COMENTARIO TECNICO

Optimización de código en el 68HC908

Por el Ing. Gabriel Dubatti

e-mail: info@ingdubatti.com.ar

Adaptación Ing. Daniel Di Lella / Electrocomponentes S.A.
e-mail: fae@electrocom.com.ar

Introducción:

La necesidad de optimizar nace de la falta de recursos o del interés de utilizar la menor cantidad posible de ellos para disminuir los costos de producción.

Por ejemplo:

- Usted tiene que realizar un programa para 68HC908 y el mayor tamaño de FLASH disponible En un 908GP32 es de 32Kbytes, y cuando compila su programa, este requiere 34KBytes. Tiene 3 opciones: Cambiar a un modelo con más memoria, o modificar su programa para que entre en 32KBytes (tal vez a costa de perder alguna prestación).
- Sus requerimientos le permiten utilizar varios modelos de controladores de la familia (con costo creciente según la capacidad de FLASH) y usted debe elegir el más económico porque piensa fabricar muchas unidades.
- Su limitación es de memoria RAM y no de FLASH.
- Su limitación es de velocidad de ejecución.

Si no tiene alguna de estas restricciones, lo mejor es elegir el modelo más cómodo para trabajar, ya que el costo del desarrollo suele ser mucho mayor al de la compra de unas pocas unidades y por lo tanto no vale la pena el esfuerzo extra de "optimizar sin sentido" (**los programas optimizados suelen ser mucho más difíciles de leer y mantener**).

Las optimizaciones mejoran la utilización de un recurso, en general, a expensas de otro.

Por ejemplo: 1) Si utiliza varios bits de un byte para guardar un valor (en lugar de un byte completo), ahorra RAM, pero el código de lectura y modificación se agranda ya que hay que enmascarar los otros bits del byte para no modificarlos. 2) Es una conocida práctica de optimización de velocidad, "desarmar" los loops de pocas iteraciones, copiando el código la cantidad de veces que sea necesaria y ahorrando el índice, su testeo y su modificación (a expensas del tamaño del código).

Este artículo muestra algunos "trucos" para optimizar el tamaño del código de los procesadores de la línea 68HC908 de Motorola™.

Optimización de código:

¿Qué optimizar?

1. Lo mejor es realizar algún tipo de **medición del código antes de comenzar** a optimizar para elegir las zonas que puedan aportar una mayor reducción y continuar luego con las menores hasta llegar al punto deseado. Por ejemplo: si uno llama muchas veces a una misma función el ahorro de un sólo byte en la carga de sus parámetros, se multiplicará por el número de llamadas.

2. Buscar código que cumpla con las condiciones de optimización que se detallan luego.
   Cuanto más opciones tengamos, más posibilidades de reducir código tendremos.
   Lo mejor es utilizar un programa que haga esta búsqueda en forma automática
   (en los casos que sea posible) ya que puede ser un proceso muy tedioso.
   vea el utilitario**: OptLst.**

3. Buscar código "grasoso".
   Un mal algoritmo optimizado puede ser mucho peor que uno bueno sin optimizar.
   Pruebe distintas opciones para realizar la misma tarea, evalúe el uso de los distintos recursos
   y elija la más conveniente.

¿Cómo optimizar?
Aquí se presentan algunas técnicas posibles:

1. **Revisión de saltos y llamadas a subrutinas.**
2. **Optimización por reorden de instrucciones**
3. **Uso de "Toda" la FLASH**
4. **Optimización por instrucciones partidas.**
5. **Optimización por extracción de PC.**

Revisión de saltos y llamadas a subrutinas:

Esta técnica consiste en buscar saltos o llamadas a subrutinas que se realicen en modo absoluto
(con JMP y JSR) y que están en alcance relativo (con BRA y BSR).
La diferencia entre un JMP o JSR y un BRA o BSR es de 1 byte, por lo que la posibilidad de
reducción es muy atractiva dado que no crea ningún efecto colateral y con solo rehubicar algunas
funciones, podemos obtener algunos bytes libres.
Cuando intentamos llamar a una subrutina en forma relativa y no se encuentra dentro del rango de
salto (PC-128 a PC+127, siendo PC el contador de programa de la *próxima* instrucción) el
ensamblador nos da error y nos obliga a cambiar la instrucción por su equivalente absoluto.
Pero si luego modificamos la ubicación de algunas rutinas y algunos saltos vuelven a estar en rango
relativo, **no nos avisa**.
Otra variante es cuando se llama a una rutina desde el final de otra. En este caso, se produce un llamado
a subrutina (JSR o BSR) seguido de un RTS, que puede reemplazarse por JMP/BRA, eliminando el RTS.
Para hacer estas búsquedas en forma automática vea el utilitario: **OptLst.**

Optimización por reorden de instrucciones:

Esta técnica consiste en rehubicar el código para eliminar saltos innecesarios.
Por ejemplo: desde el módulo de reset se salta al módulo del programa principal. Si pone el módulo
de reset (con el código del salto al final de ese módulo) y a continuación el principal
(con el punto de entrada al comienzo), puede eliminar ese salto.

```
;== myprog.asm ==
...
$INCLUDE main.asm
...
$INCLUDE reset.asm
...

;== reset.asm ==
....
jmp entrada_principal
....
```

```
    ;== main.asm ==
    ....
entrada_principal:
    ....
```

cambiar por:

```
    ;== myprog.asm ==
    ...
    $INCLUDE reset.asm
    $INCLUDE main.asm
    ...
    ...

    ;== reset.asm ==
    ....
    ....
;   CUIDADO: el siguiente salto no es necesario porque "entrada_principal"
;            viene a continuacion! (no agregar codigo aqui)
;    jmp entrada_principal

    ;== main.asm ==
;   CUIDADO: no colocar codigo antes de "entrada_principal" ya que se entra
;            sin salto!
entrada_principal:
    ....
    ....
```

Este ejemplo es muy sencillo pero permite ver la complicación básica de optimizar:
siempre hay que comentar la optimización para no modificar más tarde una parte y romper las condiciones de validez.
Otro ejemplo: Reordenando el código se puede eliminar un salto (aquí el comentario no es necesario porque no hay riesgo de generar una condición que invalide el funcionamiento).

```
EnviarString:                       ;ENVIA UNA STRING TERMINADA EN 0
    lda     ,x                      ;lee byte a enviar
    aix     #1                      ;avanza puntero (no modifica C.C.)
    bne     envia_1b                ;si "A" no es 0, lo envia
    rts                             ;si "A" es 0, termina
envia_1b:                           ;
    bsr     EnviarChar              ;envia A
    bra     EnviarString            ;ver siguiente byte
```

cambiar por:

```
envia_1b:
    bsr     EnviarChar              ;envia A
    ;<< aqui el salto ya no es necesario
EnviarString:                       ;ENVIA UNA STRING TERMINADA EN 0
    lda     ,x                      ;lee byte a enviar
    aix     #1                      ;avanza puntero (no modifica C.C.)
    bne     envia_1b                ;si "A" no es 0, lo envia
    rts                             ;si "A" es 0, termina
```

Uso de "Toda" la FLASH:

Revise las hojas de datos del procesador en busca de todas las áreas con memoria FLASH.
Puede usar, por ejemplo para tablas, el resto de los vectores de interrupción que no utiliza, teniendo cuidado de no activar dichos vectores.
Por ejemplo: el 68HC908GP32 cuenta con **36 bytes** para vectores y el 68HC908JL3 con **48**, de los cuales se suelen utilizar sólo los últimos.

Optimización por instrucciones partidas:

Esta técnica consiste en simular la operación BRA con un sólo byte.
Para ello, se insertan los opcodes de algunas instrucciones como si fueran datos (con **db**) y se produce el efecto de considerar la "siguiente" instrucción como post-byte (argumento) de la instrucción insertada, salteándola.
Utilizando las instrucciones **BRN** o **CPHX #** puede lograrse el "mismo" efecto que con un **BRA {$+1}** o **{$+2}** para saltear la siguiente instrucción de 1 o 2 bytes.
Por ejemplo:

```
    ;codigo                 ;se ensambla como:
EntradaDEC:                 ;
    deca                    ;4A
    bra seguir              ;20 01
EntradaINC:                 ;
    inca                    ;4C     (OP. de 1 byte a saltear)
seguir:                     ;
    clrx                    ;5F     (5 bytes total)
```

Se reemplaza por:

```
EntradaDEC:                 ;
    deca                    ;4A
    db $21                  ;21     ("21"+"NN" es "BRN {$+2+NN}")
EntradaINC:                 ;
    inca                    ;4C
    clrx                    ;5F     (4 bytes total)
```

Si se ingresa por **EntradaDEC** se ven las siguientes instrucciones:

4A DECA
214C **BRN {EntradaINC+$4D}**
5F CLRX

Si se ingresa por **EntradaINC** se ven las siguientes instrucciones:

4C INCA
5F CLRX

Optimización por extracción de PC:

Esta técnica consiste en utilizar las instrucciones PULH y PULX para cargar el registro HX con el PC de retorno que esta en el stack (desde una rutina) y ahorrarse la carga inmediata LDHX #. Esta opción es ideal cuando el valor a cargar en HX es una tabla a la que se desea saltar según el registro A, dado que no es necesario retornar al punto siguiente a la tabla.

```
;== CARGA de HX inmediato ==
    ldhx    #Tabla
    bsr     EnviarString
    ..

Tabla:  db 'Hola Mundo',0
```

Se reemplaza por:

```
;== EXTRACCION del PC de RETORNO ==
    bsr     EnviarINMString ;-3 bytes de carga
    db      'Hola Mundo',0  ;<< la tabla se coloca luego de la llamada
    ..                      ;<< el codigo se sigue ejecutando aqui

;-- OPCION1 -- (usando la rutina existente)
EnviarINMString:            ;+5 bytes: pulh/pulx/bsr/jmp,x
    pulh                    ;carga HX con el PC luego de la llamada (STRING)
    pulx                    ;
    bsr   EnviarString      ;envia la string, HX= byte que sigue al 0
    jmp   ,x                ;retorna luego del 0

;-- OPCION2 -- (incorporando la carga a la rutina)
EnviarINMString:            ;+3 bytes: PULH/PULX/DB $65
    pulh                    ;carga HX con el PC luego de la llamada (STRING)
    pulx                    ;
    BRA_MAS_2               ;BRA A "enviar_str"
                            ;   (CPHX #=$65 inmediato=BSR+OFFSET)
envia_1b:                   ;
    bsr     EnviarChar      ;envia A
enviar_str:                 ;
    lda     ,x              ;lee byte a enviar
    aix     #1              ;avanza puntero (no modifica C.C.)
    bne     envia_1b        ;si "A" no es 0, lo envia
    jmp     ,x              ;si "A" es 0, termina (retorna luego del 0)
```

En este ejemplo se observa que se produce un ahorro de 3 bytes en la carga del puntero a la tabla y se aumenta en 5 o en 3 bytes la rutina, según la opción que se utilice. **La ventaja se obtiene cuando hay más de una llamada**, dado que por cada una se ahorran 3 bytes.
Esto funciona dado que la secuencia **PULH + PULX + JMP ,X** es equivalente a **RTS**.

Conclusión:

En el presente artículo se mostraron varias técnicas para obtener lugar extra en la memoria FLASH.
Esta lista es sólo una muestra (su creatividad es el límite), pero representa el espectro de las distintas opciones con sus ventajas y desventajas. En próximos artículos se verán otras más.

NOTA: Para obtener los utilitarios de software que facilitarán la tarea de optimizaciones de código, visite el sitio Web **www.ingdubatti.com.ar** y podrá bajarlos en forma gratuita.