Cika Electrónica

Tutorial: CTU-005
Título: **Introducción a Holtek**
Autor: Sergio R. Caprile, Senior Engineer

Revisiones	Fecha	Comentarios
0	18/10/04	

El presente tutorial se orienta a introducir a Holtek a los ingenieros y developers familiarizados con otras tecnologías, de una forma práctica y concisa. Primero analizaremos la arquitectura de estos micros interesantes y luego haremos algunas comparaciones con micros conocidos. Finalmente, haremos una breve descripción, con ejemplos, del uso de los periféricos más comunes.

Índice de contenido

Arquitectura ... 1
Comparación con un CISC conocido ... 3
- Motorola HC08 series JL3 y QY4 ... 3
- Holtek series “Cost Effective”, “I/O” y “A/D” ... 3
- Paralelo de hardware ... 3
- Paralelo de software ... 4
  - Interrupt desde un timer ... 4
  - Bifurcación (branch) según flag ... 4
  - Salteo (skip) según flag ... 5
  - Modificar flag ... 5
  - Típico loop ... 5
  - Sumar valor a variable 8 bits ... 6
  - Sumar una variable a otra, 8 bits ... 6
  - Memory move, inmediato 8 bits ... 6
  - Memory move, directo 8 bits ... 6
  - Memory move, inmediato 16 bits ... 6
  - Memory move, directo 16 bits ... 6
  - Cálculo en 16 bits: K = next_T - advance – COMP_K ... 7
  - Máquina de estados ... 7
  - Table read, tabla en RAM ... 7
  - Table read, tabla en ROM ... 8
  - Direccionamiento indirecto en RAM ... 8
  - Conversión binario a BCD ... 8

Periféricos ... 8
- I/O ports ... 8
  - Ej.: Uso del port A: bits 2 y 3 entradas, resto salidas ... 9
- 8-bit Timer ... 9
  - Ej.: config del timer para beep a ~1 KHz con clock de 4 MHz ... 9
  - Ej.: config del timer para interrupciones cada ~50 ms con clock de 4 MHz ... 10
  - Ej.: config del timer para medición de ancho de pulso con clock de 4 MHz ... 10
  - Ej.: config del timer para demora de ~10 ms con clock de 4 MHz ... 10
- 16-bit Timer ... 10
- Conversor A/D ... 10
  - Ej.: conversión de señal en AN0 (PB0) con cristal de 4 MHz, poleado ... 11
  - Ej.: conversión de señal en AN0 con cristal de 4 MHz, por interrupciones ... 11

Arquitectura

Presentamos a continuación la arquitectura de los micros de las series “Cost Effective”, “I/O” y “A/D”, de Holtek. La arquitectura de Holtek es muy similar a la de PIC, sin embargo, en aquellos lugares en los que los usuarios de otros micros se sienten incómodos con PIC, Holtek resulta agradable. La siguiente es una descripción enumerando aquellos factores arquitectónicos standard que facilitan la comprensión:

- Se basa en la arquitectura Harvard, dado que dispone de una memoria de instrucciones, y una de datos, con buses separados y sin comunicación entre sí. Consecuencia de esta arquitectura es que en un llamado a subrutina o interrupción, el procesador no puede guardar la dirección de retorno en la memoria de datos, para lo cual requiere de un stack independiente, lo cual limita la cantidad de llamadas anidadas a la cantidad de niveles presentes en el hardware stack (2, 4, etc), como muchos DSP. Por esta misma razón, estos procesadores no poseen intrucciones de manejo de stack (PUSH, POP, etc).
- Emplea pipelining, todas las instrucciones demoran un ciclo de máquina (cuatro ciclos de clock) en ejecutarse, excepto aquellas que obligan a purgar el pipeline, como instrucciones de salto.
- Utiliza técnicas de VLIW al incorporar los datos en la instrucción, es decir, la memoria de instrucciones es de un ancho mayor, de modo de poder incluír constantes (valores inmediatos, direcciones de datos) en las instrucciones. Por ejemplo, muchos micros de esta serie tienen un ancho de palabra de instrucciones de 14-bits. En este espacio se distribuyen las más de 60 instrucciones disponibles, algunas de las cuales incluyen los 8 bits del dato correspondiente (direccionamiento inmediato) o su dirección en memoria de datos (direccionamiento directo). Esto limita la cantidad contigua de memoria de datos disponible a 256 bytes (2^8) .
- La ALU puede realizar operaciones lógicas y aritméticas sobre cualquier variable en memoria de datos y el acumulador (registro A). El resultado de la operación se especifica en el acumulador o la memoria de datos, según la operación; esto atenúa el inconveniente de disponer de sólo un registro en la CPU (respecto de otros RISC y algunos CISC), dado que puede hacerse que una determinada operación no altere el acumulador, guardando el resultado en la memoria de datos.
- Sólo es posible escribir en memoria de datos desde la ALU (el resultado de una operación) o el acumulador, no es posible cargar un dato inmediato en memoria sino que es necesario cargarlo primero en el registro A y luego cargar éste en memoria.
- Dispone de registros para funciones especiales (SFR, Special Function Registers) mapeados en la memoria de datos, el registro de estado (STATUS) de la CPU, el acumulador (ACC) y la parte baja del Program Counter (PCL) se ven como un SFR más. El estado de la última operación se obtiene accediendo a la posición de STATUS en memoria de datos, como una variable más.
- La memoria de datos se direcciona solamente de forma directa, dado que carece de punteros; sin embargo, mediante el uso de un par de SFRs, permite el acceso indirecto a la memoria de datos de una forma no demasiado complicada: un SFR contiene la dirección a acceder y otro "representa" la operación en sí, es decir, el valor del registro MP (Memory Pointer) es la dirección del dato, y al operar sobre el registro IAR (Indirect Access Register) se opera sobre el dato en la dirección que indica FSR.
- La memoria de instrucciones es de 14-bits, 15-bits, o 16-bits de ancho, y hacen falta algunos de ellos para indicar que la instrucción es de control de flujo de programa (call, jump, etc) , empleándose el resto para especificar la dirección. El ancho de palabra de programa es coincidente con la capacidad de memoria de programa, por lo que la totalidad de la misma es direccionable linealmente.
- Para permitir mayor versatilidad en el control del flujo del programa, el program counter está accesible en la memoria de datos, es un SFR más. Dado que la memoria de datos es de 8-bits, los bits (14, 15 ó 16) del PC están partidos en PCL (8-bits) y PCH (6,7 ú 8 bits). El PCH no es accesible directamente, por lo que las instrucciones que modifiquen el flujo mediante cálculo y operación sobre PCL (por oposición al uso de CALL o JMP) están limitadas a un segmento de 256 words de programa.
- Carece de direccionamiento relativo, al cual reemplaza por un "saltear si" (skip), es decir, la CPU "evita" la instrucción siguiente ante determinada condición. Por esta razón, para ejecutar operaciones comunes como saltar si la última suma dió desborde (carry) o el resultado fue cero, debe hacerse un chequeo explícito del flag en el SFR correspondiente (STATUS) mediante una instrucción skip.
- Todos los periféricos se ven desde la CPU como SFRs. Es posible modificar muchas características operativas de cada uno de ellos simplemente escribiendo en posiciones fijas de memoria de datos. De igual modo, algunas características de funcionamiento de la CPU (habilitación de interrupciones) se modifican mediante SFRs.
- La arquitectura permite interrupciones vectorizadas, aunque en realidad se trata de offsets fijos. Una interrupción ocasiona un salto a una posición fija en memoria de instrucciones, dependiente de la causa de la interrupción. El contenido del PC se guarda en el hardware stack para permitir su recuperación al ejecutar una instrucción de retorno, que restablece el estado de las interrupciones. Dado que no existen instrucciones de manejo de stack, tampoco se salva el contexto de la CPU, el programa debe salvar el registro A y el STATUS en alguna posición de memoria. Como sólo las operaciones aritméticas y lógicas afectan el STATUS, es posible prescindir de salvar este registro si el interrupt handler realiza funciones clásicas.

Comparación con un CISC conocido

Haremos a continuación una comparación con un CISC bien conocido, como por ejemplo el 68HC908 de Motorola, comunmente conocido como HC08.

Como buen CISC, el HC08 puede optimizar el uso de memoria si el programador domina la totalidad de las instrucciones y sus diversos modos de direccionamiento para cada una, generalmente diferentes entre sí dado el poco grado de ortogonalidad de la CPU y su baja cantidad de registros.

Como es un RISC simple, Holtek requiere más instrucciones para realizar lo mismo, pero estas instrucciones adicionales no significan más ciclos de ejecución, ni siquiera más palabras de código, sino todo lo contrario: Holtek resuelve lo mismo en menos ciclos de clock y con menos palabras de código. Las instrucciones son simples; todo es cuestión de pensar en RISC en vez de CISC, reduciendo el problema a un conjunto de operaciones elementales en vez de buscar cuál instrucción optimiza el acceso necesario para resolver el problema.

Motorola HC08 series JL3 y QY4

Presenta direccionamiento lineal hasta 4K flash y 128 bytes RAM. Modelos más avanzados de otras series permiten direccionamiento lineal hasta 60K flash y 2K RAM.

Es un CISC de arquitectura Von Neumann, con stack en RAM, lo que posibilita ilimitados llamados a subrutinas e interrupciones anidadas. Las interrupciones son vectorizadas (existe un vector donde se guarda la dirección en la que se encuentra el handler para periférico o condición de interrupción). La CPU posee tres registros: A, H y X, aunque en realidad H sólo puede accederse a través del stack o como parte alta de HX, un registro índice de 16-bits. El registro X puede utilizarse como índice de 8-bits u offset de 8-bits respecto de un índice fijo. Todas las operaciones aritméticas son respecto del acumulador (A), y el destino de toda operación aritmética es el acumulador. La arquitectura permite mover datos entre variables y cargar datos inmediatos en memoria con una simple instrucción (MOV). El clock acepta un cristal de hasta 32 MHz, internamente dividido por cuatro, lo cual da una ejecución a 8MIPS pico; pero dada la cantidad de ciclos de clock por instrucción empleados, se puede estimar una operación máxima sostenida de menos de 3 MIPS.

Como periféricos, encontramos un timer de 16-bits con dos registros de captura y comparación (CCRs), con capacidad de generación de PWM, y un conversor AD de 8-bits de ocho canales.

Holtek series “Cost Effective”, “I/O” y “A/D”

Ya hemos analizado la arquitectura en el párrafo anterior. Resumiendo, presenta direccionamiento lineal hasta 8K OTP y 224 bytes RAM; el modelo que incorpora una cantidad mayor de RAM emplea bank switching.

Se trata de un RISC con arquitectura Harvard modificada, stack en hardware de 2 a 16 niveles, según el modelo. Su operación es bastante similar a PICmicro de Microchip, pero sin el uso de bank switching en flash ni I/O y practicamente nulo en RAM. La CPU acepta un clock de hasta 8 MHz, internamente dividido por cuatro, lo que permite una ejecución a 2 MIPS pico, que no disminuyen demasiado en operación sostenida, dado que la gran mayoría de las operaciones se ejecutan en un ciclo de clock.

Como periféricos, encontramos un timer de 8-bits en todos los modelos; y ninguno, uno o dos timers de 16-bits en las series “I/O” y “A/D”. Los timers tienen capacidad de controlar un buzzer mediante dos salidas complementarias con 50% de ciclo de trabajo. Los modelos de la serie “A/D” incorporan un conversor AD de 9 ó 10-bits de 4 ú 8 canales, y los modelos más avanzados agregan interfaz I^2C y generador de PWM. El watchdog timer posee un oscilador independiente.

Paralelo de hardware

La siguiente es una tabla comparativa de características similares de hardware

Motorola	***Holtek sin AD***	***Holtek con AD***
JK1: 1.5K flash, 128 RAM, timer 16-bits, AD, 20 pines, 14 I/O + INT, xtal, versión RC **QY2**: ídem, 16 pines, 14 I/O osc **QT1**: ídem, 8 pines, 6 I/O, no AD	**48R06/5**: 1/0.5K flash, 64/32 RAM, timer(8), 16/18 pines, 11/13 I/O, xtal o RC	**46R47**: 2K flash, 64 RAM, timer(8), 9-bit AD, 18 pines, 13 I/O, xtal o RC
JK3: idéntico al *JK1*, 4K flash **QY4**: idéntico al *QY2*, 4K flash	**48R10/30**: 2/4K flash, 64/96 RAM, timer(8),24/28 pines, 21/25 I/O, osc	**46R47**: 2K flash, 64 RAM, timer(8), 9-bit AD, 18 pines, 13 I/O, xtal o RC

Motorola	***Holtek sin AD***	***Holtek con AD***
JL3: idéntico al *JK3*, 28 pines, 22 I/O + INT	**48R50**: 4K flash, 160 RAM, timers (8,16), 28/48 pines, 15/35 I/O, osc	**46R22/3**: 2/4K flash, 64/192 RAM, timer(8/16), 9/10-bit AD, 24/28 pines, 19/23 I/O, xtal o RC **46R24**: 8K flash, 384 RAM, timers (16,16), 10-bit AD, 28/48 pines, 20/40 I/O, xtal o RC

Paralelo de software

A continuación haremos un paralelo de software. Dijimos que como buen CISC, el HC08 puede optimizar el uso de memoria si el programador domina la totalidad de las instrucciones y sus diversos modos de direccionamiento; y que Holtek, como es un RISC simple, requiere más instrucciones elementales para realizar lo mismo, pero resuelve la misma tarea en menos ciclos de clock y con menos palabras de código. Esto es así porque en un CISC como el HC08, una simple instrucción puede llevar varias operaciones complejas implícitas, que requieren de varios bytes de programa y varios ciclos de clock para ejecutarse; mientras que en un RISC como Holtek, las instrucciones son simples y el operando siempre está en la instrucción y se ejecuta casi siempre en un ciclo de clock. Como dijimos, todo es cuestión de pensar en RISC, reduciendo el problema a un conjunto de operaciones elementales.

Las comparaciones de uso de memoria de programa serán uno a uno. Si bien Motorola emplea bytes y Holtek words, comparamos ambos procesadores por su capacidad en unidades elementales (1K byte vs. 1K word). Lo hacemos de esta forma dado que la capacidad de memoria de programa de ambos está especificada de esta forma, y nos permite realizar una comparación directa en precio y prestaciones. Como veremos, generalmente Holtek emplea menos unidades elementales para resolver la misma operación.

En cuanto a la velocidad, a igual valor de cristal corresponde igual frecuencia de clock e igual cantidad de MIPS pico, por lo que la comparación en ciclos es equivalente. Sin embargo, para ser justos, Motorola admite un clock cuatro veces mayor (32 MHz vs 8 Mhz), lo cual puede apalear la mayor cantidad de ciclos necesarios para resolver la misma operación, en la mayoría de los casos.

Los listados a continuación presentan primero la resolución de una tarea típica en programas simples de control (a lo que estos micros están destinados) para el HC08, y luego el equivalente en Holtek. De este modo, el usuario experimentado en un micro conocido como el HC08 puede tener una idea rápida de las capacidades de Holtek. Respecto al assembler de Holtek, el mismo determina mediante las declaraciones qué nombres corresponden a variables y cuáles se trata de constantes, generando automáticamente direccionamiento directo o inmediato según corresponda.

Interrupt desde un timer

Evaluamos el tiempo necesario en atender una interrupción de un timer, salvar contexto, y retomar ejecución normal del programa.

```
[03]    lda TSC
[04]    bclr 7,TSC              ; Ack TOF int
[02]    pshh                    ; salva H
        ...
[02]    pulh                    ; recupera H
[07]    rti
```

18 ciclos + overhead del procesador

```
        mov somewhere,A                 ; salva A
        mov A,STATUS
        mov somewhere2,A        ; salva STATUS
        ...
        mov A,somewhere2
        mov STATUS,A
        mov A,somewhere
        reti
```

8 ciclos + overhead = 10 ciclos

Bifurcación (branch) según flag

Este es el típico caso de modificación del flujo de programa debido a circunstancias especificadas en un flag. Dada la carencia de direccionamiento relativo, Holtek lo resuelve mediante una instrucción de salteo (skip) y un salto. Generalmente, la bifurcación debe pensarse al revés

```
[05]            brclr bit,var,l1
```

```
                ...
        l1
```
5 ciclos, 3 bytes

```
        snz var.bit             ; saltea instrucción si el bit 'bit' de la variable 'var' es 1
        jmp l1                  ; caso contrario salta a 'l1'
        ...
l1:
```
2/3 ciclos, 2 words

Salteo (skip) según flag

Esta situación se produce generalmente cuando debemos alterar el estado de un pin según nos indica un flag, pero no podemos darnos el lujo de producir glitches (modififcarlo primero asumiendo un valor y después corregirlo si el flag indica otra cosa). Consideramos sólamente el tiempo empleado en la decisión.

```
[05]            brclr bit,var,l1
                ; acción si 1
[03]            bra l2
        l1      ; acción si 0
        l2      ; continúa
```
5/8 ciclos, 5 bytes

```
        sz var.bit
        ; acción si 0
        snz var.bit
        ; acción si 1
```
3 ciclos, 2 words

En un caso más simple, cuando sí nos podemos permitir el lujo de un glitch o simplemente modificamos un flag:

```
                ; acción si 0
[05]            brclr bit,var,l1
                ; acción si 1
        l1      ; continúa
```
5 ciclos, 5 bytes

```
        ; acción si 1
        sz var.bit
        ; acción si 0
```
2 ciclos, 1 word

Modificar flag

Alteramos el estado (seteamos o reseteamos) de un bit en una variable

```
[04]    bset bit,var
```
4 ciclos, 2 bytes

```
        set var.bit
```
1 ciclo, 1 word

Típico loop

Consideramos solamente el overhead que introduce lo necesario para producir el loop, es decir, decremento del contador y salto al inicio del loop. Para el caso de HC08, evaluaremos un alternativa con el contador en memoria y otra con el contador en un registro

```
        l1
                ...
[05]            dbnz var,l1
```
contador en memoria, 5 ciclos, 3 bytes
```
[03]            dbnza l1
```
contador en registro,3 ciclos, 2 bytes

```
l1:
        ...
        sdz var                 ; decrementa 'var' y si ésta es 0, saltea la siguiente instrucción
        jmp l1                  ; vuelve al loop
```
3 ciclos, 2 words

Sumar valor a variable 8 bits

Sumamos un determinado valor constante a una variable cualquiera; el resultado debe quedar en la variable en cuestión.

```
[03]    lda var
[02]    add #value              ; resultado de la suma en A
[03]    sta var
```
8 ciclos, 6 bytes

```
        mov A,value
        addm A,var              ; resultado de la suma en var
```
2 ciclos, 2 words

Sumar una variable a otra, 8 bits

Sumamos dos variables entre sí; el resultado debe quedar en la última variable direccionada.

```
[03]    lda var1
[03]    add var2                ; resultado de la suma en A
[03]    sta var2
```
9 ciclos, 6 bytes

```
        mov A,var1
        addm A,var2             ; resultado de la suma en var2
```
2 ciclos, 2 words

Memory move, inmediato 8 bits

Colocamos un valor constante en una posición de memoria.

```
[04]    mov #value,var
```
4 ciclos, 3 bytes

```
        mov A,value
        mov var,A
```
2 ciclos, 2 words

Memory move, directo 8 bits

Colocamos en una variable el valor contenido en otra.

```
[05]    mov var1,var2
```
5 ciclos, 3 bytes

```
        mov A,var1
        mov var2,A
```
2 ciclos, 2 words

Memory move, inmediato 16 bits

Colocamos un valor constante en una posición de memoria, esta vez en 16-bits.

```
[03]    ldhx #number
[04]    sthx var16
```
7 ciclos, 5 bytes

```
        mov A,LOW number
        mov varl,A
        mov A,HIGH number
        mov varh,A
```
4 ciclos, 4 words

Memory move, directo 16 bits

Colocamos en una variable el valor contenido en otra, esta vez en 16-bits.

```
[04]    ldhx var16_1
[04]    sthx var16_2
```
8 ciclos, 4 bytes

```
        mov A,var1l
```

```
       mov var2l,A
       mov A,var1h
       mov var2h,A
```
4 ciclos, 4 words

Cálculo en 16 bits: K = next_T - advance – COMP_K

Realizamos un pequeño cálculo no muy complejo, en el cual obtenemos el valor de una variable de proceso que depende de otras dos variables y una constante de calibración. Las variables son todas de 16-bits, y la constante de calibración es de 8 bits, para mostrar un cálculo combinado. Recordemos que en Motorola los números en 16-bits se almacenan byte bajo – byte alto.

```
[03]    lda next_T+1
[03]    sub advance+1                   ; K = next_T – advance (LSB)
[01]    tax                             ; a X
[03]    lda next_T                      ;
[03]    sbc advance                     ; MSB
[02]    psha                            ; a H: HX = next_T – advance
[02]    pulh
[02]    aix #COMP_K                     ; ...  - COMP_K (8 bits)
[04]    sthx K
```
23 ciclos, 15 bytes

```
       mov A,next_Tl
       sub A,advancel               ; next_T – advance (LSB)
       mov Kl,A
       mov A,next_Th
       sbc A,advanceh               ; MSB
       mov Kh,A
       mov A,Kl
       sub A,COMP_K                 ; - COMP_K (8 bits)
       mov Kl,A
       sz C                         ; borrow ?
       jmp done                     ; no
       dec Kh                       ; sí, Kh = Kh – 1
done:
```
12 ciclos, 12 words

Máquina de estados

Analizamos a continuación un dispatcher del tipo comunmente utilizado en máquinas de estados, en el cual el flujo de programa es redirigido según el valor de una variable, es decir, el estado de la máquina.

```
[01]           clrh
[03]           ldx STATE             ; estados posibles: 0,3,6,9, etc
[06]           jsr STTAB,X

       STTAB
[03]           jmp State1
[03]           jmp State2
```
10 + 3 ciclos, 3 bytes por estado

```
       mov A,STATE           ; estados posibles: 0,1,2,3,4,etc
       addm A,PCL            ; resultado de la suma en PCL
       jmp State1
       jmp State2
```
5 ciclos, 1 word por estado

Table read, tabla en RAM

Tomamos un valor de una tabla según nos indica una variable que oficia de índice dentro de la tabla. La tabla se encuentra en RAM.

```
[01]    clrh
[03]    ldx OFFSET
[03]    lda TABLA,x             ; (1-byte table address)
```
7 ciclos, 5 bytes, resultado en A

```
       mov A,TABLA           ; tabla
       add A,OFFSET          ; + offset
       mov MP,A              ; indirecto
       mov A,IAR
```
4 ciclos, 4 words, resultado en A

Table read, tabla en ROM

Tomamos un valor de una tabla según nos indica una variable que oficia de índice dentro de la tabla. La tabla se encuentra en ROM. En el caso de Holtek, la tabla puede ocupar un espacio dentro del segmento de 256 bytes en memoria de programa en el que se está ejecutando el código, o en el último del mapa de memoria. Si la tabla no se halla al principio del segmento, el valor de *COMP* compensa esta diferencia; de esta forma mantenemos generalidad en la comparación.

```
[03]    ldx OFFSET            ; asumo H=0
[04]    lda TABLA,X           ; (2-byte table address)
```
7 ciclos, 5 bytes, resultado en A

```
        mov A,OFFSET
        add A,COMP
        mov TBLP,A
        tabrdl var            ; tabla en última página de 256 bytes memoria de programa
```
5 ciclos, 4 words, resultado en *var* (si desea en A, *var=ACC*)

Direccionamiento indirecto en RAM

Tomamos un valor de un buffer en RAM e incrementamos el puntero

```
[02]    ldx #tabla
[02]    lda ,X
[01]    incx
```
5 ciclos, 4 bytes

```
        mov A,tabla
        mov MP,A
        mov A,IAR
        inc MP                        ; increment memory pointer
```
4 ciclos, 4 words

Conversión binario a BCD

Finalmente, evaluaremos una subrutina muy común, como la empleada para presentar resultados en un display de 7-segmentos o LCD alfanumérico, es decir, la tradicional conversión de un byte a BCD. Para el caso del HC08, la llamamos con el valor a convertir en el registro X. Para el caso de Holtek, necesitaremos una posición de memoria adicional para cargar el valor a convertir (SOURCE).

```
 [04]   BINBCD  mov #8,COUNTER ; LOOP COUNTER
 [01]           clra                   ; 0 -> RESULT
 [03]           sta RESULT
 [01]   L1      lslx                   ; MSB -> carry
 [03]           adc RESULT             ; x2
 [02]           daa                    ; pero en decimal
 [03]           sta RESULT
 [05]           dbnz COUNTER,L1        ; listo?
 [04]           rts                    ; Sí, resultado en A y RESULT
124 ciclos, 16 bytes

BINBCD: mov A,8
        mov COUNTER,A          ; COUNTER=8
        clr RESULT             ; RESULT=0
L1:     rlc SOURCE             ; rota SOURCE a la izquierda (MSB a Cy)
        mov A,RESULT           ; no afecta Cy
        adc A,RESULT           ; suma A + RESULT + Cy, resultado en A (A=2 x RESULT + Cy)
        daa RESULT             ; ajusta A para que sea BCD válido, resultado en RESULT
        sdz COUNTER            ; decrementa 'COUNTER' y si éste es 0, saltea la siguiente instrucción
        jmp L1                 ; vuelve al loop
        ret                    ; resultado en RESULT
60 ciclos, 10 words
```

Periféricos

I/O ports

Al leer un port de salida, Holtek devuelve el estado del latch, como la mayoría de los microcontroladores (excepto PIC). El funcionamiento de ambos es similar

	Motorola	Holtek
Port	PORTA	PA
Control	DDRA	PAC (1=entrada)
Pull-up	PTAPUE	option memory (todos los ports)

Ej.: Uso del port A: bits 2 y 3 entradas, resto salidas

Configuramos los bits y mostramos cómo esperar un cambio de estado y setear un determinado estado en el port

```
INIT:   clr PA                  ; prepara estado bajo en salidas
        mov A,00001100b
        mov PAC,A               ; configura entradas y salidas

WLOA2:  sz PA.2                 ; espera que PA.2 esté en 0
        jmp WLOA2

WHIA3:  snz PA.3        ; espera que PA.3 esté en 1
        jmp WHIA3

        set PA.0        ; pone PA.0 en 1

        clr PA.0        ; pone PA.0 en 0
```

8-bit Timer

El timer de 8 bits permite generar interrupciones al hacer overflow. Posee un prescaler de 8 etapas, lo cual permite manejar tiempos razonables para eventos del mundo exterior. Puede tomar clock del sistema o externo, el cual es divisible por el prescaler por una potencia de 2 (2 a 256), lo cual permite su funcionamiento como timer o como contador de eventos. Posee además un tercer modo de funcionamiento en el cual puede contar (clock interno) mientras está activa la entrada de clock externo, lo cual permite realizar medición de ancho de pulsos. El mismo registro que lleva las cuentas puede ser inicializado a un determinado valor, para lograr interrupciones a la frecuencia deseada.

TMR: registro de cuentas, puede leerse o cargarse con un valor determinado
TMRC: registro de control, configura el modo de trabajo y prescaler

Posee además un par de salidas complementarias con 50% de ciclo de trabajo (divisor por 2 conectado al overflow del contador), orientadas a conectar un buzzer. Esta opción comparte los pines con el port B, y se habilita mediante el estado de uno de los pines, pudiendo habilitarse el buzzer con una sola instrucción.
La interrupción por desborde se habilita en el registro de control de interrupciones INTC, como todos los periféricos. Dicha interrupción tiene un offset propio, es decir, el flujo de programa se redirecciona a una posición fija en memoria de programa.

Ej.: config del timer para beep a ~1 KHz con clock de 4 MHz

Utilizaremos el timer para generar un beep, realizaremos el control del mismo mediante la smacros *BEEPON* y *BEEPOFF*. El buzzer se halla conectado directamente a BZ y \BZ

```
        MACRO
BEEPOFF:
        clr PB.0                ; pone BZ y \BZ a cero
        ENDM

        MACRO
BEEPON:
        set PB.0                ; habilita BZ y \BZ, produce beep
        ENDM

INIT:   mov A,10010000b         ; timer, prescaler=2 (500KHz, 2us)
        mov TMRC,A
        clr TMR
        clr PBC.0               ; PB0 y PB1 como salidas
        clr PBC.1
        BEEPOFF
```

Los pines cambiarán de estado a cada overflow del timer: $2\,us \times 256 = 512\,us$, la frecuencia es $\frac{1}{1{,}024\,ms} \simeq 1\,KHz$

Ej.: config del timer para interrupciones cada ~50 ms con clock de 4 MHz

Generamos interrupciones periódicas cada aproximadamente 50 milisegundos

```
        org 8
        goto HANDLER

        org someplace
HANDLER:
        mov somewhere,A         ; salva A
        mov A,STATUS
        mov somewhere2,A        ; salva STATUS
        mov A,195
        mov TMR,A               ; próxima interrupción
        ... tarea
        mov A,somewhere2
        mov STATUS,A            ; recupera STATUS
        mov A,somewhere         ; recupera A
        reti

INIT:   mov A,10010111b         ; timer, prescaler=256 (256us)
        mov TMRC,A
        mov A,195               ; 256us*195=49,920ms
        mov TMR,A
        set INTC.2              ; habilita int del timer
        set INTC.0              ; habilita interrupciones
```

Ej.: config del timer para medición de ancho de pulso con clock de 4 MHz

Medimos la duración del ancho de un pulso positivo (entre flanco ascendente y descendente), recibido en el pin PC1/TMR. Comenzamos la medición llamando a MEASURE.

```
INIT:   mov A,11001111b         ; pw, flanco ascendente prescaler=1 (1us)
        mov TMRC,A
        set PCC.1               ; config PC1/TMR como entrada

MEASURE:
        set TMRC.4              ; TON, comienza medición
                                ; el timer comienza a contar cuando ve
                                ; un flanco ascendente y se detiene
                                ; en un flanco descendente
WAIT:   sz TMRC.4               ; espera a que el timer pare
        jmp WAIT                ; si el bit no es zero, salta a WAIT
        ; aquí disponemos del ancho del pulso (1-256us) en TMR
```

Ej.: config del timer para demora de ~10 ms con clock de 4 MHz

Utilizamos el timer para introducir una demora de unos 10 milisegundos, sin interrupciones, para lo cual monitoreamos el flag de overflow.

```
INIT:   mov A,10010101b         ; timer, prescaler=64 (64us)
        mov TMRC,A

DEMORA: mov A,156               ; 64us*156=9,984ms
        mov TMRC,A
        clr INTC.5
WAIT:   snz INTC.5              ; chequea TF
        jmp WAIT
        ; continúa el programa
```

16-bit Timer

El timer de 16 bits es esencialmente igual al de 8 bits, excepto que no tiene prescaler. La lectura del registro TMR debe hacerse byte alto - byte bajo, dado que el byte bajo se accede a través de un latch, de modo de acceder a los 16 bits con dos operaciones de 8 bits. De igual modo, la escritura se realiza byte bajo - byte alto.

Conversor A/D

El conversor es de 9 ó 10 bits. En ambos casos, puede usarse como uno de 8 bits de manera muy simple, dado que la información está "justificada a la izquierda", es decir, el byte alto tiene siempre los 8 bits más significativos, mientras

que el byte bajo tiene 1 ó 2 bits, en la posición de bit más significativos. De esta forma, leyendo el byte alto (ADRH), puede utilizarse el conversor como uno de 8 bits; mientras que leyendo ambos (ADRL-ADRH), puede usarse como si el dato fuera una fracción de 16 bits (deben enmascararse los 6 ó 7 bits menos significativos). La elección de qué pines están conectados al conversor, y cuál de ellos es el que provee la señal para la conversión en curso, se realiza mediante el registro de control (ADCR), el mismo registro controla a su vez el inicio de la conversión (bit START). El clocking del AD se controla desde el registro ACSR, que permite dividir por 2, 8 ó 32 la frecuencia de clock. La frecuencia máxima de clock para el AD es de 1MHz. El AD puede interrumpir cuando termina una conversión.

Ej.: conversión de señal en AN0 (PB0) con cristal de 4 MHz, poleado

Configuramos los registros de control para clockear correctamente al AD, y seteamos los valores correspondientes para leer el pin AN0. La conversión se inicia en CONVERT y finaliza cuando el AD pone a cero el bit EOCB en el registro de control, momento a partir del cual los registros ADRL y ADRH contienen el valor correspondiente al resultado de la conversión.

```
INIT:   mov A,00001000b        ; AN0
        mov ADCR,a
        mov a,00000001b
        mov ACSR,a             ; clk/8 (500KHz)
CONVERT:
        clr ADCR.7
        set ADCR.7             ; reset A/D
        clr ADCR.7             ; start A/D (nueva conversión)
WAIT:   sz ADCR.6              ; EOCB
        jmp WAIT
        ; a partir de aquí disponemos del valor en ADRL, ADRH
```

Ej.: conversión de señal en AN0 con cristal de 4 MHz, por interrupciones

En este caso, iniciamos el proceso manualmente (START), y a cada fin de conversión el AD generará una interrupción que procesará su resultado

```
        org 0Ch
        jmp HANDLER

        org someplace
HANDLER:
        mov somewhere,A        ; salva A
        mov A,STATUS
        mov somewhere2,A       ; salva STATUS
        .. tarea, dato en ADRL-ADRH
        clr ADCR.7
        set ADCR.7             ; reset A/D
        clr ADCR.7             ; start A/D (nueva conversión)
        mov A,somewhere2
        mov STATUS,A           ; recupera STATUS
        mov A,somewhere        ; recupera A
        reti

INIT:   mov A,00001000b        ; AN0
        mov ADCR,A
        mov a,00000001b
        mov ACSR,a             ; clk/8 (500KHz)
        set INTC.3             ; habilita A/D interrupt
        set INTC.0             ; habilita interrupciones


START   clr ADCR.7
        set ADCR.7             ; reset A/D
        clr ADCR.7             ; start A/D

; 46R23 usa INTC0 en vez de INTC
; 46R24 también, y el "vector" del AD es 010h
```