (poner un 0 en la salida) utilizamos la función GPxCLR.

```
Project Workspace
Target 1
  Source Group 1
    startup.s
    LED.C

26        GP0SET=0x00000000;
27        GP1SET=0x00000000;
28        GP2SET=0x00000000;
29        GP3SET=0x00000000;
30        GP4SET=0x00000000;
31
32        //Setea el contenido del bit de cada pue.
33        GP0CLR=0x00000000;
34        GP1CLR=0x00000000;
35        GP2CLR=0x00000000;
36        GP3CLR=0x00000000;
37        GP4CLR=0x00000000;
38    }
39
40    void main(void)
41    {
42
43        Configurar_GPIO();
44
45        while(1){
46            GP4SET=0x00040000;
47
48            GP4CLR=0x00040000;
49
50            }
51    }
```

Ponemos dos breakpoints en cada una de estas instrucciones.

Para observar los resultados o utilizamos el simulador o utilizamos las herramientas de debug como el JTAG, para ello debemos configurarlo:



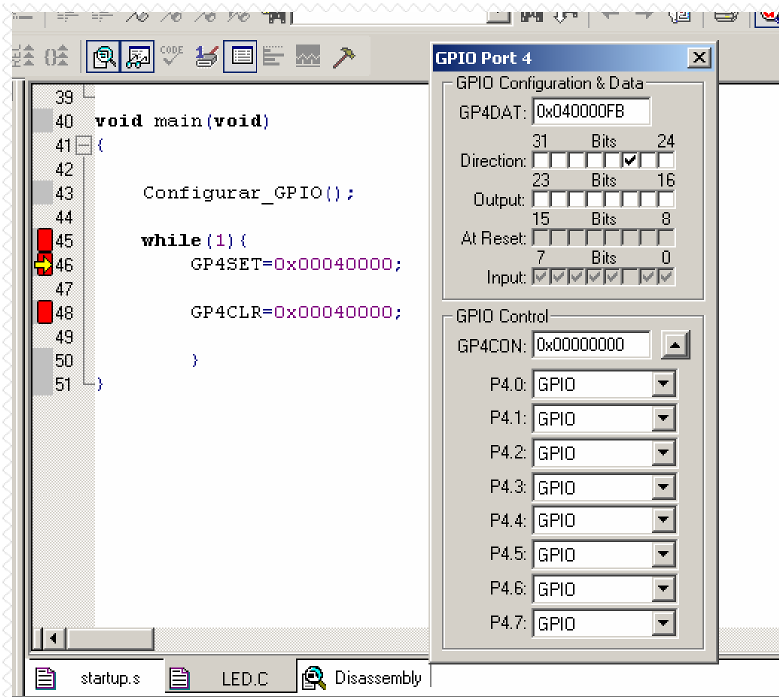
Lo hacemos seteando en **Project->Option for Target 'Target 1' -> Debug -> Settings->**Especificamos la ruta de acceso a la dll correspondiente al driver provisto con el JTAG, en nuestro caso la dirección de la instalación por default es: C:\ADuC702x\code\midaslinkrdi_v268g\mIDASLinkRDI.dll.

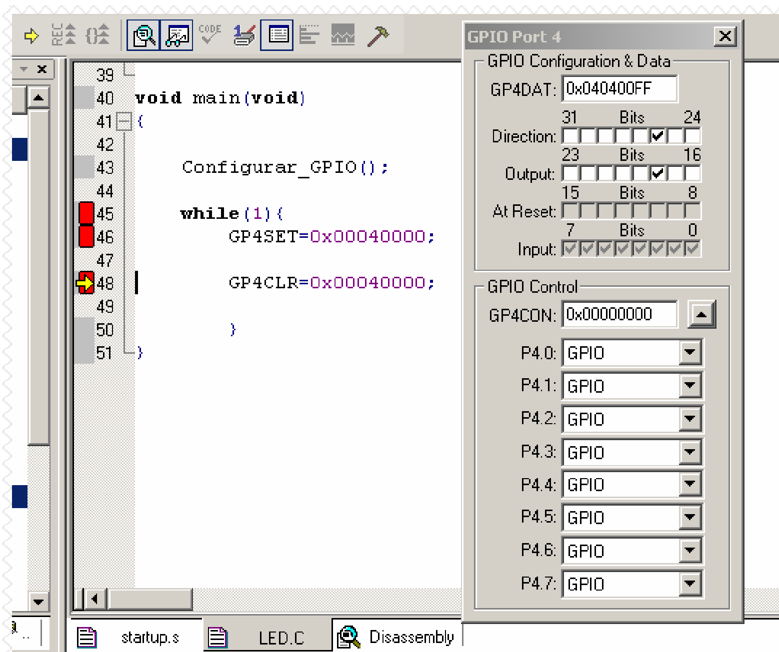
Presionamos OK y ahora tendremos habilitada la opción de Download directamente desde el menú **Flash->Download**.

Una vez cargado el programa en el micro este comenzará a ejecutarse.

Otra opción es ingresar directamente al debug (**Debug->Stara/Stop Debug Session)** una vez compilado el programa, de esta forma el KEIL descargará el firmware directamente en el micro y quedará a la espera de una acción por parte del usuario, ejecutamos nuestro programa y vemos como avanzamos de un breakpoint al otro y como cambia de estado el led en cada uno de ellos alternativamente.

Es posible en este estado evaluar los registros internos del procesador utilizando las mismas herramientas de si estuviéramos en modo Simulación.





Serie_por_Pooling

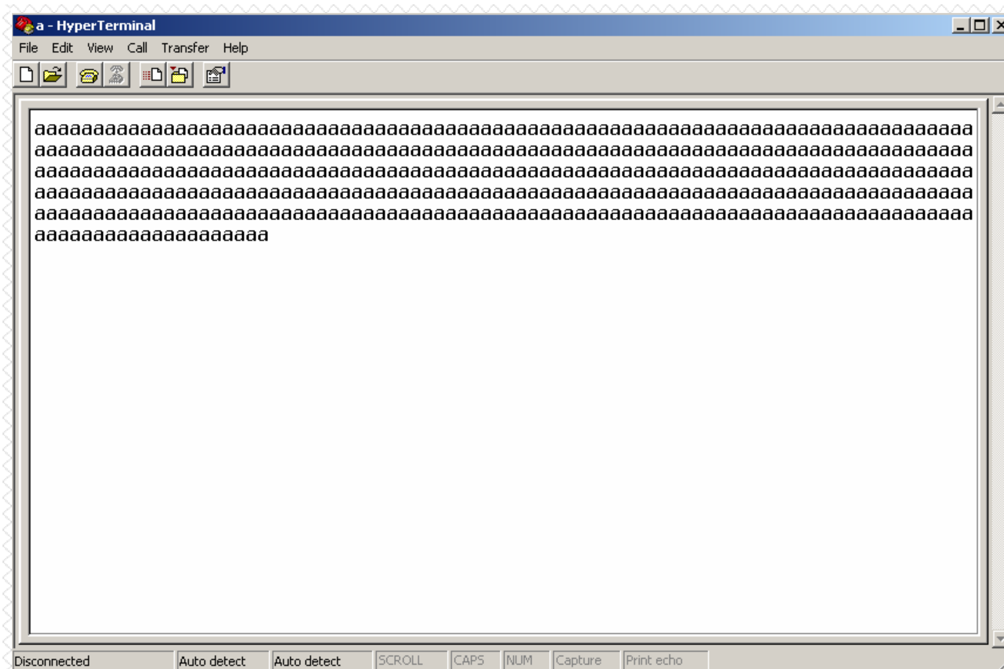
Este programa nuevamente muestra el uso de registros de 32bits y la configuración de la RAM estándar de un ARM7. Como vemos, su configuración es muy similar a la de un 16550.

```
#include <stdio.h>
#include <ADuC7026.H>

void Configurar_GPIO(void)
{
   //Control de cada pin, selecciona la función de cada pin
   GP0CON=0x00000000;//Configuro al P0.7 como SIN
   GP1CON=0x00000011;
   GP2CON=0x00000000;//Configuro al P2.0 como SOUT
   GP3CON=0x00000000;
   GP4CON=0x00000000;

   //Parámetros para los puertos 0,1 y 3
   GP0PAR=0x20000000;
   GP1PAR=0x00000000;
   GP3PAR=0x00222222;

   //Datos y Configuración del port, selecciona la dirección de los datos
   GP0DAT=0x00000000;//Configuro el pin P4.2 como salida
   GP1DAT=0x00000000;
   GP2DAT=0x00000000;
   GP3DAT=0x00000000;
   GP4DAT=0x04000000;

   //Setea el contenido del bit de cada puerto, 0 no afecta a la salida, 1 pone un 1 en la salida
   GP0SET=0x00000000;
   GP1SET=0x00000000;
   GP2SET=0x00000000;
   GP3SET=0x00000000;
   GP4SET=0x00000000;

   //Setea el contenido del bit de cada puerto, 0 no afecta a la salida, 1 pone un 0 en la salida
   GP0CLR=0x00000000;
   GP1CLR=0x00000000;
   GP2CLR=0x00000000;
   GP3CLR=0x00000000;
   GP4CLR=0x00000000;
}

void Configurar_Serie(void)
{
   COMCON0 |=0x80; //DLAB=1
   COMDIV1 = 0x00;
   COMDIV0 = 0x84;//0x084

   COMCON0 =0x03; //DLAB=0 BRK=0 SP=0 EPS=0 PEN=0 STOP=0 WLS=11

   COMIEN0 = 0x00;

}

void main(void)
{
unsigned int i;

   Configurar_GPIO();//Configuro la UART Modo 2(solo Tx y Rx)

   /*-------------------------------------------------------------------------
   Generacion del Baud Rate:
     Baud_Rate=41.78MHz/(2^CD*16*2*DL)
     De esta expresión despejamos DL
     Por ejemplo: Baud Rate:9600bps y CD=0 --> DL=0x88

   COMDIV0 y COMDIV1 son los divisores, alli cargamos el valor obtenido del CD

   COMTX:Registro de Transmisión

   COMRX: Registro de recepción

   COMIEN0: Habilita la interrupción serie
     Esta puede ser por: Modem Status - Rx - Tx - Buffer Full

   COMIEN1: Habilita el modo de trabajo en red (network)

   COMIID0: Identifica la interrupción de la que se trata

   COMIID1: Identifica interrupciones del modo network
```

```
  COMCON0: Registro de control
     el bit 7 es el DLAB, cuando este bit esta en 1 se puede acceder a COMDIV0 y COMDIV1,
     y cuando esta en 0 a los regisrtros COMTX Yy COMRX

  COMCON1: Registro de control de las líneas de control de flujo (modem)

  COMSTA0: LSR

  COMSTA1: MSR

  COMSCR: Scratchpad (se utiliza en el modo network)

  COMDIV2: Se utiliza para generar Baud Rates fraccionarios

  COMADR: Es la dirección para el modo de trabajo en red.
  ---------------------------------------------------------------------------*/

  Configurar_Serie();

  while(1){

     GP4SET=0x00040000;
     for(i=0;i<65536;i++);
     GP4CLR=0x00040000;

     while(!(0x020==(COMSTA0 & 0x020)));
     COMTX = 'a';
        while(!(0x020==(COMSTA0 & 0x020)));
     }
}
```

Descargamos el programa con nuestra unidad JTAG, o podemos hacerlo via puerto serie, ya que este microcontrolador, al igual que muchos ARM7 tienen la ventaja de poseer un ISP, es decir un sistema de programación In-System a través del port serie.

Lo que hemos capturado con el puerto serie de nuestra PC es lo siguiente:



Multi_Interrupciones

El siguiente programa permite ver el uso y configuración de una IRQ y de una FIQ.

Como vemos, todas las interrupciones son atendidas por el mismo handler, el cual debe discernir de qué tipo de interrupción se trato y ejecutar el código correspondiente.

La gestión de las interrupciones es muy similar al que hacemos con un 8259.

```
#include <ADuC7026.h>

//-------------------------------------------------------------------
// Prototipos de Funcion
//-------------------------------------------------------------------
  void IRQ_Handler(void) __irq;     //Prototipo de funcion de la IRQ
  void FIQ_Handler(void) __fiq;     //Prototipo de funcion de la FIRQ

  void delay(int);

  void Inicializacion(void);

  long ADCconvert(void);
  void ADCpoweron(int);

  void My_IRQ_Function(void);

//-------------------------------------------------------------------
// Función: void main(void)
//-------------------------------------------------------------------

void main (void)  {

  Inicializacion();                 // Inicializamos el ADC y el DAC

  //GPIOs
  GP0CON = 0x00100000;              // ADCbusy = 1 (P0.5)
  GP4DAT = 0x04000000;              // P4.2 -> Salida
  GP3DAT = 0xff000000;              // P3 -> Salida

  // Timer 1
  T1LD = 0x20000;                   // Valor del Contador (16-bit)
  T1CON = 0xC4;                     // Decreciente,Enable,Periodic(lo reinicio a mano),Binary,CoreClk/16

  //IRQ
  IRQEN = XIRQ0_BIT+GP_TIMER_BIT;   // IRQ: XIRQ0,Timer1
  FIQEN = ADC_BIT;                                      // FIQ: ADC

  while(1)
  {
    GP3DAT ^= 0x00ff0000;           // P3 = ~P3
    ADCCON = 0x4E3;                 // Single Conversion, Single_Ended,
                                    // Start Conversion, Enable ADCbusy, ADC power on,
                                    // ADC aquisition time= 2 clocks, fADC/2 = 1MSPS
    delay(0x2000);                  // Delay para el parpadeo de P3 (luego del arranque del ADC
                                    // hay que esperar 5us para tener una conversión correcta)
  }

}

void delay (int length)
{
while (length >=0)
    length--;
}

void Inicializacion(void)
{
  // Inicializamos el ADC y el DAC
  ADCpoweron(20000);                // power on ADC
  ADCCP = 0x00;                     // conversion on ADC0
  DAC1CON = 0x13;                   // AGND-AVDD range
  REFCON = 0x01;                    // Conectamos la referencia interna de 2.5v externamente
  return;
}

void ADCpoweron(int time)
{
  ADCCON = 0x20;                     // power-on ADC
  while (time >=0)                   // Luego del arranque del ADC
    time--;                         // hay que esperar 5us para tener una conversión correcta
```

```
}


//-------------------------------------------------------------------
// Función:IRQ
//-------------------------------------------------------------------
void IRQ_Handler() __irq
{
  if ((IRQSTA & GP_TIMER_BIT) != 0)  // Timer1 IRQ?
  {
    T1CLRI = 0;                      // Es un INTA de la irq del timer
    GP4DAT ^= 0x00040000;            // P4.2=~P4.2
  }
  if ((IRQSTA & XIRQ0_BIT) != 0)     // XIRQ0 IRQ?
  {
    GP4DAT ^= 0x00040000;            // P4.2=~P4_2
    while(GP0DAT & 0x00010);         // Esperamos hasta que XIRQ=0
  }
  return ;
}

//-------------------------------------------------------------------
// Función: FIQ
//-------------------------------------------------------------------
void FIQ_Handler() __fiq
{
  if ((FIQSTA & ADC_BIT) != 0)       // ADC FIQ ?
  {
    DAC1DAT = ADCDAT;                // ADC0 = DAC1
  }                                  // Debemos leerlo para limpiar la interrupción del ADC
  return ;
}
```

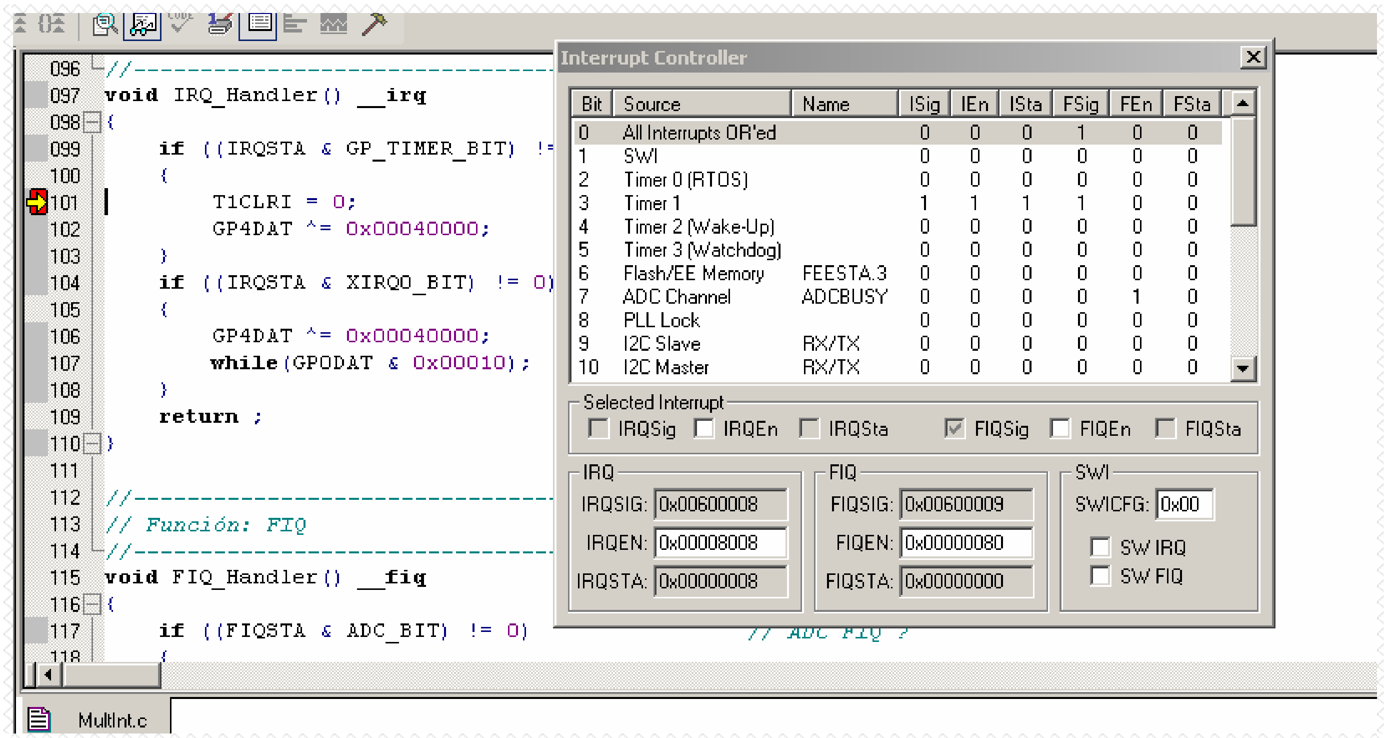
Como vemos el handler de la interrupción correspondiente al timer1 debe limpiar el bit T1CLRI, de esta manera indicamos al micro que la interrución ha sido atendida y que ya puede comenzar un nuevo ciclo de conteo, es similar a la señal INTA del 8259. Este contador no requiere de recarga manual y posee varios modos de trabajo que se detallan en la hoja de datos del microcontrolador.

Otras interrupciones también requieren de alguna indicación por parte del programa en ejecución de que han sido atendidas, es muy importante verificar esto en cada nuevo handler que programemos, de lo contrario la interrupción sucederá sólo una vez y no volverá a repetirse hasta que reiniciemos el sistema.

Compilamos y descargamos el programa con algunos de los métodos mencionados anteriormente y ejecutamos.

Lo que obtenemos es un parpadeo de un LED y la variación de intensidad de otro LED conforme variamos la posición de un potenciómetro conectado al conversor AD seleccionado.

Utilizando la interfaz JTAG y parando la ejecución en la interrupción del Timer1, vemos como quedan habilitados los registros que denotan la necesidad de la atención de la interrupción.



Vemos también que la FIQ la hemos utilizado para atender el EOC del conversor AD, así nos aseguramos de no perder ningún resultado a causa de una excepción IRQ que pudiera ocurrir simultáneamente.

Serie_por_Interrupciones

```
//-------------------------------------------------------------------
// Función: void main(void)
//-------------------------------------------------------------------

void main (void)
{

   Inicializacion();          // Inicializamos el ADC y el DAC
   InicializarSerie();        // Inicializamos la UART
   InicializarColasSerie();

   //GPIOs
   GP0CON = 0x00100000;               // ADCbusy = 1 (P0.5)
   GP1CON = 0x00000011;               // P1.0->SIN, P1.1->SOUT
   GP4DAT = 0x04000000;               // P4.2 -> Salida
   GP3DAT = 0xff000000;               // P3 -> Salida

   // Timer 1
   T1LD = 0x20000;            // Valor del Contador (16-bit)
   T1CON = 0xC4;              // ecreciente,Enable,Periodic(reinicia),Binary,CoreClk/16

   //IRQ
   IRQEN = XIRQ0_BIT+GP_TIMER_BIT+UART_BIT;  // IRQ: XIRQ0,Timer1,UART
   FIQEN = ADC_BIT;                          // FIQ: ADC

   txs("PRUEBA DE VARIAS INTERRUPCIONES");

   while(1)
   {
      GP3DAT ^= 0x00ff0000;      // P3 = ~P3
      ADCCON = 0x4E3;            // Single Conversion, Single_Ended,
                                 // Start Conversion, Enable ADCbusy, ADC power on,
                                 // ADC aquisition time= 2 clocks, fADC/2 = 1MSPS

      delay(0x2000);             // Delay para el parpadeo de P3 (luego del arranque del ADC
                                 // hay que esperar 5us para tener una conversión correcta)
   }

}
```

```
void delay (int length)
{
while (length >=0)
    length--;
}

void Inicializacion(void)
{
  // Inicializamos el ADC y el DAC
  ADCpoweron(20000);                // power on ADC
  ADCCP = 0x00;                     // conversion on ADC0
  DAC1CON = 0x13;                   // AGND-AVDD range
  REFCON = 0x01;                    // Conectamos la referencia interna de 2.5v externamente
  return;
}

void ADCpoweron(int time)
{
  ADCCON = 0x20;                    // power-on ADC
  while (time >=0)                  // Luego del arranque del ADC
    time--;                         // hay que esperar 5us para tener una conversión correcta

}


//--------------------------------------------------------------------
// Función:    IRQ
//--------------------------------------------------------------------
void IRQ_Handler() __irq
{
  if ((IRQSTA & GP_TIMER_BIT) != 0) // Timer1 IRQ?
  {
    T1CLRI = 0;                     // Es un INTA de la irq del timer
    GP4DAT ^= 0x00040000;           // P4.2=~P4.2
  }
  if ((IRQSTA & XIRQ0_BIT) != 0)    // XIRQ0 IRQ?
  {
    GP4DAT ^= 0x00040000;           // P4.2=~P4_2
    while(GP0DAT & 0x00010);        // Esperamos hasta que XIRQ=0
  }
  if ((IRQSTA & UART_BIT) != 0)
  {
    switch(COMIID0)
    {
      case 0://MODEM STATUS INTERRUPT
             //Leer COMSTA1 para limpiar la interrupción
             break;
      case 1://NO INTERRUPT
             break;
      case 2://TRANSMIT BUFFER EMPTY INTERRUPT
             //Escribir en COMTX o leer COMIID0 para limpiar la interrupción
             if (CantidadEnColaTransmision()!=0)
             {
                    COMTX=RetirarDatoColaTransmision();
                    TxEnCurso = SI;
             }
             else
             {
                    TxEnCurso = NO;
             }
             break;
      case 4://RECEIVE BUFFER FULL INTERRUPT
             //Leer COMRX para limpiar la interrupción
             AgregarDatoColaRecepcion(COMRX);
             break;
      case 6://RECEIVE LINE STATUS INTERRUPT
             //Leer COMSTA0 para limpiar la interrupción
             break;
    }
  }

  return ;
}

//--------------------------------------------------------------------
// Función: FIQ
//--------------------------------------------------------------------
```

```
void FIQ_Handler() __fiq
{
  if ((FIQSTA & ADC_BIT) != 0)      // ADC FIQ ?
  {
    DAC1DAT = ADCDAT;               // ADC0 = DAC1
                                    // Debemos leerlo para limpiar la interrupción del ADC
  }
  return ;
}

//-------------------------------------------------------------------
// Función:    InicializarSerie
//-------------------------------------------------------------------
void InicializarSerie(void)
{
// ---------------------------------------------------------------------------
// Generacion del Baud Rate:
//    Baud_Rate=41.78MHz/(2^CD*16*2*DL)
//    De esta expresión despejamos DL
//    Por ejemplo: Baud Rate:9600bps y CD=0 --> DL=0x88
//
// COMDIV0 y COMDIV1 son los divisores, alli cargamos el valor obtenido del CD
//
// COMTX:Registro de Transmisión
//
// COMRX: Registro de recepción
//
// COMIEN0: Habilita la interrupción serie
//    Esta puede ser por: Modem Status - Rx - Tx - Buffer Full
//
// COMIEN1: Habilita el modo de trabajo en red (network)
//
// COMIID0: Identifica la interrupción de la que se trata
//
// COMIID1: Identifica interrupciones del modo network
//
// COMCON0: Registro de control
//    el bit 7 es el DLAB, cuando este bit esta en 1 se puede acceder a COMDIV0 y COMDIV1,
//    y cuando esta en 0 a los regisrtros COMTX Yy COMRX
//
// COMCON1: Registro de control de las líneas de control de flujo (modem)
//
// COMSTA0: LSR
//
// COMSTA1: MSR
//
// COMSCR: Scratchpad (se utiliza en el modo network)
//
// COMDIV2: Se utiliza para generar Baud Rates fraccionarios
//
// COMADR: Es la dirección para el modo de trabajo en red.
// -----------------------------------------------------------------------------
  COMCON0 |=0x80; //DLAB=1
  COMDIV1 = 0x00;
  COMDIV0 = 0x84; //0x084

  COMCON0 = 0x03; //DLAB=0 BRK=0 SP=0 EPS=0 PEN=0 STOP=0 WLS=11

  COMIEN0 = 0x03; //Interrupción por Tx y Rx
}
```

Este programa es muy similar al anterior, con la salvedad de que hemos agregado la interrupción de puerto serie, debemos por lo tanto, no solo identificar que se trata de la interrupción de puerto serie, sino también de que tipo de interrupción del puerto serie, si es por transmisión, recepción, Error, etc.

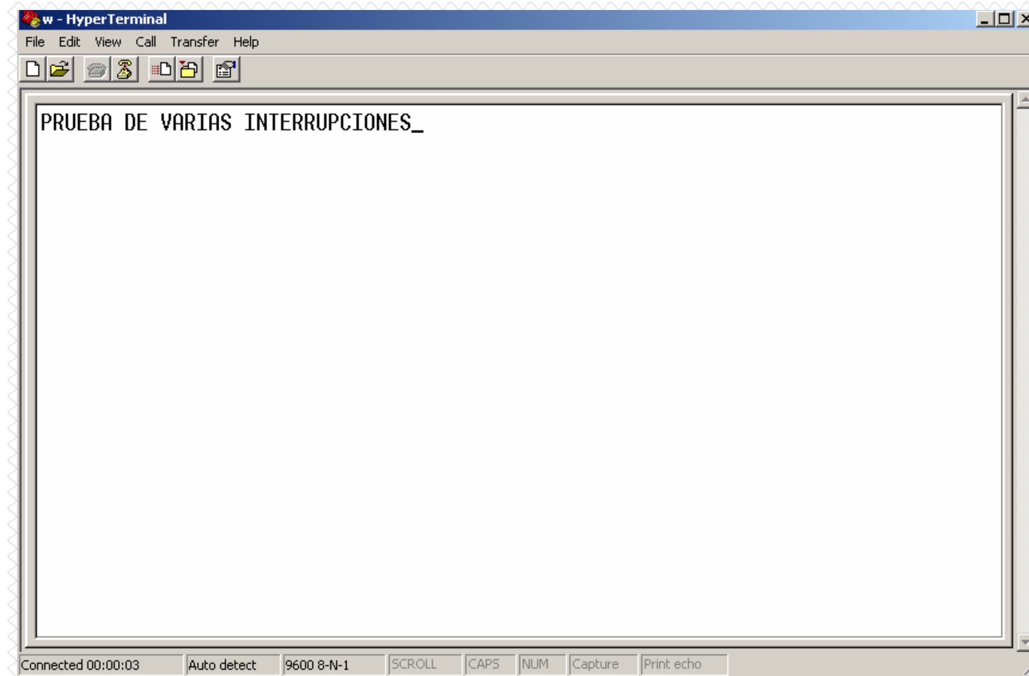
Se han hecho uso para este ejemplo de las funciones de manejo de colas de puerto serie del Ing.Cruz.

Se ha agregado a dichas funciones la siguiente, que simplemente se ocupa de manejar la transmisión de cadenas:

```
//-------------------------------------------------------------------
// Función:    txs
//-------------------------------------------------------------------
void txs(char* dato)
{
```

```
        while(*(dato) != '\0')
        {
            AgregarDatoColaTransmision (*(dato));
               while(CantidadEnColaTransmision());
            dato++;
        }
}
```

Lo que se observa luego de la ejecución como captura en el port serie de nuestra PC es:



Tipos_de_Memoria

Main.c

```
//-------------------------------------------------------------------
// MACROS
//-------------------------------------------------------------------
  #include <stdio.h>
  #include <ADuC7026.H>
  #include "defines.h"

  #include "Variables.h"


//-------------------------------------------------------------------
// Prototipos de Funcion
//-------------------------------------------------------------------
  void IRQ_Handler(void) __irq;      //Prototipo de funcion de la IRQ
  void FIQ_Handler(void) __fiq;      //Prototipo de funcion de la FIRQ

  void delay(int);

  void Inicializacion(void);
  void InicializarSerie(void);

  long ADCconvert(void);
  void ADCpoweron(int);

  void My_IRQ_Function(void);

  void Funcion_en_RAM(void);

//-----------RO-CODE -----------------------------------------------
//Es equivalente a CODE, mapeada como Read-Only Code Memory
```

```
//------------------------------------------------------------------
// Función: void main(void)
//------------------------------------------------------------------
void main (void)
{

  Inicializacion();                       // Inicializamos el ADC y el DAC
  InicializarSerie();                     // Inicializamos la UART
  InicializarColasSerie();

  //GPIOs
  GP0CON = 0x00100000;                    // ADCbusy = 1 (P0.5)
  GP1CON = 0x00000011;                    // P1.0->SIN, P1.1->SOUT
  GP4DAT = 0x04000000;                    // P4.2 -> Salida
  GP3DAT = 0xff000000;                    // P3 -> Salida

  // Timer 1
  T1LD  = 2612;                           //1mS, Valor del Contador (16-bit)
  T1CON = 0xC4;                           // Decreciente,Enable,Periodic(reinicia),Binary,CoreClk/16

  //IRQ
  IRQEN = XIRQ0_BIT+GP_TIMER_BIT+UART_BIT;  // IRQ: XIRQ0,Timer1,UART
  FIQEN = ADC_BIT;                        // FIQ: ADC

  Funcion_en_RAM();

  txs(Texto_Inicio);

  while(1)
  {
    GP3DAT ^= 0x00ff0000;                 // P3 = ~P3
    ADCCON = 0x4E3;                       // Single Conversion, Single_Ended,
                                          // Start Conversion, Enable ADCbusy, ADC power on,
                                          // ADC aquisition time= 2 clocks, fADC/2 = 1MSPS

    delay(0x2000);                        // Delay para el parpadeo de P3 (luego del arranque del ADC
                                          // hay que esperar 5us para tener una conversión correcta)
  }

}

//------------------------------------------------------------------
// Función: delay
//------------------------------------------------------------------
void delay (int length)
{
while (length >=0)
    length--;
}

//------------------------------------------------------------------
// Función: Inicializacion
//------------------------------------------------------------------
void Inicializacion(void)
{
  // Inicializamos el ADC y el DAC
  ADCpoweron(20000);                      // power on ADC
  ADCCP = 0x00;                           // conversion on ADC0
  DAC1CON = 0x13;                         // AGND-AVDD range
  REFCON = 0x01;                          // Conectamos la referencia interna de 2.5v externamente
  return;
}

//------------------------------------------------------------------
// Función: ADCpoweron
//------------------------------------------------------------------
void ADCpoweron(int time)
{
  ADCCON = 0x20;                          // power-on ADC
  while (time >=0)                        // Luego del arranque del ADC
    time--;                               // hay que esperar 5us para tener una conversión correcta

}


//------------------------------------------------------------------
// Función:    IRQ_Handler
```

```
//--------------------------------------------------------------------
void IRQ_Handler() __irq
{
  if ((IRQSTA & GP_TIMER_BIT) != 0)         // Timer1 IRQ?
  {
    T1CLRI = 0;                             // Es un INTA de la irq del timer
    GP4DAT ^= 0x00040000;                   // P4.2=~P4.2
  }
  if ((IRQSTA & XIRQ0_BIT) != 0)            // XIRQ0 IRQ?
  {
    GP4DAT ^= 0x00040000;                   // P4.2=~P4_2
    while(GP0DAT & 0x00010);                // Esperamos hasta que XIRQ=0
  }
  if ((IRQSTA & UART_BIT) != 0)
  {
    switch(COMIID0)
    {
      case 0://MODEM STATUS INTERRUPT
             //Leer COMSTA1 para limpiar la interrupción
             break;
      case 1://NO INTERRUPT
             break;
      case 2://TRANSMIT BUFFER EMPTY INTERRUPT
             //Escribir en COMTX o leer COMIID0 para limpiar la interrupción
             if (CantidadEnColaTransmision()!=0)
             {
                    COMTX=RetirarDatoColaTransmision();
                    TxEnCurso = SI;
             }
             else
             {
                    TxEnCurso = NO;
             }
             break;
      case 4://RECEIVE BUFFER FULL INTERRUPT
             //Leer COMRX para limpiar la interrupción
             AgregarDatoColaRecepcion(COMRX);
             break;
      case 6://RECEIVE LINE STATUS INTERRUPT
             //Leer COMSTA0 para limpiar la interrupción
             break;
    }
  }

  return ;
}

//--------------------------------------------------------------------
// Función: FIQ_Handler
//--------------------------------------------------------------------
void FIQ_Handler() __fiq
{
  if ((FIQSTA & ADC_BIT) != 0)              // ADC FIQ ?
  {
    DAC1DAT = ADCDAT;                       // ADC0 = DAC1
                                            // Debemos leerlo para limpiar la interrupción del ADC
  }
  return ;
}

//--------------------------------------------------------------------
// Función:    InicializarSerie
//--------------------------------------------------------------------
void InicializarSerie(void)
{
// ---------------------------------------------------------------------------
// Generacion del Baud Rate:
//    Baud_Rate=41.78MHz/(2^CD*16*2*DL)
//    De esta expresión despejamos DL
//    Por ejemplo: Baud Rate:9600bps y CD=0 --> DL=0x88
//
// COMDIV0 y COMDIV1 son los divisores, alli cargamos el valor obtenido del CD
//
// COMTX:Registro de Transmisión
//
// COMRX: Registro de recepción
//
// COMIEN0: Habilita la interrupción serie
```

```
//   Esta puede ser por: Modem Status - Rx - Tx - Buffer Full
//
// COMIEN1: Habilita el modo de trabajo en red (network)
//
// COMIID0: Identifica la interrupción de la que se trata
//
// COMIID1: Identifica interrupciones del modo network
//
// COMCON0: Registro de control
//   el bit 7 es el DLAB, cuando este bit esta en 1 se puede acceder a COMDIV0 y COMDIV1,
//   y cuando esta en 0 a los regisrtros COMTX Yy COMRX
//
// COMCON1: Registro de control de las líneas de control de flujo (modem)
//
// COMSTA0: LSR
//
// COMSTA1: MSR
//
// COMSCR: Scratchpad (se utiliza en el modo network)
//
// COMDIV2: Se utiliza para generar Baud Rates fraccionarios
//
// COMADR: Es la dirección para el modo de trabajo en red.
// ---------------------------------------------------------------------------
   COMCON0 |=0x80; //DLAB=1
   COMDIV1 = 0x00;
   COMDIV0 = 0x84; //0x084

   COMCON0 = 0x03; //DLAB=0 BRK=0 SP=0 EPS=0 PEN=0 STOP=0 WLS=11

   COMIEN0 = 0x03; //Interrupción por Tx y Rx
}

//-----------ER-SDATA--------------------------------------------
// Execute RAM, son funciones que se ejecutan desde RAM.
//-------------------------------------------------------------------
// Función:    Funcion_en_RAM
//-------------------------------------------------------------------
void Funcion_en_RAM(void) __ram
{
   delay(10);
}
```

Variables.h

```
//-------------------------------------------------------------------
// Variables
//-------------------------------------------------------------------
   //-----------RO-DATA -------------------------------------------
   // Equivale a CONST, Mapeada como Read-Only Data Memory
   const char Texto_Inicio[] =  "PRUEBA PARA DIG 2";

   const unsigned int Tabla_A[16] =
   {
      0x0123, 0x4567, 0x89AB, 0xCDEF, 0xFEDC, 0xBA98, 0x7654, 0x3210,
      0x0011, 0x2233, 0x4455, 0x6677, 0x8899, 0xAABB, 0xCCDD, 0xEEFF
   };

   const unsigned short Tabla_B[16] =
   {
      0x0123, 0x4567, 0x89AB, 0xCDEF, 0xFEDC, 0xBA98, 0x7654, 0x3210,
      0x0011, 0x2233, 0x4455, 0x6677, 0x8899, 0xAABB, 0xCCDD, 0xEEFF
   };

   const unsigned long  Tabla_C[8] =
   {
      0x01234567, 0x89ABCDEF, 0xFEDCBA98, 0x76543210,
      0x00112233, 0x44556677, 0x8899AABB, 0xCCDDEEFF
   };

   const float Seno_Con_Ruido[1024]=
   {
      #include "Seno_Con_Ruido.h"
```

```
};

//-----------RW-DATA ------------------------------------------
//Mapeada como Read/Write
unsigned int BlockSize = 1024;

//-----------ZI-DATA ------------------------------------------
//Mapeada como zona ZERO-INITIALIZED (inicializada a cero)
struct{
  unsigned int     dato1[16];
  unsigned long    dato2[8];
  }Tabla_D;

//En sistemas embebidos, donde tenemos memoria no-volatil es necesario que los datos
//retenidos en RAM no se pierdan al encender nuevamente el equipo, por lo cual
//es contraproducente la inicialización automática mencionada anteriormente,
//en estos casos se utiliza la directiva:
#pragma NOINIT
char No_Inicializada[100];        //Variable no inicializada
#pragma INIT

char Inicializada[100];           //Variable inicializada

//-----------LOCALIZACIÓN ABSOLUTA DE VARIABLES------------------
//Para acceder a periféricos mapeados como memoria utilizamos __at, además se recomienda usar 'volatile'
//para evitar que el compilador trate de optimizar innecesariamente nuestros accesos a memoria.
// volatile unsigned char RTC __at 0x1000000;

struct Nodo
{
  struct Nodo *Sig;
  char dato;
};
struct Nodo Lista[100] __at 0x00010000;
```

Seno_con_Ruido.h

```
0.07793,0.12456,0.19026,0.26301,0.26347,0.36896,0.45799,0.44297,0.56365,0.54276,0.68335,0.66916,0.68507,0.
80496,0.80138,0.88054,0.85242,0.96089,0.97666,1.01682,1.03827,1.04474,1.04923,1.08232,1.08017,1.08327,1.06
202,1.00774,1.06060,1.04172,1.00753,1.00211,0.92880,0.92529,0.88669,0.88858,0.84685,0.75286,0.74806,0.6605
8,0.64938,0.56268,0.58087,0.50181,0.41634,0.40354,0.28476,0.19582,0.22018,0.13187,0.06155,0.02671,-
0.12408,-0.15729,-0.15218,-0.29881,-0.30852,-0.37888,-0.41002,-0.44994,-0.56923,-0.59119,-0.59422,-
0.72676,-0.69447,-0.72943,-0.81294,-0.85272,-0.85538,-0.86125,-0.85659,-0.92465,-0.93254,-0.98225,-
0.94258,-0.95643,-0.97990,-0.97673,-0.88891,-0.95496,-0.85120,-0.86071,-0.80872,-0.79876,-0.84426,-
0.77647,-0.71610,-0.69832,-0.61728,-0.60285,-0.48973,-0.50043,-0.47980,-0.34450,-0.26984,-0.27783,-
0.21080,-0.11258,-0.08105,…
```

El archivo Seno_con_Ruido.h fue generado en MATLAB con las siguientes funciones:

Dig2.m

```
% Generamos una senoidal con ruido, luego la pasamos a un archivo para
% utilizarla en nuestro proyecto en Keil
% Funcion para generar una Senal senoidal
% Devuelve el contenido del buffer donde se han adquirido las N muestras
% [A]=Senoidal(Amplitud,Frecuencia, Fase,DC,N,Fs)

%Generamos una senoidal de
%Amplitud=1, Frecuencia=100Hz, Fase=0, DC=0, N(número de muestras)=1024,
%Fs(Frecuencia de muestreo)=10000
A=Senoidal(1,100,0,0,1024,10000);
%Generamos una muestra de 1024 valores de ruido aleatorio
B=rand(1,1024)/10;
%Le sumamos a nuestra señal el ruido aleatorio generado
C=A+B;

figure(1);

subplot(3,1,1);
plot(A);

subplot(3,1,2);
plot(B);
```

```
subplot(3,1,3);
plot(C);

%Generamos un archivo con las muestras del vector C
fid=fopen('Seno_Con_Ruido.h','W');
fprintf(fid,'%03.5f,',C);
fclose(fid);
```

Ejecutamos dicho .m en MATLAB 7.0 y generamos el archivo “Seno_con_Ruido.h”, esta es una herramienta que va volviendose indispensable a la hora de pensar en trabajar en procesamiento de señales. A la hora de generar patrones de muestra para probar nuestros algoritmos sería realmente muy complicado hacerlo manualmente.

Se adjuntan al proyecto los .m necesarios para generar también señales Senoidales, Cuadradas y Triangulars. Para ejecutar los scripts basta con pararse en la línea de comandos del MATLAB y escribir el nombre del scripts, este se ejecutará, dejándonos lista la señal que incluiremos en nuestro proyecto.

Incluímos el .h generado en nuestro archivo Variables.h, donde hemos generado distintos tipos de variables con distintos modos de trabajo en memoria.

const: Almacena las variables en memoria de código, en flash, es muy similar a cuando utilizábamos la palabra code en los programas realizados en 8051.

RW-DATA: Las variables aquí declaradas se almacenarán y trabajarán desde la memoria de datos, pudiendose leer y escribir.

ZI-DATA: Las variables aquí declaras se inicializan a 0 al arrancar el programa. En sistemas embebidos donde contamos con una NVRAM, es necesario resguardar el valor que tenían los datos previamente al apagado del equipo, por lo que para evitar que cada vez que prendamos el equipo estas variables en NVRAM se “borren” debemos indicarselo al compilador, esto lo hacemos con las sentencias de precompilación: #NOINIT y #INIT.

RO-CODE: Zona de código de solo lectura, es donde almacenamos el firmware del equipo.

__at: En caso de que tengamos dispositivos mapeados como memoria, debemos indicar la posición absoluta en la que se encuentran para poder direccionarlos a la hora de leer o escribir en ellos, para ello se utiliza la palabra reservada __at y a continuación la posición de memoria donde el dispositivo se encuentra mapeado. Esto es similar al _at_ de los programas realizados en 8051.
Tambien es posible fijar la dirección en memoria de una variable haciendo uso de la misma directiva __at.

Es recomendable que en los casos en que queremos utilizar dispositivos mapeados como memoria, en el momento de la declaración de la variable con la que se accede al dispositivo antepongamos la palabra reservada **volatile** de esta forma el compilador no tratará de optimizar los accesos a memoria cuando utilicemos dicha variable y lo hará solo de la manera en la que nosotros lo indiquemos.

ERAM: RAM ejecutable. Es posible ejecutar funciones desde la SRAM del micro, esto sirve para ejecutar programas cuyo tiempo de ejecución sea crítico, como la RAM es una memoria mas rápida que la FLASH podemos ganar tiempo de ejecución, si además conjugamos esto con el modo de trabajo mas conveniente (ARM o THUMB) de acuerdo a la longitud de palabra de la memoria con la que trabajemos (16 o 32 bits) ganaremos velocidad en la ejecución de esa rutina.

Debe recordarse que el código generado siempre se encuentra en FLASH, la única diferencia es que al iniciar el programa el micro copia este código en RAM para su ejecución, con lo que no ahorramos espacio de código al declarar funciones en RAM, de hecho el compilador reserva ese espacio en RAM también, así que perdemos espacio de código y de datos. Esto solo se recomienda en los casos en que la velocidad de ejecución sea crítica.

Sin embargo otra aplicación podría ser la de la actualización del firmware, para lo que tendremos una función corriendo en RAM que puede comunicarse por puerto serie y descargar el nuevo firmware que será copiado en FLASH, una vez terminado este proceso reiniciamos la ejecución del microcontrolador. Esta operación no sería posible si la rutina encargada de grabar la flash estuviera también en memoria de código, ya que se pisaría en medio del proceso y este quedaría inconcluso junto con un sistema sin firmware.

Otro detalle interesante es que al cambiar a una arquitectura de 32bits los tipos de variables con los que nos manejábamos ya no son los mismos, son muy pocos los cambios pero vale la pena revisarlos:

Tipo	Tamaño	Alineación
Char	1 byte	Cualquier dirección
Short	2 bytes	Cualquier dirección divisible por 2
Float Int Long Pointer	4 bytes	Cualquier dirección divisible por 4
Long long Double	8 bytes	Cualquier dirección divisible por 4

Esto significa que si tratáramos de hacer: **unsigned int var1 __at 0x0001;** obtendríamos un error de alineación como el siguiente:"Insufficient aligned absolute ardes". Para resolverlo basta con colocar esta variable en una dirección de memoria que corresponda con la alineación del tipo que corresponda.

Estos microcontroladores no poseen un área de memoria reservada para trabajar con bits, con lo que al declarar una variable de este tipo lo que hacemos es en realidad es trabajar con una variable del tipo que el compilador crea necesario.

ARM y THUMB

FFT

```
//------------------------------------------------------------------
// MACROS
//------------------------------------------------------------------
   #include <stdio.h>
   #include <math.h>
   #include <ADuC7026.H>
   #include "defines.h"

   #define TAMANO_VECTOR       256
   #define Pi                  3.141592

//------------------------------------------------------------------
// Variables
//------------------------------------------------------------------
   const char Texto_Inicio[] =  "PRUEBA PARA DIG 2\n";

   const float xr[TAMANO_VECTOR]=
   {
      #include "Seno_Con_Ruido.h"
   };
   float xi[TAMANO_VECTOR];//Lo necesito con 0s, pero como esta en la zona ZI ya me lo hace automáticamente
   float XR[TAMANO_VECTOR];
   float XI[TAMANO_VECTOR];
   float Aux1[TAMANO_VECTOR],Aux2[TAMANO_VECTOR];

//------------------------------------------------------------------
// Prototipos de Funcion
//------------------------------------------------------------------
   void IRQ_Handler(void) __irq;      //Prototipo de funcion de la IRQ

   void Inicializacion(void);
   void InicializarSerie(void);

   void My_IRQ_Function(void);

   #pragma ARM
   void fft(float* xr,float* xi,float* XR,float* XI);
   __inline unsigned int IBR(unsigned int m, unsigned int v);
   #pragma THUMB

   void ModuloFase(float* xr,float* xi,float* Modulo,float* Fase);
```

```
//------------------------------------------------------------------
// Función: main
//------------------------------------------------------------------
void main (void)
{
   unsigned char texto[10];
    unsigned short i;

   InicializarSerie();                                // Inicializamos la UART
   InicializarColasSerie();

   //GPIOs
   GP1CON = 0x00000011;                               // P1.0->SIN, P1.1->SOUT

   //IRQ
   IRQEN = UART_BIT;//UART

   txs(Texto_Inicio);

   fft(xr,xi,XR,XI);
   ModuloFase(XR,XI,Aux1,Aux2);

   txs("Modulo\n");
   for(i=0;i<TAMANO_VECTOR;i++)
   {
      sprintf(texto,"%1.2f\n",Aux1[i]);//El módulo
      txs(texto);
   }

   txs("Fase\n");
   for(i=0;i<TAMANO_VECTOR;i++)
   {
      sprintf(texto,"%1.2f\n",Aux1[i]);//La Fase
      txs(texto);
   }

   while(1)
   {
   }

}


//------------------------------------------------------------------
// Función:    IRQ
//------------------------------------------------------------------
void IRQ_Handler() __irq
{

   if ((IRQSTA & UART_BIT) != 0)
   {
      switch(COMIID0)
      {
         case 0://MODEM STATUS INTERRUPT
                //Leer COMSTA1 para limpiar la interrupción
                break;
         case 1://NO INTERRUPT
                break;
         case 2://TRANSMIT BUFFER EMPTY INTERRUPT
                //Escribir en COMTX o leer COMIID0 para limpiar la interrupción
                if (CantidadEnColaTransmision()!=0)
                {
                       COMTX=RetirarDatoColaTransmision();
                       TxEnCurso = SI;
                }
                else
                {
                       TxEnCurso = NO;
                }
                break;
         case 4://RECEIVE BUFFER FULL INTERRUPT
                //Leer COMRX para limpiar la interrupción
                AgregarDatoColaRecepcion(COMRX);
                break;
         case 6://RECEIVE LINE STATUS INTERRUPT
                //Leer COMSTA0 para limpiar la interrupción
                break;
```

```
    }
  }

  return ;
}

//-------------------------------------------------------------------
// Función:    InicializarSerie
//-------------------------------------------------------------------
void InicializarSerie(void)
{
// ---------------------------------------------------------------------------
// Generacion del Baud Rate:
//    Baud_Rate=41.78MHz/(2^CD*16*2*DL)
//    De esta expresión despejamos DL
//    Por ejemplo: Baud Rate:9600bps y CD=0 --> DL=0x88
//
// COMDIV0 y COMDIV1 son los divisores, alli cargamos el valor obtenido del CD
//
// COMTX:Registro de Transmisión
//
// COMRX: Registro de recepción
//
// COMIEN0: Habilita la interrupción serie
//    Esta puede ser por: Modem Status - Rx - Tx - Buffer Full
//
// COMIEN1: Habilita el modo de trabajo en red (network)
//
// COMIID0: Identifica la interrupción de la que se trata
//
// COMIID1: Identifica interrupciones del modo network
//
// COMCON0: Registro de control
//    el bit 7 es el DLAB, cuando este bit esta en 1 se puede acceder a COMDIV0 y COMDIV1,
//    y cuando esta en 0 a los regisrtros COMTX Yy COMRX
//
// COMCON1: Registro de control de las líneas de control de flujo (modem)
//
// COMSTA0: LSR
//
// COMSTA1: MSR
//
// COMSCR: Scratchpad (se utiliza en el modo network)
//
// COMDIV2: Se utiliza para generar Baud Rates fraccionarios
//
// COMADR: Es la dirección para el modo de trabajo en red.
// ----------------------------------------------------------------------------
  COMCON0 |=0x80; //DLAB=1
  COMDIV1 = 0x00;
  COMDIV0 = 0x84; //0x084

  COMCON0 = 0x03; //DLAB=0 BRK=0 SP=0 EPS=0 PEN=0 STOP=0 WLS=11

  COMIEN0 = 0x03; //Interrupción por Tx y Rx
}


//-------------------------------------------------------------------
// Función:    IBR
//-------------------------------------------------------------------
#pragma ARM
__inline unsigned int IBR(unsigned int m, unsigned int v) __ram
{
  //Al usar la directiva __inline del CARM, lo que hacemos es eliminar
  //el llamado a la función, entonces en cada lugar donde se pretende
  //acceder a ella se reemplazará con el código de la rutina.
  //De esta manera aumentamos la velocidad del sistema, reduciendo
  //el tiempo del llamado a la función.
  //Esto es recomendable en funciones donde buscamos la máxima
  //velocidad de procesamiento.
  unsigned int i,p=0,c;

  for(i=0;i<v;i++)
  {
    c=(m>>i)&1;
    p+=c<<(v-i-1);
  }
```

```
  return(p);
}
//-----------------------------------------------------------------
// Función:    fft
//-----------------------------------------------------------------

void fft(float* xr,float* xi,float* XR,float* XI) __ram
{
  long i;
  int I;
  int k;
  int N = TAMANO_VECTOR;
  int L;
  int N2=TAMANO_VECTOR/2;
  int m,p;
  int aux;
  float T1R,T1I;

  float v;

  v=(float)log(N)/((float)log(2));

  for(i=0;i<N;i++)
  {
    XR[i]=xr[i];
    XI[i]=xi[i];
  }

  for(i=0;i<N;i++)
  {
    Aux1[i]=cos(2*Pi*i/N);
    Aux2[i]=sin(2*Pi*i/N);
  }

  for(L=1;L<=v;L++)
  {
    for(k=0;k<N-1;k=k+N2)
    {
      m=k>>((unsigned int)v-L);
      p=IBR(m,(unsigned int)v);
      for(I=0;I<N2;I++,k++)
      {
              T1R=Aux1[p]*XR[k+N2]-Aux2[p]*XI[k+N2];
              T1I=Aux2[p]*XR[k+N2]+Aux1[p]*XI[k+N2];
              XR[k+N2]=XR[k]-T1R;
              XI[k+N2]=XI[k]-T1I;
              XR[k]=XR[k]+T1R;
              XI[k]=XI[k]+T1I;
      }
    }
    N2=N2/2;
  }

  //Antes de volver ordeno los resultados
  for(k=0;k<N-1;k++)
  {
    m=IBR(k,(unsigned int)v);
    if(m>k)
    {
      aux=XR[k];
      XR[k]=XR[m];
      XR[m]=aux;
      aux=XI[k];
      XI[k]=XI[m];
      XI[m]=aux;
    }
  }

}
#pragma THUMB

//-----------------------------------------------------------------
// Función:    ModuloFase
//-----------------------------------------------------------------
void ModuloFase(float* xr,float* xi,float* Modulo,float* Fase)
{
  unsigned int k;
```

```
   unsigned int N = TAMANO_VECTOR;

   for (k = 0 ; k <= N-1 ; k++)
   {
     Modulo[k]=sqrt(pow(xr[k],2)+pow(xi[k],2));

     if(xr[k]!=0)
       Fase[k]=atan(xi[k]/xr[k]);
     else
       Fase[k]=0;
   }
}
```

En este programa hemos definido zonas de código que serán compiladas y ejecutadas utilizando el modo ARM y otras usando el modo THUMB.

La principal ventaja del modo THUMB es que permite ahorrar código, sin embargo a la hora de ejecutar rutinas de gran complejidad de cálculo o que requieran una alta velocidad se recomienda utilizar el modo ARM y las funciones ejecutándose desde RAM.

Esto se debe a que las memorias RAM normalmente son de 32bits de largo de palabra, con lo que es posible leer un código de ejecución en un solo ciclo, mientras que si lo hicieramos con la FLASH en modo ARM, como estas normalmente son de 16bits tendríamos que hacer dos accesos consecutivos a memoria.

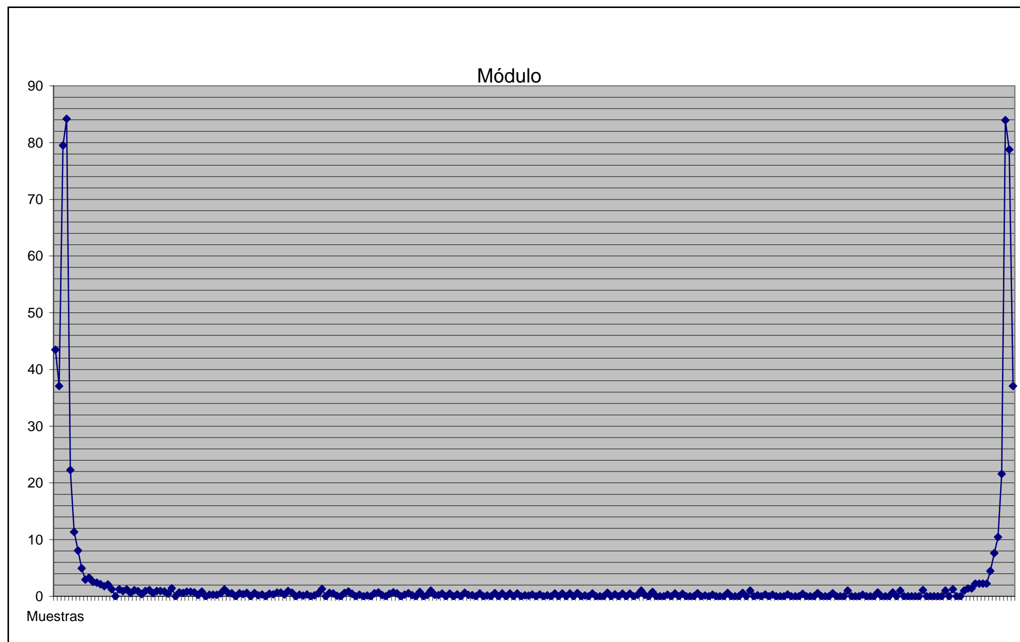
Por otro lado, el modo THUMB, que es de 16bits, se ejecuta mas rápido de FLASH, aunque utiliza mayor cantidad de instrucciones para la misma tarea, con lo que el modo ARM es lo mas recomendable a la hora de trabajar con aplicaciones donde el tiempo de ejecución es crítico.

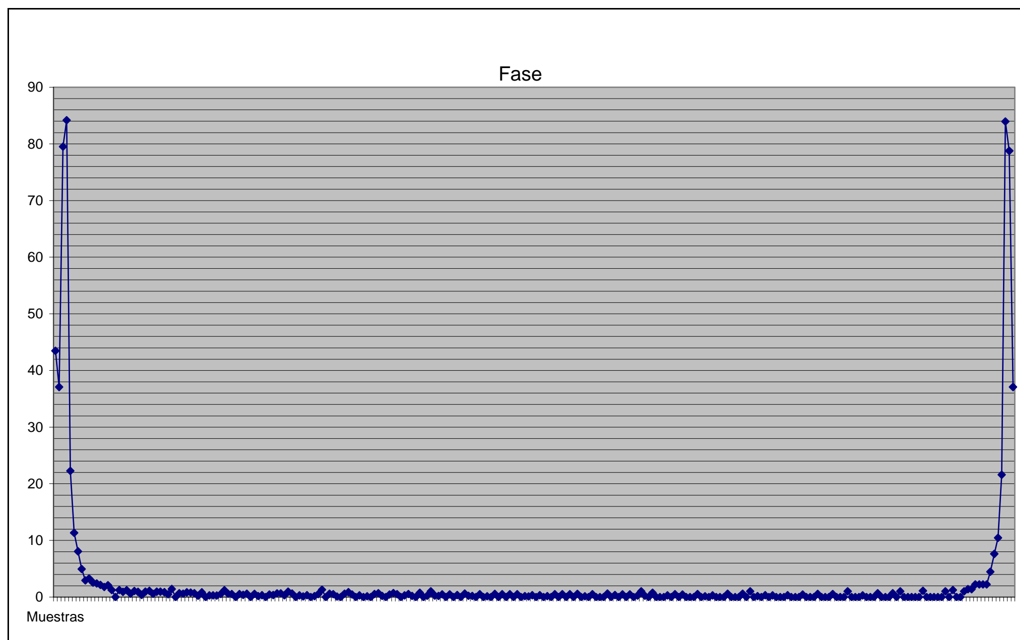
En este ejemplo las rutinas que ejecutan la fft de nuestra señal almacenada en memoria se ejecutan desde RAM. La función IBR (bit revearsal) es necesaria para decodificar la información entregada por el algoritmo de la fft y debe hacerse por cada resultado obtenido, por lo que también es crítica, sin embargo como esta función solamente la utilizamos en este lugar podemos ahorrarnos el tiempo del salto a la subrutina (ahorrándonos el tiempo de resguardo de registros en la pila), con lo que mantenemos la estructura de función, pero en la compilación esta pasa a formar parte de las funciones donde la estemos llamando, generando la repetición de código en cada lugar.

Esto es particularmente útil para generar las funciones de atención de interrupción a llamar desde el handler general de la IRQ. Lo que hacemos es generar nuestros handlers anteponiendo en su declaración la directiva __inline y estaremos generando ese código dentro de la interrupción sin crear saltos a función, ni perder tiempo de ejecución.

La función ModuloFase convierte los vectores que contienen la parte imaginaria y real resultantes de la fft y devuelven el mismo resultado en módulo y fase, esta función ya no es crítica, se hace para adaptar los datos para que sean comprensibles por el usuario.

Por último sacamos la fft resuelta por puerto serie, abrimos una ventana de hyperterminal y guardamos e resultado en un archivo, luego lo importamos a Excel y graficamos las muestras recibidas:





Usuario&Supervisor

Main.c

```
//--------------------------------------------------------------------
// MACROS
//--------------------------------------------------------------------
   #include <stdio.h>
   #include <math.h>
   #include <ADuC7026.H>
   #include "defines.h"

//--------------------------------------------------------------------
// Variables
//--------------------------------------------------------------------
   const char Texto_Inicio[] =  "PRUEBA PARA DIG 2\n";
```

```
//-----------------------------------------------------------------
// Prototipos de Funcion
//-----------------------------------------------------------------
  void IRQ_Handler(void) __irq;      //Prototipo de funcion de la IRQ

  void Inicializacion(void);
  void InicializarSerie(void);

  void My_IRQ_Function(void);

//-----------------------------------------------------------------
// Función: main
//-----------------------------------------------------------------
void main (void)
{

  InicializarSerie();                // Inicializamos la UART
  InicializarColasSerie();

  // Timer 1
  T1LD = 0x20000;                    // Valor del Contador (16-bit)
  T1CON = 0xC4;                      // Decreciente,Enable,Periodic(lo reinicio a mano),Binary,CoreClk/16

  //GPIOs
  GP1CON = 0x00000011;               // P1.0->SIN, P1.1->SOUT
  GP4DAT = 0x04000000;               // P4.2 -> Salida

  //IRQ
  IRQEN = GP_TIMER_BIT+UART_BIT;     // IRQ: Timer1,UART

  txs(Texto_Inicio);//Comienza la función

  TimerInit();
  TimerStart(0,10);

  while(1)
  {
  }
}


//-----------------------------------------------------------------
// Función:   IRQ
//-----------------------------------------------------------------
void IRQ_Handler() __irq
{
  if ((IRQSTA & GP_TIMER_BIT) != 0)   // Timer1 IRQ?
  {
    T1CLRI = 0;                       // Es un INTA de la irq del timer
    Timer_Tick();
    TimerEvent();    //La SWI (Modo supervisor)
  }

  if ((IRQSTA & UART_BIT) != 0)
  {
    switch(COMIID0)
    {
      case 0://MODEM STATUS INTERRUPT
              //Leer COMSTA1 para limpiar la interrupción
              break;
      case 1://NO INTERRUPT
              break;
      case 2://TRANSMIT BUFFER EMPTY INTERRUPT
              //Escribir en COMTX o leer COMIID0 para limpiar la interrupción
              if (CantidadEnColaTransmision()!=0)
              {
                      COMTX=RetirarDatoColaTransmision();
                      TxEnCurso = SI;
              }
              else
              {
                      TxEnCurso = NO;
              }
              break;
      case 4://RECEIVE BUFFER FULL INTERRUPT
              //Leer COMRX para limpiar la interrupción
              AgregarDatoColaRecepcion(COMRX);
              break;
```

```
        case 6://RECEIVE LINE STATUS INTERRUPT
                //Leer COMSTA0 para limpiar la interrupción
                break;
      }
   }

   return ;
}

//--------------------------------------------------------------------
// Función:    Tareas
//--------------------------------------------------------------------
//Las tareas no guardan ni recuperan regitros y no vuelven a ninguna parte.
//Se utilizan como pequeños programas tipo "void main(void)"
//Es el SO el que debe encargarse de resguardar los registros que correspondan.
void Tarea0 (void) __task
{
   while (1);
}

//--------------------------------------------------------------------
// Función:    InicializarSerie
//--------------------------------------------------------------------
void InicializarSerie(void)
{
// ------------------------------------------------------------------------
// Generacion del Baud Rate:
//    Baud_Rate=41.78MHz/(2^CD*16*2*DL)
//    De esta expresión despejamos DL
//    Por ejemplo: Baud Rate:9600bps y CD=0 --> DL=0x88
//
// COMDIV0 y COMDIV1 son los divisores, alli cargamos el valor obtenido del CD
//
// COMTX:Registro de Transmisión
//
// COMRX: Registro de recepción
//
// COMIEN0: Habilita la interrupción serie
//    Esta puede ser por: Modem Status - Rx - Tx - Buffer Full
//
// COMIEN1: Habilita el modo de trabajo en red (network)
//
// COMIID0: Identifica la interrupción de la que se trata
//
// COMIID1: Identifica interrupciones del modo network
//
// COMCON0: Registro de control
//    el bit 7 es el DLAB, cuando este bit esta en 1 se puede acceder a COMDIV0 y COMDIV1,
//    y cuando esta en 0 a los regisrtros COMTX Yy COMRX
//
// COMCON1: Registro de control de las líneas de control de flujo (modem)
//
// COMSTA0: LSR
//
// COMSTA1: MSR
//
// COMSCR: Scratchpad (se utiliza en el modo network)
//
// COMDIV2: Se utiliza para generar Baud Rates fraccionarios
//
// COMADR: Es la dirección para el modo de trabajo en red.
// -------------------------------------------------------------------------
   COMCON0 |=0x80; //DLAB=1
   COMDIV1 = 0x00;
   COMDIV0 = 0x84; //0x084

   COMCON0 = 0x03; //DLAB=0 BRK=0 SP=0 EPS=0 PEN=0 STOP=0 WLS=11

   COMIEN0 = 0x03; //Interrupción por Tx y Rx
}
```

TimerTick.c

```
#include <ADuC7026.H>
#include "defines.h"

unsigned char Evento;
unsigned long TmrRun[8];

//-------------------------------------------------------------------
// Función:    TIMER TICK
//-------------------------------------------------------------------
//El código de esta función se copia en el lugar desde donde la
//estamos llamando.
extern __inline void Timer_Tick(void)
{
   unsigned char j;

   for(j=0;j<8;j++)
   {//Tengo 8 timers que puedo usar
      if(TmrRun[j])
      {//Si hay alguno habilitado
         TmrRun[j]--;
         if(!TmrRun[j])
                Evento=Evento|((unsigned char)0x01<<j);
      }
   }
}

//-------------------------------------------------------------------
// Función:    TimerInit
// Inicializa los Timers
//-------------------------------------------------------------------
void TimerInit(void) __swi (1)
{
   char i;

   for(i=0;i<8;i++)
      TmrRun[i]=0;
}

//-------------------------------------------------------------------
// Función:    kTimerStart
// Configura los Timers(la usan las funciones de modo supervisor)
//-------------------------------------------------------------------
void kTimerStart(unsigned char Evento, unsigned int Ticks)
{
   unsigned char mascara = 1;

   TmrRun[Evento]=Ticks;

   // Me aseguro un 0 en la bandera. Podría estar disparándolo antes
   // de haber atendido la finalización del mismo.
   Evento=Evento & (~(mascara<< Evento));

   return;
}

//-------------------------------------------------------------------
// Función:    TimerStart
// Configura los Timers(la usan las funciones de modo usuario)
//-------------------------------------------------------------------
void TimerStart(unsigned char Evento, unsigned int Ticks) __swi (2)
{
   unsigned char mascara = 1;

   TmrRun[Evento]=Ticks;

   // Me aseguro un 0 en la bandera. Podría estar disparándolo antes
   // de haber atendido la finalización del mismo.
   Evento=Evento & (~(mascara<< Evento));

   return;
}

//-------------------------------------------------------------------
// Función:    kTimerStop
// Detiene alguno de los Timers (la usan las funciones de modo supervisor)
```

```
//------------------------------------------------------------------
void kTimerStop(unsigned char Evento)
{
   TmrRun[Evento]=0;
   Evento=Evento & (~(((unsigned char)0x01)<<Evento));
}

//------------------------------------------------------------------
// Función:    TimerStop
// Detiene alguno de los Timers(la usan las funciones de modo usuario)
//------------------------------------------------------------------
void TimerStop(unsigned char Evento) __swi (3)
{
   TmrRun[Evento]=0;
   Evento=Evento & (~(((unsigned char)0x01)<<Evento));
}

//------------------------------------------------------------------
// Función:    TimerEvent
// Atiende los eventos programados
//------------------------------------------------------------------
//*Las Interrupciones de software (swi) aceptan parámetros y devuelven valores.
//*Los parámetros se le pasan por registros, con lo cual el máximo
//número de parámetros que podemos pasar son 8 bytes.
//*Se ejecutan en Modo Supervisor (Están protegidas de otras interrupciones
//del sistema).
//*Es muy similar a una CallGate en un sistema IA-32
void TimerEvent (void) __swi(0)
{
   unsigned char j;

   for(j=1;j;j<<=1){
      switch(Evento&j){
         case 0x01://Evento 0
                //---------LA FUNCIÓN DEL TIMER-------------
                GP4DAT ^= 0x00040000;         // P4.2=~P4.2
                kTimerStart(0,10);
                //------------------------------------------
                Evento=Evento&0xfe;
                break;
         case 0x02://Evento 1
                //---------LA FUNCIÓN DEL TIMER-------------
                //------------------------------------------
                Evento=Evento&0xfd;
                break;
         case 0x04://Evento 2
                //---------LA FUNCIÓN DEL TIMER-------------
                //------------------------------------------
                Evento=Evento&0xfb;
                break;
         case 0x08://Evento 3
                //---------LA FUNCIÓN DEL TIMER-------------
                //------------------------------------------
                Evento=Evento&0xf7;
                break;
         case 0x10://Evento 4
                //---------LA FUNCIÓN DEL TIMER-------------
                //------------------------------------------
                Evento=Evento&0xef;
                break;
         case 0x20://Evento 5
                //---------LA FUNCIÓN DEL TIMER-------------
                //------------------------------------------
                Evento=Evento&0xdf;
                break;
         case 0x40://Evento 6
                //---------LA FUNCIÓN DEL TIMER-------------
                //------------------------------------------
                Evento=Evento&0xbf;
                break;
         case 0x80://Evento 7
                //---------LA FUNCIÓN DEL TIMER-------------
                //------------------------------------------
                Evento=Evento&0x7f;
                break;
         }
      }
}
```

Estas funciones contenidas en TimerTick.c estan basadas en las funciones para manejo de Timers del Ing. Marcelo Trujillo.

A este proyecto debemos agregarle el archivo SWI_VEC.S provisto con KEIL, que será el encargado de generar el vector de interrupciones de software.

Las interrupciones de software se utilizan para generar una cierta protección en el acceso desde el nivel de usuario y el de supervisor. Su verdadera aplicación se da en los sistemas operativos embebidos, donde las funciones de modo usuario se comportan como tareas independientes del kernel y en el momento en que alguna de ellas necesite realizar alguna tarea que comprometa la estabilidad del sistema (según los criterios del programador del sistema operativo) deberá hacerlo a través de una interrupción de software. Estas se comportan como una puente entre el núcleo del sistema operativo y las tareas del usuario.

Estas interrupciones de software deberán evaluar si el llamado desde donde se requirieron cumple con todas las condiciones para la tarea requerida y que dicha tarea no generará una inestabilidad, de lo contrario no realiza la tarea requerida.

Las interrupciones de software se identifican con la palabra reservada __swi al final de la declaración y las tareas lo hacen con __task.

El llamado nivel de privilegio de modo supervisor se verifica en que las excepciones no pueden interrumpir la ejecución de una interrupción de software. Esta es la aplicación que le hemos dado en este programa.

Arrancamos en el main uno de los timers, la interrupción es la encargada de ir decrementando cada uno de los 8 timers disponibles que esten corriendo y constantemente verificamos si hay alguno que haya vencido.

Cuando vence el primer timer(Evento0), cambia el estado de un led y recarga el timer.

Vemos también aquí el uso de la palabra reservada __inline para ejecutar el handler de la interrupción de timer. De esta manera, desde la interrupción vemos una llamada a una función, pero en la realidad ese código se ejecuta directamente desde ese punto (de una manera muy similar a una macro).