Manual de Programación

Lenguaje Ensamblador HP Saturno

⊖HP49G⊖

Escrito por

Alberto Villalba Kong

Gustavo Portales Villar

Autores:

Alberto Villalba Kong
Ciudad de Arequipa
Dpto. de Arequipa
Perú
tifosis@yahoo.com

Gustavo Portales Villar
Ciudad de Trujillo
Dpto. de La Libertad
Perú
hp@gaak.org

Primera Edición: Abril 22, 2006

Prefacio

El Lenguaje Ensamblador es muy diferente a otros lenguajes de programación. Cuando el usuario aprende un lenguaje típico, mayormente se empieza por un programa simple que muestre “Hola Mundo”. Pero en lenguaje ensamblador, esto ya no es tan simple. Recuerde que antes de escribir cualquier código en este lenguaje, se debe entender de cómo es que trabaja el procesador y cómo trabaja el sistema operativo.

En este libro, la primera parte es un material de introducción acerca de los sistemas numéricos usados y operaciones. Es importante que usted conozca estos temas para manipular la CPU de su calculadora. Pero, no se preocupe, pronto empezará a programar!. Después de los conceptos de introducción, verá los comandos del lenguaje ensamblador, conocidos como instrucciones, que se encuentran en pequeños grupos, y empezando con programas extremadamente sencillos.

Debido a que el código máquina es binario, usted debe aprender a usar los mnemónicos, que son simplemente la representación de este código binario. Este libro está dirigido a propietarios de las calculadoras HP49G, usando el ensamblador conocido como MASD. La sintaxis de este ensamblador es bastante fácil, además, la sintaxis HP para lenguaje ensamblador, usado por Jazz y HP Tools (Debug2 ó Debug4x), es muy similar, así usted podrá usar la sintaxis HP con unos simples cambios.

Nosotros pensamos que mucha gente aprenderá el lenguaje ensamblador leyendo este libro, el cual está basado en la experiencia de los autores, que arrastra consigo, las referencias de programadores franceses e ingleses. Una gran parte de este libro, está inspirada en la documentación de F. H. Gilbert para las calculadoras HP48. También, la documentación SASM en español, fue muy útil.

La información de este libro, es sólo para HP49G, pero algunos ejemplos podrían usarse en la HP49G+. Información de ésta última, hay muy poco, y no trabaja con procesador Saturno, por lo que esperamos “no arriesguen”.

Agradecimientos en orden alfabético a: Andree Schoorl, Carsten Dominik, Christoph Giesselink, Cyrille de Brebisson, Dan Kirkland, David Winter, Eric Rechlin, F.H. Gilbert, Jean-Ives Avenard, Joe Horn, John H. Meyers, Jurjen Bos, Mario Mikocevic, Mika Heiskanen, Peter Geelhoed, Randy Ding, Raymond Hellstern, Richard Steventon, Rick Grevelle, Thomas Rast.

Actualizaciones de este libro estarán disponibles en <http://www.hpcalc.org>

Si tiene algo que decir, puede escribirnos a:

Alberto Villalba <tifosis@yahoo.com>
Gustavo Portales <hp@gaak.org>

INDICE

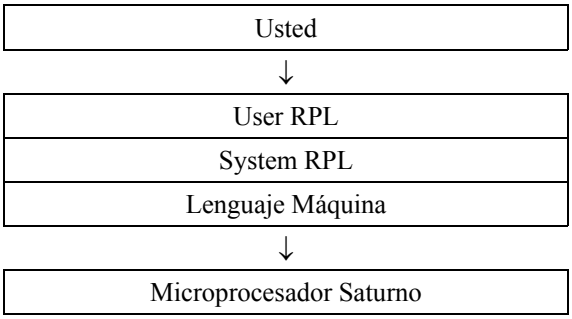
Parte I: Conceptos de Programación HP49G

1 Introducción a User RPL y System RPL

El usuario básico de la HP49G puede usar User RPL. Es fácil de entender, y es el primer lenguaje que puede usar para programar su HP.
Cada vez que usted use un comando User RPL, se realiza una verificación previa de argumentos, por consiguiente: User RPL es un lenguaje bastante seguro.

Si vemos el software interno en la HP, descubriremos que cada comando User RPL está dividido en varios comandos internos. A esta estructura interna, la conocemos como System RPL. Usualmente, cada uno de los comando System RPL, terminan llamando código máquina, que es ejecutado directamente por el procesador de las HP's, el Saturno.

Para ayudarle a entender, mire esto:



Las HP48/HP49G ofrecen tres lenguajes de desarrollo. User RPL es seguro y fácil, pero el más lento de todos. System RPL es rápido debido a que no verifica argumentos, es por eso que se debe tener cuidado con este lenguaje, ya que debemos estar seguros de tener todos los argumentos necesarios, antes de ejecutar el comando.

Finalmente, tenemos lenguaje ensamblador, que produce el lenguaje máquina (ML: Machine Language). Éste es el código más rápido, directamente ejecutado por el procesador, y por supuesto, nos da el control total de la HP.

Si usted desea aprender System RPL (conociendo bien el User RPL), todo lo que se necesita es una lista completa de los comandos System RPL, llamados puntos de entrada (entry points), en la ROM.

Lenguaje ensamblador, no está relacionado a User RPL o System RPL, así que usted está a punto de aprender algo completamente distinto. Ensamblador no tiene todos los comandos que están disponibles para el System RPL.

No es un lenguaje complejo, pero cuando se escribe un programa en ensamblador, se puede obtener dos resultados: Uno, que es lo que se deseaba, o ¿?

Un buen programa, debe ser rápido y pequeño. Lastimosamente, lo que se obtiene con el ensamblador, mayormente es grande (en relación al resto de lenguajes), pero su velocidad, es de suma importancia.
Al inicio, usted no se preocupe por el tamaño del programa... Primero consiga lo que intenta hacer, y luego, optimice su código. La HP49G nos proporciona bastante espacio de almacenamiento.

Con el ensamblador, no sólo se consiguen programas rápidos, sino que, también, dormirá menos, pero es como se inicia todo programador.
Así es que, le deseo suerte! (y paciencia).

2 Lenguaje ensamblador y lenguaje máquina

El procesador Saturno, como cualquier otro procesador, usa una serie de 0's y 1's para trabajar

Recuerde que:
- Lenguaje máquina
- Existe uno para cada comando disponible en el procesador Saturno.

Es importante no mezclar "lenguaje ensamblador" y "lenguaje máquina". El último, es la forma resultante del anterior, después de haberlo traducido con un compilador.

Ese es el porqué lenguaje ensamblador es llamado un lenguaje de "nivel-bajo". En lenguaje ensamblador, cada mnemónico corresponde a una instrucción. Ese es el porqué lenguaje ensamblador es **específico** para cada procesador. Cuando escribimos programas en lenguaje ensamblador, primero escribimos mnemónicos en un archivo fuente (fuente porque es la fuente de nuestro trabajo). Luego usaremos un programa llamado un "ensamblador" o "compilador", el que traducirá mnemónicos al lenguaje máquina, en una serie de 0's y 1's (dígitos binarios), que son unidos y guardados usando paquetes de cuatro-bits en la HP.

3 La base binaria y los nibbles

Antes de empezar a programar en ML es necesario que se sepa manejar bien los números hexadecimales y binarios, también recordar lo siguiente:

BIT Es la mínima unidad, este puede ser cero (0) o uno (1)
BYTE Esta conformado por 8 Bits
WORD Conformado por 16 Bits

Un nibble es igual a un cuarteto y esta conformado por 4 bits.

Decimal	Binario	BCD	Hexadecimal
0	0000	0000	0
1	0001	0001	1
2	0010	0010	2
3	0011	0011	3
4	0100	0100	4
5	0101	0101	5
6	0110	0110	6
7	0111	0111	7
8	1000	1000	8
9	1001	1001	9
10	1010	0001 0000	A
11	1011	0001 0001	B
12	1100	0001 0010	C
13	1101	0001 0011	D
14	1110	0001 0100	E

15	1111	0001 0101	F

- La base de un número decimal es **10,** representado por 10 símbolos: **0 1 2 3 4 5 6 7 8 9**
- La base de un número binario es **2,** representado por 2 símbolos: **0 1**
- La base de un número hexadecimal es **16,** representado por 16 símbolos: **0 1 2 3 4 5 6 7 8 9 A B C D E F**
- El código BCD se usa para representar valores decimales en formato binario, cada cuarteto representa un digito.

4 Convirtiendo de binario a decimal

Empecemos examinando dos nibbles juntos, lo que se conoce como "byte".

4 bits	→	un cuarteto	→	un nibble
8 bits	→	un octeto	→	un byte

Demos un número a cada uno de los bits, de derecha a izquierda, que llamaremos "posición" de un bit.
Ejemplo: Tengo dos nibbles juntos (un byte) que es: 01001101, así tendremos:

```
pos: 7 6 5 4 3 2 1 0
bit: 0 1 0 0 1 1 0 1
```

Como podemos ver, los valores posiciónales deben empezarse en "cero", luego contemos de derecha a izquierda. Cada bit tiene una posición, una posición específica.

Podemos decir entonces que:
bit0 = 1
bit1 = 0
bit2 = 1
bit3 = 1
bit4 = 0
bit5 = 0
bit6 = 1
bit7 = 0

Para convertir un número binario, a su forma decimal, simplemente hacemos lo siguiente: multiplicar cada uno de los bits por "2 elevado a su posición", luego sumar todos los términos. Aplicándolo en el ejemplo, sería:

```
pos: 7 6 5 4 3 2 1 0
bit: 0 1 0 0 1 1 0 1
```

$(0 * 2^7) + (1 * 2^6) + (0 * 2^5) + (0 * 2^4) + (1 * 2^3) + (1 * 2^2) + (0 * 2^1) + (1 * 2^0)$

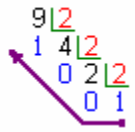
Esto es: 0 + 64 + 0 + 0 + 8 + 4 + 0 + 1 = 77

En efecto: # 77d = # 01001101b

Como convertir un número decimal a otra base

Si nosotros queremos convertir cualquier número decimal a otro tipo de base, tenemos que dividir el número entre la base a la que deseemos convertir, se ira dividiendo hasta que la división ya no sea posible.
Ejemplo:
Convertir el número decimal 9 a binario.

Solución:
Los números binarios tienen como base 2, por lo tanto se dividirá entre 2.



El resultado se tomara de derecha a izquierda como muestra la flecha, en este caso sera 1001, Practique y compare con el cuadro de arriba.

5 Bit más significativo y bit menos significativo

Como ya hemos visto, cada bit tiene un valor posicional.

El bit del extremo derecho, es llamado Bit Menos Significativo (LSB: Least Significant Bit).
El bit del extremo izquierdo, es llamado Bit Más Significativo (MSB: Most Significant Bit).

En el nibble, el LSB y MSB será:	En el byte, el LSB y MSB será:
pos: 3 2 1 0 MSB LSB	pos: 7 6 5 4 3 2 1 0 MSB LSB

6 Números fraccionales

Un número fraccional tiene dos partes: la parte entera, antes de la marca fraccional (un punto o coma decimal, dependiendo de su región), y la parte fraccional, después de la marca fraccional.

Pero... cómo podemos tener números fraccionales usando bits?

Simple, los valores posiciónales de la parte entera son positivos, y en la parte fraccional, los valores posiciónales son negativos. Un ejemplo sería conseguir el valor decimal del binario: 100.1011

```
pos: 2 1 0   -1 -2 -3 -4
bit: 1 0 0 .  1  0  1  1
```

Luego:

$(1 * 2^{2}) + (0 * 2^{1}) + (0 * 2^{0}) . (1 * 2^{-1}) + (0 * 2^{-2}) + (1 * 2^{-3}) + (1 * 2^{-4})$

Usted debe recordar que: $(2^{-n}) = (1 / 2^{n})$

Tenemos entonces:
(4 + 0 + 0) . (0.5 + 0 + 0.125 + 0.0625)

Los resultados son: (4) . (0.6875)

Finalmente, “unimos” los resultados: 4.6875

Entonces: # 4.6875d = # 100.1011b

Rápidamente, desarrollar lo siguiente: Convertir el binario 0.0110 a decimal.

```
pos: 0   -1 -2 -3 -4
bit: 0 .  0  1  1  0
```

$(0 * 2^{0}) . (0 * 2^{-1}) + (1 * 2^{-2}) + (1 * 2^{-3}) + (0 * 2^{-4})$

(0) . (0 + 0.25 + 0.125 + 0)

(0) . (0.375)

Entonces: # 0.375d = # 0.0110b

7 Cálculos usando base 2

En esta sección, usted aprenderá a sumar, restar, multiplicar y dividir dos valores binarios. Esto le ayudará a entender cómo es que trabaja el procesador Saturno.

7.1 Sumando

Recuerde las siguientes reglas:

0 + 0 = 0 → no acarrea
0 + 1 = 1 → no acarrea
1 + 0 = 1 → no acarrea
1 + 1 = 0 → acarrea

Ejemplo:

```
  0101
+ 1101
------
 10010
```

Nota: Cuando se dice carry, esto indica que hubo un desbordamiento en la memoria, ósea este ya alcanzo su límite máximo. Cuando yo tengo 1 + 1, yo pongo un cero y muevo el carry 1 a la izquierda, se trabaja de derecha a izquierda, agregando dos bits cada tiempo.

7.2 Restando

Se seguirán estas reglas:

0 - 0 = 0
0 - 1 = -1 → resultado negativo
1 - 0 = 1
1 - 1 = 0

Existen varias formas de representar números negativos en base binaria, a continuación se vera solo un método:

7.3 Números Negativos

También se pueden representar números negativos, para ello veremos dos métodos.

7.3.1 El signo binario: primer método

Cuando el MSB es cero, el valor es *positivo*. Cuando el MSB es uno, el valor es *negativo*.

Si nosotros tenemos un nibble:
Aquí puede haber un cero positivo o negativo.

Si tenemos un nibble,

un cero positivo es: 0000
un cero negativo es: 1000

(Si,)

Binario	Decimal
de 0000 a 0111 cuando es positivo	de cero positivo a 7 cuando es positivo
de 1000 a 1111 cuando es negativo	de cero negativo a -7 cuando es negativo

Si usamos un byte (dos nibbles):

Binario	Decimal
de 00000000 a 01111111 cuando es positivo	de cero positivo a 127 cuando es positivo
de 10000000 a 11111111 cuando es negativo	de cero negativo a -127 cuando es negativo

Recuerde!! Tenemos *dos* ceros aquí: un cero negativo y un cero positivo.

7.3.2 El complemento de 2: Segundo método

Cuando todos los bits de una palabra (word) se invierten (los ceros se convierten en unos, y los unos en ceros), conseguimos una palabra invertida, por ejemplo:
0101 se convierte en 1010
1001 se convierte en 0110

Pero el complemento de 2 es el complemento verdadero para una palabra, y este es el complemento de uno más uno:
(complemento de 2) = (complemento de 1) + 1

Ejemplo:
Hallar el complemento de 2 de 01101110 ?

Primero, se invierten todos los bits, y se obtiene: 10010001
Luego sumamos uno:

```
10010001
+       1
--------
10010010
```

Entonces el complemento de 2 de 01101110 es 10010010

Veamos como esta codificado un byte (8 bits).

Si nosotros tenemos 0000 0000 y restamos uno obtenemos:
`0000 0000 - 1 = 1111 1111`

Como se puede observar en el complemento de 2, los números negativos comienzan con 1 en MSB. Si el MSB es 0, el valor es positivo.

Los valores positivos van a partir de 0000 0000 a 0111 1111, o 0 a 127 en decimal.
Los valores negativos van a partir de 1000 0000 a 1111 1111, o -128 a -1 en decimal.

Note que solo estamos usando 8 bits, pero podríamos utilizar menos (como 4) o más (como 64).

Nosotros no conseguimos un cero positivo o un cero negativo aquí.

No podemos hacer (1000 0000 - 1) !!
Por qué? Iríamos de un valor negativo a un valor positivo.
También, no podemos hacer (0111 1111 + 1) porque iríamos de un valor positivo a uno negativo

Mejor veamos un ejemplo, calculemos 8-11 (que son valores decimales) usando 8 bits y el complemento de 2

8 es 0000 1000
11 es 0000 1011

Para restar 11, voy ha agregar su valor negativo:
(8 - 11) se convierte en (8 + -11)
El “complemento de 2” de 0000 1011 es 1111 0101

Ahora sumo los dos valores:

```
  0000 1000
```

```
+ 1111 0101
  ---------
  1111 1101
```

Como se observa el MSB es igual a 1, por lo tanto el resultado es negativo. Pero no puedo utilizar este valor final directamente. Tengo que:

1) Calcular su complemento de 2
2) Recordar que es negativo y
3) Agregar el signo – delante de el.

El complemento de 2 de 1111 1101 es: 0000 0011
Finalmente es igual a: -3

RECUERDE!
Cuando usamos el complemento de 2 para hacer cálculos, el resultado final no es directamente usable.
Tenemos que:

1) Calcular el "valor absoluto"
2) Poner el signo de a cuerdo al valor del MSB

7.4 Multiplicando

La multiplicación se puede realizar a través de sumas, como se puede observar que cuando se suma un número a si mismo, este se esta multiplicando por 2, en binario basta con desplazar todos los bits que lo componen hacia la izquierda y añadir un cero a la derecha.

Se puede realizar una multiplicación por 2 con la instrucción A=A+A

También se puede multiplicar por 4 el registro C limitado al campo A:
C=C+C.A C=C+C.A

La HP esta conformada por cuartetos, cada cuarteto es un grupo de 4 bits, lo que es igual a un número hexadecimal.

Se puede realizar una multiplicación por 16 con la instrucción ASL.campo. (así mismo para B,C y D)

Imaginémonos que quisiéramos multiplicar el número que se encuentra en el registro A por 33 recordando que:
Para multiplicar por 2 (A+A)
Para multiplicar por 16 (ASL)

Ya que no es necesario olvidar que la multiplicación es una serie de adiciones, bien cómo se pueden hacer 33 con nuestras herramientas básicas:
33=1+1 +.. (33).. + 1 demasiado largo

33=2*2 *.. (16).. * 2+1 demasiado largo
33=16*2+1 ah no esta mal !!!

El truco esta en observar cuál es el múltiplo de 16 más cercano posible aquí 33/16=2.06
Entonces 32=2*16 luego 33-32=1 el programa seria:

```
                    %El número que debe multiplicarse se encuentra en A
C=A.W               %Guardamos A para la adición final
ASL.W               % aquí A=A*16
A=A+A.W             % aquí A=A*16*2
A=A+C.W             % finalmente A=A*16*2+1
                    % entonces A contiene A*33
```

Ejemplo 2:
Multipliquemos el registro A por 254.

```
C=A.W               % Guardamos A en C
ASL.W               %  A=A*16
ASL.W               % A=A*256
C=C+C.W             % C=C*2
C=C+C.W             % C=C*4
A=A-C.W             % finalmente A=A*16*15-A*4 = 254
```

7.5 Dividiendo

Se puede realizar una división entre 2 con la instrucción ASRB (En el caso del registro A)
Se puede realizar una división entre 16 con la instrucción ASR.campo. (en el caso del registro A)

8 Base hexadecimal

Para representar un número en base hexadecimal (base dieciséis), es necesario disponer de un conjunto, o alfabeto, de 16 símbolos.

El sistema hexadecimal utiliza 16 estados, y su conjunto de sus símbolos está compuesto por:

0,1,2,3,4,5,6,7,8,9, A, B, C, D, E, F.

Como necesitamos 16 dígitos diferentes y sólo poseemos 10 y no era cuestión de inventar símbolos raros, se escogieron como símbolos numéricos extra las 6 primeras letras del abecedario (en mayúsculas), la A representa al dígito 10, la B al 11, la C al 12, la D al 13, la E al 14 y la F al 15.

Esta base se usa para abreviar números binarios, cada digito representa un conjunto de 4 bits.

Para indicar explícitamente que un número está en base 16 (Hexadecimal) lo representaremos con “h“ por ejemplo 12h.

9 Convirtiendo de hexadecimal a decimal

En el sistema hexadecimal, cada dígito tiene asociado un peso equivalente a una potencia de 16, entonces para convertir de hexadecimal a decimal se multiplica el valor decimal del dígito correspondiente por el respectivo peso y realizar la suma de los productos.

Ejemplo:

Convertir el número **31Fh** a decimal.

$31Fh = 3x16^2 + 1x16 + 15 \times 16^0 = 3x256 + 16 + 15 = 768 + 31 = 799d$

10 Escribiendo grupos de cuatro bits (cuartetos)

Nosotros podemos escribir nuestros cuartetos usando dígitos hexadecimales, como sabemos un cuarteto consta de cuatro bits, podemos tener 16 diversos valores, y tenemos 16 símbolos con la base hexadecimal, veamos unos ejemplos.

Representar el grupo de 4 bits igual a 1111 en forma hexadecimal.
Como sabemos 1111 es igual Fh, pero que pasa si queremos representar el siguiente grupo de bits:

11110110101.

La solución es fácil, solo se deben separar en grupos de 4 bits de derecha a izquierda y en caso que no se complete el grupo de 4, los completamos agregando ceros, luego reemplazamos por su equivalente en hexadecimal:

111 1011 0101 = 0111 1011 0101

0111 ➔ 7
1011 ➔ B
0101 ➔ 5

Entonces finalmente 1110110101 = 7B5h

11 BCD

El código BCD no es nada más que una formación de números decimales codificados con su equivalente en binario, se necesitan 4 bits para codificar cada dígito.

12 Convirtiendo un decimal a forma BCD

Convertir un número decimal a la forma BCD es lo mas simple que puede existir, veamos un ejemplo de cómo codificar el número 874.

8 = 1000
7 = 0111
4 = 0100

El resultado en BCD es: **1000 0111 0100** como se puede apreciar cada número es reemplazado por su equivalente en binario.

13 Operaciones lógicas

Ésta es una parte importante

George Boole (1815-1864), publicó un libro llamado “Analyse mathematique de la logique”, algo así como “Análisis matemático de la lógica”. Lo que conocemos como álgebra Booleana.

En el álgebra booleana, tenemos dos valores: 0 y 1.

0 representa “falso”
1 representa “verdadero”

Nombre	Símbolo usado
AND	•
OR	+
NOT	–

13.1 OR

Tabla de verdad: OR		
A	B	S
0	0	0
0	1	1
1	0	1
1	1	1

```
S = A + B
```

Ejemplo:
```
A = 0101
B = 0011
S = 1000
```

13.2 AND

Tabla de verdad: AND		
A	B	S
0	0	0
0	1	0
1	0	0
1	1	1

$S = A \cdot B$

13.3 NOT (Inversor)

Tabla de verdad: NOT	
A	S
0	1
1	0

13.4 NOR (NOT - OR)

Tabla de verdad: NOR		
A	B	S
0	0	1
0	1	0
1	0	0
1	1	0

13.5 NAND (NOT - AND)

Tabla de verdad: NAND		
A	B	S
0	0	1
0	1	1
1	0	1
1	1	0

13.6 XOR (OR exclusivo)

Tabla de verdad: XOR		
A	B	S
0	0	0
0	1	1
1	0	1
1	1	0

Parte II: Herramientas de Programación

MASD, puede usarse en cualquier HP49G. Como parte de su configuración, tiene al flag –92, el cual, debe estar desactivado, para los ejemplos de este documento (leer la documentación MASD para más detalles.).

Si se desean utilizar otras herramientas para facilitar la programación, se puede usar Emacs o EditPro, ambas requieren de la librería Extable...., también se puede usar Jazz, esta requiere de la librería HPTabs…

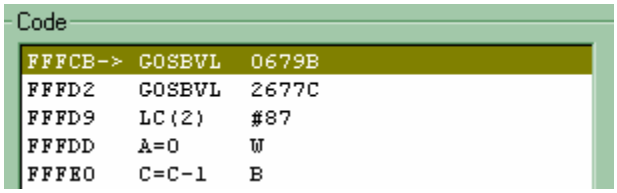
En la PC:
Debug2 o Debug4x.
Emu48 v1.34 (o superior) es muy, pero muy útil, para realizar un seguimiento preciso de lenguaje máquina, todas estos programas se encuentran en hpcalc.

14 Usando Emu48 para Lenguaje Máquina

Se puede hacer un seguimiento de cómo corre el lenguaje máquina, el EMU48 es muy útil para esto, para hacer el seguimiento se debe de colocar el programa escrito en la pila, y ejecutamos DUP ➔A 10 +, obtendremos una dirección:

Después en el Emy48 nos vamos a Tools ➔ Debbuger… ➔ Breakpoints ➔ Edit Breakpoints… ➔ ADD
Agregamos la dirección obtenida:

Elegimos OK. Luego presionamos F5, y en el emulador DROP EVAL, el Debbuger se detendrá en el inicio de nuestro código y podremos hacer un seguimiento de cómo se ejecuta nuestro programa.



Presionamos F7 o F8 para pasar a la siguiente instrucción, F7 entra a ejecutar los saltos, y F8 no entra.

15 Un Ensamblador: MASD

Para ensamblar nuestros códigos, podemos usar MASD que ya trae nuestra HP incorporada, solo basta con ejecutar el comando ASM.

16 Parte III: El procesador Saturno

REGISTROS
Ahora preste mucha atención a este capitulo que es la parte mas importante, se debe saber para que sirve cada uno de los registros que se mencionan a continuación por que son estos registros los que usaremos para programar en assembler.

1	4 Registros de calculo	A B C D
2	2 Punteros	D0 D1
3	5 Registros de almacenamiento	R0 R1 R2 R3 R4
4	16 Bits bandera	ST0 hasta ST15
5	2 Registros	IN OUT
6	1 Pila	RSTK
7	1 Puntero de Campo	P
8	1 Contador de programa	PC
9	8 Bits de estado	CARRY INTON INTOFF SHUTDN INT HST

Existen 3 Flags: ST, HST, CARRY
Estos pueden tener solo un bit, esto quiere decir que solo pueden tener un valor, cero o uno:
0 Desactivado
1 Activado

CAMPOS
Los campos se usan para representar el tamaño que consideraremos de cada registro, cada campo tiene un tamaño establecido.

DIVISION EN CAMPOS DE UN REGISTRO DE 64 BITS															
15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
W															
S	**M**												**X**		
												A			
													XS	**B**	

CAMPOS		CAPACIDAD DEL CAMPO		CUARTETOS QUE CONTIENE
W	*(Word)*	16 cuartetos	(64 bits)	0 - 15
S	*(Mantisa Signo)*	1 cuarteto	(4 bits)	15
M	*(Mantisa)*	12 cuartetos	(48 bits)	3 - 14
X	*(eXponente)*	3 cuartetos	(12 bits)	0 - 2
XS	*(eXponente Signo)*	1 cuarteto	(4 bits)	2
B	*(Bytes)*	2 cuartetos	(8 bits)	0 - 1
A	*(Direcciones)*	5 cuartetos	(20 bits)	0 - 4

Por ejemplo cuando se dice **B** campo **A,** esto quiere decir los 5 primeros cuartetos de **B**, como se ve en la tabla el campo **A** tiene un tamaño de 5 cuartetos.

A continuación una breve explicación de cada uno de los registros:

1	A,B,C,D	Son los más importantes pues estos hacen las principales operaciones matemáticas. No son exactamente iguales, ya que ciertas instrucciones solo operan con algunos registros y no con otros. Con ellos se puede sumar, restar y realizar operaciones lógicas.
2	D0, D1	Estos registros son los punteros de dirección y se utilizan para acceder a la memoria, es decir para leer y escribir datos en la memoria. También se encuentran los punteros de memoria DAT0, DAT1
3	R0. R1, R2, R3, R4	Estos registros sirven para almacenar temporalmente los datos para un futuro uso. Con ellos no se pueden hacer cálculos, solo se puede leer y escribir en ellos.
4	Bits de Bandera (ST)	Constituido de 16Bits: ST0 ST1 ST2 ST3 ST4 ST5 ST6 ST7 ST8 ST9 ST10 ST11 ST12 ST13 ST14 ST15. Son testeables y utilizables por separado. Sirven al programador para guardar un valor, pero en un estado, por ejemplo si una tecla ya había estado apretada en el paso anterior Los últimos Bits (15,14,13,12). Tienen funciones específicas: 12 - Deep Sleep Override. Active o DeepSleep debe continuar activo (desativado - fuerza un wakeup). 13 - Activar si una interrupcion ha ocurrido. 14 - Activar si una interrupcion esta pendiente. 15 - Con 1 se habilitan las interrupciones y con 0 las interrupciones son invalidas (ON C, ON A F, etc..).
5	IN	Sirven para la comunicación con el sistema para controlar cualquier periférico conectado a un Chip (teclado, IR, etc...). El registro **IN** es de apenas Lectura. Es un registro de 16 bits. De todas formas solo los bits del 0 al 5 y el 15 se utilizan al testear el teclado.
	OUT	El registro **OUT** es para ser escrito. Es un registro de 12 bits, solo los bits del 0 al 8 y el 11 se utilizan. Los bits del 0 al 8 sirven para testear el teclado y el bit 11 sirve para producir sonidos.
6	Pila o RSTK (Return Stack)	La pila o RSTK (ReturnSTacK) Es igual que la pila pero solo contiene 8 niveles, cada uno de 20 bits, es útil para guardar información temporalmente.
7	Puntero de registro o campos (P)	**P** es un registro de 4 bits que sirve principalmente para definir el tamaño de los campos **P** y **PW**. Sirve como un selector de limite a ser utilizado por los registros **A** y **C**. En ciertos casos en el cual se necesita de muchas operaciones o bucles, este puntero puede ser utilizado como contador.
8	Puntero de Instrucción (PC):	También llamado "Program Counter". El registro **PC** es de 20 bits, contiene la dirección de memoria de la próxima instrucción a ser ejecutada. Su modificación conduce al programa a otra dirección.

<table>
<tr><th colspan="3">9</th></tr>
<tr><td colspan="3">BITS DE ESTADO
Se Dividen en 8, A continuación se describen cada uno de ellos.</td></tr>
<tr><td>CARRY:</td><td colspan="2">Carry es un flag que se puede presentar en varios eventos.
Puede ser fácilmente testeado por GOC y GONC.
Este flag se activa si existe un desbordamiento en el registro, es decir cuando se excede el numero de bits disponibles al realizar una operación. Por ejemplo:

Si tenemos en C(1)=F y en A(1)=1 ósea en el primer nibble de C y de A tenemos un F y un 1, respectivamente y queremos sumar los dos registros pero solo en el primer nibble, la operación en binario seria:

<pre> 1111
+ 0001

 10000</pre>
Pero como solo tenemos 4 bits disponibles el procesador te avisa que hubo un desbordamiento encendiendo el CARRY, eso se puede notar al hacer esta operación y depurarla con algún debuger de ML.
También sirve como control para un programa.
En operaciones aritméticas este sera activado cuando una operacion resulta en un error: (pasa el límite, como se indico mas arriba).
Ejemplo: LA(5) FFFFF A=A+1 A
O cuando la operacion resulta en un número negativo.
Ejemplo: LA(5) 0 A=A-1 A.
Este sera desactivado cuando las operaciones no resultan en errores. Tambiém sirve para operaciones de test, este sera activado cuando el test es verdadero.</td></tr>
<tr><td>INTON:</td><td colspan="2">Este bit activa las interrupciones del teclado.</td></tr>
<tr><td>INTOFF:</td><td colspan="2">Este bit desactiva las interrupciones del teclado.</td></tr>
<tr><td>SHTDN:</td><td colspan="2">Este bit indica si el procesador esta en modo SHUTDOWN, es decir en modo de vigilia.
Este bit no es testeable, solo se puede activar, poniéndolo a 1.</td></tr>
<tr><td>INT:</td><td colspan="2">Este bit indica que el procesador esta realizando una interrupción.
Se puede poner a 0 mediante un RTI pero esto provoca también un retorno al subprograma.
No se puede testear.</td></tr>
<tr><td rowspan="5">HST</td><td colspan="2">Consta de 4 bits que son SB, MP, SR, XM.
Hay sólo un bit que nosotros podemos poner a 0 : XM.
Nosotros usaremos una instrucción llamada RTNSXM (Retorna XM con 1).</td></tr>
<tr><td>SB:</td><td>Indica que una rotación hacia la derecha ha provocado la perdida de un bit.
Se puede testear.</td></tr>
<tr><td>MP:</td><td>Se pone a 1 al insertar una tarjeta. Se puede testear y poner a 0.</td></tr>
<tr><td>SR:</td><td>Se pone a 1 si se produce una avería en un chip de la HP
(No sirve para nada). Se puede testear y poner a 0.</td></tr>
<tr><td>XM:</td><td>Tarjeta retirada (HP48), Se puede testear y poner a 0.</td></tr>
</table>

PUNTOS IMPORTANTES ANTES DE PROGRAMAR EN ML

* Cuando se inicia un programa en ML algunos registros y algunos indicadores son usados internamente por la HP:
D1 apunta al primer nivel de la pila.
D0 Apunta a la siguiente dirección a ejecutar en RPL.
B campo **A** apunta a la pila de retorno **RSTK**,
D campo **A** contiene una estimación de la memoria libre en bloques de 5 nibbles.

Cuando un programa empieza, estos valores se usan, por lo tanto la primera cosa a hacer es guardar estos registros a no ser que queramos modificarlos.
Aveces nosotros tendremos que guardar algún valor del registro porque si nosotros perdemos **B** o **D0**, nosotros entraremos en problemas (la HP se colgara o perderemos la memoria).

Para guardar estos registros se hace con una llamada a una subrutina en ROM mediante la orden **SAVE,** si se usa JAZZ seria **GOSBVL =SAVPTR**. Aquí vemos una instrucción nueva **GOSBVL** que significa GOSuB Very Long, pues esto realiza el salto a una subrutina en ROM.
En este caso la subrutina vendría a ser **SAVPTR** que lo que hace es salvar los registros **D0 D1 D.A B.A** en la memoria del sistema.

Para recuperar los registros guardados se usa **LOAD** y si se usa JAZZ tendremos que usar **GOSBVL =GETPTR**

D0 apunta a los próximos objetos RPL a ejecutar. Cuando nosotros queremos salir, nosotros tenemos que leer la dirección del próximo objeto RPL a ejecutar en **A**.

El siguiente código es el que generalmente se usa para retornar al RPL:

```
A=DAT0 A
D0=D0+ 5
PC=(A)
```

Pero toda esta rutina se puede reemplazar por un **RPL** o **LOOP**

* Cada Nibble va de 0 a F (15) facilitando asi trabajar com números muy grandes.
* Cuando se diga **D.A** se refiere a **D** campo **A**.
* Usted puede poner tantas instrucciones como quiera en una sola línea.
* Usted puede poner cada instrucción en cada línea.
* Para colocar un comentario se debe anteponer el carácter **%**
* Las etiquetas o subprogramas se declaran anteponiendo el carácter *.
* Cuando se usan instrucciones en Saturn son del formato: Registro=Registro...
  (Donde registro puede ser **A, B, C, D, P, D1** o **D0**). Usted puede omitir Registro=.
  Ejemplo: **A=A+5** es equivalente a **A+5**.
* Usted puede separar el campo de la instrucción por el carácter punto "**.**" o con un espacio.
  Ejemplo:
  A+5.A

17 Parte IV: Lista de instrucciones del Saturno

A continuación se muestra un resumen de todas las instrucciones en ML, mas adelante se vera cada una de ellas.

f es un campo (A, B, M, P, S, W, X, XS, WP)
x es una expresión entre 0 y 15 o 1 y 16 (dependiendo de la instrucción y del flag! 0-15)

?B=A.f	A+B.f	ASRB.f	R0=A.f	RTNSXM
?C=B.f	B+C.f	BSRB.f	R1=A.f	RTN
?A=C.f	C+A.f	CSRB.f	R2=A.f	RTNSC
?C=D.f	D+C.f	DSRB.f	R3=A.f	RTNCC
?B#A.f	A+A.f	PC=A	R4=A.f	SETHEX
?C#B.f	B+B.f	PC=C	R0=C.f	SETDEC
?A#C.f	C+C.f	A=PC	R1=C.f	RSTK=C
?C#D.f	D+D.f	C=PC	R2=C.f	C=RSTK
?A=0.f	B+A.f	APCEX	R3=C.f	CLRST
?B=0.f	C+B.f	CPCEX	R4=C.f	C=ST
?C=0.f	A+C.f	HST=0.x	A=R0.f	ST=C
?D=0.f	C+D.f	XM=0	A=R1.f	CSTEX
?A#0.f	A=0.f	SB=0	A=R2.f	P+1
?B#0.f	B=0.f	SR=0	A=R3.f	P-1
?C#0.f	C=0.f	MP=0	A=R4.f	A&B.f
?D#0.f	D=0.f	CLRHST	C=R0.f	B&C.f
?A>B.f	A=B.f	?HST=0.x	C=R1.f	C&A.f
?B>C.f	B=C.f	?XM=0	C=R2.f	D&C.f
?C>A.f	C=A.f	?SB=0	C=R3.f	B&A.f
?D>C.f	D=C.f	?SR=0	C=R4.f	C&B.f
?A<B.f	B=A.f	?MP=0	AR0EX.f	A&C.f
?B<C.f	C=B.f	ST=0.x	AR1EX.f	C&D.f
?C<A.f	A=C.f	ST=1.x	AR2EX.f	A!B.f
?D<C.f	C=D.f	?ST=0.x	AR3EX.f	B!C.f
?A>=B.f	ABEX.f	?ST=1.x	AR4EX.f	C!A.f
?B>=C.f	BCEX.f	OUT=CS	CR0EX.f	D!C.f
?C>=A.f	ACEX.f	OUT=C	CR1EX.f	B!A.f
?D>=C.f	CDEX.f	A=IN	CR2EX.f	C!B.f
?A<=B.f	A-B.f	C=IN	CR3EX.f	A!C.f
?B<=C.f	B-C.f	UNCNFG	CR4EX.f	C!D.f
?C<=A.f	C-A.f	CONFIG	D0=A	RTI
?D<=C.f	D-C.f	C=ID	D1=A	A+Expresión.f
?A=B.f	B-A.f	SHUTDN	AD0EX	B+Expresión.f
?B=C.f	C-B.f	INTON	AD1EX	C+Expresión.f
?C=A.f	A-C.f	RSI	D0=C	D+Expresión.f
?D=C.f	C-D.f	LA Hexadecimal	D1=C	A-Expresión.f
?A#B.f	A=B-A.f	LA(x) Expresión	CD0EX	B-Expresión.f
?B#C.f	B=C-B.f	LAASC(x) ASCII	CD1EX	C-Expresión.f
?C#A.f	C=A-C.f	BUSCB	D0=AS	D-Expresión.f
?D#C.f	D=C-D.f	ABIT=0.x	D1=AS	GOINC Etiqueta
?B>A.f	ASL.f	ABIT=1.x	AD0XS	COINA Etiqueta

?C>B.f	BSL.f	?ABIT=0.x	AD1XS	G5 Etiqueta
?A>C.f	CSL.f	?ABIT=1.x	D0=CS	G4 Etiqueta
?C>D.f	DSL.f	CBIT=0.x	D1=CS	G3 Etiqueta
?B<A.f	ASR.f	CBIT=1.x	CD0XS	G2 Etiqueta
?C<B.f	BSR.f	?CBIT=0.x	CD1XS	GOIN5 Etiqueta
?A<C.f	CSR.f	?CBIT=1.x	DAT0=A.x	GOIN4 Etiqueta
?C<D.f	DSR.f	PC=(A)	DAT1=A.x	GOIN3 Etiqueta
?B>=A.f	A=-A.f	PC=(C)	A=DAT0.x	GOIN2 Etiqueta
?C>=B.f	B=-B.f	BUSCD	A=DAT1.x	D1+Expresión
?A>=C.f	C=-C.f	INTOFF	DAT0=C.x	D0+Expresión
?C>=D.f	D=-D.f	C+P+1	DAT1=C.x	
?B<=A.f	A=-A-1.f	RESET	C=DAT0.x	
?C<=B.f	B=-B-1.f	BUSCC	C=DAT1.x	
?A<=C.f	C=-C-1.f	C=P.x	D0=Hexadecimal	
?C<=D.f	D=-D-1 .f	P=C.x	D1=Hexadecimal	
GOTOL Etiqueta		SREQ?	D0=(x) Expresión	
GOLONG Etiqueta		CPEX.x	D1=(x) Expresión	
GOVLNG Etiqueta		ASLC	LC Hexadecimal	
GOSUBL Etiqueta		BSLC	LC(x) Expresión	
GOSBVL Etiqueta		CSLC	LCASC(x) Ascii	
		DSLC	RTNC	
		ASRC	GOC Etiqueta	
		BSRC	RTNNC	
		CSRC	GONC Etiqueta	
		DSRC	GOTO Etiqueta	
			GOSUB Etiqueta	

NUEVAS INSTRUCCIONES DE MASD

GOINC Etiqueta		Equivalente a **LC(5)Etiqueta-&**. (**&** es la dirección de la instrucción)
GOINA Etiqueta		Equivalente a **LA(5)Etiqueta-&**. (**&** es la dirección de la instrucción)
$hhhh...hhh o **NIBHEX hhh…hh**		Incluye datos hexadecimales. Ejemplo: **$12ACD545680B**.
$/hhhh...hhh		Incluye datos hexadecimales en orden inverso. Ejemplo: **$/123ABC** es equivalente a **$CBA321**.
$(x)Exp o **CON(x)Exp** o **EXP(x)Exp**		Coloca el valor de **Exp** en el código, en **x** nibbles
¢Ascii		Incluye datos ASCII. El fin del string es el siguiente **¢** o retorna. Ejemplo: **¢**Hola**¢**. Para salir al carácter **¢**, pórgalo dos veces. Para poner un carácter por el numero, use **\xx** donde xx es un numero hex. Para poner \, ponga el carácter dos veces.
INCLUDE Nombre		Nombre es el nombre del archivo que se va a incluir.
GOIN5 etiq **GOIN4 etiq**	**G5 etiq** **G4 etiq**	Igual a **$(x)Etiqueta-&** con x=5, 4, 3 o 2.

GOIN3 etiq **GOIN2 etiq**	**G3 etiq** **G2 etiq**	Util para crear un salto de tabla
SAVE		Equivalente a **GOSBVL SAVPTR**
LOAD		Equivalente a **GOSBVL GETPTR**
RPL o **LOOP**		Equivalente a **A=DAT0.A D0+5 PC=(A)**
LOADRPL		Equivalente a **GOVLNG GETPTRLOOP**
INTOFF2		Equivalente a **GOSBVL DisableIntr**
INTON2		Equivalente a **GOSBVL AllowIntr**
ERROR_C		Equivalente a **GOSBVL ErrjmpC**
A=IN2		Equivalente a **GOSBVL AINRTN**
C=IN2		Equivalente a **GOSBVL CINRTN**
OUT=C=IN		Equivalente a **GOSBVL OUTCINRTN**
RES.STR		Equivalente a **GOSBVL MAKE$N**
RES.ROOM		Equivalente a **GOSBVL GETTEMP**
RESRAM		Equivalente a **GOSBVL MAKERAM$**
SHRINK$		Equivalente a **GOSBVL SHRINK$**
COPY<-, **COPY←**, **COPYDN**		Equivalente a **GOSBVL MOVEDOWN**
COPY->, **COPY→**, **COPYUP**		Equivalente a **GOSBVL MOVEUP**
DISP		Equivalente a **GOSBVL DBUG (solo si debug es encendido)**
DISPKEY		Equivalente a **GOSBVL DBUG.KEY (solo si debug es encendido)**
SRKLST		Equivalente a **GOSBVL SHRINKLIST**
SCREEN		Equivalente a **GOSBVL D0->Row1**
MENU		Equivalente a **GOSBVL D0->Sft1**
ZEROMEM		Equivalente a **GOSBVL WIPEOUT**
MULT.A		Equivalente a **GOSBVL MULTBAC**
MULT		Equivalente a **GOSBVL MPY**
DIV.A		Equivalente a **GOSBVL IntDiv**
DIV		Equivalente a **GOSBVL IDIV**
BEEP		Equivalente a **GOSBVL makebeep**

18 Trabajando instrucciones de los registros

18.1 Cargando un valor en un registro

Es muy fácil cargar un valor dentro de un registro.
Existen dos instrucciones disponibles que son **LA** y **LC**.

LA (Cargar en **A**) se usa para cargar un valor en el registro **A**,
LC (Cargar en **C**) se usa para cargar un valor en el registro **C**,

Ejemplo:
Cargar #ABCDE dentro del registro **C**.

Solución:
LC ABCDE

Después de que el código se corre, el registro de **C** contendrá a #ABCDE.

LC o **LA** solo se pueden cargar de 1 a 16 nibbles dentro de **A** o **C**.

Una cosa importante para considerar al usar **LA** o **LC** es el valor de **P**.
Cuando uno empieza un programa **P** es igual a cero, uno puede cambiar el valor de **P** pero al salir del programa uno debe restaurar **P** colocando su valor a cero.

Por ejemplo Si **P=0**, cuando usted carga algo en **A** o **C** está cargado desde el nibble 0 del registro, ahora si **P=4**, usted empezara a cargar desde el nibble 4 del registro.

Por ejemplo, esto es lo que pasa si nosotros cargamos **ABCDE** en **C** cuando **P=4**:

F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	**E**	**D**	**C**	**B**	**A**	0	0	0	0

Cuando uno carga un valor en un registro, los valores se sobrescriben, pero si nosotros le damos el valor de **E** a **P** y cargamos 6 nibbles en **C**, (**LC ABCDEF**) nosotros conseguimos:

F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
B	**A**	0	0	0	0	0	0	0	0	0	0	**F**	**E**	**D**	**C**

18.1.1 LA, LC

Mnemónico	Hex-form	Campos	Carry	Ciclos
LA n..1	8082 n 1...n	Según el valor de P	No	7.5+1.5n
LC n..1	3 n 1...n	Según el valor de P	No	3+1.5n

1 es el primer nibble cargado y **n** es el último.
LC es más rápido que **LA**, si se necesita velocidad use **LC** en lugar de **LA**.

También se puede usar:
LCASC(x) Caracteres o **LAASC(x) Caracteres**
Esto carga el valor hexadecimal de **x** caracteres en C. **x** debe estar entre 1 y 8.
Ejemplo:
LCASC(8) AREQUIPA

18.1.2 Poniendo a cero un registro

Aquí nosotros vamos aprender a dar el valor de cero a un registro que contenga de 1 a 16 nibbles.
Primero se coloca el registro, **A**, **B**, **C**, o **D**. después (**=**) y **0** .
Ejemplo:
Igualar a cero los 16 nibbles del registro **C**:

Solución:
C=0 W
Nota: recuerde que el campo **W** es igual a 16 nibbles. Primero se coloca a que va ha ser igual un registro y después la cantidad.

Mnemónico	Hex-form	Campos (f)	Carry	Ciclos
A=0 f	D0 Ab0	Solo A P,WP,XS,S,M,B,W	No No	8 4.5+n
B=0 f	D1 Ab1	Solo A P,WP,XS,S,M,B,W	No No	8 4.5+n
C=0 f	D2 Ab2	Solo A P,WP,XS,S,M,B,W	No No	8 4.5+n
D=0 f	D3 Ab3	Solo A P,WP,XS,S,M,B,W	No No	8 4.5+n

18.1.3 Cambiando el valor de un bit

Nosotros podemos cambiar el valor de cualquier bit dentro de los primeros 4 nibbles de los registros **A** o **C**. Nosotros usaremos las instrucciones **ABIT** (para el registro A) y **CBIT** (para el registro C).

Mnemónico	Hex-form	Campos	Carry	Ciclos
ABIT=0 n	8084n	bit número n	No	7.5
ABIT=1 n	8085n	bit número n	No	7.5
CBIT=0 n	8088n	bit número n	No	7.5
CBIT=1 n	8089n	bit número n	No	7.5

Donde **n** es el número de bits puestos a **1** o **0**, y **n** puede tener los valores de **0** a **15** (número de bits)

18.1.4 Intercambiando dos registros

Nosotros podemos intercambiar valores entre dos registros.
También podemos intercambiar el registro entero o sólo un campo de él.

Mnemónico	Hex-form	Campos (f)	Carry	Ciclos
ABEX f	DC AbC	A P,WP,XS,S,M,B,W	No No	8 4.5+n
BCEX f	DD AbD	A P,WP,XS,S,M,B,W	No No	8 4.5+n
ACEX f	DE AbE	A P,WP,XS,S,M,B,W	No No	8 4.5+n
CDEX f	DF AbF	A P,WP,XS,S,M,B,W	No No	8 4.5+n

Todo esto vendría a ser como un SWAP pero entre registros. Nosotros sabemos que no podemos cargar un valor en **D** porque no existe el comando **LD**, para hacer esto nosotros podemos cargar los nibbles en **C** o **A** y entonces intercambiar registros.

Ejemplo:
LC 124
CDEX X

Esto intercambia 3 nibbles entre **C** y **D**, "moviendo" **#124h** en **D** y el contenido de **D** lo mueve a **C**.

18.1.5 Copiando el valor de un registro a otro

No sólo se puede intercambiar dos registros, nosotros también podemos sobrescribir directamente los nibbles de un registro a otro. A continuación las instrucciones para hacer esto:

Mnemónico	Hex-form	Campos (f)	Carry	Ciclos
A=B f	D4 Ab4	A P,WP,XS,S,M,B,W	No No	8 4.5+n
A=C f	D8 Ab8	A P,WP,XS,S,M,B,W	No No	8 4.5+n
B=A f	D5 Ab5	A P,WP,XS,S,M,B,W	No No	8 4.5+n
B=C f	D9 Ab9	A P,WP,XS,S,M,B,W	No No	8 4.5+n
C=A f	DA AbA	A P,WP,XS,S,M,B,W	No No	8 4.5+n
C=B f	D6 Ab6	A P,WP,XS,S,M,B,W	No No	8 4.5+n
C=D f	D7 Ab7	A P,WP,XS,S,M,B,W	No No	8 4.5+n
D=C f	DB AdB	A P,WP,XS,S,M,B,W	No No	8 4.5+n

18.2 Trabajando con registros

18.2.1 Incrementando el valor de un registro

Esto es igual que agregar 1 a un registro. Aquí, el campo especificará cuántos nibbles se usan.
Por ejemplo, si yo tengo **#FFh** en **C** y yo hago **C=C+1 B** yo conseguiré **#00h** en **C** y el carry se encenderá a 1 (porque ocurrió un desbordamiento en la memoria).
Si yo uso el campo **X** yo conseguiré **#100h**.

Mnemónico	Hex-form	Campos (f)	Carry	Ciclos
A=A+1 f	E4 Ba4	A P,WP,XS,S,M,B,W	Si Si	8 4.5+n
B=B+1 f	E5 Ba5	A P,WP,XS,S,M,B,W	Si Si	8 4.5+n
C=C+1 f	E6 Ba6	A P,WP,XS,S,M,B,W	Si Si	8 4.5+n
D=D+1 f	E7 Ba7	A P,WP,XS,S,M,B,W	Si Si	8 4.5+n

18.2.2 Especial con el acarreo en bucles

Cuando usted utiliza el acarreo en bucles, usted tiene que tener en cuenta que el bucle se repetirá hasta que ocurra un desbordamiento o un desbordamiento de capacidad inferior; en el ejemplo, cuando C(A) alcanza cero, repite otra vez, porque todavía no ocurrió acarreo.

```
LC F
*LOOP
C=C-1 P
GONC LOOP
```

El bucle se ejecuta F + 1 veces. Esto es 16 veces, no 15, porque cuando C(P) alcanza cero, repite el bucle, y recién ocurre un desbordamiento (se enciende el carry). Si quisiéramos que un pedazo del código se repita 4 veces, si utilizamos el acarreo (carry) entonces cargaremos 3, no 4, dentro del registro que será utilizado como contador

```
LA 00003
*LOOP
(Inserte el código que se repetirá 4 veces)
A=A-1 A
GONC LOOP
```

En la tabla que se muestra se puede observar cuando se produce el acarreo.

# veces	Contenido de A, después de A-1 A	Carry
1	00002	No
2	00001	No
3	00000	No
4	FFFFF	Si

Como se observa, cargando un valor de **3** en **A**, el bucle se repite 4 veces.

18.2.3 Sumando dos registros

Aquí nosotros sumamos el contenido de un registro a otro

Mnemónico	Hex-form	Campos (f)	Carry	Ciclos
A=A+B f	C0 Aa0	A P,WP,XS,S,M,B,W	Si Si	8 4.5+n
A=A+C f	CA AaA	A P,WP,XS,S,M,B,W	Si Si	8 4.5+n
B=B+A f	C8 Aa8	A P,WP,XS,S,M,B,W	Si Si	8 4.5+n
B=B+C f	C1 Aa1	A P,WP,XS,S,M,B,W	Si Si	8 4.5+n
C=C+B f	C2 Aa2	A P,WP,XS,S,M,B,W	Si Si	8 4.5+n
C=C+D f	CB AaB	A P,WP,XS,S,M,B,W	Si Si	8 4.5+n
D=D+C f	C3 Aa3	A P,WP,XS,S,M,B,W	Si Si	8 4.5+n

18.2.4 Sumando un registro asimismo

Esto se usará para multiplicar el contenido de un registro.

Mnemónico	Hex-form	Campos (f)	Carry	Ciclos
A=A+A f	C4 Aa4	A P,WP,XS,S,M,B,W	Si Si	8 4.5+n
B=B+B f	C5 Aa5	A P,WP,XS,S,M,B,W	Si Si	8 4.5+n
C=C+C f	C6 Aa6	A P,WP,XS,S,M,B,W	Si Si	8 4.5+n
D=D+D f	C7 Aa7	A P,WP,XS,S,M,B,W	Si Si	8 4.5+n

18.2.5 Sumando una constante a un registro

Esto permite agregar una constante al contenido de un registro.

Mnemónico	Hex-form	Campos (f)	Carry	Ciclos
A=A+c f	818F0(c-1) 818a0(c-1)	A W,M,X,B	Si Si	8+n 8+n
B=B+c f	818F1(c-1) 818a1(c-1)	A W,M,X,B	Si Si	8+n 8+n
C=C+c f	818F2(c-1) 818a2(c-1)	A W,M,X,B	Si Si	8+n 8+n
D=D+c f	818F3(c-1) 818a3(c-1)	A W,M,X,B	Si Si	8+n 8+n

2 <c <16

18.2.6 Decrementando un registro

Esto es igual que quitar uno.

Mnemónico	Hex-form	Campos (f)	Carry	Ciclos
A=A-1 f	CC AaC	A P,WP,XS,S,M,B,W	Si Si	8 4.5+n
B=B-1 f	CD AaD	A P,WP,XS,S,M,B,W	Si Si	8 4.5+n
C=C-1 f	CE AaE	A P,WP,XS,S,M,B,W	Si Si	8 4.5+n
D=D-1 f	CF AaF	A P,WP,XS,S,M,B,W	Si Si	8 4.5+n

18.2.7 Decrementando dos registros

Resta un registro de otro.

Mnemónico	Hex-form	Campos (f)	Carry	Ciclos
A=A-B f	E0 Ba0	A P,WP,XS,S,M,B,W	Si Si	8 4.5+n
B=B-C f	E1 Ba1	A P,WP,XS,S,M,B,W	Si Si	8 4.5+n
C=C-A f	E2 Ba2	A P,WP,XS,S,M,B,W	Si Si	8 4.5+n
D=D-C f	E3 Ba3	A P,WP,XS,S,M,B,W	Si Si	8 4.5+n
B=B-A f	E8 Ba8	A P,WP,XS,S,M,B,W	Si Si	8 4.5+n
C=C-B f	E9 Ba9	A P,WP,XS,S,M,B,W	Si Si	8 4.5+n
A=A-C f	EA BaA	A P,WP,XS,S,M,B,W	Si Si	8 4.5+n
C=C-D f	EB BaB	A P,WP,XS,S,M,B,W	Si Si	8 4.5+n
A=B-A f	EC BaC	A P,WP,XS,S,M,B,W	Si Si	8 4.5+n
B=C-B f	ED BaD	A P,WP,XS,S,M,B,W	Si Si	8 4.5+n
C=A-C f	EE BaE	A P,WP,XS,S,M,B,W	Si Si	8 4.5+n
D=C-D f	EF BaF	A P,WP,XS,S,M,B,W	Si Si	8 4.5+n

18.2.8 Decrementando una constante

Mnemónico	Hex-form	Campos (f)	Carry	Ciclos
A=A-C f	818F8(c-1) 818a8(c-1)	A W,M,X,B	Si Si	8+n 8+n
B=B-C f	818F9(c-1) 818a9(c-1)	A W,M,X,B	Si Si	8+n 8+n
C=C-C f	818FA(c-1) 818aA(c-1)	A W,M,X,B	Si Si	8+n 8+n
D=D-C f	818FB(c-1) 818aB(c-1)	A W,M,X,B	Si Si	8+n 8+n

18.2.9 Complemento de 2

Mnemónico	Hex-form	Campos (f)	Carry	Ciclos
A=-A-1 f	FC BbC	A P,WP,XS,S,M,B,W	Si Si	8 4.5+n
B=-B-1 f	FD BbD	A P,WP,XS,S,M,B,W	Si Si	8 4.5+n
C=-C-1 f	FE BbE	A P,WP,XS,S,M,B,W	Si Si	8 4.5+n
D=-D-1 f	FF BbF	A P,WP,XS,S,M,B,W	Si Si	8 4.5+n

18.2.10 Complemento de 1

Todos los bits son invertidos:

Mnemónico	Hex-form	Campos (f)	Carry	Ciclos
A=-A f	F8 Bb8	A P,WP,XS,S,M,B,W	Si Si	8 4.5+n
B=-B f	F9 Bb9	A P,WP,XS,S,M,B,W	Si Si	8 4.5+n
C=-C f	FA BbA	A P,WP,XS,S,M,B,W	Si Si	8 4.5+n
D=-D f	FB BbB	A P,WP,XS,S,M,B,W	Si Si	8 4.5+n

18.2.11 Operaciones lógicas

18.2.11.1 OR lógico

También llamado suma lógica, se representa con el símbolo "**!**".
El resultado se almacena en el registro a la izquierda del signo igual:

Mnemónico	Hex-form	Campos (f)	Carry	Ciclos
A=A!B f	0EF8 0Ea8	A P,WP,XS,S,M,B,W	No No	11 6+n
A=A!C f	0EFE 0EaE	A P,WP,XS,S,M,B,W	No No	11 6+n
B=B!C f	0EF9 0Ea9	A P,WP,XS,S,M,B,W	No No	11 6+n
B=B!A f	0EFC 0EaC	A P,WP,XS,S,M,B,W	No No	11 6+n
C=C!A f	0EFA 0EaA	A P,WP,XS,S,M,B,W	No No	11 6+n
C=C!B f	0EFD 0EaD	A P,WP,XS,S,M,B,W	No No	11 6+n
C=C!D f	0EFF 0EaF	A P,WP,XS,S,M,B,W	No No	11 6+n
D=D!C f	0EFB 0EaB	A P,WP,XS,S,M,B,W	No No	11 6+n

18.2.11.2 Lógico AND

También llamado producto lógico, se representa con el símbolo "**&**".

Mnemónico	Hex-form	Campos (f)	Carry	Ciclos
A=A&B f	0EF0 0Ea0	A P,WP,XS,S,M,B,W	No No	11 6+n
A=A&C f	0EF6 0Ea6	A P,WP,XS,S,M,B,W	No No	11 6+n
B=B&C f	0EF1 0Ea1	A P,WP,XS,S,M,B,W	No No	11 6+n
B=B&A f	0EF4 0Ea4	A P,WP,XS,S,M,B,W	No No	11 6+n
C=C&A f	0EF2 0Ea2	A P,WP,XS,S,M,B,W	No No	11 6+n
C=C&B f	0EF5 0Ea5	A P,WP,XS,S,M,B,W	No No	11 6+n
C=C&D f	0EF7 0Ea7	A P,WP,XS,S,M,B,W	No No	11 6+n
D=D&C f	0EF3 0Ea3	A P,WP,XS,S,M,B,W	No No	11 6+n

18.2.12 Moviendo un nibble a la izquierda o derecha

Hay cuatro instrucciones para cambiar un nibble a la izquierda que multiplica el valor por 16 y otros cuatro para cambiar un nibble a la derecha que divide el valor por 16.

Cuando cambiamos a la izquierda, todos los nibbles se mueven un nibble a la izquierda y el último nibble (el número #Fh) se pierde. El número del nibble #0h recibe un nibble nulo (el valor es #0h).

Cuando cambiamos a la derecha, todos los nibbles se mueven un nibble a la derecha y el último nibble (el número #Fh) recibe un nibble nulo. El primer nibble (el número #0h) se pierde.

Primero escribimos la letra del registro, seguido de **S** (para el Cambio), y finalmente la letra de la dirección, con **R** para la derecha y **L** para la izquierda.
El código mnemotécnico para mover el registro **C** a la derecha es **C + S + R**, (**CSR f**.)

Mnemónico	Hex-form	Campos (f)	Carry	Ciclos
ASL f	F0 Bb0	A P,WP,XS,S,M,B,W	No No	8 4.5+n
BSL f	F1 Bb1	A P,WP,XS,S,M,B,W	No No	8 4.5+n
CSL f	F2 Bb2	A P,WP,XS,S,M,B,W	No No	8 4.5+n
DSL f	F3 Bb3	A P,WP,XS,S,M,B,W	No No	8 4.5+n
ASR f	F4 Bb4	A P,WP,XS,S,M,B,W	No No	8 4.5+n
BSR f	F5 Bb5	A P,WP,XS,S,M,B,W	No No	8 4.5+n
CSR f	F6 Bb6	A P,WP,XS,S,M,B,W	No No	8 4.5+n
DSR f	F7 Bb7	A P,WP,XS,S,M,B,W	No No	8 4.5+n

18.2.13 Rotando un nibble a la izquierda o derecha

Para rotar un nibble, nosotros escribimos primero la letra del registro, después la letra **S** (para el Cambio), después la dirección (**R** o **L**) y finalmente la letra **C** (Rotando).

Mnemónico	Hex-form	Campos	Carry	Ciclos
ASLC	810	Todos	No	22.5
BSLC	811	Todos	No	22.5
CSLC	812	Todos	No	22.5
DSLC	813	Todos	No	22.5
ASRC	814	Todos	No	22.5
BSRC	815	Todos	No	22.5
CSRC	816	Todos	No	22.5
DSRC	817	Todos	No	22.5

Ejemplo de un registro antes de rotar

F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
3	A	8	B	B	3	8	2	0	0	0	A	F	0	F	C

Así quedaría el registro al rotar a la derecha

F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
C	3	A	8	B	B	3	8	2	0	0	0	A	F	0	F

18.2.14 Desplazando un bit a la derecha en un campo

Desplazar un bit a la derecha, es lo mismo que la división por 2.
Este proceso deriva de (Shifting Right Bit); entonces para formar los mnemónicos, se debe colocar la letra del registro "A" o "C", seguido de "S" (desplazar), luego la dirección "R" (derecha), y por último "B" (bit).
En caso de que se desee mover un bit a la derecha del registro A (usando sólo el campo X), el mnemónico sería: ASRB.X

Mnemónico	Campos (f)	Forma-HEX	Carry
ASRB.f	A	819F0	No
	P,WP,XS,X,S,M,B	819a0	No
	W	81C	No
BSRB.f	A	819F1	No
	P,WP,XS,X,S,M,B		No
	W	81D	No
CSRB.f	A	819F2	No
	P,WP,XS,X,S,M,B		No
	W	81E	No
ASRB.f	A	819F2	No
	P,WP,XS,X,S,M,B		No
	W	81F	No

Si usamos el campo W, los 64 bits del registro, serán movidos a la derecha.

Estas instrucciones dividen el valor del campo f, por 2.

Aquí, el acarreo no es modificado, pero si el bit ¿? es igual a 1, entonces SB es encendido.

Esto es como los comandos anteriores, pero aquí nosotros usamos un campo, y el cambio sólo ocurre dentro del campo seleccionado.

Note que no existe ninguna instrucción para cambiar un bit un campo a la izquierda.

19 Guardando los registros

Los registros **A** y **C** son muy importantes para el procesador, estos registros se pueden guardan en los siguientes registros: **R0**, **R1**, **R2**, **R3** y **R4**.
El registro de la izquierda recibe el valor del registro de la derecha

Guardando un registro de trabajo

Mnemónico	Hex-form	Campos	Carry	Ciclos
R0=A	100	Todos	No	20.5
R1=A	101	Todos	No	20.5
R2=A	102	Todos	No	20.5
R3=A	103	Todos	No	20.5
R4=A	104	Todos	No	20.5
R0=C	108	Todos	No	20.5
R1=C	109	Todos	No	20.5
R2=C	10A	Todos	No	20.5
R3=C	10B	Todos	No	20.5
R4=C	10C	Todos	No	20.5

19.1 Guardando un registro de trabajo dentro de un campo

Mnemónico	Hex-form	Campos (f)	Carry	Ciclos
R0=A f	81AF00 81Aa00	A P,WP,XS,S,M,B,W	No No	14 9+n
R1=A f	81AF01 81Aa01	A P,WP,XS,S,M,B,W	No No	14 9+n
R2=A f	81AF02 81Aa02	A P,WP,XS,S,M,B,W	No No	14 9+n
R3=A f	81AF03 81Aa03	A P,WP,XS,S,M,B,W	No No	14 9+n
R4=A f	81AF04 81Aa04	A P,WP,XS,S,M,B,W	No No	14 9+n
R0=C f	81AF08 81Aa08	A P,WP,XS,S,M,B,W	No No	14 9+n
R1=C f	81AF09 81Aa09	A P,WP,XS,S,M,B,W	No No	14 9+n
R2=C f	81AF0A 81Aa0A	A P,WP,XS,S,M,B,W	No No	14 9+n
R3=C f	81AF0B 81Aa0B	A P,WP,XS,S,M,B,W	No No	14 9+n
R4=C f	81AF0C 81Aa0C	A P,WP,XS,S,M,B,W	No No	14 9+n

Ejemplo:
Supongamos que yo quiero usar **D1**, pero necesito guardar su valor:

```
CD1EX        %Cambia D1 al registro C. (SWAP entre D1 y C)
R0=C A       %Guardamos los 5 cuartetos de C que contiene a D1
```

19.2 Copiando a un registro el valor de un registro guardado

Uno no puede trabajar con registros guardados, a continuación las instrucciones para recuperar un registro guardado:

Mnemónico	Hex-form	Campos	Carry	Ciclos
A=R0	110	Todos	No	20.5
A=R1	111	Todos	No	20.5
A=R2	112	Todos	No	20.5
A=R3	113	Todos	No	20.5
A=R4	114	Todos	No	20.5
C=R0	118	Todos	No	20.5
C=R1	119	Todos	No	20.5
C=R2	11ª	Todos	No	20.5
C=R3	11B	Todos	No	20.5
C=R4	11C	Todos	No	20.5

19.3 Copiando a un registro un campo de un registro guardado.

Mnemónico	Hex-form	Campos (f)	Carry	Ciclos
A=R0 f	81AF10 81Aa10	A P,WP,XS,S,M,B,W	No No	14 9+n
A=R1 f	81AF11 81Aa11	A P,WP,XS,S,M,B,W	No No	14 9+n
A=R2 f	81AF12 81Aa12	A P,WP,XS,S,M,B,W	No No	14 9+n
A=R3 f	81AF13 81Aa13	A P,WP,XS,S,M,B,W	No No	14 9+n
A=R4 f	81AF14 81Aa14	A P,WP,XS,S,M,B,W	No No	14 9+n
C=R0 f	81AF18 81Aa18	A P,WP,XS,S,M,B,W	No No	14 9+n
C=R1 f	81AF19 81Aa19	A P,WP,XS,S,M,B,W	No No	14 9+n
C=R2 f	81AF1A 81Aa1A	A P,WP,XS,S,M,B,W	No No	14 9+n
C=R3 f	81AF1B 81Aa1B	A P,WP,XS,S,M,B,W	No No	14 9+n
C=R4 f	81AF1C 81Aa1C	A P,WP,XS,S,M,B,W	No No	14 9+n

19.4 Intercambio entre registros guardados y registros de trabajo

Primero escribimos la letra del registro de trabajo y luego el nombre del registro guardado y **EX** para intercambiar los registros.

Mnemónico	Hex-form	Campos	Carry	Ciclos
AR0EX	120	Todos	No	20.5
AR1EX	121	Todos	No	20.5
AR2EX	122	Todos	No	20.5
AR3EX	123	Todos	No	20.5
AR4EX	124	Todos	No	20.5
CR0EX	128	Todos	No	20.5
CR1EX	129	Todos	No	20.5
CR2EX	12A	Todos	No	20.5
CR3EX	12B	Todos	No	20.5
CR4EX	12C	Todos	No	20.5

19.5 Intercambio entre un registro guardado y un registro de trabajo con un campo

Mnemónico	Hex-form	Campos (f)	Carry	Ciclos
AR0EX f	81AF20 81Aa20	A P,WP,XS,S,M,B,W	No No	14 9+n
AR1EX f	81AF21 81Aa21	A P,WP,XS,S,M,B,W	No No	14 9+n
AR2EX f	81AF22 81Aa22	A P,WP,XS,S,M,B,W	No No	14 9+n
AR3EX f	81AF23 81Aa23	A P,WP,XS,S,M,B,W	No No	14 9+n
AR4EX f	81AF24 81Aa24	A P,WP,XS,S,M,B,W	No No	14 9+n
CR0EX f	81AF28 81Aa28	A P,WP,XS,S,M,B,W	No No	14 9+n
CR1EX f	81AF29 81Aa29	A P,WP,XS,S,M,B,W	No No	14 9+n
CR4EX f	81AF2A 81Aa2A	A P,WP,XS,S,M,B,W	No No	14 9+n
CR3EX f	81AF2B 81Aa2B	A P,WP,XS,S,M,B,W	No No	14 9+n
CR4EX f	81AF2C 81Aa2C	A P,WP,XS,S,M,B,W	No No	14 9+n

20 Punteros

D0 y **D1** pueden contener sólo cinco nibbles.
A y **C** son los únicos registros que pueden usarse con **D0** y **D1**.

20.1 Asignando un valor a D0 o D1

Nosotros podemos cargar 2, 4 o 5 nibbles en **D0** o **D1**. Si nosotros sólo cargamos dos, los otros tres quedan intactos; si nosotros cargamos sólo cuatro, el último queda intacto.
Nosotros simplemente escribiremos:
D0 =…
D1 =…
Quizás puede parecer extraño para cargar sólo dos o cuatro nibbles. De hecho, a veces usted cargará un valor de cinco nibbles, y entonces sólo cambia lo que necesita ser cambiado.
Todo lo que se tiene que escribir es **D0** =… (Con 2,4 o 5 nibbles).
También se puede hacer lo mismo con **D1**.
El espacio después del signo = es importante.
Las formas hexadecimales dependen del número de nibbles que nosotros carguemos:

Mnemónico	Hex-form	Campos	Carry	Ciclos
D0= yz	19zy	Primeros 2 nibbles	No	6
D0= wxyz	1Azyxw	Primeros 4 nibbles	No	9
D0= vwxyz	1Bzyxwv	Primeros 5 nibbles	No	10.5
D1= yz	1Dzy	Primeros 2 nibbles	No	6
D1= wxyz	1Ezyxw	Primeros 4 nibbles	No	9
D1= vwxyz	1Fzyxwv	Primeros 5 nibbles	No	10.5

Cuando teclea **D1 = 00120**, el objeto se codificara como "**1F02100**" para que se inviertan los nibbles a ser cargados en **D1**.

20.2 Agregando a o substrayendo de D0 o D1

Nosotros no siempre necesitamos cargar un valor en **D0** o **D1**, nosotros podemos simplemente mover el indicador. ¿Recuerda este pedazo de código?

```
A=DAT0 A
D0=D0+ 5
PC=(A)
```

La segunda línea incrementa **D0**.
Nota: No se olvide del espacio después del signo +, como **D0=D0+ 5** es en lugar de **D0=D0+5**. Si usted se olvida del espacio, al momento de ensamblar su código informará un error.
Usted puede agregar o puede quitar de **1** a **16** a **D0** o **D1**; si usted quiere quitar más, usted tendrá que repetir estas instrucciones tantas veces como sea necesario.

Mnemónico	Hex-form	Campos (f)	Carry	Ciclos
D0=D0+ n	16(n-1)	El registro entero	Si	8.5
D0=D0- n	18(n-1)	El registro entero	Si	8.5
D1=D1+ n	17(n-1)	El registro entero	Si	8.5
D1=D1- n	1C(n-1)	El registro entero	Si	8.5

20.3 Copiando A (o C) a D0 (o D1)

El registro **A** es útil, tiene solo 5 nibbles, como **D0** y **D1**.
Nosotros podemos hacer que **D0** o **D1** apunte a los 5 nibbles dentro de **A** o **C**:

Mnemónico	Hex-form	Campos (f)	Carry	Ciclos
D0=A	130	Campo A de D0	No	9.5
D0=C	134	Campo A de D0	No	9.5
D1=A	131	Campo A de D0	No	9.5
D1=C	135	Campo A de D0	No	9.5

También es posible copiar sólo cuatro nibbles, dejando el quinto intacto.
Los mnemónicos aquí son similares, con la única diferencia que "**S**" es agregada al extremo.

Mnemónico	Hex-form	Campos	Carry	Ciclos
D0=AS	138	Solo 4 nibbles	No	8.5
D0=CS	13C	Solo 4 nibbles	No	8.5
D1=AS	139	Solo 4 nibbles	No	8.5
D1=CS	13D	Solo 4 nibbles	No	8.5

20.4 Intercambio entre el campo A de A/A con D0/D1

Esto es muy útil si uno quiere verificar el prologo de un objeto.

Mnemónico	Hex-form	Campos	Carry	Ciclos
AD0EX	132	5 nibbles	No	9.5
AD1EX	133	5 nibbles	No	9.5
CD0EX	136	5 nibbles	No	9.5
CD1EX	137	5 nibbles	No	9.5

Cada objeto empieza con un prólogo que consta de cinco nibbles.
La pila no contiene los objetos, sino a las direcciones de los objetos.

Si nosotros queremos hacer un **DROP**, nosotros podemos mover **D1** al nivel 2 y liberar los 5 nibbles de memoria, ya que estos ya no se usan

20.5 Intercambiando 4 nibbles LSB A de A/C con D0/D1

Estas instrucciones cambian los 4 primeros nibbles de los registros **A** o **C** con **D0** o **D1**.

Mnemónico	Hex-form	Campos	Carry	Ciclos
AD0XS	13A	Solo 4 nibbles	No	8.5
AD1XS	13B	Solo 4 nibbles	No	8.5
CD0XS	13E	Solo 4 nibbles	No	8.5

21 Leyendo memoria dentro de un campo

Nosotros podemos modificar **D0** o **D1** para que apunte a dónde nosotros queramos en la memoria.
Nosotros también podemos cargar los valores en **D0** o **D1** (usando D0/D1 =...).
Aquí, sólo se usara los registros **A** y **C**, nosotros también podemos copiar instrucciones de **DAT0** y **DAT1**.

Mnemónico	Hex-form	Campos (f)	Carry	Ciclos
A=DAT0 A	142	A	No	23.5,3.5
A=DAT0 B	14A	B	No	19.5
A=DAT0 f	152a	P,WP,X,XS,S,W,M,B	No	20+n,1+n/2
A=DAT1 A	143	A	No	23.5,3.5
A=DAT1 B	14B	B	No	19.5
A=DAT1 f	153a	P,WP,X,XS,S,W,M,B	No	20+n,1+n/2
C=DAT0 A	146	A	No	23.5,3.5
C=DAT0 B	14E	B	No	19.5
C=DAT0 f	156a	P,WP,X,XS,S,W,M,B	No	20+n,1+n/2
C=DAT1 A	147	A	No	23.5,3.5
C=DAT1 B	14F	B	No	19.5
C=DAT1 f	157a	P,WP,X,XS,S,W,M,B	No	20+n,1+n/2

El registro que recibe los nibbles se localiza a la izquierda del signo igual (=).
Ejemplo:
Si yo quiero leer 5 nibbles de #00100h, yo hago:
D0 = 00100
C=DAT0 A
D0 contiene #00100h, pero **C** contendrá los cinco nibbles que están en #00100h en la memoria.

22 Leyendo de la memoria un valor

Nosotros también podemos usar un número en lugar de un campo, para que en lugar de:
C=DAT0 A
Uno puede usar:
C=DAT0 5

¿Cuál es la diferencia? En el objeto del código no se codifica de la misma manera.
Además, **C=DAT0 A** necesita 23.5 ciclos, y **C=DAT0 5** necesita 24 ciclos, como usted ve mejor es usar **A**, pero usar **C=DAT0 16** es mejor que usar **C=DAT0 W**, como **C=DAT0 16** necesita 35 ciclos, y **C=DAT0 W** necesita 36 ciclos. Esto es sólo una diferencia de un ciclo, pero cuando se usan loops por ejemplo, 100 veces, hay 100 ciclos menos usados.

Cuando uno necesita una rutina rápida empezara a elegir que códigos va ha usar.

Mnemónico	Hex-form	Campos	Carry	Ciclos
A=DAT0 n	15A(n-1)	n nibbles	No	19+n
A=DAT1 n	15B(n-1)	n nibbles	No	19+n
C=DAT0 n	15E(n-1)	n nibbles	No	19+n
C=DAT1 n	15F(n-1)	n nibbles	No	19+n

23 Escribiendo en la memoria un campo

Aquí nosotros copiamos los nibbles de un registro de trabajo (**A** o **C**) a la memoria:

Mnemónico	Hex-form	Campos (f)	Carry	Ciclos
DAT0=A A	140	A	No	19.5
DAT0=A B	148	B	No	16.5
DAT0=A f	150a	P,WP,X,XS,S,W,M,B	No	19+n
DAT1=A A	141	A	No	19.5
DAT1=A B	149	B	No	16.5
DAT1=A f	151a	P,WP,X,XS,S,W,M,B	No	19+n
DAT0=C A	144	A	No	19.5
DAT0=C B	14C	B	No	16.5
DAT0=C f	154a	P,WP,X,XS,S,W,M,B	No	19+n
DAT1=A A	145	A	No	19.5
DAT1=A B	14D	B	No	16.5
DAT1=A f	155a	P,WP,X,XS,S,W,M,B	No	19+n

Como usted puede ver aquí, los campos **A** y **B** tienen instrucciones específicas:

A es útil para las direcciones.
B con dos nibbles, normalmente se usa para los bytes.

24 Escribiendo la memoria con un valor

Mnemónico	Hex-form	Campos	Carry	Ciclos
DAT0=A n	158(n-1)	n nibbles	No	18+n
DAT1=A n	159(n-1)	n nibbles	No	18+n
DAT0=C n	15C(n-1)	n nibbles	No	18+n
DAT1=A n	15D(n-1)	n nibbles	No	18+n

Suponga que usted quiere escribir un nibble:
DAT0=A P

Si P no es igual a cero (porque usted lo usa para algo más), usted puede escribir:
DAT0=A 1

Esto escribirá un nibble y pondrá "1590" dentro del objeto del código (159 y 1-1 = 1590)

25 Saltos y pruebas

Cada instrucción se localiza en alguna dirección en memoria, nosotros podemos hacer saltos dentro de nuestros programas. Normalmente, las instrucciones de un programa se ejecutan una después de otra. Pero a veces, después de una prueba o simplemente porque nosotros queremos el código podrá continuar en alguna otra parte si nosotros queremos saltar a alguna otra parte.

El registro **PC** se usa aquí. Como usted sabe este registro contiene la próxima instrucción a ser ejecutada, si nosotros modificamos **PC**, nosotros podemos cambiar el flujo de la ejecución del programa.
Nosotros discutiremos tres tipos de saltos:

SALTO RELATIVO.
Esto salta +n o -n nibbles de nuestra posición actual. Nosotros realmente no sabemos precisamente a que lugar de la memoria vamos a saltar, pero nosotros sabemos cuántos nibbles antes o después de que será.
Para esto usaremos el desplazamiento, este puede ser positivo o negativo.
Así, nosotros podemos saltar hacia atrás o adelante.
Por ejemplo, si nosotros queremos saltar 100 nibbles, el desplazamiento es +100.

SALTO ABSOLUTO
Esto salta a una dirección específica. Cuando una dirección es un número de 5nibbles, como #ABCDE, y se usa un salto absoluto este salta para dirigirse a ABCDE y continua con la ejecución.

SALTO INDIRECTO
Aquí uno no está saltando a una dirección, pero si leyendo la dirección a la que nosotros saltaremos.
Por ejemplo:
Si yo hago un salto absoluto a ABCDE, yo iré directamente a ABCDE en la memoria y continuaré la ejecución.
Si yo hago un salto indirecto, yo leeré 5 nibbles a #ABCDE y entonces saltaré a la dirección encontrada en esos 5 nibbles.
Cuando usted realiza saltos en su código usando las Etiquetas, el ensamblador generará los saltos automáticamente para usted.
Por ejemplo, suponga que yo tengo el siguiente salto en mi código:

```
?C=0 A
GOYES SALIR
...
*SALIR
```

El ensamblador generará el código y entonces calcula el desplazamiento para que **GOYES** salte al código que se encuentre después de la Etiqueta ***SALIR**.

A veces el salto no podrá realizarse por que el desplazamiento es demasiado grande, en esos casos usted tendrá que escribir algo diferente para trabajar con su código.
Esto se explica mas adelante.

25.1 Saltos condicionales: GOC y GONC

Un salto condicional no es un salto automático: el salto ocurrirá según el valor de algo más: aquí, Cuando se usa el carry existen dos instrucciones para saltar según el estado del carry: **GOC** y **GONC**

Mnemónico	Hex-form	Cuando?	Ciclos
GONC Etiqueta	5yz	Si el carry esta apagado	4.5 o 12.5
GOC Etiqueta	4yz	Si el carry esta encendido	4.5 o 12.5

Hay dos valores del ciclo:
Se necesitan 3 Ciclos si el salto no es hecho, y se necesitan 10 Ciclos si el salto es hecho.
Como usted puede ver aquí, se usan dos nibbles, **yz**, para codificar el salto relativo.
Esto significa que nosotros tenemos 8 bits para codificar el desplazamiento.
GONC y **GOC** pueden saltar al revés 128 nibbles, o 127 adelante.
Si la Etiqueta es de más de 127 nibbles adelante o 128 nibbles atrás, el ensamblador le dirá que el salto es demasiado largo.
Así, esto:

```
GOC siCarry       %Si no se enciende el carry aquí entonces vuelve
GONC noCarry
GOTO siCarry
*noCarry          %Nosotros continuamos aquí si no se enciende el carry
```

GOTO acostumbra a usar un nibble mas para codificar los saltos para un total de 3 nibbles.
Si el ensamblador todavía dice que el salto es demasiado largo, nosotros usaremos **GOVLNG** que puede saltar a cualquier punto en la memoria para codificar la dirección del salto con 5 nibbles, recuerde que todas las direcciones usan un máximo de 5 nibbles.

Recuerde que **GOC** y **GONC** sólo pueden saltar 128 nibbles atrás y 127 nibbles después de la posición presente.
Si usted necesita un salto más grande, usted debe cambiar el test, usando **GOTO**, **GOLONG** o **GOVLNG** (GO Very LonG) instrucciones que se describen en la próxima sección.

25.2 Saltos incondicionales: GOTO, GOLONG, GOVLNG

Estos saltos incondicionales se harán sin ninguna comprobación.
Una diferencia entre **GOTO**, **GOLONG** y **GOVLNG** es el número de nibbles que necesitan para codificar los saltos: 3, 4 o 5.

25.2.1 GOTO

Mnemónico	Hex-form	Cuando?	Ciclos
GOTO Etiqueta	6xyz	Siempre	14

GOTO hará que el programa continué la ejecución desde de la ETIQUETA dentro del código fuente. Se necesitan de 14 ciclos. Porque se usan 3 nibbles para codificar los saltos, puede saltar 2048 nibbles antes de la posición del **GOTO** o 2047 nibbles después de la posición del **GOTO**.

25.2.2 GOLONG

Mnemónico	Hex-form	Cuando?	Ciclos
GOLONG Etiqueta	8Cwxyz	Siempre	17

Éste es un salto más grande, se usan 4 nibbles para codificar los saltos, para que nosotros podemos saltar 32768 nibbles antes de la posición del **GOLONG** o 32767 nibbles después.

Cuando usted codifique, primero pruebe con **GOTO,** si al ensamblar dice que el salto es demasiado largo, entonces pruebe con **GOLONG**, si todavía dice que el salto es demasiado largo, usted tendrá que usar **GOVLNG.**

25.2.3 GOVLNG

Mnemónico	Hex-form	Cuando?	Ciclos
GOVLNG Etiqueta	8Dvwxyz	Siempre	18.5

Aquí, como todas las direcciones se codifican usando 5 nibbles, nosotros hacemos un salto directo a cualquier punto de la memoria. En este caso no es un desplazamiento puesto en código, sino una dirección. Veamos una aplicación `GOVLNG`:

```
*ini
GOSUB bm
?A=0 W  ➔ ini
RPL
*bm
...
GOVLNG =ADDRESS
```

Cuando ejecutamos un GOSUB, debe encontrarse un RTN para ReTorNar, sin embargo, esto es innecesario si se encuentra un GOVLNG =??... Lo que quiere decir que sería lo mismo colocar en bm:

```
...
GOSBVL =ADDRESS
RTN
...
```

25.2.4 Salto incondicional a la dirección en A o C

Se usan dos instrucciones: **PC=A** y **PC=C**. Nosotros modificamos el registro de **PC** directamente y cargamos **A** o **C** dentro de él.
La próxima instrucción se sacará de la dirección en **A** o **C**.

Mnemónico	Hex-form	Cuando?	Ciclos
PC=A	81B2	Campo A del registro A	26,3.5
PC=C	81B3	Campo A del registro C	26,3.5

5 nibbles se mueven de **A** o **C** directamente a **PC**.

25.2.5 Salto incondicional con intercambio

Aquí, en lugar de copiar el campo **A** de **A** o **C** a **PC**, nosotros intercambiamos los valores.
PC hace el flujo de las instrucciones, continua a la dirección en el campo **A** de **A** o **C**, y el valor anterior de **PC** se guarda dentro de **A** o **C**, porque es un intercambio.

Instrucciones para mover desde **A** o **C** directamente a **PC**.

Mnemónico	Hex-form	Cuando?	Ciclos
APCEX	81B6	Campo A del registro A	19
CPCEX	81B7	Campo A del registro C	19

25.3 Salto indirecto

Ahora para el salto indirecto nosotros primeros leemos 5 nibbles de la dirección de A o C y entonces los cargamos en PC, así saltamos ahí.

Mnemónico	Hex-form	Cuando?	Ciclos
PC=(A)	808C	Campo A del registro A	26,3.5
PC=(C)	808E	Campo A del registro C	26,3.5

Por ejemplo si dentro de la dirección **#ABCDE** de **A** o **C** hay 5 nibbles **#FGHIJ**, entonces la ejecución continúa a **#FGHIJ**

25.4 Guardando el contenido de PC

Estas dos instrucciones no se usan para hacer los saltos, pero son útiles para guardar el valor de PC antes de un salto.

Mnemónico	Hex-form	Cuando?	Ciclos
A=PC	81B4	Campo A del registro A	11
C=PC	84B5	Campo A del registro C	11

26 Llamando un subprograma

Cuando uno crea un subprograma, normalmente es para ahorrar espacio dentro de el código, esto cuando algúna parte del código se repite mucho. Si usted quiere hacer esto en ML simplemente coloque una etiqueta, seguida por el código que tendrá dicho subprograma.
Cuando usted quiera, usted puede volver atrás dónde llamó el subprograma y puede continuar con el funcionamiento de su programa.

Al saltar a un subprograma, el procesador carga la dirección de la instrucción que sigue la instrucción del salto en **RSTK** (Área de Pila de Retorno). Entonces, la ejecución continúa a dónde usted saltó, hasta que una instrucción de retorno se encuentre.
Cuando la instrucción de retorno se encuentra, está se carga en **PC** y el programa del subprograma y la ejecución continúa.
La diferencia aquí entre un salto usual es que una dirección debe grabarse para que nosotros podemos regresar.

RSTK consta de ocho niveles, cada nivel de 5 nibbles, pero pueden usarse sólo cinco niveles. Esto significa que usted puede tener solo cinco niveles de subprogramas, u ocho si usted no permite ninguna interrupción.

26.1 GOSUB

Aquí se usan tres nibbles para codificar el desplazamiento del salto, para que nosotros podamos saltar a un subprograma 2048 nibbles antes o 2047 nibbles después de la posición de GOSUB.

Mnemónico	Hex-form	Cuando?	Ciclos
GOSUB Etiqueta	7xyz	El nivel 1 de RSTK se uso	15

26.2 GOSUBL

Esto simboliza GOSUB Long. Aquí se usa 4 nibbles para codificar el salto, para que nosotros podamos saltar a un subprograma 32768 nibbles antes o 32767 nibbles después de GOSUBL.

Mnemónico	Hex-form	Cuando?	Ciclos
GOSUBL Etiqueta	8Ewxyz	El nivel 1 de RSTK se uso	18

26.3 GOSBVL

Aquí se usan 5 nibbles para codificar la dirección.

Mnemónico	Hex-form	Cuando?	Ciclos
GOSBVL Etiqueta	8Fvwxyz	El nivel 1 de RSTK se uso	19.5

26.4 Volviendo de un subprograma

Hay varias instrucciones para hacer esto:

Mnemónico	Hex-form	Cómo?	Ciclos
RTN	01	Retorno usual	11
RTNSC	02	Enciende el carry bit	11
RTNCC	03	Apaga el carry bit	11
RTI	0F	Permite las interrupciones	11
RTNSXM	00	Enciende el bit XM	11

RTN simplemente remueve la dirección de retorno del RSTK y salta después de la instrucción que llamó al subprograma.
RTNSC y **RTNCC** volverán y encenderán o apagaran el carry bit, para que pueda usarse para llamar a un subprograma que usará el carry bit para darle información según el estado del carry bit
RTI permitirá las interrupciones de nuevo.
RTNSXM volverá y encenderá el bit XM

26.5 Volviendo según el valor del CARRY BIT

Usted puede escoger volver dependiendo del valor del carry bit. Las instrucciones son:

Mnemónico	Hex-form	Cuando?	Ciclos
RTNC	400	Si el carry se enciende	4.5 o 12.5
RTNNC	500	Si el carry se apaga	4.5 o 12.5

RTNC sólo volverá si el carry esta encendido. Si está apagado la comprobación usa 4.5 ciclos; si esta encendido, para retornar necesita 12.5 ciclos.
RTNNC sólo volverá si el carry esta apagado.

27 Pruebas (Tests)

También se pueden comparar registros o comparar que un registro sea igual a cero.
Aquí usted verá dos valores por los ciclos necesitados. El primero son los ciclos usados para la prueba para completar si no ocurre ningún salto y el segundo es el número de ciclos usados si la prueba se completa y si el salto es realizado.

Cada prueba empezará con un signo de interrogación (?), después viene un registro de trabajo (**A** o **C**), y después el operador (=, >, <, etc.).
En la línea que sigue una prueba, debe haber un orden de acción. Puede ser un salto, usando **GOYES**, pero también puede ser un retorno de un subprograma si nosotros usamos **RTNSI**.

27.1 Comprobando si un registro es igual a cero

Aquí nosotros verificamos si un registro de trabajo es igual a cero.
El símbolo para verificar la igualdad es =.

Mnemónico	Hex-form	Campos (f)	Carry	Ciclos
?A=0 A	8A8 yz	A	Si	13.5/21.5
?A=0 f	9a8 yz	P,WP,X,XS,S,W,M,B	Si	8.5+n/16.5+n
?B=0 A	8A9 yz	A	Si	13.5/21.5
?B=0 f	9a9 yz	P,WP,X,XS,S,W,M,B	Si	8.5+n/16.5+n
?C=0 A	8AA yz	A	Si	13.5/21.5
?C=0 f	9aA yz	P,WP,X,XS,S,W,M,B	Si	8.5+n/16.5+n
?D=0 A	8AB yz	A	Si	13.5/21.5
?D=0 f	9aB yz	P,WP,X,XS,S,W,M,B	Si	8.5+n/16.5+n

Si una prueba es verdad (TRUE), entonces el carry bit se encenderá.
Estas instrucciones necesitan 8.5+n ciclos (n = el número de nibbles del campo que se usó para la comparación), si el resultado de la prueba es falso y 16.5+n si el resultado de la prueba es verdadero y el salto se realiza.

Usted puede usar "**GOYES Etiqueta**" o "**RTNSI**" después de las instrucciones.
GOYES saltara a la Etiqueta y ejecutara su código y **RTNSI** volverá de un subprograma.

COMPROBANDO SI UN REGISTRO ES DIFERENTE DE CERO

Mnemónico	Hex-form	Campos (f)	Carry	Ciclos
?A#0 A	8AC yz	A	Si	13.5/21.5
?A#0 f	9aC yz	P,WP,X,XS,S,W,M,B	Si	8.5+n/16.5+n
?B#0 A	8AD yz	A	Si	13.5/21.5
?B#0 f	9aD yz	P,WP,X,XS,S,W,M,B	Si	8.5+n/16.5+n
?C#0 A	8AE yz	A	Si	13.5/21.5
?C#0 f	9aE yz	P,WP,X,XS,S,W,M,B	Si	8.5+n/16.5+n
?D#0 A	8AF yz	A	Si	13.5/21.5
?D#0 f	9aF yz	P,WP,X,XS,S,W,M,B	Si	8.5+n/16.5+n

El carry bit se encenderá si la prueba resulta verdadera.

27.2 Comprobando igualdad entre dos registros

Mnemónico	Hex-form	Campos (f)	Carry	Ciclos
?A=B A	8A0 yz	A	Si	13.5/21.5
?A=B f	9a0 yz	P,WP,X,XS,S,W,M,B	Si	8.5+n/16.5+n
?B=C A	8A1 yz	A	Si	13.5/21.5
?B=C f	9a1 yz	P,WP,X,XS,S,W,M,B	Si	8.5+n/16.5+n
?A=C A	8A2 yz	A	Si	13.5/21.5
?A=C f	9a2 yz	P,WP,X,XS,S,W,M,B	Si	8.5+n/16.5+n
?C=D A	8A3 yz	A	Si	13.5/21.5
?C=D f	9a3 yz	P,WP,X,XS,S,W,M,B	Si	8.5+n/16.5+n

27.3 Comprobando desigualdad entre dos registros

Mnemónico	Hex-form	Campos (f)	Carry	Ciclos
?A#B A	8A4 yz	A	Si	13.5/21.5
?A#B f	9a4 yz	P,WP,X,XS,S,W,M,B	Si	8.5+n/16.5+n
?B#C A	8A5 yz	A	Si	13.5/21.5
?B#C f	9a5 yz	P,WP,X,XS,S,W,M,B	Si	8.5+n/16.5+n
?A#C A	8A6 yz	A	Si	13.5/21.5
?A#C f	9a6 yz	P,WP,X,XS,S,W,M,B	Si	8.5+n/16.5+n
?C#D A	8A7 yz	A	Si	13.5/21.5
?C#D f	9a7 yz	P,WP,X,XS,S,W,M,B	Si	8.5+n/16.5+n

27.4 Comprobando si registro es menor que (<) o mayor que (>)

Mnemónico	Hex-form	Campos (f)	Carry	Ciclos
?A>B A	8B0 yz	A	Si	13.5/21.5
?A>B f	9b0 yz	P,WP,X,XS,S,W,M,B	Si	8.5+n/16.5+n
?B>C A	8B1 yz	A	Si	13.5/21.5
?B>C f	9b1 yz	P,WP,X,XS,S,W,M,B	Si	8.5+n/16.5+n
?C>A A	8B2 yz	A	Si	13.5/21.5
?C>A f	9b2 yz	P,WP,X,XS,S,W,M,B	Si	8.5+n/16.5+n
?D>C A	8B3 yz	A	Si	13.5/21.5
?D>C f	9b3 yz	P,WP,X,XS,S,W,M,B	Si	8.5+n/16.5+n
?A<B A	8B4 yz	A	Si	13.5/21.5
?A<B f	9b4 yz	P,WP,X,XS,S,W,M,B	Si	8.5+n/16.5+n
?B<C A	8B5 yz	A	Si	13.5/21.5
?B<C f	9b5 yz	P,WP,X,XS,S,W,M,B	Si	8.5+n/16.5+n
?C<A A	8B6 yz	A	Si	13.5/21.5
?C<A f	9b6 yz	P,WP,X,XS,S,W,M,B	Si	8.5+n/16.5+n
?D<C A	8B7 yz	A	Si	13.5/21.5
?D<C f	9b7 yz	P,WP,X,XS,S,W,M,B	Si	8.5+n/16.5+n

27.5 Comprobando si registro es menor o igual o mayor o igual

Mnemónico	Hex-form	Campos	Carry	Ciclos
?A>=B A	8B8 yz	A	Si	13.5/21.5
?A>=B f	9b8 yz	P,WP,X,XS,S,W,M,B	Si	8.5+n/16.5+n
?B>=C A	8B9 yz	A	Si	13.5/21.5
?B>=C f	9b9 yz	P,WP,X,XS,S,W,M,B	Si	8.5+n/16.5+n
?C>=A A	8BA yz	A	Si	13.5/21.5
?C>=A f	9bA yz	P,WP,X,XS,S,W,M,B	Si	8.5+n/16.5+n
?D>=C A	8BB yz	A	Si	13.5/21.5
?D>=C f	9bB yz	P,WP,X,XS,S,W,M,B	Si	8.5+n/16.5+n
?A<=B A	8BC yz	A	Si	13.5/21.5
?A<=B f	9bC yz	P,WP,X,XS,S,W,M,B	Si	8.5+n/16.5+n
?B<=C A	8BD yz	A	Si	13.5/21.5
?B<=C f	9bD yz	P,WP,X,XS,S,W,M,B	Si	8.5+n/16.5+n
?C<=A A	8BE yz	A	Si	13.5/21.5
?C<=A f	9bE yz	P,WP,X,XS,S,W,M,B	Si	8.5+n/16.5+n
?D<=C A	8BF yz	A	Si	13.5/21.5
?D<=C f	9bF yz	P,WP,X,XS,S,W,M,B	Si	8.5+n/16.5+n

Como de costumbre, después de cada una de estas instrucciones se puede poner un **GOYES** o un **RTNSI**. El salto se realiza si la prueba es verdadera.

27.6 Probando un bit

Tenemos instrucciones que pueden probar el valor de un bit de **C** o **A**.
Sólo los primeros 16 bits, o 4 nibbles de los registros de trabajo puede probarse.
Se necesitan 12 ciclos para terminar la prueba y continuar si es falso y se necesitan 21 ciclos si es verdad y se realiza un salto.

Mnemónico	Hex-form	Campos	Carry	Ciclos
?ABIT=0 n	8086n yz	Bit número n	Si	12.5/20.5
?ABIT=1 n	8087n yz	Bit número n	Si	12.5/20.5
?CBIT=0 n	808An yz	Bit número n	Si	12.5/20.5
?CBIT=1 n	808Bn yz	Bit número n	Si	12.5/20.5

27.7 El registro P

El registro de **P** puede contener sólo un nibble. Cuando un programa empieza, su valor es cero. **P** no sólo define cómo trabajan las instrucciones **LC** y **LA**, también define el ancho del registro **WP** (del nibble 0 hasta el nibble P).

A menudo el registro **P** se usa como un contador de vueltas, también existen instrucciones para incrementar o decrementar su valor.
Usted también puede probar el registro **P** y otras cosas.

Mnemónico	Hex-form	Campos	Carry	Ciclos
P= n	2n	El nibble de P	No	3
P=P+1	0C	El nibble de P	Si	4
P=P-1	0D	El nibble de P	Si	4
?P# n	88n yz	El nibble de P	Si	7.5 o 15.5
?P= n	89n yz	El nibble de P	Si	7.5 o 15.5
C=P n	80Cn	El nibble de P	No	8
P=C n	80Dn	El nibble de P	No	8
C+P+1	809	El nibble de P	Si	9.5
CPEX n	80Fn	El nibble de P	No	8

27.8 P = n

No se olvide del espacio después del signo "=" **n** es el valor de **P**.

27.9 P=P+1

Esto incrementa el valor de **P** en **1**. No importa si el procesador Saturn está trabajando en hexadecimal o decimal, el incrementando se hace bajo el modo hexadecimal.

27.10 P=P-1

Decrementa el valor de **P** en **1**. No importa si el procesador Saturn está trabajando en hexadecimal o decimal, el incrementando se hace bajo el modo hexadecimal.

27.11 ?P #n

Esta prueba verifica si **P** es diferente de **n**. El carry se enciende si el test es verdadero.

27.12 ?P = n

Verifica si **P** es igual a **n**. El carry se enciende si el test es verdadero.

27.13 C=P n

El registro de trabajo **C** va ha ser igual al valor del número de **n** nibbles del registro **P**.
n está entre 0 y 15.

27.14 P=C n

Da a **P** el valor del número de **n** nibbles del registro de trabajo **C**.

27.15 C+P+1

Agrega el valor de **C(A)**, **P** y **1** al campo **A** de **C**. Usted verá que puede ser muy útil.

27.16 CPEX n

Esta instrucción intercambia el número de **n** nibbles del registro de trabajo **C** con el registro **P**.

28 LA PILA RSTK

RSTK es la pila de retornos dónde se guardan las direcciones de retorno mientras los subprogramas están corriendo.
También se puede usar **RSTK** para guardar el registro de trabajo **C** durante algún tiempo.

Mnemónico	Hex-form	Campos	Carry	Ciclos
RSTK=C	06	Campo A de C	No	9
C=RSTK	07	Campo A de C	No	9

29 Registros IN y OUT

Se usara mucho el campo **X** para el registro **OUT**, debido a que es de 2 nibbles.
El registro **IN** es de 4.

Mnemónico	Hex-form	Campos	Carry	Ciclos
OUT=CS	800	Nibble #0h de C	No	5.5
OUT=C	801	Campo X de C	No	7.5
A=IN	802	4 nibbles de A	No	8.5
C=IN	803	4 nibbles de A	No	8.5

Un BUG

Usted no puede usar las instrucciones **A=IN** o **C=IN** a menos que estén localizadas en una dirección igual, por que de lo contrario no trabajaran.
Así que, en lugar de escribir **A=IN** o **C=IN** en el código fuente, nosotros usaremos siempre los subprogramas en ROM que harán **A=IN** o **C=IN** y entonces devolverá a nuestro programa.
Nosotros usaremos:
GOSBVL 01160 en lugar de **C=IN**
GOSBVL 0115A en lugar de **A=IN**
Esas rutinas contienen:

```
C=IN    A=IN
RTN     RTN
```

30 Bits de Estado (ST)

Nosotros tenemos muchos flags que usaremos en nuestros programas. Hay cuatro nibbles de **ST** (4 * 4 = 16 flags); pero algunos ya se usan por el Saturn. Hasta que usted no aprenda a usarlos bien, nosotros evitaremos usar los bits 12, 13 y 14 (usualmente se utiliza el bit 15 para habilitar o desactivar las interrupciones),

Mnemónico	Hex-form	Campos	Carry	Ciclos
CLRST	08	Primeros 3 nibbles de ST	No	7
C=ST	09	Campo X de C	No	7
ST=C	0A	Campo X de C	No	7
CSTEX	0B	Campo X de C	No	7
ST=0 n	84n	Bit n de ST	No	5.5
ST=1 n	85n	Bit n de ST	No	5.5
?ST=0 n	86n yz	Bit n de ST	Si	8.5 o 16.5
?ST=1 n	87n yz	Bit n de ST	Si	8.5 o 16.5

30.1 CLRST

Borra los primeros 3 nibbles de **ST.**
El último no lo borra porque contiene valores especiales para el procesador.

30.2 C=ST

Copia 3 nibbles de **ST** a **C**.

30.3 ST=C

Copia 3 nibbles de **C** a **ST**.

30.4 CSTEX

Intercambia 3 nibbles de **C** y **ST**.

30.5 ST=1 n

Enciende el bit **n** de **ST** a **1**.

30.6 ST=0 n

Borra el bit **n** de **ST**, poniéndolo por consiguiente a **0**.

30.7 ?ST=0 n y ?ST=1 n

Estas 2 instrucciones se usan para verificar el valor de un solo bit del registro **ST**.
Nosotros podemos probar si esta encendido o apagado. Usted puede poner un **GOYES** o un **RTNSI** con una Etiqueta después de estas instrucciones si usted lo desea.

31 Hardware Status Bits (HST)

Existen 11 instrucciones que pueden usarse para afectar los bits de **HST**.
Los bits útiles de HST son:
SR (Service Request)
SB (Sticky Bit)
MP (Module Pulled)
XM (eXternal Module missing)

Mnemónico	Hex-form	Campos	Carry	Ciclos
CLRHST	82F	4 bits de HST	No	4.5
XM=0	821	bit XM	No	4.5
SB=0	822	bit SB	No	4.5
SR=0	824	bit SR	No	4.5
MP=0	828	bit MP	No	4.5
HST=0 n	82n	1 o mas bits	No	4.5
?XM=0	831	bit XM	Si	7.5 o 15.5
?SB=0	832	bit SB	Si	7.5 o 15.5
?SR=0	834	bit SR	Si	7.5 o 15.5
?MP=0	838	bit MP	Si	7.5 o 15.5
?HST=0 n	83n	1 o mas bits	Si	7.5 o 15.5

CLRHST aclarará los 4 bits de **HST**. Usted también puede usar otras instrucciones, como **XM=0** o **SB=0** para aclarar un solo bit de **HST**.
Hay 4 bits, entonces existen 4 instrucciones para aclarar los bits.
Hay también otra manera de poner algunos bits de **HST** a cero usando la siguiente fórmula:

Valor = XM + (2*SB) + (4 * SR) + (8 * MP)

Ejemplo:
Si usted quiere tener:
XM = 0
SB = 0
SR = 1
MP = 1

Entonces calculará:
Valor = 0 + (2 * 0) + (4 * 1) + (8 * 1)
= 0 + 0 + 4 + 8
= 12
Entonces colocando **HS=0 12** se pueden encender sólo los bits **SR** y **MP**.
Note que la instrucción **HS=1 n** no existe.

Hay instrucciones para testear cada uno de los 4 bits disponibles y la instrucción **?HS=0 n** puede ser usada para probar si algunos o todos los bits de **HST** son iguales a cero.
Después de testear estas instrucciones se puede poner un **GOYES** o **RTNSI** si la prueba es verdadera.

Estas instrucciones necesitan 6 ciclos para realizar la prueba y continuar si es falso o 13 ciclos si la prueba es verdadera y si el salto se realiza. Puede ser un salto a una Etiqueta o un retorno que usa **RTNSI** si nosotros estamos dentro de un subprograma.

Ejemplo:
Escribamos algún código para volver de un subprograma si el bit **XM** se pone a **1**:
Valor = 1 + (0 * 2) + (0 * 4) + (0 * 8)
= 1
Esto significa que el código puede escribirse así:

```
?HST=0 1
RTNSI
```

Si nosotros queremos saltar a la Etiqueta "**tifosis**" si **XM=1** y **SR=1**, nosotros primero debemos calcular el valor:
Valor = 1 + (0 * 2) + (1 * 4) + (0 * 8)
= 1 + 4
= 5
Así que nuestro código se parecería a:

```
?HST=0 5
GOYES   tifosis
```

Para probar **SB** después de una rotación de bits o nibbles en un registro de trabajo nosotros tenemos:

```
?HST=0 2
GOYES result_loss
etc.
```

32 Saturn modo DEC/HEX

El procesador Saturn puede trabajar bajo dos modos: el modo hexadecimal (base 16) o el modo decimal (base 10).Recuerde que con BCD se usan 4 bits para codificar cada número decimal.

Mnemónico	Hex-form	Campos	Carry	Ciclos
SETHEX	04	Modo Hexadecimal	No	4
SETDEC	05	Modo Compacto BCD	No	4

33 Interrupciones

Mnemónico	Hex-form	Campos	Carry	Ciclos
INTOFF	808F	Desactiva las interrupciones del teclado	No	7
INTON	8080	Habilita las interrupciones del teclado	No	7
RSI	80810	Restablece las interrupciones	No	8.5

INTOFF e **INTON** simplemente desactivan o habilitan las interrupciones unidas al teclado. Para desactivar las interrupciones de todos, nosotros tenemos que poner el bit **15** de **ST** a **0** usando **ST=0 15**

34 Parte V: Objetos en la HP49G

Los objetos son una parte esencial de la HP49G. Todo lo puesto en la pila es un objeto, y ellos están codificados de la misma manera: empiezan con un prólogo de 5-nibbles que identifica el objeto, y lo que sigue es conocido como el cuerpo (body) del objeto.

En cierto modo, el objeto es una cáscara, y los datos se ponen dentro.

Cuál es el prólogo?
El prólogo es lo que identifica a un objeto, a continuación se verán los prólogos y a que tipos de objetos pertenecen.

PROLOGOS		TIPO DE OBJETOS
029E8	DOARRY	Serie (Array)
02B62	DOBAK	Backup
02911	DOBINT	Binario (BINT)
02DCC	DOCODE	CODE
02D9D	DOCOL	Secundario
02A2C	DOCSTR	String
0299D	DOECMP	Long complex
02955	DOEREL	Long real
02ADA	DOEXT	Unidad
026AC	DOFLASHP	Puntero Flash
02B1E	DOGROB	GROB
02A4E	DOHSTR	HXS
02E48	DOIDNT	Nombre Global (ID)
02614	DOINT	Entero (ZINT)
02E6D	DOLAM	Nombre Local (LAM)
02A0A	DOLNKARRY	Linked array
02B40	DOLIB	Librería
02A74	DOLIST	Lista
02686	DOMATRIX	Matriz
02933	DOREAL	Número Real
02E92	DOROMP	XLIB
02A96	DORRP	Directorio
02AB8	DOSYMB	Simbólico
02AFC	DOTAG	Etiquetado
026D5	DOAPLET	
02B88	DOEXT0	
02BAA	DOEXT1	
	DOACPTR	
02BCC	DOEXT2	
02BEE	DOEXT3	
02C10	DOEXT4	
02660	DOLNGCMP	
0263A	DOLNGREAL	

Algunos objetos que describiré, son *compuestos*: como el objeto 'directorio HOME' o el objeto 'biblioteca' (el más interesante de todos). Pero otros, como las cadenas de caracteres, son *simples*.

La siguiente tabla, muestra el nombre de cada tipo de objeto y el valor conseguido por el comando `TYPE`:

Nombre del Tipo de objeto	Número	Clase
Número real	0	Simple
Número complejo	1	Compuesto
Cadena de caracteres	2	Simple
Lista	5	Compuesto
Nombre global	6	Simple
Nombre local	7	Simple
Programa RPL	8	Compuesto
Expresión algebraica	9	Compuesto
Cadena Hexad Entero binario	10	Simple
Gráfico	11	Simple
Objeto etiquetado	12	Compuesto
Objeto unidad	13	Compuesto
Objeto Puntero ROM (XLIB)	14	Simple
Directorio	15	Compuesto
Biblioteca	16	Compuesto
Backup		
	20	
Número real extendido	21	
Número complejo extendido	22	
Objeto carácter	24	Simple
Objeto Código	25	Simple
Biblioteca de datos	26	Simple
Minifuente	27	Simple
Número entero	28	Simple
Formación	29	
Fuente	30	Simple

35 Objetos Simples

35.1 Número Real

Prologo: 02933
Tamaño: 21 nibbles

EN MEMORIA	
Prologo	5 nibbles
Exponente	3 nibbles
Mantisa	12 nibbles
Signo	1 nibble

El número real es dividido en tres campos: la mantisa, el signo y el exponente.
La mantisa es el número, el exponente es **n** en **10n** y el signo se usa codificando un nibble dónde el positivo es **#0h** y el negativo es **#9h**

La mantisa usa 12 dígitos y el número se codifica en memoria usando el modo compacto BCD (Binary Coded Decimal). El exponente también se codifica usando BCD, pero de una manera especial según su signo.
Si el exponente es positivo, los 3 nibbles son codificados usando BCD.
Si el exponente es negativo, el exponente se codifica como sigue:
1000 - ABS(exponente)
ABS es el valor absoluto del exponente cuyo valor va de 0 a 499

Ejemplo:
Codifiquemos el número **2.79233x10-9**.
Primero nosotros colocamos el prologo que es #33920h (recuerde que debe invertirse).
Después tenemos el exponente que es **1000 - ABS(-9)**. Esto es **1000 - 9 = 991**, nosotros debemos colocar **#199**. La mantisa es **279233**, esto se debe codificar en memoria como **#000000332972h**. y finalmente el signo se codifica como: **#0h**.
En la memoria nosotros tenemos **#33920 199 000000332972 0h** (Los espacios son para dar claridad)

NOTA: El Saturn invierte el orden de nibbles en la memoria. Esto es por qué cada campo aquí se invierte en la memoria

35.2 Carácter

Prologo: 029BF
Tamaño: 7 nibbles

EN MEMORIA	
Prologo	5 nibbles
código ASCII del carácter	2 nibbles

Cada carácter consta de 2 nibbles, es uno de los objetos más fáciles de escribir en la HP49.

35.3 Cadena de Caracteres

Conocido como “string”
Prologo: 02A2C
Tamaño: 12 nibbles

EN MEMORIA	
Prologo	5 nibbles
Tamaño	5 nibbles
1er Carácter	2 nibbles
…	
Carácter n	2 nibbles

El tamaño de un string es 5 nibbles y cada carácter es de 2 nibbles.
Como con todos los otros objetos, el prólogo no es considerado en el tamaño.
También se puede usar ASCII, nosotros podemos ver la tabla ASCII de la hp presionando la tecla roja y **CHARS**
Ejemplo:
Para codificar el string "HP" se colocaría primero el prologo de un string **#02A2Ch,** recuerde que nosotros tenemos 2 caracteres, estamos ocupando 4 nibbles. Se usan 5 nibbles para codificar el tamaño (**#9h**) y finalmente los 4 nibbles de los caracteres (**#4850**).
Esto significa que el string "HP" quedaría como sigue:
#C2A20 90000 8405H
Nota: Se invierten los dos nibbles para cada carácter,
#48h es **#72d** que es el carácter "**H**"
#50h es **#80d** que es el carácter "**P**"
Ahora nosotros podemos manipular strings o cadenas en ensamblador, ahora veamos unos ejemplos.

EJEMPLO BLANKSTR.
Este ejemplo crea un string de #n espacios.
* Entrada: Un entero binario (#n)
* Salida: string de #n espacios

CODIGO PARA MASD

```
CODE
 GOSBVL   SAVPTR
 A=DAT1   A          %A(A) contiene la dirección del bint del nivel 1
 D1=A                %D1 contiene al bint del nivel 1
 D1=D1+   5          %Salta el prologo del bint
 C=DAT1   A          %Lee el valor del bint
 C=C+C    A          %* 2 por nibs
 GOSBVL   MAKE$N     %crea un string en tempob
 A=R0     A          %->string nuevo
 DAT1=A   A          %string
 D1=A                %string
 D1=D1+   5          %salta el prologo del string
```

```
 C=DAT1   A          %lee la longitud creada en nibs
 C=C-5    A          %-5 nibs por la longitud del campo
 CSRB     A          %divide nibs por 2 para # caracteres (bytes)
 C=C-1    A          %-1 para el contador
 GOC      EXIT       %Sale si es nulo
 D1=D1+   5          %salta la longitud del campo
 LA(2)    32         %carácter espacio
 *CLEARLUP
 DAT1=A   B          %escribe un carácter
 D1=D1+   2          %siguiente byte
 C=C-1    A          %decrementa el contador
 GONC     CLEARLUP
 *EXIT
 GOVLNG GETPTRLOOP
ENDCODE
@
```

EJEMPLO TAMAÑO:

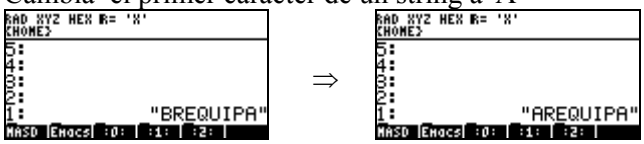
Programa que devuelve el tamaño de un string.

CODIGO PARA MASD

```
CODE
GOSBVL   SAVPTR
GOSBVL   GetStrLenStk    %C(A) Numero de caracteres en el string
A=C A                    %prepara para empujar
GOVLNG   PUSH#ALOOP      %Empuja el tamaño del string a la pila
                         %como un Bint y restaura el sistema.
ENDCODE
@
```

EJEMPLO CAMBIO DE CARACTER

Cambia el primer carácter de un string a 'A'

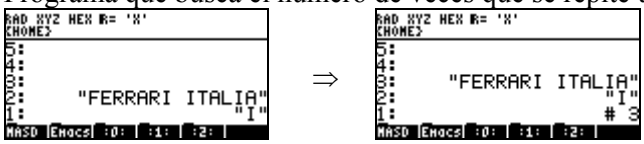


CODIGO PARA MASD

```
CODE
SAVE
C=DAT1 A        %Graba en C(A) la dirección del objeto del nivel 1
D1=C            %D1 ➔ string
D1=D1+10        %Salta el prologo y la longitud
LC 41           %Código del carácter 'A'
DAT1=C  B       %Cambia el carácter en el string
LOADRPL
ENDCODE
@
```

EJEMPLO CARACTER

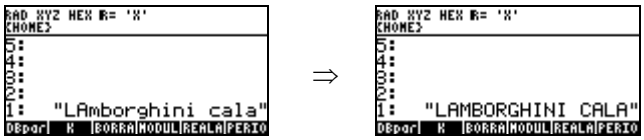
Programa que busca el número de veces que se repite un carácter en un string



CODIGO PARA MASD

```
CODE
SAVE
A=DAT1 A
D0=A                    %D0 ➔ Buscar carácter
D0=D0+ 10               %Salta el prologo y la longitud
A=DAT0 B                %Busca el carácter
D1=D1+ 5                %D1 ➔ Nivel 2 de la pila
GOSBVL GetStrLenStk     %C(A) Numero de caracteres en el string
B=0 A                   %Inicia la cuenta
D=C A                   %D(A) Caracteres en el string
D=D-1 A                 %Ajusta longitud de base 0
GOC Salir               %si el string es nulo sale
*Buscar
C=DAT1 B                %Lee los caracteres para buscar en el string
*string
?C#A B                  %Pregunta si no es el mismo carácter
GOYES Siguiente
B=B+1  A                %SI, busca en el siguiente carácter
*Siguiente
D=D-1 A                 %NO, Incrementa el contador en uno
GOC Salir               %Sale si se alcanzo el final del string
D1=D1+ 2                %D1 ➔ Siguiente carácter en la búsqueda
GOTO Buscar             %Continua buscando
*Salir
A=B A                   %A(A) Cuenta de los caracteres.
GOVLNG PUSH#ALOOP       %Empuja el contador del carácter como un Bint
ENDCODE
@
```

EJEMPLO MINISCULAS A MAYUSCULAS:



La distancia entre una letra minúscula y mayúscula es de 32 nibbles.
El siguiente ejemplo es interesante, convierte un string que contenga minúsculas en mayúsculas:

CODIGO PARA MASD

```
CODE
C=DAT1 A          % lee la dirección de objeto del primer nivel de la pila
?C#0 A            % si no está vacío (diferente de cero)
GOYES STRING?     % verifica si es una cadena
LA 00201          % de lo contrario, carga #201h en A (#201h: "Muy pocos argumentos")
*ERROR            % ésta etiqueta muestra el #error de A
GOVLNG 05023      % Errjmp
*STRING?          %Etiqueta  STRING?
CD1EX             %D1 apunta al objeto
A=DAT1 A          % lee su prólogo
D1=C              %y restaura D1
LC 02A2C          %El prólogo del string se carga en C
?A=C A            %Si el objeto es un string
GOYES OK          %ir a OK
LA 00202          %de lo contrario, carga #202h en A (#202h: "Argumento incorrecto")
GOTO ERROR        %Saltamos a la Etiqueta ERROR
*OK               %El tipo de rutina principal
SAVE              %Guarda los registros D1, D0, B, D
C=DAT1 A          %Lee la dirección del string
D1=C              %Apunto al string
D1=D1+ 5          %Salto el prólogo
C=DAT1 A          %Lee el tamaño del string
C=C-5 A           %Remuevo su longitud
D=C A             %y lo guarda en D
DSRB A            %Divide por dos
D=D-1 A           %y quita uno porque yo voy a usar  el carry
GOC END           %si el carry se enciende el string esta vació y entonces salta al final
D1=D1+ 3          %mueve 3 nibbles adelante.
*LOOP             %Etiqueta  LOOP
D1=D1+ 2          % Apunta al carácter
C=DAT1 B          % lo lee en C
LA 61             % #61h es #97d
?C<A B            %si es menor que 97.
GOYES CHECK       %Salta a la Etiqueta CHECK
LA 7A             % #7Ah es #122d
?C>A B            % si es mayor a 122.
GOYES CHECK       %Salta a la Etiqueta CHECK
                  % aquí nosotros tenemos que escribir las letras mayúsculas del carácter,
                  %para eso se debe que quitar 32.
```

```
C=C-16 B
C=C-16 B          %(valor del carácter) - 32
DAT1=C B          % escribe en el string
*CHECK            %Etiqueta CHECK
D=D-1 A           %hay otro carácter?
GONC LOOP         %en ese caso, salta a la Etiqueta LOOP
*END              %Etiqueta  END
LOADRPL           %restaura los registros  y sale
ENDCODE
@
```

Ahora probemos el programa, con una cadena de caracteres, como: “Lenguaje Español”. Después de ejecutar el programa, conseguiremos “LENGUAJE ESPAÑOL”

El código puede ser optimizado, y vocales con tilde, podrían ser agregados en la verificación. Si usted consigue mejorar este código, se dará cuenta que ya aprendió, y que lenguaje ensamblador es fácil.

Como práctica, escriba un programa que haga lo inverso: que convierta cadena de caracteres de Mayúsculas a minúsculas.

35.4 Nombre Global

Prologo: 02E48
Tamaño: 7 nibbles

EN MEMORIA	
Prologo	5 nibbles
Tamaño	2 nibbles
1er Carácter	2 nibbles
…	
Carácter n	2 nibbles

La diferencia principal entre un string y los nombres globales es que estos solo pueden tener 255 caracteres.

En el siguiente ejemplo convierte un string a un nombre global, coloque un string en la pila.

```
CODE
C=DAT1 A  % Lee la dirección del nivel 1 de la pila
R0=C A    % Guarda en R0 dicha dirección  para un futuro uso
?C#0 A    % Si el nivel 1 contiene algo
GOYES TEST  %verifica si es un string
LA 00201  % Si no da el error "Too Few Arguments"
*ERROR
GOVLNG 05023  %Esta rutina hace el error
*TEST
CD1EX     % Apunta al objeto
A=DAT1 A  % Lee el prologo
D1=C      % Restaura la dirección del nivel 1 de la pila
LC 02A2C  % 02A2C = prologo de un string
?A=C A    %Si es un string salta a
GOYES TODOSOCATE
LA 00202  % Si no error #202h, que es
GOTO ERROR  % "Bad Argument Type"
*TODOSOCATE
SAVE      % Guarda D0,D1,B,D
C=R0 A    % C contiene la dirección de nivel 1
D1=C      % D1 apunta al objeto
D1=D1+ 5  %Apunta al tamaño del string
C=DAT1 A  % Aa contiene el tamaño del string
C=C-5 A   % remueve la longitud del tamaño del campo
CSRB A    % y divide por dos, ahora Ca contiene el numero de caracteres dentro del string
R1=C A    % Guarda el numero de caracteres en R1a
C=C+C A   % Multiplicamos por 2, nosotros necesitamos el tamaño en nibbles para reservar memoria
C=C+7 A   % Agregamos 7 nibbles,  5 para el prologo de un nombre global
          %l y 2 para la longitud del campo
GOSBVL 039BE  %esta rutina se llama RES_ROOM.
              % Ca asigna y produce "Memoria Insuficiente"
              % error si no hay bastante RAM. También verifica si
              % existen 5 nibbles libres, para que nosotros podamos empujar
              % el objeto reservado hacia la pila.
CD0EX         % D0 = objeto recientemente reservado, ahora C contiene su dirección
```

```
R2=C A          %Guardamos en R2a
D0=C            %D0 apunta a l nueva memoria reservada
LC 02E48        %02E48 = prologo de un nombre global
DAT0=C A        % escribimos
D0=D0+ 5        % y movemos 5 nibbles
C=R1 B          % recupera el tamaño que se calculo
DAT0=C B        % y lo escribe al campo de longitud de nombre global
C=C-1 B         % quitamos uno, si el carry se enciende
GOC PUSH        % entonces el string está vacío, así que empujamos al nombre global a la pila
A=R0 A          % recupera la dirección del string
D1=A            %D1 ahora apunta a la dirección del string
D1=D1+ 8        % y movemos 8 nibbles. ¿Por qué no 10?
                % porque dentro de LOOP nosotros tenemos D1=D1+ 2
                % nosotros conseguimos finalmente D1=D1+ 10
*LOOP           % en esta vuelta, nosotros copiamos los caracteres uno en un tiempo
D1=D1+ 2        % movemos 2 nibbles adelante, cuando entramos en LOOP
                % nosotros entramos en el primer carácter; por otra parte, nosotros avanzamos
                % a los próximos dos caracteres .
D0=D0+ 2        % movemos 2 nibbles adelante en el nuevo objeto reservado
A=DAT1 B        % lea un carácter en el string
DAT0=A B        % y lo escribe en el nombre global
C=C-1 B         % quitamos 1 del string
GONC LOOP       % y volvemos hasta que el carry se encienda
*PUSH           %aquí empujamos el nombre global a la pila
LOAD            % recuperamos los registros D0, D1, B, D.
D=D-1 A         % quitamos 5 nibbles de RAM
D1=D1- 5        % y empuja un objeto a la pila
C=R2 A          % recupera la dirección del nombre global
DAT1=C A        % lo empuja a la pila
RPL             % retorna al rpl
ENDCODE
@
```

35.5 Nombre Local

Prologo: 02E6D
Tamaño: 7 nibbles

EN MEMORIA	
Prologo	5 nibbles
Tamaño	5 nibbles
1er Carácter	2 nibbles
…	
Carácter n	2 nibbles

Como se puede apreciar la diferencia entre un nombre local y un nombre global es el prologo, pero el nombre local no esta disponible todo el tiempo, como el nombre global

35.6 Entero Binario

Hay dos tamaños de enteros binarios uno de 15-nibbles y otro ilimitado (dentro de la memoria)
La HP solo nos permite crear el entero binario mas pequeño, nosotros podemos crear el grande pero manualmente.

Prologo: 02A4E
Tamaño: 11 nibbles

EN MEMORIA	
Prologo	5 nibbles
Tamaño	2 nibbles
1er Digito	1 nibble
…	
Digito n	1 nibble

Cuando uno crea un número con #, la HP lo convertirá en base hexadecimal, si teclea # sin h al final la hp lo considerara como número hexadecimal, si se pone d lo considerara como decimal y o como octal, entonces la HP lo convertirá automáticamente a la base que se este usando
Cada digito usa un nibble.
Como un ejemplo, codifiquemos #BAh.
Este programa crea 85-dígitos en base binaria.

```
CODE
SAVE          % Guardamos los registros
LC 0005A      % reserva 90 nibbles
GOSBVL 039BE  % llamada de RES_ROOM
CD0EX         % nosotros guardamos la dirección del  objeto
R0=C A        % en R0a
D0=C          % restauramos D0
LC 02A4E      % prólogo del número binario
DAT0=C A      % lo escribe
D0=D0+ 5      % movemos 5 nibbles
LC 00055      % El tamaño es 90-5=85, #55h,
DAT0=C A      % escribimos el tamaño del objeto
LOAD          % recuperamos los registros
D=D-1 A       % liberamos 5 nibbles de la memoria ram
D1=D1- 5      % Ponemos al objeto en la pila
C=R0 A        % su dirección está dentro de R0a
DAT1=C A      % nosotros lo empujamos a la pila
RPL           %Salimos
ENDCODE
@
```

35.7 XLIB name

Prologo: 02E92
Tamaño: 11 nibbles

EN MEMORIA	
Prologo	5 nibbles
ID librería	3 nibbles
Número de comando	3 nibbles

Toma 6 dígitos en forma binaria (#...) como un argumento; los primeros tres dígitos son del número de la librería y los últimos tres dígitos del número del comando de la librería, ambos en hexadecimal.

35.8 Sistema entero binario

Prologo: 02911
Tamaño: 10 nibbles

EN MEMORIA	
Prologo	5 nibbles
Número	5 nibbles

Contiene un entero binario que se codifica usando cinco nibbles (20 bits).
El número se mostrará según la base activa, pero siempre se codifica en hexadecimal.

35.9 Long real

Prologo: 02955
Tamaño: 26 nibbles

EN MEMORIA	
Prologo	5 nibbles
Exponente	5 nibbles
Mantisa	15 nibbles
Signo	1 nibbles

También llamado real extendido, el long real es más preciso que el real normal. El usuario no tiene acceso a este número, la HP lo usa internamente para una mayor precisión durante los cálculos. Su codificación es similar en cierto modo al real normal, pero aquí algunos campos tienen tamaños mayores, aquí se usa el BCD.

35.10 Objeto Código (Code)

Prologo: 02DCC
Tamaño: 10+ nibbles

EN MEMORIA	
Prologo	5 nibbles
Tamaño	5 nibbles
Código Saturn	Varia

Tiene la misma estructura que las cadenas (Strings), algunas personas esconden un string en forma de un code, o cualquier otro objeto, esto es fácil de hacer solo hay que cambiar el prologo.

35.11 Datos de librería (Library Data)

Prologo: 02B88
Tamaño: 10 nibbles

EN MEMORIA	
Prologo	5 nibbles
Tamaño	5 nibbles
Datos	Varia

HP ha hecho este objeto disponible a programadores para que las librerías que necesiten guardar valores puedan guardar información dentro de un objeto separado. Esto es bueno cuando uno quiere proteger sus datos.

35.12 Backup

Prologo: 02B62
Tamaño: varia

EN MEMORIA	
Prologo (02B62)	5 nibbles
Tamaño	5 nibbles
Número de caracteres del nombre del objeto	2 nibbles
Primer carácter del nombre	2 nibbles
…	
Ultimo carácter del nombre	2 nibbles
Objeto	Varia el tamaño
Prologo del sistema binario (02911)	5 nibbles
0	1 nibble
CRC	4 nibbles

Pueden ponerse varios objetos dentro de un Backup. La HP pondrá sólo un objeto y lo seguirá con un sistema binario oculto cuyo primer nibble es 0 junto con otros cuatro nibbles que contienen el CRC del objeto. El primero de los cinco nibbles del binario usado del sistema siempre será el cero porque el CRC se codifica usando sólo cuatro nibbles.

35.13 Puntero Extendido (Extended pointer)

Prologo: 02BAA
Tamaño: 15 nibbles

EN MEMORIA	
Prologo	5 nibbles
Dirección del objeto	5 nibbles
Dirección del programa	5 nibbles

Algunas áreas de memoria son cubiertas por otras, pero a veces nosotros necesitamos apuntar algo que se cubre. Éste es el objeto que la HP usa internamente para ese propósito. En este caso, el objeto es a lo que nosotros queremos tener acceso y el programa es la rutina de ROM que se asociará con la dirección del objeto,

35.14 Números Complejos

Prologo: 02977
Tamaño: 37 nibbles

EN MEMORIA	
Prologo	5 nibbles
Exponente real	3 nibbles
Mantisa real	12 nibbles
Signo real	1 nibble
Exponente imaginario	3 nibbles
Mantisa imaginaria	12 nibbles
Signo imaginario	1 nibble

Cuando se representa un número complejo lo que se esta haciendo es mantener unidos dos números reales.
El primero es la parte real del complejo, y el segundo es la parte imaginaria.
Por ejemplo, codifiquemos **(100, 1000)**.

Primero viene el prólogo, **#77920h** (invertido, como de costumbre), después el exponente real, #200h, seguido de la mantisa, #000000000001 (para la parte real), y el signo, #0h. Seguido de la parte imaginaria que se codifica como #300h para el exponente #000000000001h para la mantisa, y finalmente el signo, #0h. Todo esto se codifica en la memoria así:
#77920 200 000000000001 0 300 000000000001 0

35.15 Array

Prologo: 029E8
Tamaño: 27 nibbles

EN MEMORIA	
Prologo	5 nibbles
Tamaño	5 nibbles
Tipo de objetos	5 nibbles
Numero de dimensiones	5 nibbles
1ra Dimensión	5 nibbles
…	
Dimensión n	5 nibbles
Contenido del primer objeto	Varia el Tamaño
…	
Contenido del ultimo objeto	Varia el Tamaño

Puede contener cualquier número de objetos que sean del mismo tipo. Cuando sólo tiene una dimensión, se llama un "vector". Si hay dos dimensiones, nosotros tenemos una "matriz". Nosotros también podemos darle más de dos dimensiones.
Normalmente, se usan para mantener unidos una serie de números reales o complejos. Nosotros debemos definir primero qué objetos van a ser guardados, entonces, nosotros tenemos que decir el tamaño de la serie entera, pero depende del tipo de objetos que nosotros vamos a coleccionar, y claro, cuántos de ellos serán codificados.

Lo que es interesante aquí es que aun cuando nosotros estamos coleccionando diez reales, no hay diez prólogos reales guardados en la serie: su prólogo sólo se guarda una vez. Esto es por qué el tipo de objetos que usa cinco nibbles se pone en código y también por qué sólo un tipo de objeto puede guardarse dentro de una serie. Una vez que nosotros hemos puesto el tipo de objetos dentro de la serie, nosotros codificamos el número de dimensiones que usan cinco nibbles: #00001h para vectores o #00002h para el matrices. Después de las dimensiones, nosotros debemos decir cuántos objetos están en cada dimensión. Si nosotros tenemos una matriz, nosotros codificaremos el número de líneas y columnas.

Una vez que nosotros hemos hecho todo esto, nosotros podemos empezar la codificación de los objetos dentro de la serie. Los valores de la manera que se guardan varían según el tipo de objetos guardado. Los prólogos ya no se necesitan porque nosotros ya los hemos grabado.

Una cosa importante es que no importa el tamaño de la serie, si en algunas partes esta vacía estas todavía usan memoria.

Por ejemplo, creemos una serie de strings, con algunos nombres dentro de él.
[HP48S HP48G]

Primero, nosotros tenemos el prólogo de la serie: **#029E8h**. El tamaño total es **#00032h**. Después tenemos que decir que tipo de objeto es, así que nosotros usamos **#02A2Ch** (5 nibbles). hay sólo una dimensión así que nosotros debemos escribir **#00001h** (5 nibbles), y hay dos elementos así que escribimos **#00002h** (5 nibbles).

Finalmente nosotros codificamos las dos cadenas. Hay dos nibbles por cada carácter, cada cadena toma diez nibbles más cinco nibbles más adelante de él, es decir su longitud. Porque cada cadena toma 15 nibbles, incluso el campo de la longitud es **#0000Fh**. Esto significa un total de 50 nibbles (longitud de **#32h**, como **#50d** = **#32h**), más el prólogo quedaría así:
#8E920 23000 C2A20 10000 20000 F00008405438335 F00008405438374

35.16 Lista

Prologo: 02A74
Epilogo: 0312B
Tamaño: 10 nibbles

EN MEMORIA	
Prologo	5 nibbles
1er Objeto	Varia el tamaño
…	
Objeto n	Varia el tamaño
Epilogo	5 nibbles

Puede contener cualquier tipo de objeto y tantos objetos como la RAM lo permita.
Esto se hace usando un prólogo para marcar el principio y un epílogo para marcar el final.
Una lista vacía será {} y 10 nibbles de longitud: hay un prólogo y un epílogo pero nada entre ellos.

Ejemplo
Pongamos la lista {10.2 "dX"}.
Como se puede ver la lista contiene un número real y una cadena. Es bastante fácil.
Primero codifique cada objeto y entonces ponga los dos entre el prólogo de la lista y epílogo:
10.2 es **#33920 100 000000000201 0**
"dX" es **#C2A20 90000 4685**
Y para que quede como una lista se codificaría así:
#47A20 339201000000000002010 C2A20900004685 B2130

35.17 Programa RPL

Prologo: 02D9D
Tamaño: 20 nibbles para arriba

EN MEMORIA	
Prologo	5 nibbles
<< Inicio del User (2361E)	5 nibbles
…	
>> Final del User (23639)	5 nibbles
Epilogo	5 nibbles

Es similar a una lista, mientras contiene un prólogo y un epílogo. Dentro del programa de RPL, se encuentran los objetos, uno después de otro, pero cada objeto puede ser una dirección en lugar del propio objeto. Por ejemplo, suponga usted crea el siguiente programa RPL:
<< OFF >>
En lugar de encontrar el código que apaga la HP48 dentro de este objeto RPL, hay sólo una dirección que apunta al código. Cuando la HP despliega un programa RPL, descifra cada dirección con el nombre correspondiente, si tiene uno.
El programa << OFF >> es codificado de esta manera en la memoria:
D9D20
E1632 % <<
E13A1
93632 % >>
B2130
Aquí, los nibbles se invierten. Los primeros y últimos 5 nibbles son del objeto de RPL: **#02D9Dh** es el prólogo y **#0312Bh** es el epílogo.

35.18 Expresión Algebraica

Prologo: 02AB8
Epilogo: 0312B
Tamaño: 10 nibbles

EN MEMORIA	
Prologo	5 nibbles
Objeto 1	Varia el tamaño
…	…
Objeto n	Varia el tamaño
Epilogo	5 nibbles

Los símbolos matemáticos se codifican colocando su dirección en ROM

Ejemplo
Codifiquemos ' A+2 ':
El Prologo es **021B8**, luego nosotros debemos codificar el nombre global **A**, que se codifica como **02E48** Prologo **01** para el Tamaño y **41** para el código de ASCII del carácter '**A**'. mire la sección de cómo se codifican objetos globales. Luego viene "**+**" esto se codifica usando su dirección en ROM, **#2A2Deh**, y el número **2**, codificando su dirección de ROM, **#1AB67h**. y finalmente el epílogo **#0312Bh**. Esto significa que el objeto completo, en ROM es:
#8BA20 84E201014 ED2A2 76BA1 B2130

35.19 Objeto Etiquetado

Prologo: 02AFC
Tamaño: varia

EN MEMORIA	
Prologo	5 nibbles
Tamaño de la Etiqueta	2 nibbles
1er carácter de la Etiqueta	2 nibbles
…	
Ultimo carácter de la Etiqueta	2 nibbles
Objeto	Varia el tamaño

Ejemplo:
Etiquetemos el valor 10 que usa "Diez". Se codificara así:
#CFA20 30 4556E6 33920 100 000000000001 0
Note que esto empieza con el prólogo de la Etiqueta **#02AFCh**, el número de caracteres usado por la Etiqueta (**#03h**, invertido) y el valor ASCII de los caracteres que forman la Etiqueta se codifica (Diez se vuelve **#6E6554h**, y se invierte entonces). Finalmente, el número real 10 se codifica como se aprendió antes.

35.20 Objeto Unidad

Prologo: 02ADA
Epilogo: 0312B
Tamaño: varia

EN MEMORIA	
Prologo	5 nibbles
Objeto	Varia el tamaño
Unidad	Varia el tamaño
Epilogo	5 nibbles

Este objeto se define con un prólogo y un epílogo y entre ellos nosotros encontramos dos valores. El primero es el objeto que recibirá la unidad; puede ser un valor real, como 2, el segundo es la unidad.

El símbolo como (+ / * -) se codifica usando su dirección en ROM, y los caracteres se codifican como si fueran símbolos.

Ejemplo:
Codifiquemos 300 000 000 metros por segundo.
Primero, ponga el prólogo que es **#02ADAh**, después ponga el objeto que es el número real 300 000 000, este numero se codificaría como **#33920 800 000000000003 0**.
Finalmente la unidad, esto es importante: nosotros debemos usar la notación RPN, nosotros vemos que la unidad "metros por segundo" es entonces: **m s /**
Nosotros tenemos dos caracteres, **m** y **s**. Estos vendrían a ser dos cadenas, estas dos cadenas o strings quedarían así:
#C2A20 70000 D6
#C2A20 70000 37
El signo de la división (/) viene luego, para dividir **m** por **s** y el signo de la multiplicación (*) para multiplicar el número por la unidad.
El epílogo se inserta al final, nosotros conseguimos finalmente:
#ADA20 (prólogo)
#33920080000000000030 (número 300000000)
#C2A2C70000D6 (M)
#C2A2C7000037 (S)
#86B01 (/)
#68B01 (*)
#B2130 (epílogo)

35.21 Directorios

Nosotros tenemos dos tipos de directorios: el directorio de **HOME**, y los otros directorios que creamos. **HOME** se llama a menudo el "directorio de la raíz."
Solo veremos el prologo ya que es un poco complejo este tema:

Directorio HOME
Prologo: 02A96
Tamaño: 57 nibbles

Subdirectorios
Prologo: 02A96

35.22 Long complex

Prologo: 0299D
Tamaño: 47 nibbles

EN MEMORIA	
Prologo	5 nibbles
exponente real	5 nibbles
Mantisa real	15 nibbles
Signo real	1 nibble
Exponente Imaginario	5 nibbles
Mantisa imaginaria	15 nibbles
Signo Imaginario	1 nibble

Vendría a ser como dos reales extendidos juntos, el primero viene a ser la parte real y el segundo la parte imaginaria.

35.23 Linked array

Prologo: 02A0A
Tamaño: 32+ nibbles

EN MEMORIA	
Prologo (02A0A)	5 nibbles
Tamaño	5 nibbles
Tamaño de objetos	5 nibbles
Numero de dimensiones	5 nibbles
Dimensión 1	5 nibbles
…	
Dimensión n	5 nibbles
Puntero del primer objeto	5 nibbles
…	
Puntero del ultimo objeto	5 nibbles
Contenido del Primer objeto	Varia el Tamaño
…	
Contenido del ultimo objeto	Varia el Tamaño

El linked Array es un tipo especial de serie. En el linked Array hay indicadores, cada indicador es un atajo a un objeto que se codifico. Si usted tiene una serie grande que contiene varios valores idénticos, este tipo de serie le ahorrará mucho espacio de memoria.

35.24 Objeto Gráfico

Prologo: 02B1E
Tamaño: 22 nibbles

EN MEMORIA	
Prologo	5 nibbles
Tamaño	5 nibbles
Número de líneas	5 nibbles
Número de columnas	5 nibbles
…	Píxeles

El objeto gráfico contiene primero un prólogo, **#02B1Eh**, seguido por el tamaño del objeto entero, pero no se cuentan los cinco nibbles del prólogo.

Los próximos dos de cinco nibbles codifican el número de líneas y filas.

Después siguen los píxeles codificados uno después de otro, desde la esquina superior-izquierda a la esquina inferior-derecha.

Los datos de un grob son codificados usando bits en lugar de nibbles, cada byte codifica ocho bits que forman ocho píxeles del grob.

Un grob (Objeto Gráfico), contiene píxeles.

El bit **1** representa un píxel encendido.
El bit **0** representa un píxel apagado.

Cada grob debe ser un múltiplo de 8 píxeles. Si un grob no es múltiplo de 8 se debe agregar tantos ceros como sea necesario hasta que sea un múltiplo de 8.

Ejemplo.
Recuerde que **0** indica apagado y **1** encendido.

Si usted quiere: **0101 0111**

Esto tiene ocho píxeles, cuando nosotros codificamos esto debemos invertir cada nibble, esto quedaría **1010 1110**. Ese código es igual a #AEh. Después de agregar el prólogo (#02B1Eh), tamaño (#19d, o #13h nibbles), anchura (#4h), y altura (#1h) al principio, nuestro gráfico es codificado como sigue:

#E1B20 31000 40000 10000 AE00

Note que se agregaron dos ceros al final.

36 Parte VI: Escribiendo Programas

37 Realizar bucles

Bucles (LOOPS)
Hay varias maneras de hacer bucles, estas tienen varios propósitos. Un bucle puede usarse como espera o se puede repetir simplemente un juego de instrucciones varias veces, etc.
Cuando usted quiere hacer un bucle, se debe usar algo para guardar un valor.
Puede ser un registro como **A**, **B**, **C**, **D**, o **P**.
Ejemplo:
Aquí, nosotros vamos a usar el registro **C** como el contador de la vuelta:

```
LC FF
*LOOP
C=C-1 B
GONC LOOP
RPL
@
```

Éste es un programa simple. El número **FF** se carga en el registro **C** y después se entra en un bucle, donde **C** se ira decrementando de 1 en 1, el campo **B** indica que **C** contiene 2 valores. Una cosa importante para recordar al usar el carry es que la vuelta se hace 1 vez más que el valor del contador. ¿Por qué? **FF** estaba cargado en el campo **B**. se quitará 1 cada vuelta, pero cuando se lee cero, el carry todavía no se enciende por que todavía no hubo un desbordamiento en la memoria entonces cuando **C** contenga **#00h** y cuando se quite 1 el carry recién se encenderá. Por consiguiente, la vuelta se ejecuta #FFh + 1 veces.
Entonces cuando se use el carry para hacer vueltas en sus programas, no se olvide de quitar una unidad del valor del contador antes de empezar. Porque la vuelta se hará un tiempo más.

Aquí un ejemplo de cómo usar el contador **P**. Nosotros vamos a crear un programa que hace un clic al pulsar una tecla (emite un sonido en el zumbador de la HP).
Para encender el zumbador, nosotros usamos el registro **OUT** del procesador Saturn. Nosotros mandamos **#800h** a **OUT** y el zumbador se enciende.

Si nosotros queremos apagarlo, nosotros enviamos **#000h** y se apaga

Ejemplo:

```
CODE
LC 800              %Cargamos en C el valor de 800
OUT=C               %El registro OUT ahora es igual a 800
*LOOP
P=P+1               %Incrementa P en 1
GONC LOOP
C=0 X               %Cargamos C on 000
OUT=C               %OUT es igual a C
RPL                 %Regresamos al RPL
ENDCODE
@
```

38 Leyendo del teclado

DETECTAR UNA TECLA

Tenemos los registros **OUT/IN**, en el registro **OUT** se enviaran valores se salida, y en el registro **IN** se enviaran valores de entrada, entonces para detectar si se presiono una tecla, tenemos que enviar un valor específico al registro **OUT**, y después leer el registro **IN**, es así como comprobamos que una tecla fue presionada, comparando **IN** con el valor que se envió a **OUT**.

Debajo está la disposición del teclado de la HP49G, abajo de cada nombre encontramos dos números: uno a la izquierda, y uno a la derecha. El que esta a la izquierda será el valor de **OUT**, y el de la derecha el valor de **IN** que nos confirmara si se presiono alguna tecla.

F1 020/0001	F2 002/0002	F3 002/0004	F4 002/0008	F5 002/0010	F6 002/0020
				∧ 040/0008	
APPS 020/0080	MODE 010/0080	TOOL 008/0080		< 040/0004	> 040/0001
VAR 002/0080	STO 002/0080	NXT 001/0080		∨ 040/0002	
HIST 010/0040	CAT 008/0040	EQW 002/0040	SYMB 002/0040	 001/0040	
 010/0020	 008/0020	SIN 002/0020	COS 002/0020	TAN 001/0020	
EEX 010/0010	+/- 008/0010	X 002/0010	1/X 002/0010	/ 001/0010	
ALPHA 008/0008	7 008/0008	8 002/0008	9 002/0008	* 001/0008	
(CI) 008/0004	4 008/0004	5 004/0004	6 002/0004	- 001/0004	
(CD) 080/0002	1 008/0002	2 004/0002	3 002/0002	+ 001/0002	
ON /8000	0 008/0001	. 004/0001	SPC 002/0001	ENTER 001/0001	

Usted puede usar el programa GKey para ayuda interactiva.

- El teclado puede leerse con el registro **IN**, pero también debe ejecutarse desde una dirección, es por eso que nosotros usamos una rutina en ROM "**CINRTN**" o "**AINRTN**"

- El registro **IN** es de 4 nibbles y los bits del **0** al **8** se usan para el teclado, el bit **11** se enciende cuando la tecla [ON] es presionada.

- El registro **OUT** es de 3 nibbles y se usa para determinar la "columna" de la tecla

La tabla que se muestra a continuación contiene los códigos de las teclas de las HP48 y HP49:

TECLAS	HP48		HP49	
	OUT hex	IN bit	OUT hex	IN bit
MTH	004	4	Teclas solo disponibles en la HP48	
PRG	080	4		
CST	080	3		
TECLA '	001	4		
EVAL	040	3		
DEL	010	1		
APPS	Teclas solo disponibles en la HP49		020	7
MODE			010	7
TOOL			008	7
HIST			010	6
CAT			008	6
EQW			004	6
SYMB			002	6
TECLA X			004	4
F1	002	4	020	0
F2	100	4	020	1
F3	100	3	020	2
F4	100	2	020	3
F5	100	1	020	4
F6	100	0	020	5
VAR	080	2	004	7
NXT	080	0	001	7
STO	040	4	002	7
ARRIBA	080	1	040	3
IZQUIERDA	040	2	040	2
DERECHA	040	0	040	0
ABAJO	040	1	040	1
DROP	010	0	001	6
SIN	008	4	004	5
COS	020	4	002	5
TAN	020	3	001	5
SQRT	020	2	008	5
Y^X	020	1	010	5
1/X	020	0	002	4
+/-	010	3	008	4
EEX	010	2	010	4
ENTER	010	4	001	0
ALPHA	008	5	080	3
L SHFT	004	5	080	2
R SHFT	002	5	080	1
ON	ANY	15	ANY	15
SPC	001	1	002	0
TECLA 0	001	3	008	0
TECLA 1	002	3	008	1
TECLA 2	002	2	004	1
TECLA 3	002	1	002	1
TECLA 4	004	3	008	2
TECLA 5	004	2	004	2
TECLA 6	004	1	002	2
TECLA 7	008	3	008	3
TECLA 8	008	2	004	3
TECLA 9	008	1	002	3
TECLA .	001	2	004	0
TECLA +	001	0	001	1
TECLA -	002	0	001	2
TECLA *	004	0	001	3
TECLA /	008	0	001	4

TECLAS	HP48		HP49	
	OUT hex	IN hex	OUT hex	IN hex
MTH	004	0010	Teclas solo disponibles en la HP48	
PRG	080	0010		
CST	080	0008		
TECLA '	001	0010		
EVAL	040	0008		
DEL	010	0002		
APPS	Teclas solo disponibles en la HP49		020	0080
MODE			010	0080
TOOL			008	0080
HIST			010	0040
CAT			008	0040
EQW			004	0040
SYMB			002	0040
TECLA X			004	0010
F1	002	0010	020	0001
F2	100	0010	020	0002
F3	100	0008	020	0004
F4	100	0004	020	0008
F5	100	0002	020	0010
F6	100	0001	020	0020
VAR	080	0004	004	0080
NXT	080	0001	001	0080
STO	040	0010	002	0080
ARRIBA	080	0002	040	0008
IZQUIERDA	040	0004	040	0004
DERECHA	040	0001	040	0001
ABAJO	040	0002	040	0002
DROP	010	0001	001	0040
SIN	008	0010	004	0020
COS	020	0010	002	0020
TAN	020	0008	001	0020
SQRT	020	0004	008	0020
Y^X	020	0002	010	0020
1/X	020	0001	002	0010
+/-	010	0008	008	0010
EEX	010	0004	010	0010
ENTER	010	0010	001	0001
ALPHA	008	0020	080	0008
L SHFT	004	0020	080	0004
R SHFT	002	0020	080	0002
ON	ANY	8000	ANY	8000
SPC	001	0002	002	0001
TECLA 0	001	0008	008	0001
TECLA 1	002	0008	008	0002
TECLA 2	002	0004	004	0002
TECLA 3	002	0002	002	0002
TECLA 4	004	0008	008	0004
TECLA 5	004	0004	004	0004
TECLA 6	004	0002	002	0004
TECLA 7	008	0008	008	0008
TECLA 8	008	0004	004	0008
TECLA 9	008	0002	002	0008
TECLA .	001	0004	004	0001
TECLA +	001	0001	001	0002
TECLA -	002	0001	001	0004
TECLA *	004	0001	001	0008
TECLA /	008	0001	001	0010

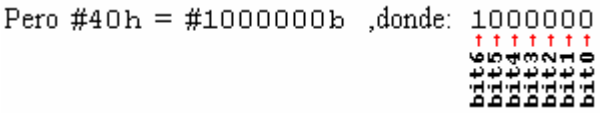
Nosotros usamos INTOFF para deshabilitar las interrupciones del teclado, e INTON para volver a habilitarlas, siempre que usemos INTOFF no nos olvidemos de utilizar INTON en el final del código.

Cuando deseamos probar una sola tecla, bastara con verificar el bit correspondiente a dicha tecla.

Por ejemplo, los valores de entrada y salida de la te tecla [DROP] son:

OUT	IN
001	0040

Entonces si deseamos verificar la tecla [DROP], tendremos que enviar el valor #001h al registro **OUT**, y después leer el valor del registro **IN**, si el valor de **IN** es #40h, entonces esto nos indicara que se presiono la tecla DROP.



Entonces tan solo se tiene que comprobar si el bit 6 es igual a uno o cero.

-Si es uno, entonces la tecla se ha presionado.
-Si es cero, entonces la tecla no se ha presionado y tengo que escanear otra vez.

Nosotros ya vemos de acuerdo a nuestra conveniencia el tipo adecuado a usar.

Ahora veamos un programa, en el que no se podrá salir, hasta que presionemos la tecla [DROP].

CODIGO PARA MASD

```
CODE
 INTOFF                 % Deshabilitamos las interrupciones del teclado
*LOOP
 LC 001                 % 001 viene a ser el valor a cargar en OUT.
 OUT=C                  % cargamos 001 en OUT
 C=IN                   % C va ha ser igual al valor de IN (tecla presionada)
 ?CBIT=1 6   ->EXIT     % Si se presiona [DROP] entonces salimos.
 GOTO LOOP              % de lo contrario volvemos a escanear
*EXIT
 INTON                  % habilitamos las interrupciones del teclado
 RPL                    % Salimos al RPL
ENDCODE
@
```

Nosotros podemos reemplazar: `OUT=C  C=IN`
por la rutina en `ROM` *`GOSBVL OUTCINRTN`* o por su equivalente en MASD *`OUT=C=IN`*

Veamos un ejemplo en donde no se podrá salir del programa hasta que se presione la tecla [S].

CODIGO PARA MASD

```
CODE
 INTOFF              % Deshabilita  las interrupciones del teclado
*TECLAS
 LC 004              % Valor de entrada de la tecla [S]
 GOSBVL OUTCINRTN    % Hacemos  OUT=C   C=IN
 ?CBIT=1 5           % Valor de salida de la tecla [S]
 GOYES SALIR         % Si se presiona la tecla [S], entonces salimos
 GOTO TECLAS         % De lo contrario seguimos escaneando
*SALIR
 INTON               % Habilita las interrupciones del teclado
 RPL                 % Vuelve al RPL
ENDCODE
@
```

EJEMPLO ESPERAR UNA TECLA

Este programa espera que se presione una tecla y devuelve el valor del registro **IN.**

CODIGO PARA MASD

```
CODE
 SAVE
 LC 0FF      % Necesitamos poner el registro OUT con ocho unos (allkeys).
 OUT=C       % Guardamos el valor en OUT
 A=0 A       % Borramos 5 nibbles de A, por que el registro IN  es solo de 4 nibbles
*MAIN
 GOSBVL AINRTN          % leemos el registro IN
 ?A=0 X -> MAIN         % Si no se presiona una tecla volvemos a escanear
 GOSBVL Flush           % vaciamos el buffer del teclado
 GOVLNG PUSH#ALOOP      % ponemos el número en la pila
ENDCODE
@
```

Si deseamos comprobar, si se ha presionado cualquier tecla, tenemos que hacer un **OR** de todos los valores de salida, y después comprobamos si el valor leído desde el teclado es diferente de cero.

Un **OR** de todos los valores de salida es:
#1h OR #2h OR #4h OR #8h OR #10h OR #20h OR #40h OR #80h, esto es igual a: **#FFh**

```
CODE
 INTOFF
*LOOP
 LC 0FF                 % Valor para todas las teclas, (allkeys)
 OUT=C=IN
 ?C=0 A ->LOOP          % Si IN es cero, entonces no se presiono ninguna tecla.
 INTON
 RPL
ENDCODE
@
```

Como se puede ver al ejecutar el código, tan pronto como lancemos el programa, este sale, para solucionar esto, tenemos que limpiar el almacenador intermediario del teclado (buffer) antes que comencemos el bucle. El siguiente código limpiará el buffer del teclado:

```
D1= 80669       % KEYBUFFER
C=0 A           % Vamos a limpiar el buffer del teclado
DAT1=C A        % Limpiamos el buffer del teclado
```

O también podemos usar una rutina en ROM, que realiza lo mismo.

```
GOSBVL Flush
```

Como podemos observar, se utiliza D1, entonces debemos guardar su valor antes:

```
CODIGO PARA MASD
CODE
 SAVE                  % Guardamos D0, D1, Ba, Da
 INTOFF
 GOSBVL Flush          % Vaciamos el buffer del teclado
*LOOP
 LC 0FF                % valor para todas las teclas
 OUT=C=IN
 ?C=0 A  ->LOOP        % Si no se presiono ninguna tecla, volvemos a escanear
 INTON
 LOADRPL               % Restauramos D0, D1, Ba, Da, y salimos al RPL
ENDCODE
@
```

DETECTAR MULTIPLES TECLAS

Cuando se quiere detectar más de una tecla presionada al mismo tiempo (teclas múltiples), tenemos que realizar un **OR**, tanto con los valores de entrada, como con los valores de salida, el resultado de esta operación serán los valores ha cargar en **OUT**, y en **IN**.

Para entender mejor veamos un ejemplo.
Vamos a escribir un programa que salga si presionamos las teclas [1] y [9] al mismo tiempo.

Los valores de salida y entrada:

TECLAS	OUT	IN
[1]	008	0002
[9]	002	0008

El valor de salida **(OUT)** es encontrado realizando un **OR**:

TECLAS	OUT
[1]	008h = 1000b
[9]	002h = 0010b
	OR = 1010b

Aplicando **OR** obtenemos: #1010b = #Ah

El valor de entrada **(IN)** también es encontrado realizando un **OR**:

TECLAS	IN
[1]	0002h = 0010b
[9]	0008h = 1000b
	OR = 1010b

Aplicando **OR** obtenemos: #1010b = #Ah

Entonces nuestro código es:

```
CODIGO PARA MASD
CODE
 INTOFF              % Deshabilita las interrupciones del teclado
*LOOP
 LC 00A              % Valor a cargar en OUT
 OUT=C=IN            % Hacemos  OUT=C  C=IN
 LA 000A             % Valor a comprobar en IN
 ?C#A A              % Si IN es diferente a A, entonces no se presionaron las teclas
 GOYES LOOP          % y volvemos a escanear
*EXIT                % de lo contrario salimos
 INTON               % Habilita las interrupciones del teclado
 RPL                 % Salimos al RPL
ENDCODE
@
```

NOTA:
No nos olvidemos que el valor de **IN** se almacena en **C**, es por eso que al hacer `?C#A A,` estamos leyendo el valor de **IN** que se almaceno en **C**, y lo comparamos con el valor de **A**.

Ahora hagamos un programa que salga si se presionan al mismo tiempo las teclas [SPC], [8] y [4].

Hallando los valores de entrada y salida.

TECLAS	OUT	IN
[SPC]	002h = 0010b	0001h = 0001b
[8]	004h = 0100b	0008h = 1000b
[4]	008h = 1000b	0004h = 0100b
	OR = 1110b	**OR** = 1101b

Aplicando **OR** con los valores de **OUT** obtenemos: #1110b = #Eh
Aplicando **OR** con los valores de **IN** obtenemos: #1101b = #Dh

```
CODE
 INTOFF                 % Deshabilita las interrupciones del teclado
*LOOP
 LC 00E                 % Valor a cargar en OUT
 OUT=C=IN               % Hacemos  OUT=C  C=IN
 LA 000D                % Valor a comprobar en IN
 ?C#A A                 % Si IN es diferente a A, entonces no se presionaron las teclas
 GOYES LOOP             % y volvemos a escanear
*EXIT                   % de lo contrario salimos
 INTON                  % Habilita las interrupciones del teclado
 RPL                    % Salimos al RPL
ENDCODE
@
```

Existen otras maneras de comprobar que se presionaron varias teclas al mismo tiempo, otra forma es comprobar cada tecla individualmente, una después de otra, como el lenguaje maquina va tan rápidamente, detectará teclas múltiples.

El programa que sigue, no sale hasta que se presionen las teclas [EEX] y [ENTER]. Primero, esperamos [EEX], y tan pronto se detecta, comprobamos para [ENTER].

TECLAS	OUT	IN
[EEX]	010	0010
[ENTER]	001	0001

```
CODE
 INTOFF
*LOOP
 LC 010                 % valor OUT para [EEX]
 OUT=C=IN
 LA 0010                % valor IN para [EEX]
 ?C#A A  -> LOOP        % Si [EEX] no se presiona, seguimos escaneando.
 LC 001                 % OUT para [ENTER]
 OUT=C=IN
```

```
 ?CBIT=0 0 ->LOOP      % Si [ENTER] no se presiona, volvemos a escanear.
 INTON                 % restauramos las interrupciones
 RPL                   % Salimos al PRL
ENDCODE
@
```

LA TECLA [ON]

La tecla [ON] es especial, si deseamos probar la tecla [ON], INTOFF no es suficiente, tenemos que inhabilitar todas las interrupciones, para eso utilizamos el flag 15.

```
ST=0 15          %Inhabilitamos todas las interrupciones.
ST=1 15          %Habilitamos todas las interrupciones.
```

Debemos tener cuidado al usar ST=0 15, porque una vez que se haga, ya no se podrá hacer *ON-C* o *ON-A-F* y si el programa se encierra en un bucle infinito, entonces se tendrá que resetear la HP usando el agujero ubicado debajo de la HP.

Para leer la tecla [ON], tenemos que hacer C=IN y entonces comprobar si el bit 15 es cero, si es así, entonces la tecla [ON], no se ha presionado.

Código para verificar la tecla [ON]:

```
CODE
 ST=0 15          % deshabilitamos todas las interrupciones
*LOOP
 C=IN
 ?CBIT=0 15       % Si el bit 15 es cero, entonces [ON] no se ha presionado.
 GOYES LOOP       % y volvemos a escanear, de lo contrario
 ST=1 15          % habilitamos todas las interrupciones
 RPL              % Salimos al RPL
ENDCODE
@
```

39 Manipulando la Pila

LA PILA (STACK)
Ahora que ya conocemos para que sirven cada una de las instrucciones ahora se vera como trabaja la pila. Se profundizara un poco en este tema para que sirva como introducción para los temas posteriores.

Siempre que se inicia un programa en ML, el puntero D1 apunta al primer nivel de la pila, pero lo que en verdad tenemos en la pila no son objetos, lo que tenemos en verdad son las direcciones de dichos objetos y como sabemos cada dirección ocupa 5 nibbles, esto quiere decir que si nosotros queremos avanzar al segundo nivel de la pila tendremos que sumar 5nibbles al puntero D1 para que este apunte a la siguiente dirección.

Veamos un cuadro referido a este caso a continuación:

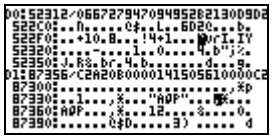
D1 + 15	00000	5 nibbles
D1 + 10	Dirección 3ro objeto	5 nibbles
D1 + 5	Dirección 2do objeto	5 nibbles
D1	Dirección 1er objeto	5 nibbles

Como se puede observar en el cuadro cada vez que sumamos 5 nibbles a D1 este avanza un nivel en la pila, si nosotros quisiéramos retroceder un nivel tendríamos que restar 5 nibbles en ese caso. También se puede observar que en el 4 nivel de la pila la dirección es 00000 esto nos indica que en el 4to nivel no se encuentra ningún objeto.

Ahora si nosotros quisiéramos modificar un objeto tendríamos que hacer que D1 apunte a la dirección del objeto que nosotros queremos modificar. Por ejemplo si nosotros quisiéramos modificar el objeto del nivel 2 de la pila tenemos que hacer que D1 apunte a dicha dirección.

El gráfico que se observa a continuación contiene las direcciones del D1.

Como se observa en la grafica D1=DBFF0/08378653786B37200000…
en donde DAT1=08378653786B37200000… siendo cada 5 nibbles una dirección.
65378 viene a ser la dirección del objeto que se encuentra en el 2do nivel de la pila, como nosotros queremos modificar dicho objeto tenemos que hacer que D1 apunte a esa dirección, para eso tendríamos que sumar 5 nibbles a D1 para avanzar al 2do nivel. (D1=D1+5) una vez realizado esto D1=DBFF5/653786B37200000… aquí nosotros ya nos encontramos en el 2do nivel, ahora nosotros tenemos que hacer que D1=65378 para lograr esto nosotros copiamos los 5 datos de DAT1 a un registro (A=DAT1.A) y después hacer que D1 apunte a esos datos para eso intercambiamos registros (AD1EX) ahora nos queda:



como vemos **D1**=87356/C2A20B00001415…. y ahora **DAT1**= C2A20B00001415….
Ahora modificando **DAT1** es que nosotros modificamos el objeto.

Ejemplo DROP:

Este ejemplo realiza un DROP, asegúrese de tener un objeto antes de ejecutar el programa.

CODIGO PARA MASD

```
CODE                %Inicia un programa en assembler
 D1=D1+5            %Subimos al 2do nivel de la pila.
 D=D+1 A            %Liberamos una dirección de memoria. (5 nibbles)
 RPL                %Regresamos al modo RPL
ENDCODE             %Termina un programa en assembler
@                   %Indica que el código es para compilar con MASD
```

Nota: *En realidad en el primer paso ya esta borrado el objeto, pero tenemos que indicar a la HP que un objeto a sido borrado, por eso es que tenemos que liberar una dirección de la memoria.*

Ejemplo DROP con TEST:

Este ejemplo primero verifica que exista un objeto y si es que existe realiza un DROP de lo contrario termina el programa.

CODIGO PARA MASD

```
CODE
 C=DAT1 A           %Copiamos en C la dirección del objeto del 1er nivel de la pila
 ?C=0 A -> FIN      %Si C es igual a cero salta a la etiqueta FIN, de lo contrario continua.
 D1=D1+5            %Subimos al 2do nivel de la pila.
 D=D+1 A            %Liberamos una dirección de memoria. (5 nibbles)
*FIN                %Etiqueta FIN
 RPL                %Regresamos al modo RPL
ENDCODE
@
```

Ejemplo SWAP:

CODIGO PARA MASD

```
CODE
 A=DAT1 A           %Copiamos en A la dirección del objeto del 1er nivel de la pila
 D1=D1+5            %Subimos al 2do nivel de la pila.
 C=DAT1 A           %Copiamos en C la dirección del objeto del 2do nivel de la pila
 DAT1=A A           %Pegamos la dirección del objeto del 1er nivel de la pila
 D1=D1-5            %Regresamos al 1er nivel de la pila
 DAT1=C A           %Pegamos la dirección del objeto del 2do nivel de la pila
 RPL                %Regresamos al modo RPL
ENDCODE
@
```

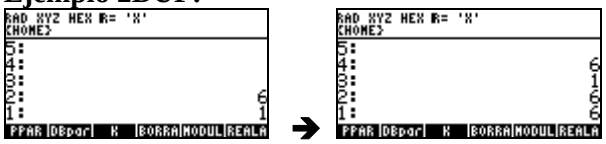
Ejemplo DUP:

Este ejemplo realiza un DUP, asegúrese de tener un objeto antes de ejecutar el programa.

CODIGO PARA MASD

```
CODE
 A=DAT1 A          %Copiamos en A la dirección del objeto del 1er nivel de la pila
 D=D-1 A           %Reservamos una  dirección de memoria. (5 nibbles)
 D1=D1-5           %Bajamos un nivel en la pila.
 DAT1=A A          %Pegamos  la dirección del objeto del 1er nivel de la pila
 RPL               %Regresamos al modo RPL
ENDCODE
@
```

Ejemplo 2DUP:



Este ejemplo realiza un doble DUP pero del objeto que se encuentra en el 2do nivel de la pila, asegúrese de tener 2 objetos antes de ejecutar el programa.

CODIGO PARA MASD

```
CODE
 D1=D1+5           %Subimos al 2do nivel de la pila.
 A=DAT1 A          %Copiamos en A la dirección del objeto del 2do nivel de la pila
 D=D-2 A           %Reservamos dos  direcciones de memoria.
 D1=D1-10          %Bajamos dos niveles en la pila.
 DAT1=A A          %Pegamos  la dirección del objeto del 2do  nivel de la pila
 D1=D1-5           %Bajamos otro nivel.
 DAT1=A A          %Pegamos  la dirección del objeto del 2do  nivel de la pila
 RPL               %Regresamos al modo RPL
ENDCODE
@
```

EJEMPLO STACK

Entrada: Tres Enteros Binarios (#a #b #c) Salida: #a #b #c (#a+#b+#c) 2*(#a+#b+#c)

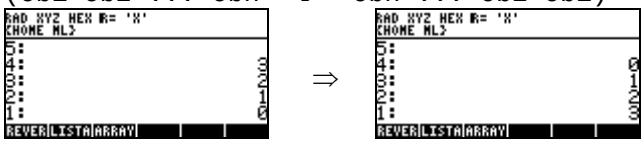
CODIGO PARA MASD

```
CODE
 GOSBVL   SAVPTR
 C=0   A           %Para el resultado, la suma de los bints se guardará en C(A)
 GOSUB    ADDSUB   %Salta a la etiqueta ADDSUB
 D1=D1+   5        %Nivel 2
 GOSUB    ADDSUB
 D1=D1+   5        %Nivel 3
 GOSUB    ADDSUB
 R0=C     A        %copia el resultado de C(A) a R0(A)
 GOSBVL   PUSH#    %hace GETPTR, coloca R0(A) como # en la pila
 GOSBVL   SAVPTR   %Se debe hacer SAVEPTR después de cada PUSH#
```

```
 C=R0     A          %Copia en C el resultado de R0
 C=C+C    A          %multiplica por 2
 R0=C     A
 GOSBVL   PUSH#      %Coloca el resultadox2 a la pila
 GOSBVL   SAVPTR
 GOVLNG   GETPTRLOOP   %Regresa al RPL, continua después de ENDCODE
*ADDSUB
 A=DAT1   A          %Llama al puntero del objeto de la pila.
 D0=A                %colocamos el puntero en D0.
 D0=D0+   5          %Saltamos el prologo.
 A=DAT0   A          %Lee el valor, apuntado por D0
 C=C+A    A          %suma ese valor al resultado en C(A)
 RTN                 %Retorna de donde fue llamado ADDSUB
ENDCODE
@
```

EJEMPLO REVERSYM

(ob1 ob2 ... obn ➔ obn ... ob2 ob1)



CODIGO PARA MASD

```
CODE
 SAVE
 CD1EX
 D1=C              %D1 apunta al primer nivel de la pila
 D0=C              %D0 apunta al primer nivel de la pila
 A=DAT0 A
 ?A=0 A -> EXIT    %Si es 0 no hay objetos en la pila y sale
*TOP
 D1+5              %Sube un nivel en la pila
 A=DAT1 A
 ?A#0 A -> TOP     %Termina cuando no exista ningún elemento en la pila
 D1-5              %Baja un nivel y este es el objeto mas alto en la pila
*MAIN
 A=DAT1 A          %Lee el objeto mas alto (nivel n)
 C=DAT0 A          %Lee el objeto mas bajo (nivel 1)
 DAT1=C A
 DAT0=A A
 D1-5              %apunta al nivel (n-1)
 D0+5              %apunta al nivel 2
 AD0EX
 D0=A
 CD1EX
 D1=C
 ?A<C A -> MAIN
*EXIT
 LOADRPL
ENDCODE
@
```

Con estos ejemplos ya queda demasiado claro como es el funcionamiento de la pila en ML.

39.1 Ejemplo de temporizador

```
CODE
 SAVE INTOFF2
*again
 D1=(5) TIMERCTRL.1
 LC 4 DAT1=C.1
 D1=(2) TIMER1 LC(1)8-1
 DAT1=C.1
 SHUTDN
 D1=(2) ANNCTRL C=DAT1.B C=-C-1.P DAT1=C.P
 GOSBVL OnKeyDown?
 GONC again
 INTON2 LOADRPL
ENDCODE
@
```

Cuando colocamos #4h en **TIMERCTRL.1** activamos el temporizador, luego tengamos en cuenta lo siguiente: **TIMER1** decrementa 16 veces cada segundo. Entonces si deseamos crear un temporizador de medio segundo, colocamos en **TIMER1** el valor de (8-1) que sería: 8/16 = 0.5 segundos.
(Si colocamos 16-1 en **TIMER1**, tendremos un temporizador de 16/16 = 1 segundo)
SHUTDN duerme la HP, hasta que ocurra alguna interrupción, la cual se suscitará cada medio segundo (el valor del temporizador)....

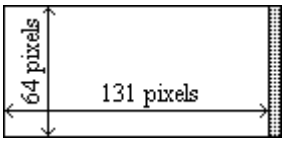
40 Gráficos

La HP49G, nos brinda facilidades para el entorno gráfico, en lenguaje ensamblador, usaremos ciertas direcciones RAM, o sólo comandos, veremos también la forma de cómo conseguir las escalas de grises, de una manera simple.

40.1 Píxeles y octetos

La pantalla de la HP49G esta conformada por varios elementos unitarios llamados píxeles. Un píxel puede estar encendido (Negro) o apagado (Blanco).

La resolución de la pantalla es de 131 píxeles de largo por 64 píxeles de alto, siendo:
131x64 = 8384 píxeles en total.



El número **1** se va a asignar para indicar encendido (Negro) y el **0** para indicar apagado (blanco).
Por ejemplo: 10101100 seria igual a: 

Cada elemento unitario **0** o **1** se llama a un BIT.
Un grupo de cuatro bits forman un cuarteto o dos pares un octeto.
La HP trabaja con octetos en la pantalla, un octeto esta formado por dos cuartetos.

40.2 Memoria gráfica

En realidad nuestra memoria gráfica esta conformada por 136x64 píxeles, también consideremos que internamente nuestras líneas siempre van ha ser múltiplos de 8, por lo tanto: 136/8 = 17 octetos por línea y 64 líneas, son un total de 1088 octetos,
(1 kiloocteto = 1024 octetos). Por eso la HP posee una zona de 1088 octetos

En la zona de la memoria de la pantalla se va a avanzar octeto por octeto
Aplicando una regla simple:

1: Enciende un píxel,
0: Apaga un píxel

Para borrar la pantalla de la memoria grafica se deberán formar octetos exclusivamente de 0: 00000000

Para encender la pantalla de la memoria grafica se deberán formar octetos exclusivamente de 1: 11111111

40.3 Utilización hexadecimal

Para escribir en la pantalla, vamos a optar por la notación de bits, para escribir 1088 cifras en la pantalla nosotros tenemos que utilizar la notación adecuada, en este caso la base hexadecimal

Consideremos un cuarteto: bbbb (b significa **BIT**). b puede ser el valor **0** o **1**, El número de combinaciones posibles es 2^4, que son 16 combinaciones posibles que se muestran en la siguiente tabla, recordemos que nosotros vamos a usar la notación hexadecimal

Decimal	Binario	Hexadecimal	Decimal	Binario	Hexadecimal
0	0000	0	8	1000	8
1	0001	1	9	1001	9
2	0010	2	10	1010	A
3	0011	3	11	1011	B
4	0100	4	12	1100	C
5	0101	5	13	1101	D
6	0110	6	14	1110	E
7	0111	7	15	1111	F

Esto nos permite cada combinación de píxeles de un cuarteto (1cuarteto = 1nibble)

Ejemplo:
0110 : 6 : 
1010 : A : 

Usted se preguntara como represento un octeto, pues es muy fácil, el octeto se comporta como dos cuartetos, entonces el ejemplo seria:

En octeto: 01101010 
Esta compuesto por 0110 mas 1010
0110 corresponde a **6** y 1010 corresponde a **A**. (Esto en hexadecimal)
Entonces: 01101010 vendría a ser **6A** : 

Otro ejemplo, 0100 mas 1101, es 01001101 : **4D** : 

Cuando queremos mostrar un grob o escribir algo en la pantalla la técnica consiste en apuntar a la dirección de la memoria grafica con la ayuda de los punteros **D0** o **D1**,
(Ejemplo: **D0 = 8068D**)

Después leer la dirección apuntada, (Ejemplo: **A=DAT0 A**) para apuntar entonces a esta dirección lo hacemos con la ayuda de **D0** o **D1**, (Ejemplo: **D0=A)**

Después se recolectan los datos con la ayuda de los registros de trabajo **A(A)** o **C(A)**, esto se puede hacer por ejemplo con **A=DAT0 A** (el Campo **A** indica que uno quiere recuperar los primeros 5 nibbles, *(5x 4 = 20 píxeles)*).

En el ejemplo que veremos a continuación se puede hacer lo siguiente:

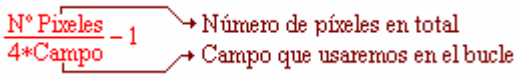
LA 0000000000000000 se puede reemplazar por LA FFFFFFFFFFFFF.

También se puede intercalar los píxeles  (valor 1010 = A), en ese caso se puede reemplazar LA 0000000000000000 por LA AAAAAAAAAAAAAAAA.

Supongamos que se quiera escribir lo siguiente: 

Se tendrá que descomponer en cuartetos
1101 = D
0110 = 6
1010 = A
1011 = B
Resultaría 1101011010101011 que corresponde a D6AB.
Entonces se reemplazaría LA 0000000000000000 por LA D6ABD6ABD6ABD6AB

También en ocasiones vamos a tener que generar bucles, para escribir en la pantalla, para saber cuantas veces se tiene que repetir el bucle, usamos la siguiente formula:



40.4 Borrando la pantalla

En modo normal, el menú se encuentra activo, usando 131 píxeles de ancho, con 8 píxeles de alto.

Veamos un ejemplo de cómo borrar la pantalla.

EJEMPLO:

CODIGO PARA MASD

```
CODE
 SAVE                    % Guardamos D0, D1, Ba, Da
 D0=8068D                % Recupera la dirección de la memoria grafica
 C=DAT0 A                % C apunta a la memoria grafica
 D0=C                    % D0 contiene la dirección de la memoria grafica
 LA 0000000000000000     % A  contiene 16 cuartetos de valor 0
                         % Son 16*4=64 BITs Por eso va a borrar
                         % una serie de 64 píxeles
 LC(2) 136*64/(4*16)-1   % Para escribir Aw 136 veces en D0
                         % (Pantalla: 136*64 píxeles, y Aw: 16 nibbles
                         % (136*64)/(4*16)=136, en hexadecimal = 88
                         % menos 1 porque la HP empieza a contar a 0.
{                        % Iniciamos un bucle
 DAT0=A 16               % Copiamos datos en  la memoria grafica, con el contenido de A
 D0=D0+16                % El puntero avanza 16 nibbles en la memoria grafica
 C=C-1 B                 % Decrementa el contador
 UPNC                    % Recomienza el bucle si C es mayor o igual a 0
}                        % Fin del bucle
 LOADRPL                 % Recuperamos D0, D1, Ba, Da,  y salimos
ENDCODE
@
```

A **D0** se le puede sumar o restar máximo 16, si se quiere sumar mas se tendrá que hacer por partes como se vera mas abajo.
Cuando se suma al indicador del registro avanza hacia la derecha y si se le resta avanza hacia la izquierda, Si se quiere ir una línea hacia abajo (136 píxeles), se tendrá que incrementar el registro en 34 nibbles. 34x4=136 píxeles y si se quiere subir se tendrá que hacer lo contrario.
Atención: #34d => #22h

Eso se haría de la siguiente manera con el registro D0:

Si se quiere bajar: (Sumamos #34d)

```
D0=D0+ 16
D0=D0+ 16
D0=D0+ 2
```

O si se quiere subir: (Restamos #34d)

```
D0=D0- 16
D0=D0- 16
D0=D0- 2
```

Otro método más eficaz sería intercambiar el contenido del puntero del registro de trabajo, para modificar este registro de trabajo y para intercambiar el contenido del registro se haría lo siguiente:

EJEMPLO:

```
D0= 8068D   %Dirección de la pantalla grafica
A=DAT0 A    %Grabamos en A(A) la dirección de la pantalla grafica
D0=A        %Apuntamos a esa dirección
AD0EX       %Coloca la dirección de D0 en A(A))
LC 00022    %Carga el registro C con #22h una línea
A=A+C A     %Se desplaza  una línea hacia abajo
AD0EX       %Reemplaza la nueva dirección en D0
```

EJEMPLO:

Este ejemplo limpia la pantalla realizando un barrido.

```
CODE
 GOSBVL 0679B      %SAVPTR  guarda los registros D0, D1, Ba, Da
 GOSBVL 2677C      %D0->Row1  apunta a la dirección de la pantalla grafica
 LC(5)  64*34-1
 B=C    A
 A=0    P
*sgte_nibble
 DAT0=A P
 D0=D0+ 1
 LC(5)  $02000     %Generamos un retardo
*retardo
 C=C-1  A
 GONC   retardo
 B=B-1  A
 GONC   sgte_nibble
 GOVLNG 05143      %GETPTR  recupera los registros D0, D1, Ba, Da,  y sale
ENDCODE
@
```

40.5 Como mostrar un grob en toda la pantalla

Primero que nada miremos un GROB, un GROB es un objeto gráfico conseguido con la ayuda del ambiente PICTURE de la HP49. En este objeto se almacenan todos los datos acerca del diagrama. Es por consiguiente que cuando nosotros leemos estos datos es que uno puede desplegar un GROB en la pantalla

Cuando uno obtiene un diagrama de este entorno lo que obtiene en la pila es un
GROB DE 131x64 píxeles

Que al editarlo adopta la siguiente forma:
GROB 131 64 123454100….000

Entonces para usarlo en ML usted lo tiene que convertir a :
$1234541000....000

Como se a dado cuenta se tiene que eliminar el prologo GROB 131 64 que es inservible en ML y reemplazarlo por el carácter **$**

Ahora existen varios métodos para mostrar el Grob a continuación se verán 2 métodos:

Método 1

```
GOSUBL  Etiqueta1          %Uno pone el principio del grob en  RSTK
$0000000322514052314 30    %Este es el código del grob
*Etiqueta1
C=RSTK                     %Aquí uno recupera en C(A) la dirección del  GROB
                           %A la salida C(A) contiene la dirección del principio del GROB.
                           %Entonces uno puede guardar esta dirección (En R0 por ejemplo)
                           %o para apuntar directamente con la ayuda de los indicadores D0
                           %o D1. (Ejemplo: D0=C o D1=C).
```

Método 2

El segundo método consiste leyendo directamente un GROB desde la pila, y mostrarlo en la pantalla. Para esto es necesario que el principio del programa contenga el registro **D1**, porque normalmente este registro contiene la dirección del objeto que se encuentra en el primer nivel de la pila. Esto se puede hacer con la ayuda de la siguiente instrucción:

A=DAT1 A

Después obtenemos el puntero del GROB con la ayuda de la instrucción:

D1=A

En esta fase se tiene que efectuar la copia de los datos del GROB en los datos de la pantalla
A continuación se vera un programa que muestra un GROB de 131 * 56 en la pantalla:

EJEMPLO GROB

CODIGO PARA MASD

```
CODE
 SAVE                  % Guardamos D0, D1, Ba, Da
 GOSUBL Inicio         % Salta a Inicio y el dato que seguía se va a RSTK
 $12345411534...012    % Aquí va el código del grob
 *Inicio
 C=RSTK                % ahora C  contiene los datos del GROB
 D1=C                  % D1 apunta al GROB a mostrar
 D0= 8068D             % Recupera la dirección de la memoria grafica
 A=DAT0 A              % Ahora A apunta a la  memoria grafica
 D0=A                  % D0 contiene la dirección de la memoria grafica
 C=0 W                 % Borramos el registro C.
 LC 76F                % 56 líneas para mostrar: 56*136=7616 píxeles, son 1904
                       % nibbles, o  #770h. #770-1 = #76Fh.
*Datos
 A=DAT1 A              % Copia los datos del GROB apuntados por el registro D1 en A(A)
 DAT0=A 1              % Copia cuarteto por cuarteto del GROB a la pantalla de
                       % la dirección contenida en el registro D0
 D0=D0+ 1              % mueve a la derecha un cuarteto en la pantalla
 D1=D1+ 1              % mueve a la derecha un cuarteto en el GROB
 C=C-1 A               % Si se terminan de copiar los datos sale después de Datos de lo
                       % contrario salta a la siguiente instrucción.
 GONC Datos            % Salta a Datos
 LOADRPL               % Recuperamos D0, D1, Ba, Da,  y salimos
ENDCODE
@
```

Nosotros podemos usar campos más grandes para que nuestro programa sea más rápido. (Ejemplo: Campos **B**, **X**, **W**)
de la manera siguiente:

```
A=DAT1 B
DAT0=A B
```

Aquí, los campos **B** reagrupan 2 nibbles, el despliegue va a tener lugar 2 veces más rápido, la condición para ajustar el contador del bucle:
(#770h 2) -1 = #3B7h

40.6 Como mostrar un símbolo gráfico

Se considera un símbolo gráfico, a aquel grob cuyo ancho es menor a 131 píxeles.

En primer lugar, es necesario señalar el ancho del grob en nibbles, es decir por cada grupo de 4 píxeles. (4, 8, 12, 16, 20, 24...), esto por una razón simple, esto corresponde al tamaño de campos del registro, por ejemplo, para un gráfico de 8x12 píxeles, uno usará el campo **B** para desplegar una línea del gráfico, debido a que el campo **B** es igual a 2 nibbles que es igual a 8 píxeles.

Supongamos que queremos mostrar el símbolo que se muestra en la figura:

Para hacer el despliegue de este gráfico, nosotros usaremos el campo **B,** ya que este campo es igual a 8 píxeles.

Para hacer esto, existen varios métodos, solo veremos 2 métodos.

1er Método

Para desplegar un símbolo gráfico, es necesario tomar en cuenta el tamaño, en primer lugar, si el ancho de nuestro gráfico es igual a 8 píxeles, nosotros usaremos el campo **B,** ya que este campo es el que mejor se adapta a la situación.

Nuestro gráfico va a tener una altura de 12 píxeles, o #Ch - 1 = #Bh.

Para una altura de 4 píxeles uno mueve dos nibbles:

0>4	píxeles:	2 nibbles
4>8	píxeles:	2 nibbles
8>12	píxeles:	4 nibbles
12>16	píxeles:	4 nibbles
16>20	píxeles:	6 nibbles
20>24	píxeles:	6 nibbles
24>28	píxeles:	8 nibbles
etc ...		

El largo de nuestro gráfico es 8 píxeles, entonces cada línea será igual a 2 nibbles = campo B.
Nosotros desplegaremos nuestro gráfico en la esquina superior izquierda de la pantalla.

```
CODE
 SAVE              % Guardamos D0, D1, Ba, Da
 GOSUBL gráfico
 $E7185A5A5ADBDB5A5A5A18E7      % Datos del gráfico (Grob)
*gráfico
 C=RSTK            % recupera la dirección del gráfico
 D1=C              % D1 apunta al gráfico
 SCREEN            % D0 apunta a la pantalla
 LC B              % Tiene 12 líneas, 12-1=11 o #Bh
*Datos
 A=DAT1 B          % Lee el gráfico. (una línea)
 DAT0=A B          % Lo copia en la pantalla (una línea)
 D0=D0+ 16
 D0=D0+ 16         % Pasa a la línea siguiente de la pantalla
 D0=D0+ 2
 D1=D1+ 2          % Pasa a la línea siguiente del gráfico
 C=C-1 B           % Decrementa el contador
 GONC Datos
 LOADRPL           % Recuperamos D0, D1, Ba, Da,  y salimos
ENDCODE
@
```

Método 2

Este método consiste en colocar la copia del símbolo gráfico encima de los datos de la pantalla. Para esto se usará la instrucción '**!**' llamado "y" lógico.
El campo **B** jugará el papel de contador del bucle

EJEMPLO:


```
CODE
SAVE                % Guardamos D0, D1, Ba, Da
GOSUBL grob
$E7185A5A5ADBDB5A5A5A18E7   % Datos del gráfico
*grob
C=RSTK              % Recupera la dirección del gráfico
D1=C                % D1 apunta al gráfico
SCREEN              % D0 apunta a la pantalla
C=0 W               % Borra el registro C
B=0 W               % Borra el registro B
LC B                % 12 líneas, 12-1=11 o #Bh (Carga en C la cantidad de píxeles del gráfico)
B=C B
*Datos
A=DAT1 B            % Lee el gráfico
C=DAT0 B            % Lee la pantalla
A=A!C B
DAT0=A B            % Copia en la pantalla
D0=D0+ 16
D0=D0+ 16           % Pasa a la línea siguiente de la pantalla
D0=D0+ 2
D1=D1+ 2            % Pasa a la línea siguiente del gráfico
B=B-1 B
GONC Datos          % 12 líneas para copiar
B=0 W               % Borra el registro B
LC 1FFFF            % Valor para generar un retardo
{ C-1.A UPNC }      % Generamos un retardo
LOADRPL             % Recuperamos D0, D1, Ba, Da, y salimos
ENDCODE
@
```

40.7 Como desplazar un símbolo gráfico

Para desplazar un grafico o que este cambie de posición en la pantalla, depende de la dirección en cuál apunta a la pantalla. Modificar esta dirección implica modificar la posición del desplazamiento del grafico y por consiguiente cambiarlo de sitio, es de antemano necesario hacer un resguardo de la dirección del grafico que debe desplegarse. Por ejemplo uno podría guardar esta dirección en un registro como **R0**.

```
SCREEN
AD0EX
R0=A
```

Entonces ahora se podría desplazar el gráfico agregando 1 a la dirección guardada en R0:

```
A=R0
A=A+1 A
R0=A
```

Nuestro gráfico cambiará de sitio un cuarteto (4 píxeles) a la derecha. Inversamente, para cambiar de sitio el gráfico un cuarteto a la izquierda:

```
A=R0
A=A-1 A
R0=A
```

Para mover el gráfico una línea en la pantalla hacia abajo es necesario agregar 34 (#22h) a esta dirección:

```
A=R0
A=A+16 A
A=A+16 A
A=A+2 A
R0=A
```

Inversamente para mover una línea hacia arriba se deberá restar 34:

```
A=R0
A=A-16 A
A=A-16 A
A=A-2 A
R0=A
```

Ahora veremos un ejemplo para entender mejor el funcionamiento:

```
CODE
 SAVE                  % Guardamos los registros D0, D1, Ba, Da
 ST=0.15               % Deshabilitamos las interrupciones
 GOSUBL Grob
```

```
 $E7185A5A5ADBDB5A5A5A18E7 %Datos del grob
*Grob
 C=RSTK    R1=C.A        % Guardamos la dirección del grob en R1
 D1=C                    % D1=dirección del grob
 SCREEN    R0=A.A        % Guardamos en R0 la dirección de la pantalla actual
*Move
 D0=A                    % D0 apunta a la pantalla actual
 LC(2) $B                % 12 líneas, 12-1=#11d = #Bh (C= #píxeles del grob)
 B=C B                   % B= contador (Generamos un bucle de 12 vueltas)
*Datos                   % Rutina para mostrar el grob en la pantalla
 A=DAT1 B                % Lee los datos del grob
 DAT0=A B                % y los copia en la pantalla
 D0=D0+ 34               % Pasa a la línea siguiente de la pantalla
 D1=D1+ 2                % Pasa a la línea siguiente del grob
 B=B-1 B                 % Decrementa el contador
 GONC Datos              % Vuelve a copiar
 B=0 W                   % Borra el registro B
*Keys                    % Rutina para escanear el teclado
 LC 040                  % Código de entrada de la tecla DOWN y RIGHT
 GOSBVL OUTCINRTN        % OUT=C, C=IN
 ?CBIT=1 1  ->DOWN       % Código de salida de la tecla DOWN
 ?CBIT=1 0  ->RIGHT      % Código de salida de la tecla RIGHT
 LC 001                  % Código de entrada de la tecla ENTER
 GOSBVL OUTCINRTN        % OUT=C, C=IN
 ?CBIT=1 0  ->EXIT       % Código de salida de la tecla ENTER
 GOTO Keys               % Vuelve a escanear el teclado
*RIGHT                   % Rutina para mover el grob a la derecha
 A=R0.A    A+1.A         % Recuperamos la pantalla actual y movemos hacia la derecha
*RUN                     % Muestra el grob desplazado en la pantalla
 R0=A.A                  % Guardamos la dirección de la pantalla desplazada
 GOSUB     Delay         % Generamos un retardo
 GOSUB     DGrob         % Leemos los datos del grob
 GOTO      Move          % Movemos el grob
*DOWN                    % Rutina para mover el grob hacia abajo
 A=R0.A    A+34.A        % Recuperamos la pantalla actual y movemos hacia abajo
 GOSUB     RUN
*EXIT                    % Rutina para salir al RPL
 ST=1.15                 % Habilitamos las interrupciones
 LOADRPL                 % Recuperamos los registros D0, D1, Ba, Da y salimos al RPL
*DGrob
 C=R1.A    D1=C          % Recuperamos la dirección del grob en D1.
 RTN                     % Retorna al lugar donde fue llamado
*Delay                   % Rutina para generar un retardo
 LC(5) $FFF              % Valor para generar un retardo
 { C-1.A UPNC }          % Generamos un retardo
 RTN
ENDCODE
@
```

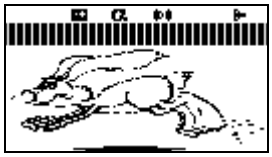
40.8 Animaciones gráficas

Se pueden generar animaciones, si colocamos en la pantalla gráficos consecutivos, de tal manera que den el "efecto" de movimiento (así es como funcionan los gif's animados).

La mejor manera para generar animaciones, consiste en tener todos los grobs que se quieren mostrar, en uno solo grob, y después ir mostrándolos consecutivamente.

En el ejemplo que veremos usamos un grob de 131 x 424, donde se encuentran 8 grobs, cada grob tiene una dimensión de 131x53 píxeles (53x8=424 pixeles)

EJEMPLO ANIMACION RABBIT:



```
!ASM
% Declaración de variables
DC LNC      128          %Line Counter
DC ADDR     120          %display ADDR
DC RDisp    1A45         %Restaura el display
DC oneline  #34          %cantidad de nibbles para avanzar una línea (131 píxeles)
DC bodies   #8           %grobs 131*53
DC onebody  #53*online   % cantidad de nibbles para avanzar un grob (131x53 píxeles)

STROBJ $02DCC
{
 ST=0.15                 %  Inhabilitamos las interrupciones
 SAVE                    %  guardamos registros

 SKUBL { /rab.gro }      %  aquí se incluye el gráfico almacenado en rab.gro
                         %  RSTK contiene la dirección del grob general  (131x424)
 SCREEN                  %  D0 y A ➔ pantalla

% Apagando el menú
 D1=(5)LNC               %  (D1➔ LINECOUNT)
 LC(2)64-1               %  Vamos usar toda la pantalla
 DAT1=C.B                %  Apagamos el menú

% Dibujando líneas en pantalla
 D1+16                   %  (D1➔ TIMER2)
 C=DAT1.S                %  C(S)=Número al azar
```

```
 LC(5)oneline*11-1     % Usamos C como contador (Vamos a dibujar 11 líneas)
 {                     % Iniciamos un bucle
  DAT0=C.S             % Dibujamos en pantalla
  D0+1                 % Avanzamos a los siguientes píxeles
  C-1.A                % Decrementazos el contador
  UPNC                 % Sale del bucle si se termino de dibujar
 }                     % Fin del bucle

 CD0EX D=C.A           % Da=Pantalla en posición (0,11)
 C=RSTK CD0EX          % D0 ➔ grob1 grob2 …. grob8
 LC(3)$600 B=C.A       % Bx=valor inicial del retardo

*again
 C=0.S C+7.S           % Cs=grobs-1(Cs=7➔grob1, Cs=6➔grob2 ..... Cs=0➔grob8)
{
 C=D.A CD1EX           % D1 ➔ Pantalla en posición (0,11)
 LC(5)onebody COPYDN         %Copia un grob en la pantalla

 D1=(5)TIMER2 C=DAT1.B       %Cb=número aleatorio para encender indicadores
 D1=(2)ANNCTRL
 CBIT=1.7              % Siempre enciende los indicadores
 DAT1=C.B              % escribe el número con el bit 7 encendido

 LC(3)$800             % sonido si el grob es 0 o 7 (rabbit en el piso)
 A=0.S ?A#C.S { OUT=C }
 A+7.S ?A#C.S { OUT=C }

 A=B.A
  {
   LC(3)1 OUT=C=IN     % apaga el sonido y verifica el teclado
   ?CBIT=0.1  { B-1.A }      % tecla [+]
   ?CBIT=0.2  { B+1.A }      % tecla [-]
   ?CBIT=0.15 { GONC exit } % tecla [ON]
   A-1.X UPNC
  }

 C-1.S UPNC
}

 LC(5)onebody*8 AD0EX A-C.A AD0EX      %D0 ➔ grob1 grob2 ... grob8

 GOTO again

*exit
 GOSBVL RDisp
 ST=1. 15                    % habilitamos las interrupciones
 LOADRPL                     % restauramos los registros y salimos al RPL.
}
!RPL
@
```

40.9 Escala de grises

La HP normalmente difunde dos colores en pantalla: negro y blanco, los niveles de grises son el resultado de jugar con estos colores en la pantalla.
También la ROM de la HP49G, contiene rutinas que permiten fácilmente programar aplicaciones en cuatro niveles de grises.

Nosotros veremos como mostrar 4 colores en niveles de grises.
- Blanco □
- Gris claro □
- Gris oscuro ■
- Negro ■

Para obtener 4 escala de grises, simplemente debemos mostrar consecutivamente en la pantalla dos grobs de manera muy rápida superponiéndolos, el superponer un gráfico tras otro debe ser rápido, pero teniendo en cuenta que ya haya sido mostrado totalmente en la pantalla, es así que se genera el "efecto" de grises. En realidad, se utiliza el sistema booleano para regular los niveles de grises, como lo presenta el pequeño grafico:

Como se puede observar en el cuadro, es necesario mostrar una vez el primer grob y dos veces el segundo, esto continuamente.

Si por superposición un píxel se encuentra sobre el ***Grob N°1,*** y no se encuentra sobre el ***Grob N°2***, tendremos el gris claro. Si se aplica el mismo razonamiento para el gris oscuro, entonces el píxel se encontrará sobre el segundo grob. Por último, para hacer blanco, ningún píxel debe ser mostrado mientras que para el negro se debe mostrar dos píxeles.

En general, no se toman dos grobs diferentes, más bien un grob que contiene a los dos. Usaremos las direcciones de memoria: **00100**, **00120**, **00125**, **00128**.

Mayormente, usamos gráficos de 131x128 para obtener 4 grises. Un gráfico de 131x128 es como dos grobs 131x64 unidos verticalmente (64+64=128):

```
...GROB1
...GROB2
```

Pues bien, como el ancho no varía, usaremos 64. También recordemos que cuando capturamos el grob de la pila, hacemos algo como:

```
A=DAT1.A
D1=A
D1=D1+20
```

Con esto hacemos que **D1** apunte al cuerpo del grob **131x128**, que es el inicio del *GROB1*, entonces para conseguir el inicio del *GROB2*, debemos sumarle **64x34**, que es **64** de alto, por **34** de ancho.

Recordemos que 1 línea = 136 píxeles = 34 nibbles , si exactamente 136 píxeles y no 131 píxeles, solo q la HP difunde los 131 primeros píxeles.

Teniendo las direcciones del *GROB1* y *GROB2*, debemos displayarlos consecutivamente. Para displayar sólo basta con colocar estas direcciones en **#120** (=DISP1CTL).... lastimosamente se debe tener en cuenta que estas direcciones sean pares... pero porque?:

Al colocar la dirección de un grob en **#120**, el bit **0** es ignorado, entonces, coloquemos por ejemplo la dirección #00003, que es #000..0011 en binario. Si anulamos el bit 0 quedaría #000..0010... vemos que es como una resta. Si la dirección hubiera sido par, no se habría alterado.
En estos casos, para las direcciones impares.... se debe colocar en **#100** el valor **12** , y en **#125** el valor **#FFF** ...

En condiciones normales, la HP49G se encuentra con la siguiente configuración:

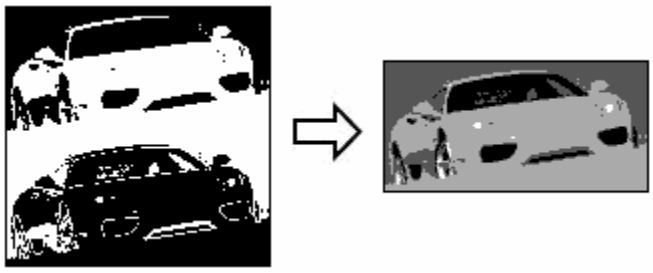
Dirección en memoria:	Conocido como:	Tiene el valor:
00100	BITOFFSET	8
00125	LINENIBS	000
00128	LINECOUNT	55

Nosotros, fácilmente podemos manipular las direcciones RAM, que son direcciones pares, por lo que ya no, nos preocuparemos si el gráfico es impar. Con *extable*, estas direcciones son conocidas como:

```
SCREEN1
SCREEN2
SCREEN3
```

Para mostrar una imagen en cuatro niveles de grises, de dimensiones 131x64 píxeles, es necesario utilizar un grob de dimensiones 131x128 píxeles, (el doble).

La imagen de la izquierda es el grob bruto, y el de la derecha es el resultado de superponer este grob, así conseguimos cuatro niveles de gris:



A continuación veremos 2 métodos de cómo poder mostrar escala de grises, el primero es el método tradicional (clásico), y el segundo, es un método usando la rutina en ROM que trae incorporado la HP49G.

MÉTODO CLÁSICO

Consiste en crear un subprograma que muestre los cuatro niveles de grises, este seguramente es el método más utilizado y más extendido. Nuestro ejemplo se basará en una simple imagen de dimensiones 131x64 (o sea 131x128 en tamaño real).

Al principio, procederemos a detectar si las dimensiones del grob presente sobre el nivel 1 de la pila corresponden a las dimensiones que deseamos. A continuación, es necesario regular los márgenes, ¿Qué es esto? En nuestro caso, los márgenes sirven para reajustar una imagen si se almacena en dirección impar, pero en otras aplicaciones permiten realizar magníficas presentaciones de manera muy elegante. Existen dos tipos de márgenes:

Margen Izquierdo

El margen izquierdo se expresa en bits, (3bits) en la dirección # 100h
Sirve para desplazar una imagen un píxel a la vez.

Margen Derecho

El margen derecho se expresa en cuartetos, (3 cuartetos) en la dirección # 125h
Permite desplazar una imagen cuatro píxeles a la vez (un cuarteto).

En lo que nos concierne, si el grob se coloca en dirección impar, es necesario desplazar la imagen un cuarteto hacia la izquierda de la pantalla.

Pasemos ahora a lo que se refiere mostrar el grob. Como se menciono mas arriba, tenemos que esperar que un grob sea mostrado totalmente para poder mostrar el siguiente, pero ¿Cómo sabemos que un grob ha sido mostrado totalmente?, bueno eso se va a conocer con el barrido de la pantalla, que se encuentra en la dirección #128h, para obtener el número de la línea en curso de barrido (*refresco o actualización de la pantalla*), tenemos que leer 6 bits (ejemplo: D0= 00128 A=DAT0 6). El registro **A** vale **0** cuando la última línea ha sido actualizada. Un barrido se efectúa en 1/64 de segundo.

Una vez que el primer grob ha sido mostrado totalmente, procedemos a mostrar el segundo grob, esperamos que se muestre totalmente y volvemos a mostrar el segundo grob.

A continuación se vera un ejemplo:

```
::
CK1&Dispatch grob
 ::
 TURNMENUOFF
  CODE
   SAVE INTOFF2              % Guardamos registros, inhabilitamos las interrupciones
   A=DAT1.A D1=A             % D1=Grob
   D1+5 LC 0110F A=DAT1.A    % A= tamaño del grob
   ?A#C.A ->FIN              % Si este no es el tamaño salimos, de lo contrario
   AD1EX A+15.A B=A.A        % se recupera la dirección del primer grob
   D1=00100                  % D1=BITOFFSET
   C=DAT1.1 R0=C.P           % Se guarda el margen izquierdo
   ?ABIT=0.0 ->BUCLE         % si el grob está en dirección impar
   LC C DAT1=C.1             % se desplaza el margen de la izquierda
   LC FFF D1=25 DAT1=C.X     % y el margen de la derecha
  *BUCLE                     % iniciamos un bucle
   GOSUB DISP                % llamamos al subprograma de visualización
   LC 00880 A+C.A            % avanzamos al segundo grob
   GOSUB DISP GOSUB DISP     % y lo visualizamos dos veces,
   A=B.A                     % se recupera la dirección del primer grob
   GOTO BUCLE                % y repetimos.
  *FIN2 D1=00                % D1=BITOFFSET
   C=R0.P DAT1=C.1           % recuperamos el margen izquierdo,
   D0=8068D A=DAT0.8         % la pantalla inicial
   D1=20 DAT1=A.8            % y el margen derecho
  *FIN INTON2                % volvemos a habilitar las interrupciones
   GOSBVL Flush              % vaciamos el buffer del teclado
   LOADRPL                   % recuperamos los registros y salimos al RPL.
  *DISP D0=00120 DAT0=A.A    % se muestra el grob que debe mostrarse
   D1=00129                  % #128=#3F ... entonces #129=#3
  *BAR1 C=DAT1.1             % Leemos el estado del barrido
   ?CBIT=1.1 ->BAR1          % verificamos el cambio #20-#1=#1F
  *BAR2 C=DAT1.1             % Leemos el estado del barrido
   ?CBIT=0.1 ->BAR2          % verifica el cambio #0-#1=#3F , barrido completo
   LC 001 OUT=C=IN           % test de la tecla [DROP]
   ?CBIT=1.6 ->FIN2          % si la tecla se presiona, salimos
   RTN                       % regresamos
  ENDCODE
  ;
 ;
@
```

MÉTODO POR INTERRUPCIONES

La HP49G integra una rutina interna que permite realizar aplicaciones en cuatro niveles de grises, esto lo realiza por intermedio de las interrupciones, cunado ocurre una interrupción, la máquina detiene momentáneamente la tarea que estaba realizando para ejecutar otra tarea, una vez que termina de ejecutar dicha tarea, vuelve a ejecutar la tarea donde se detuvo.

En este caso las interrupciones las vamos a utilizar para mostrar los niveles de grises, en una aplicación, se puede definir cualquier tarea para las interrupciones (en nuestro caso, será mostrar los niveles de grises) y así ya no nos preocupamos por esto, sabiendo que se hará automáticamente.

Basta con decir a la hp, que estamos trabajando en niveles de grises, eso se indicara encendiendo un bit, y con declarar la dirección de cada uno de los grobs.

Las direcciones son:

Nombre:	Dirección (HEX):	Tamaño:
GreyOn?	8069C	1 cuarteto
GreyScr1	8069D	5 cuartetos
GreyScr2	806A7	5 cuartetos
GreyScr3	806B1	5 cuartetos

¿Para mostrar una imagen en niveles de grises, es necesario colocar un valor diferente de cero en GreyOn? (las otras rutinas internas que utilizan los niveles de grises allí colocan el valor F, cosa que haremos también).

A continuación, se colocan exactamente las direcciones del primer grob en GreyScr1 y la dirección del segundo grob en GreyScr2 y GreyScr3. Observe que disponemos de tres direcciones, esto en caso de que un usuario utilizara tres imágenes para mostrar cuatro niveles de grises. Y como en el primer método, no nos olvidemos de retirar la barra de menús y de regular los márgenes por las mismas razones. ¿Ahora, para volver de nuevo a una visualización clásica en blanco y negro, es necesario volver a poner el valor **0** en GreyOn? pero también recuperar la dirección de la pantalla y restablecer los márgenes que se encuentran en # 8068Dh, no nos olvidemos de restablecer los menús.

También contamos con una rutina interna que nos permite también mostrar niveles de grises para los menús.

He aquí, la a otra parte las direcciones respectivas a los grobs de los menús, que funcionan exactamente igual como la parte de la pantalla, utilizando GreyOn?

Nombre:	Direccion (HEX):	Tamaño:
GreyOn?	8069C	1 cuarteto
GreySoft1	806A2	5 cuartetos
GreySoft2	806AC	5 cuartetos
GreySoft3	806B6	5 cuartetos

Pasemos ahora al código para mostrar una imagen de 131x64 píxeles en cuatro niveles de grises *(un grob de 131x128 píxeles):*

```
::
CK1&Dispatch grob
 ::
 TURNMENUOFF
  CODE
   SAVE INTOFF
   A=DAT1.A D1=A                  % D1=Grob
   D1+5 LC 0110F A=DAT1.A         % A= tamaño del grob
   ?A#C.A -> FIN                  % Si este no es el tamaño salimos, de lo contrario
   GOSUB DISP                     % se llama la rutina de visualización del grob
  *KEY LC 001 OUT=C=IN            % test de la tecla [DROP]
   ?CBIT=1.6 ->EXIT               % si la tecla se presiona, salimos, de lo contrario
   GOSBVL Flush                   % se vacía el buffer del teclado
   GOTO KEY                       % generamos un bucle de esta tecla
  *EXIT D0=8069C C=0.B DAT0=C.1   % se vuelve a poner la pila en blanco y negro
   D1=00100 C=R0.P DAT1=C.1       % se vuelve a poner el margen izquierdo
   D0=8D A=DAT0.8 D1=20 DAT1=A.8  % luego la pantalla y el margen derecho
   D0=95 A=DAT0.A D1=30 DAT1=A.A  % y se vuelven a poner los menús
  *FIN INTON                      % habilitamos las interrupciones
   GOSBVL Flush                   % se vacía el buffer teclado
   LOADRPL                        % salimos al RPL
  *DISP D1+15 AD1EX               % se avanza sobre el contenido el grob
   D0=00100 C=DAT0.1 R0=C.P       % se guarda el margen izquierdo
   ?ABIT=0.0 ->OK                 % si la dirección es impar
   LC C DAT0=C.1                  % se desplaza el margen izquierdo
   LC FFF D0=25 DAT0=C.X          % y el margen derecho
  *OK D0=8069D DAT0=A.A           % se coloca la primera pantalla (GROB Nº1)
   LC 00880 A+C.A                 % avanzamos al segundo grob
   D0=A7 DAT0=A.A                 % se coloca la segunda pantalla (GROB Nº2)
   D0=B1 DAT0=A.A                 % se coloca la última pantalla (GROB Nº2)
   D0=9C LC F DAT0=C.1            % y se indica el color. (Grises)
   RTN
  ENDCODE
   ;
 ;
@
```

Los dos métodos no se utilizan obviamente de la misma forma y cada método pose una ventaja como una desventaja.

Con el primer método, se obliga a llamar la rutina que administra los niveles de grises en cada bucle, esto no es problema si el programa contiene pocos bucles.

La ventaja de este método es que puede ser generalizado para crear una rutina de visualización de grobs en ocho niveles de grises (o por qué no más!). Para mostrar ocho niveles de grises, el principio es el mismo que para cuatro niveles de grises, excepto que es necesario un grob de 131x192 píxeles, este es un grob que contiene tres grobs de 131x64. Solo esta parte del código cambiaría:

```
[...]
 SAVE INTOFF2
 A=DAT1.A D1=A              % D1=Grob
 D1+5 LC 0198F A=DAT1.A     % A= tamaño del grob
 ?A#C.A ->FIN               % Si este no es el tamaño salimos, de lo contrario
 AD1EX A+15.A B=A.A         % se recupera la dirección del primer grob
 D1=00100                   % D1=BITOFFSET
 C=DAT1.1 R0=C.P            % Se guarda el margen izquierdo
 ?ABIT=0.0 ->BUCLE          % si el grob está en dirección impar
 LC C DAT1=C.1              % se desplaza el margen de la izquierda
 LC FFF D1=25 DAT1=C.X      % y el margen de la derecha
*BUCLE
 GOSUB DISP                 % llamamos al subprograma de visualización
 LC 00880 A+C.A             % avanzamos al 2 grob
 GOSUB DISP GOSUB DISP      % llamamos al subprograma de visualización 2 veces
 LC 00880 A+C.A             % avanzamos al 3 grob
 GOSUB DISP GOSUB DISP      % llamamos al subprograma de visualización 4 veces
 GOSUB DISP GOSUB DISP
 A=B.A                      % recuperamos la dirección del 1er grob
 GOTO BUCLE
[...]
```

El resto del código es completamente idéntico.

Para el segundo método, el problema viene de las interrupciones: si se prohíben las interrupciones los niveles de grises no son visualizados. Por el contrario, si se los autoriza, se los autoriza también para el teclado, por eso es que en cada bucle es necesario vaciar el buffer del teclado.

Por el contrario, una vez llamada la rutina, no es más necesario preocuparnos de ello, esto es ideal.

El segundo método no esta presente en la HP48, pero el primer método es totalmente compatible con la HP48. Una vez adquirido el principio de funcionamiento de visualización en niveles de grises, es totalmente posible lanzar animaciones o presentaciones de toda clase.

41 Declaración de variables

En ML también se pueden declarar variables para facilitar la programación. Existen formas distintas de declarar variables de acuerdo al compilador que usemos.

1) Método Usando JAZZ
Sintaxis:
nombre EQU valor

Ejemplos:
=Contraste EQU #00101
Retardo EQU #FFFF

2) Método Usando MASD
Sintaxis:
DC nombre valor

Ejemplos:
DC contraste 00101
DC retardo FFFF

42 Sección ejemplos

EJEMPLO AUMENTAR EL CONTRASTE

CODIGO PARA MASD

```
CODE
SAVE
D1= 00101          %Dirección que apunta al contraste
C=DAT1 2           %Copiamos en C el valor actual del contraste
C+1 B              %Sumamos 1 al valor del contraste
DAT1=C 2           %Pegamos el nuevo valor del contraste en la dirección
LOADRPL
ENDCODE
@
```

EJEMPLO SUMA

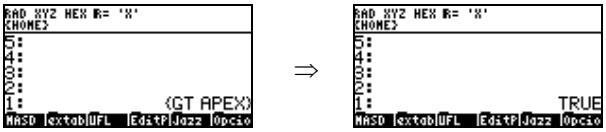
Suma dos números reales

CODIGO PARA MASD

```
CODE
GOSBVL POP2%          %coloca dos números reales en A y C, después hace un SAVPTR=SAVE
GOSBVL SPLTAC         %Convierte dos reales(%) en longreales (%%)
GOSBVL ADDF           %Suma dos long reales
GOSBVL PACK           %Pasa de (%%) a (%)
GOSBVL PUSH%LOOP      %Empuja el resultado a la pila
ENDCODE
@
```

EJEMPLO LISTA

Verifica si el argumento que se encuentra en el nivel 1 es una lista, Si es una lista entonces TRUE.



CODIGO PARA MASD

```
CODE
GOSBVL PopASavptr      %Coloca la dirección del  objeto del nivel 1 en A.  SAVE
AD1EX                  %D1 apunta al objeto
A=DAT1 A               %A= prologo del objeto
LC(5) DOLIST           %C =prologo de una lista
?C=A A                 %pregunta si C=A
GOYES TRUE             %Si se cumple sala a la etiqueta TRUE
GOVLNG GPPushFLoop     %Coloca FALSE en la pila y después hace LOOP
*TRUE
GOVLNG GPPushTLoop     %Coloca TRUE en la pila y después hace LOOP
ENDCODE
@
```

EJEMPLO BORRADO DE PANTALLA

Este ejemplo muestra como borrar el display.

CODIGO PARA MASD

```
CODE
SAVE
SCREEN          % A.A = dirección de la pantalla
D1=A            % D1 apunta a la pantalla
A=0.W           % A=ceros para borrar
LC 87           % numero de veces a repetir la acción
*borrar         % Etiqueta del bucle
DAT1=A.W        % borra 16 cuartetos
D1=D1+ 16       % avanza 16 cuartetos
C=C-1.B         % resta 1 al bucle
?C#0.B          % pregunta si termino el bucle
GOYES borrar    % si no repite
LOADRPL
ENDCODE
@
```

Otra variante seria:

```
CODE
SAVE
SCREEN          % A.A = dirección de la pantalla
D1=A            % D1 apunta a la pantalla
A=0.W           % A=ceros para borrar
LC 87           % numero de veces a repetir la acción
{               % Inicia un bucle
DAT1=A.W        % borra 16 cuartetos
D1=D1+ 16       % avanza 16 cuartetos
C=C-1.B         % resta 1 al bucle
UPNC }          % Finaliza un bucle si C=0. (Cuando se encienda el CARRY)
LOADRPL
ENDCODE
@
```

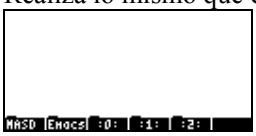
Otra variante seria:

```
CODE
SAVE
SCREEN          % A.A = dirección de la pantalla
D1=A            % D1 apunta a la pantalla
LC 00880
ZEROMEM
LOADRPL
ENDCODE
@
```

`ZEROMEM` llama a una rutina para poner a ceros la dirección apuntada por **D1** el número de cuartetos indicados por el registro **C**.

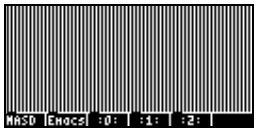
EJEMPLO PANTALLA:

Realiza lo mismo que el ejemplo anterior, pero con una pausa para poder observar el resultado.



```
CODIGO PARA MASD
CODE
SAVE                %Graba el estado de la HP
SCREEN              %En Aa (La partida A del registro A), tiene la
                    %dirección de la pantalla
D0=A                %Pone Aa en D0
A=0 W               %en A: 0000000000000000
LC 00087            %Para escribir 136 veces Aw en D0 (Pantalla: 136*64 píxeles,
                    %y Aw: 16 nibbles  (136*64)/(16*4)=136, en hexadecimal = 88
                    %menos 1 porque la HP empieza a contar a 0 => 87.
{
DAT0=A W            %escribir
D0=D0+16            %64 píxeles siguientes
C=C-1 A
UPNC
}
LC 0FFFF            %Para tener el tiempo de ver el resultado
{
C=C-1 A
UPNC
}
LOADRPL             %Recupera el estado grabado de la HP
ENDCODE
@
```

Se puede sustituir en **A=0 W** por cualquiera de los siguientes códigos:

RESULTADO	CODIGO	DESCRIPCION
	LA FFFFFFFFFFFFFFFF	Pantalla negra
	LA AAAAAAAAAAAAAAAA	Negro, blanco, negro…

EJEMPLO 20PIXELES:
Enciende los primeros 20 píxeles del display.

CODIGO PARA MASD

```
::
CLEARVDISP
CODE
SAVE                  % Salva los punteros (B, D, D0 y D1)
LC FFFFF              % Carga en el registro C el numero #FFFFF  píxeles)
GOSBVL "D0->Row1"     % D0 apunta al primer píxel de la primera línea.
DAT0=C A              % Carga el contenido de C(A) en D0
LOADRPL               % Restaura los punteros y sale del RPL
ENDCODE
SetDAsTemp
;
@
```

OBSERVACIONES: Recuerda que el numero #FFFFF al pasarlo a binario es #11111111111111111111 (esto quiere decir que el procesador encenderá los primeros 20 píxeles)

EJEMPLO DIBUJA UN GROB EN LA PANTALLA.

CODIGO PARA MASD

```
::
CLEARVDISP
CODE
SAVE
SCREEN
GOSUB DatosGrob
NIBHEX 80        %1ra línea del grob
NIBHEX E3        %2da  línea del grob
NIBHEX B6        %3ra línea del grob
NIBHEX E3        %4ta línea del grob
NIBHEX 80        %5ta línea del grob
*DatosGrob       %Retorna la dirección del inicio del grob
C=RSTK
D1=C             %D1 ( Inicio del grob
P=(16-5)       %5 líneas de los datos del grob para dibujar
*GrobBucle
C=DAT1 B       %Lee una línea de los datos del grob(2 nibbles)
DAT0=C B       %Escribe en la pantalla
D1=D1+ 2       %Siguiente línea de los datos del grob
D0=D0+ 16
D0=D0+ 16      %Próxima fila en la pantalla
D0=D0+ 2
P=P+1          %Incrementa el contador
GONC GrobBucle
GOVLNG GETPTRLOOP
ENDCODE
SetDAsTemp
;
@
```

EJEMPLO COMPARANDO REGISTROS

Un entero binario tiene un cuerpo de 5 nibbles. Te aconsejo que revises el MLTUT y también la documentación del MASD, te van a ayudar bastante.

CODIGO PARA MASD

```
::
#65
CODE
GOSBVL POP#        %Guarda un bint en A.A
SAVE
LC 00065           %Cargamos en C #00065
?C=A A             %Pregunta si el registro C es igual a al registro A
GOYES IGUAL        %SI es igual salta a la etiqueta igual
GOVLNG GPPushFLoop   %Si no Sale y devuelve la bandera FALSE
*IGUAL
GOVLNG GPPushTLoop %Sale y devuelve la bandera TRUE
ENDCODE
;
@
```

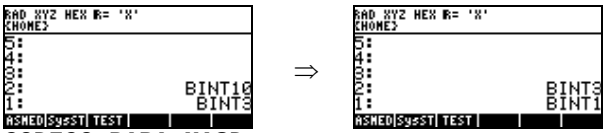
OBSERVACIONES: El comando GPPushTLoop lo que hace primero es un GETPTR (Restaura los registros), después coloca la bandera TRUE y después ejecuta un LOOP.

Otra variante de este ejemplo seria:

```
CODE
SAVE
LA 00065                %Cargamos en A #00065
LC 00065                %Cargamos en C #00065
?C=A A                  %Pregunta si el registro C es igual a al registro A
GOYES IGUAL             %SI es igual salta a la etiqueta igual
GOVLNG GPPushFLoop      %Si no sale y devuelve la bandera FALSE
*IGUAL                  %Etiqueta igual
GOVLNG GPPushTLoop      %Sale y devuelve la bandera TRUE
ENDCODE
@
```

EJEMPLO DIVISION:

Divide dos enteros binarios.

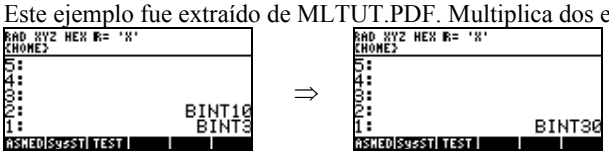


CODIGO PARA MASD

```
CODE
GOSBVL POP#        %Guarda BINT3 en Aa
R4=A A             %Guarda BINT3 en R4
GOSBVL POP#        %Guarda BINT10 en A
SAVE               %Guarda los punteros
C=R4 A             %C es BINT3 y A todavía es BINT10
GOSBVL IntDiv      %divide A por C
R0=C A             %Coloca los valores en R0 y R1
R1=A A             %Porque PUSH2# empuja a aquellos
GOSBVL GETPTR      %obtiene los punteros RPL
GOSBVL PUSH2#      %y empujamos R0 y R1
GOVLNG Loop        %retorna al RPL
ENDCODE
@
```

EJEMPLO MULTIPLICACION

Este ejemplo fue extraído de MLTUT.PDF. Multiplica dos enteros binarios.



CODIGO PARA MASD

```
CODE
GOSBVL POP#          %Guarda BINT3 en A
R4=A A               %Guarda BINT3 en R4
GOSBVL POP#          %Guarda BINT10 en Aa
SAVE
C=R4 A               %llama a BINT3.
GOSBVL MULTBAC       %Multiplica A y C, el resultado lo guarda en B
A=B A                %Ponemos el resultado de B en A
GOVLNG PUSH#ALOOP    %Coloca A en la pila
ENDCODE
@
```

EJEMPLO BEEP INDICADOR:

CODIGO PARA MASD

```
CODE
SAVE
LA 08
LC 80
A=A!C B
D0= 0010B            %Ésta es la dirección del registro del hardware que
                     %controla los indicadores en la cima del LCD
DAT0=A B             %Escribe en D0.
LC(5) 1000
D=C.A                %El registro D sostiene el valor de la frecuencia del sonido
LC(5) 500            %El registro C sostiene el valor de la duración del sonido
GOSBVL makebeep      %o 267F3, esta es la rutina del beep
LA 0
DAT0=A B             %Apaga el indicador.
LOADRPL
ENDCODE
@
```

OBSERVACIONES:

Mejor seria hacer:
LC 88 y DAT0=C.B 2 líneas después,

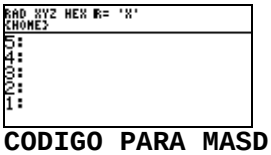
O más rápido aun:
LC 8 y DAT0=C.1 después (para cambiar al ultimo nibble)

EJEMPLO SWAP CON TEST.

CODIGO PARA MASD

```
CODE
C=DAT1 A        %Grabamos en C.A la dirección del nivel 1
?C=0 A          %Preguntamos si C.A es igual a cero
GOYES SALIR     %Si es vacía vamos a la etiqueta SALIR
D1=D1+ 5        %De lo contrario Subimos un nivel en la pila
A=DAT1 A        %Grabamos en A.A la dirección del nivel 2
D1=D1- 5        %Bajamos un nivel en la pila
?A=0 A          %Preguntamos si A.A es igual a cero
GOYES SALIR     %Si es vacía vamos a la etiqueta SALIR
DAT1=A A        %Escribimos la dirección de A.A en nivel 1
D1=D1+ 5        %Subimos un nivel en la pila
DAT1=C A        %Escribimos la dirección de A.A en nivel 2
D1=D1- 5        %Regresamos  al nivel 1
*SALIR          %Etiqueta Salir
LOOP            %Regresamos al RPL
ENDCODE
@
```

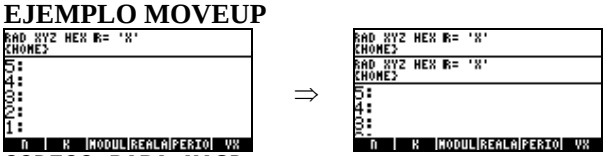
EJEMPLO TECLADO MÁS MENU



CODIGO PARA MASD

```
CODE
SAVE
LC 3F               %Valor para borrar el menú
D0=00128            %LINECOUNT, esta dirección controla al menú
DAT0=C.B            %borra el menú
*Teclas
LC 001
GOSBVL OUTCINRTN
?CBIT=1 6           %No sale hasta que se presione DROP
GOYES Exit
GOTO Teclas
*Exit
LC 37               %Valor para encender el menú
D0=00128
DAT0=C.B            %Enciende el menú
GOSBVL Flush        %Vacía el buffer del teclado
LOADRPL
ENDCODE
@
```

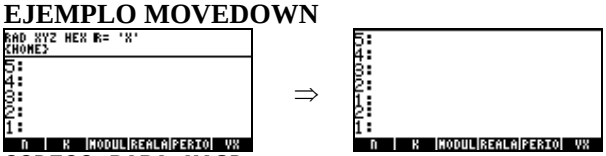
EJEMPLO MOVEUP



CODIGO PARA MASD

```
CODE
SAVE
SCREEN              %A(A) dirección de la pantalla
LC(5)    34*55+34   %Destino del desplazamiento
C=C+A    A
D1=C                %Destino
LC(5)    34*41+34   %Desplazamiento del origen
C=C+A    A
D0=C                %origen
LC(5)    34*41      %nibbles a mover
GOSBVL   MOVEUP     %Copia el bloque
GOVLNG   GETPTRLOOP
ENDCODE
@
```

EJEMPLO MOVEDOWN



CODIGO PARA MASD

```
CODE
SAVE
SCREEN              %A(A) dirección de la pantalla
D1=A                %Destino
LC(5)    34*14      %Destino del desplazamiento
C=C+A    A          %Desplazamiento del origen + dirección  de la pantalla
D0=C                %origen
LC(5)    34*41      %nibbles a mover
GOSBVL   MOVEDOWN   %Copia el bloque
GOVLNG   GETPTRLOOP
ENDCODE
@
```

OBSERVACIONES:

Depende por donde usted inicie y termine *(destino y origen)* el bloque copiado, Surgirán los efectos de cómo este deberá desplazarse. Note que en ambos ejemplos se mueven 34 Nibbles x 41 líneas.

43 Descripción de algunas rutinas

ERROR_A
Ponga el código del error en **Aa** y llame a esta rutina usando un GOVLNG. Usted no debe tener los registros guardados cuando lo llama, si los registros necesitan ser recuperados, hágalo primero y entonces llame esta rutina.

SKIPOB (Skip OBject)
Registros usados: **D0**, **A** y **C**.
Antes de llamar a esta rutina, **D0** debe apuntar al primer nibble. Cuando vuelve, **D0** apunta al primer nibble después del objeto.

RES_STR (REServe STRing)
Registros usados: **D0**, **D1**, **A**, **B**, **C**, **D**, **R0**.
Antes de llamarlo, ponga el número de caracteres que se necesitaran en **Ca**; deben guardarse los registros antes de llamar a esta rutina.
Si hay bastante espacio en el área de memoria temporal, el subprograma vuelve y **D0** apunta dentro del string a su primer nibble y su dirección se guarda en **R0a**.

COPY DOWN y COPY UP
Hay dos rutinas para copiar los datos en la memoria: **COPY DOWN** y **COPY UP**.

COPY DOWN sirve para copiar un objeto a otro, del principio de cada uno al extremo.
Si yo quisiera copiar al revés en la memoria entonces uso **COPY UP**.
Cuando se llama a **COPY DOWN** o **COPY UP**, **D0** debe apuntar dónde se leerán los datos y **D1** dónde se escribirán. Nosotros debemos indicar en **Ca** cuántos nibbles nosotros deseamos copiar.

GET_REG
Esta rutina restaura los registros guardados anteriores en ROM.

44 Subprogramas en ROM

En esta sección que veremos contiene una lista de subprogramas en ROM, que lo que hacen generalmente es reemplazar varias rutinas por una sola llamada en ROM, si se quiere aprender mas del ML se pueden desmontar estar rutinas con algún debuger como el **DB** del **JAZZ** para saber como trabajan estas direcciones de ROM.

Información necesaria:
D = Cantidad de memoria libre (direcciones) entre la pila de retornos y la pila (FREETOP)
D1 = Puntero que apunta a la superficie de pila, orden creciente en memoria (DSKTOP)
D1.A es la dirección del objeto del nivel 1 de la pila
D0 = puntero para el actual stream (RPLTOP)
D0.A es el próximo puntero/prologo del objeto
B = puntero a la superficie de la pila de retornos, orden decreciente en memoria (RETTOP)
B.A es libre, @ (**B.A-5**) dirección de superficie de la pila de retornos.
(**B.A-5**) es el siguiente puntero/prologo en la pila de retornos.

44.1 Propósito general

0679B SAVPTR	**D0** a RPLTOP **D1** a DSKTOP **B** a RETTOP **D** a FREETOP Apaga el carry
067D2 GETPTR	Lo contrario de **SAVPTR**. Apaga el Carry.
05143 GETPTRLOOP	Hace **GETPTR**, **Loop**
36897 D0=DSKTOP	Obtiene un nuevo **D0** desde DSKTOP, Registros Usados: **A**
368A6 D1=DSKTOP	Obtiene un nuevo **D1** desde DSKTOP, Registros Usados: **C**
26767 AllowIntr	Permite las interrupciones.
26791 DisableIntr	Deshabilita las interrupciones.
0020A AINRTN	**A=IN** También vea **CINRTN** Por razones del hardware (Bug) **A=IN** debe estar en la misma dirección
00212 CINRTN	**C=IN** También vea **AINRTN** Por razones del hardware (Bug) **C=IN** debe estar en la misma dirección

44.2 Errores

GENERANDO ERRORES

04FBB DOMEMERR	Error “Insufficient Memory”
26CA7 DOSIZEERR	Error “Bad Argument Value”
Errjmp	Error y sale. **A.A** = Número del error a mostrar.
266DB GPErrjmp	**C A=C.A,** vea **GETPTR** vea **Errjmp**
065AA GPMEMERR	**GETPTR**, Error Insufficient Memory vea **GETPTR** vea **DOMEMERR**

CONSTANTES DE ERROR

00202 argtypeerr	"Bad Argument Type"
00203 argvalerr	"Bad Argument Value"

00B02 constuniterr	"Inconsistent Units"
00305 infreserr	"Infinite Result"
00A03 intrptderr	"Interrupted"
00C14 lowbaterr	"Low Battery"
00302 negunferr	"Negative Underflow"
00303 ofloerr	"Overflow"
0000A portnotaverr	"Port Not Avaliable"
00301 posunferr	"Positive Underflow"
00C13 prtparerr	"Invalid PRTPAR"
00C02 timeouterr	"Timeout"
00C06 xferfailerr	"Transfer Failed"

44.3 Matemática hexadecimal

26A2A ADIV3	A.A = A.A/3	Registros Usados: A.6 C.6 P
26A23 ADIV6	A.A = A.A/6	Registros Usados: A.6 C.6 P
26A15 ADivC	B.A=A.A/C.A	Registros Usados: A.A C.A
269F2 AMULT34	A.A=A.A*34	Registros Usados: C.A
26A1C BMULT34	B.A=B.A*34	Registros Usados: C.A
269F9 CMULT34	C.A=A.A*34	Registros Usados: A.A

26F00 DCHXW

Convierte BCD en **C.W** para hexadecimal en
A.W B.W C.W. vea **HXDCW**
Registros Usados: P CRY

06A8E DIV5 — C.A = C.A/5 — Registros Usados: A.10 C.10 D.10 P

26A0E HEXTODEC

Convierte hexadecimal en **C.A** para BCD en **A.A**
Registros Usados: A.6 B.6 P

2DEAA HXDCW

Convierte hexadecimal en A.W para BCD en
A.W B.W C.W. vea **DCHXW**
Registros Usados: P CRY
Note que **HXDCW** requiere la entrada en **A**
pero **DCHXW** requiere en **C**

03F24 IntDiv

A.A/C.A ⇒ A.A=resto
C.A=cociente, Registros Usados: D.A P SB

2709E MPY

Multiplica A.W y C.W (⇒A.W=C.W)
Registros Usados: D.W, SB. Devuelve el carry eliminado

03991 MUL#	B.A = A.A*C.A
26A07 MULTB+A*C	B.A=B.A+(A.A*C.A)
26A00 MULTBAC	B=0.A vea **MULTB+A*C**

44.4 Long reales

MANEJO ALMACENAMINETO

31348 STAB0	A.W ⇒ R0	B.W ⇒ R1
31356 STAB2	A.W ⇒ R2	B.W ⇒ R3
31364 STCD0	C.W ⇒ R0	B.W ⇒ R1
31372 STCD2	C.W ⇒ R2	B.W ⇒ R3
3139C RCAB0	R0 ⇒ A.W	R1 ⇒ B.W
313A7 RCAB2	R2 ⇒ A.W	R3 ⇒ B.W
313B2 RCCD0	R0 ⇒ C.W	R1 ⇒ D.W
313BD RCCD2	R2 ⇒ C.W	R3 ⇒ D.W
31380 EXAB0	A.W ⇔ R0	B.W ⇔ R1
3138E EXAB2	A.W ⇔ R2	B.W ⇔ R3
3133A XYEX	A:B ⇔ C:D	
314CA GETAB1	x = @D1+	
314E4 GETAB0	x = @D0+	
31518 GETCD0	y = @D0+	
31532 PUTAB0	@D0+ = x	

CALCULANDO

315A9 RADDF (%%)	x=x+y D0
315BB ADDF (%)	x=x+y D0 0 resultado negativo si **ST10** o **ST11**
31756 DIVF	x=x/y
316FD MULTF	x=x*y
3158F RADD1	x=x+1 también vea **RADDF**
31586 RSUB1	x=x-1 también vea **RADDF**

CONVERSION

2F4A2 PACK	(A,B ⇒ A) Convierte %% a %. vea **PACKSB** sin redondear
2F47D PACKSB	(A,B ⇒ A) Convierte %% a %.
31131 SPLITA	(A ⇒ x) Convierte % a %%
31193 (SPLITC)	(C ⇒ y) Convierte % a %%
SPLITC_	(C ⇒ y) Convierte % a %%
31187 SPLTAC	(A,C ⇒ x, y) Convierte 2 reales a long reales

44.5 Manejo de la memoria

MEMORIA GENERAL MANEJO DE RUTINAS

069F7 ADJMEM
D= @FREETOP = vea **ROOM** / 5
Registros Usados: A.10 B.10 C.10 D.10 vea **DIV5**
Recalcula **D** en FREETOP después
de los movimientos RAM

0554C DOGARBAGE
Si **ST=1 10** entonces **GPMEMERR**
de lo contrario **GARBAGECOL** y **GETPTR**

0613E GARBAGECOL
Colecciona la Basura, no Registros Usados: R1..R4

06806 ROOM
⇒ **C.A** = @DSKTOP-@RETTOP
Registros Usados: A.A D0
Calcula la memoria libre entre los datos y la pila de retornos.

03019 SKIPOB
Salta el objeto en **D0**, Elimina **ST1**, Elimina el carry, **P=0**
⇒ **D0** = dirección del objeto pasado
Registros Usados: A.A C.A P ST1 RSTK2

MOVIENDO E INTERCAMBIANDO AREAS DE MEMORIA

2682B BLKSWAP+
vea SWAPMEM_D0D1C y ajusta todas referencias

26871 EndTempOb
Mueve TEMPOB a D0, vea NEWADR

0670C MOVEDOWN
Copia hacia arriba **C.A** nibbles de **D0** a **D1**
D0 y D1 apuntaran a las próximas localizaciones
Registros Usados: A.W C.A P
Use esto para mover hacia arriba

06992 MOVERSD
Mueve **A.A** – pila de retornos hacia bajo **C.A**
nibbles Borrar un bloque debajo de RSK
A.A=fin **C.A**=nibbles
Ajusta todas las referencias, vea ADJMEM
Registros Usados: A.W B.A C.W D.10 D0 D1 P

06A53 MOVERSU
Mueve A.A - pila de retornos hacia arriba **C.A** nibbles
Abra un bloque debajo de RSK
A.A=inicio **C.A**=nibbles
Ajusta todas las referencias, vea ADJMEM
Registros Usados: A.W B.A C.10 D.10 D0 D1 P

06A1D MOVEDSD
Mueve datos de la pila - A.A hacia abajo C.A nibbles
Abra un bloque sobre la pila
A.A=fin **C.A**=nibbles
Ajusta todas las referencias, vea ADJMEM
Registros Usados: A.W B.A C.10 D.10 D0 D1 P

069C5 MOVEDSU — Mueve datos de la pila - A.A hacia arriba C.A nibbles Anule un bloque sobre la pila
A.A=inicio **C.A**=nibbles
Ajusta todas las referencias, vea ADJMEM
Registros Usados: A.W B.A C.10 D.10 D0 D1 P

066B9 MOVEUP — Copia hacia abajo **C.A** nibbles de **D0** a **D1**
D0 y **D1** apuntarán al inicio del área
Registros Usados: A.W C.A P
Use esto para mover hacia abajo

269B3 SWAPMEM — Cambia 2 áreas de memoria
Area 1: **R1.A** a **R2.A**
Area 2: **R2.A** a **R3.A**
vea **SWAPMEM_D0D1C**

269DD SWAPMEMEQ — Cambia dos áreas de memoria del mismo tamaño
vea SWAPMEMEQ_D0D1C
R1.A⇒Area1 R2.A⇒Area2

269E4 SWAPMEMEQ_D0D1C — Cambia dos áreas de memoria del mismo tamaño
D0⇒Area1 D1⇒Area2 C=(D1-D0)
Registros Usados: A.W B.A C.W P CRY

269BA SWAPMEM_D0D1C — D=C.A vea **SWAPMEM_D0D1D**
269C1 SWAPMEM_D0D1C_ nofree — D=C.A vea **SWAPMEM_D0D1D_nofree**
269C8 SWAPMEM_D0D1D — Cambia dos áreas de memoria
Area 1: D0 a D1
Area 2: D1 a (D1+D.A)
Registros Usados: A.W B.A C.W D.W P CRY

269CF SWAPMEM_D0D1D_nofree — vea **SWAPMEM_D0D1D** pero no altera la memoria @RSKTOP

269D6 SWAPMEM_nofree — vea **SWAPMEM** pero no altera la memoria @RSKTOP

SITUAR LA MEMORIA EN TEMPOB

06AD8 CREATETEMP — Sitúa los nibbles de **C.A** si no hay suficiente memoria
⇒ D0=fondo, D1=cima del área
⇒ B.A = C.A = @D1 = desplazamiento
al previo tempob = #nibbles+6
Asigna **C.A** nibbles, enciende el carry si no hay suficiente memoria.

039BE GETTEMP — CREATETEMP con GARBAGECOL si es necesario
C.A = nibbles ⇒ C.A = nibbles
⇒ **D0** = bottom, **D1** = top
Si no hay bastante memoria vea **GPMEMERR**

268CC GETBOTTEMP — Asigna los nibbles de C.A al fondo de TEMPOB,
Error si no hay bastante memoria
Retorna D0=cima D1=fondo del área
C.A=nibbles
Registros Usados: A.W B.A C.W D.10 R1.A

05B79 MAKE$ — Crea una cadena de caracteres en el área temporal.
Hace **SETHEX**, C=C+C.A

05B7D MAKE$N — Crea una cadena de caracteres en el área temporal.
C.A = nibbles
C=D1= dirección de la pila (stack)
⇒ A = nibbles+5 (longitud el campo del string)
⇒ B = nibbles+16 (longitud del área temporal para el $)
⇒ C = D1 = dirección de la pila (stack)
⇒ D0 = dirección del cuerpo del $ (prologo y longitud)
⇒ R0 = Dirección del $
No Registros Usados: R1-R4

26919 MAKEBOT$N — Crea un string extenso de C.A nibbles
D0⇒cuerpo **R0.A**⇒string **R1.A**=longitud
Registros Usados: A.W B.A C.W D.10
Vea también WIPEOUT, GETBOTTEMP

26920 MAKERAM$ — Asigna toda la memoria en un string,
Deja 5 nibbles para empujar,
Vea también **MAKE$N**, **ROOM**

CAMBIANDO AREAS EN TEMPOB

26840 Clean$ — disminuye un objeto del tipo string en la cima de TEMPOB
R1=dirección de la longitud del campo
A.A=nueva dirección del extremo
Registros Usados: A.W B.A C.W D.A D0 D1

26847 Clean$R0 — R1=R0+5

26721 Shrink$ — disminuye un objeto del tipo string.
R0.A⇒$ **D0**=fin del $
Registros Usados: A.W B.A C.W D.10 D0 D1

26990 Stretch$ — Extiende un objeto del tipo string
R0.A⇒$ D0=fin del $
Registros Usados: A.W B.A C.10 D.10 D0 D1
Vea SIZEPLUS

44.6 Rutinas CRC

05981 DoCRC
Calcula el CRC del nibble **A.A** de **D0**.
Retorna CRC en **A.A**
Registros Usados: C.W P
Enciende las interrupciones las y las apaga.

0597E DoCRCc
D0=C vea **DoCRC**

266B8 CKLBCRC
Verifica el CRC de la librería en **D0**
CC: Ok CS: CRC es erróneo
Registros Usados: A.A C.W D0 P
Desactiva y re-habilita las interrupciones

44.7 Trabajando con la memoria

26C53 CompareACbBytes
Compara los bytes **A.B=C.B** en **D0** y **D1**
CC: Igual CS: No igual
D0/D1 siempre apunta al final del extremo
Registros Usados: A.M A.A C.M C.B P

2690B INV.ZONE
Invierte los **C.A** nibbles en **D0**
Registros Usados: A.W C.A P

0675C WIPEOUT
Pone a cero **C.A** nibbles de **D1**
Registros Usados: A.W C.A P

269EB WIPESPACE
Inicia **C.A** nibbles de **D1** con espacios (#20h)

44.8 Otras rutinas

26808 aBZU
Descomprime un string BZ-comprimido
D0 ⇒ comprimido
D1 ⇒ Lugar a descomprimir
Registros Usados: A-D R0-R2

083D1 GETRRP
Retorna el RRP.
Si objeto es SysRRP, retorna CS
A.A no cambia; de lo contrario CC y
A.A ⇒RRP **B.A**⇒RAM-WORD
Registros Usados: A.A B.A C.A D.A D0
RRP es un directorio, el retorna la dirección del pasado-objeto dentro del directorio.
SysRRP es igual que HOME.

26C68 RclAssembly — Llama un objeto del directorio actual
D1 ⇒ Nombre (ID etc)
Retorna el objeto en **D0**
Registros Usados: A.W B.A C.W D.A D0 D1 ST P

44.9 Cambiando de banco

26BB9 ACCESSBank0 — **P=0**: cambia al banco 0
P=1: cambia atrás
Registros Usados: D0 C.A P

26BC0 ACCESSBank1 — Bank 1, vea ACCESSBank0
26BC7 ACCESSBank2 — Bank 2, vea ACCESSBank0
26BCE ACCESSBank3 — Bank 3, vea ACCESSBank0
26BD5 ACCESSBank4 — Bank 4, vea ACCESSBank0
26BDC ACCESSBank5 — Bank 5, vea ACCESSBank0
26BE3 ACCESSBank6 — Bank 6, vea ACCESSBank0
26BEA ACCESSBank7 — Bank 7, vea ACCESSBank0
26BF1 ACCESSBank8 — Bank 8, vea ACCESSBank0
26BF8 ACCESSBank9 — Bank 9, vea ACCESSBank0
26BFF ACCESSBank10 — Bank 10, vea ACCESSBank0
26C06 ACCESSBank11 — Bank 11, vea ACCESSBank0
26C0D ACCESSBank12 — Bank 12, vea ACCESSBank0
26C14 ACCESSBank13 — Bank 13, vea ACCESSBank0
26C1B ACCESSBank14 — Bank 14, vea ACCESSBank0
26C22 ACCESSBank15 — Bank 15, vea ACCESSBank0

44.10 Display

266B1 $5x7 — (D.A B.A C.A D0 D1 ⇒)
Visualiza el cuerpo del string de **D1** en
el grob de **D0**
C.A = caracteres
B.A = localización en x
D.A = longitud de la fila en nibbles (normalmente #22)
⇒ **D1** = dirección después del $
D0 = localización del próximo carácter
D.A = longitud de la fila

2677C D0->Row1 — (⇒ D0) Llama a la dirección de
la pantalla actual (1ra fila)

26783 D0->Sft1 — (⇒ D0) Llama a la dirección del
grob del menú (1ra fila)

26A38 DISP_DEC — Muestra hexadecimal de **C.A** como decimal
D0⇒GROB
Registros Usados: A.6 B.W C.W CRY RSTK2 ST
Vea también **MINI_DISP_AWP**

2679F DispOn — Vuelve a encender la pantalla

26798 DispOff — Vuelve a apagar la pantalla

26894 GET_HEADER — vea **GET_HEADERTYPE** y **ST9** (normal/minifont)

2689B GET_HEADERTYPE — Retorna el tipo de cabecera en **A.A** Registros Usados: D0
El tipo del título es la altura del título en píxeles,
incluye la línea separadora negra

2687F GET_@FONTE — Retorna la dirección del sistema de fuente en **A.A**
Generalmente **LA 84D82 RTN**

268A2 GET_HFONTE — Retorna la altura del sistema de fuente en **A.A**,
Registros Usados: **D0**

268A9 GET_HFONTECMD — Retorna la altura de la línea de
comandos de la fuente, Registro Usado: **D0**
⇒ **A.A**=altura **ST9**=normal/minifont

268B0 GET_HFONTESTK — Retorna la altura de la fuente de la pila, Registro Usado: D0
⇒ **A.A**=altura **ST9**=normal/minifont

268B7 GET_HFONTESTKD1C — Retorna la altura de la fuente de la pila,
Registro Usado: D1
⇒ **C.A**=altura **ST9**=normal/minifont

2674B makegrob — **R0.A** = x, **R1.A** = y
⇒ **D0** = cuerpo
Crea un grob de tamaño x, y
El Prologo esta en D0-20

26927 MINI_DISP — Muestra un string en minifont
D1⇒string **D0**⇒GROB **C.A**=caracteres
ST11=normal/invertido
Solo trabaja encima de 34 nibble de ancho de pantalla,
en un nibble alineado la posición Avanza
D0 y D1 al próximo carácter

2692E MINI_DISP_AWP — Muestra A.WP en minifont
D0⇒GROB, **ST11**=normal/invertido
ST10=muestra/esconde iniciando con ceros
Registros Usados: A.A B.W C.W CRY RSTK2

2693C MINI_DISP_VAL — Muestra C.A dígitos de B.W en minifont, D0⇒GROB, ST

26974 SCREEN.MARGIN — ST9=0 ⇒ C.A=00016
C.A es el número de caracteres que pueden mostrarse con
MINI_DISP (ST9=1) o $5x7 (ST9=0)

2696D SCREEN.MARGIN2 — Ceros R0.A entonces realiza SCREEN.MARGIN
Registros Usados: R0.W

269AC STYLE.MINIFONT — Cambia los datos del carácter minifont en A.6,
Registros Usados: P
ST1=1 ⇒ italic
ST2=2 ⇒ underline
ST3=3 ⇒ invertido

44.11 Herramientas gráficas

26B7A Arrows
Dibuja flechas para señalar si el desplazamiento es posible
D0⇒GROB ST4-7=arrows:
4=Arriba 5=Abajo 6=Izquierda 7=Derecha
ST9=normal/minifont
Registros Usados: D1 A.A B.A C.A D.A ST0-7 P RSTK2
ST9 selecciona la flecha grande o pequeña

26AB6 aCircleB
Dibuja un circulo negro encima del GROB de **D0**
A.A = cx, **B.A** = cy, **C.A** = r
Registros Usados: RSTK2 D0 D1 R3.A R4.A A.S C.S

26AC4 aCircleG1
Dibuja un circulo gris claro. vea **aCircleB**
26ACB aCircleG2
Dibuja un círculo gris oscuro. vea **aCircleB**
26ABD aCircleW
Dibuja un circulo blanco. vea aCircleB
26AD2 aCircleXor
Invierte el circulo. vea aCircleB

26B0A aDistance
C.A=sqrt(A.A^2+B.A^2)
Registros Usados: A.6 B.6 C.6 D.6 CRY SB P

26B34 aFBoxB
Dibuja un rectángulo o cuadrado lleno de color negro
D0⇒GROB
A.A=x1 **B.A**=y1 **C.A**=x2 **D.A**=y2
Registros Usados:
RSTK2 A.W B.W C.W D.A D.S D0 D1 R3.A R4.A

26B42 aFBoxG1
Dibuja un rectángulo o cuadrado lleno de color gris claro.
vea aFBoxB

26B49 aFBoxG2
Dibuja un rectángulo o cuadrado lleno de color gris oscuro.
vea aFBoxB

26B3B aFBoxW
Dibuja un rectángulo o cuadrado lleno de color blanco.
vea aFBoxB

26B50 aFBoxXor
Invierte los colores de un rectángulo o cuadrado lleno.
vea aFBoxB

26AF5 aGrey?
Retorna información acerca del GROB de **D0**
ST0: 0=B&W 1=Gris
R4.A= Longitud del Plano **R3.A**= Longitud de la línea

26AFC aGNeg
Invierte el GROB de **D0**
Registros Usados: RSTK2 A.W B.A C.A D0 R3.A R4.A

26B57 aLBoxB
Dibuja un rectángulo negro
D0⇒GROB
A.A=x1 **B.A**=x2 **C.A**=y1 **D.A**=y2
Su uso es igual a aFBoxB

26AA1 aLineG1
Dibuja una línea gris clara. vea aLineB
26AA8 aLineG2
Dibuja una línea gris oscura. vea aLineB
26A9A aLineW
Dibuja una línea blanca. vea aLineB

26B65 aLBoxG1	Dibuja un rectángulo gris claro. vea aLBoxB
26B6C aLBoxG2	Dibuja un rectángulo gris oscuro. vea aLBoxB
26B5E aLBoxW	Dibuja un rectángulo blanco. vea aLBoxB
26B73 aLBoxXor	Invierte un rectángulo. vea aLBoxB
26A93 aLineB	Dibuja una línea negra encima del GROB de **D0** **A.A**=x1, **B.A**=x2, **C.A**=y1, **D.A**=y2 Registros Usados: RSTK2 D0 D1 R3.A R4.A A.A A.S B.A B.S C D.A
26AAF aLineXor	Invierte una línea. vea aLineB
26B18 aPixonB	Dibuja un píxel encima del GROB de **D0** **A.A** = coordenada x, **B.A** = coordenada y Registros Usados: RSTK2 C.W D0 D1 R3.A R4.A
26B1F aPixonG1	Dibuja un píxel gris claro. vea aPixonB
26B26 aPixonG2	Dibuja un píxel gris oscuro. vea aPixonB
26B11 aPixonW	Dibuja un píxel blanco. vea aPixonB
26B2D aPixonXor	Invierte un píxel. vea aPixonB
26B03 aScroolVGrob	Desplaza el grob GROB de **D0** **R0.A**=h **R1.A**=Ys **R2.A**=Yd **R3.A**=X **R4.A**=w Registros Usados: A.A B.A B.S C.W D.A D.S RSTK2 R3.A R4.A D0 D1
26AE0 aSubReplGor	
26AE7 aSubReplGxor	
26AD9 aSubReplRepl	
26760 w->W	Calcula el ancho del GROB A.A=ancho en nibbles

44.12 Popping y pushing

Pop es usado para obtener datos de la pila, y push para colocar datos en la pila.

PUNTEROS

03249 DropLoop	Hace un drop al nivel 1 de la pila y luego **Loop**
34202 4DropLoop	Pop 4, Loop
03672 GPOverWrALp	GETPTR, OverWr **A**, Loop
0366F GPOverWrR0Lp	GETPTR, OverWr **R0**, Loop
266E2 GPPushA	GETPTR, Coloca **A** en la pila, Borra el carry
268EF GPPushALp	GETPTR, Coloca **A** en la pila, Loop
268E8 GPPushR0Lp	GETPTR, Coloca **R0** en la pila, Loop
26705 PopASavptr	Pop a **A.A**, SAVPTR
2670C PopSavptr	Pop, SAVPTR
03A86 PUSHA	Push A, Loop

IR, WIRE, PRINTING, TIME

settimeout		
clrtimeout		
SavPtrTime*	(⇒ C.13)	Obtiene TICKS en C.13
GetTime++	(⇒ C.13)	Obtiene TICKS en C.13

TRUE y FALSE

GPPushFTLp	GETPTR, Loop a FalseTrue
266CD GETPTRFALSE	vea GETPTR , Do FALSE
266D4 GETPTRTRUE	vea GETPTR , Do TRUE
35213 GPOverWrFLp	vea GETPTR , OverWr FALSE, Loop
351F3 GPOverWrTLp	vea GETPTR , OverWr TRUE, Loop
351F0 GPOverWrT/FL	vea GETPTR , OverWr, TRUE/FALSE, Loop
3524F GPPushFLoop	vea GETPTR , Coloca FALSE, Loop
35236 GPPushTLoop	vea GETPTR , Coloca TRUE, Loop
35233 GPPushT/FLp	vea GETPTR , Coloca TRUE/FALSE, Loop
3521D OverWrFLoop	OverWr FALSE, Loop
351FD OverWrTLoop	OverWr TRUE, Loop
3521A OverWrT/FLp	OverWr TRUE/FALSE, Loop
34A68 popflag	Pop a **A.A**, Si es TRUE entonces enciende el carry
35259 PushFLoop	Coloca FALSE, Loop DOFALSE
3523D PushF/TLoop	Coloca FALSE (CRY)/TRUE, Loop
35240 PushTLoop	Coloca TRUE, Loop
35256 PushT/FLoop	Coloca TRUE (CRY)/FALSE, Loop

44.13 Enteros binarios (BINT)

03991 MUL#	(B.A = A.A*C.A)
26A07 MULTB+A*C	B.A=B.A+(A.A*C.A)
26A00 MULTBAC	B=0.A
316FD MULTF	x=x*y
03F24 IntDiv	(A.A/C.A ⇒ A.A C.A)
269F9 CMULT34	C.A=A.A*34 Registros Usados: A.A
26A1C BMULT34	B.A=A.A*34 Registros Usados: B.A
269F2 AMULT34	A.A=A.A*34 Registros Usados: C.A
06641 POP#	Pop # a **A.A**
03F5D POP2#	(#1 #2 ➔) Pop #1 a **A.A** y #2 a **C.A**
06537 PUSH#	GETPTR , Push **R0** como **#**
03DC7 #PUSHA-	SAVPTR , R0=A, PUSH# , Loop
06529 PUSH2#	GETPTR , Push R0 y R1 como #
0357F PUSH#LOOP	GETPTR , Push R0 como #, Loop
0357C PUSH#ALOOP	GETPTR , Push A como #, Loop
03F14 Push2#Loop	GETPTR , Push R0 y R1 como #, Loop
35812 Push2#aLoop	GETPTR , Push R0 y A como #, Loop
036F7 Push#TLoop	GETPTR , Push R0 como #, Coloca TRUE, Loop
283A3 Push#FLoop	GETPTR , Push R0 como #, Coloca FALSE, Loop

44.14 Hexadecimales y enteros

0596D PUSHhxsLoop	Push P+1 nibbles de **A** como hxs, Loop
266FE PUSHhxs	Push P+1 nibbles de A como hxs
2709E MPY	Multiplica A.W y C.W (⇒A.W=C.W) Registros Usados: D.W, SB. Retorna si el carry esta apagado
266FE PUSHhxs	Push A.WP como hxs
0596D PUSHhxsLoop	Push A.WP como hxs, Loop
26951 PUSHzint	Push A.WP como ZINT
26958 PUSHzintLoop	Push A.WP como ZINT, Loop

44.15 Reales y complejos

2F62C POP1%SPLITA	(%pop ⇒ x) Pop %, convierte a %%, **SAVPTR**
2F636 POP1%	(%pop ⇒ A) Pop %, **SAVPTR**
2F65E POP2%	(%pop1 %pop2 ⇒ A,C) Pop 2 reales, **SAVPTR**
2F7E4 PUSH%	(A ⇒ %push) Push A as %, **GETPTR**
2F899 PUSH%LOOP	(A ⇒ %push) Push A como %, **GETPTRLOOP**
26A62 POPC%	(C%pop ⇒ A:C) Pop C% (SETDEC)
26A70 POPC%%	(C%%pop ⇒ A:B C:D) Pop C%% (SETDEC)
26A69 PUSHC%	(A:C ⇒ C%push) Push C%
26A77 PUSHC%%	(A:B:C:D ⇒ C%%push) Push C%%

44.16 Manejo del teclado

2A4AA ATTNchk	Sale si se presiona CANCL.
2678A Debounce	Escanea el teclado hasta que no se detecte ninguna inestabilidad retorna el plano de las teclas presionadas en **A.W**
04999 KeyInBuff?	Carry si es verdadero
267C2 OnKeyDown?	Carry si es verdadero
267C9 OnKeyStable?	Carry si es verdadero
267A6 Flush	Vacía el buffer del teclado.
267AD FlushAttn	Vacía el mostrador attn.
04840 POPKEY	(⇒ C.A) Enciende el carry si el buffer esta vació. De lo contrario retorna la tecla en C.B (y en @KEYSTORE) Registros Usados:: A.S C.S C.A D1 (poner P=0)
267DE SrvcKbdAB	(A.W ⇒) poner KEYSTATE y KEYBUFFER
26D1E (ThisKeyDn?)	CS si la tecla en A.B es abajo. Registros Usados:: A.A C.A D1 P OR
26D17 (ThisKeyDnCb?)	A=C.B vea ThisKeyDn?

44.17 Cadena de caracteres (STRINGS)

266F7	GetStrLenStk	Pop $ ⇒ C.A D1=dirección($)⇒C.A = longitud, D1 = cuerpo
266F0	GetStrLenC	D1 = C, C=dirección($)⇒ C.A = longitud, D1 = cuerpo
266E9	GetStrLen	D1=$ ⇒ C.A = longitud, D1 = cuerpo
268D3	GetStrLenL	D1=$ ⇒ C.A = longitud de caracteres

44.18 Otras entradas

	Warmstart	Realiza un ON-C
	Coldstart	Realiza un ON-A-F
26E60	ASRW5	ASR.W 5 veces
26E71	ASLW5	ASL.W 5 veces
313C8	CCSB1	Registros Usados: D.S para encender SB, Apaga el carry
26832	CHANGE_FLAG	Cambia el flag ST # A.B (1-4) Si A.B > 10, A.B-11 se guarda en R0.B. Apaga el carry si es ok Vea también CHANGE_FLAG2
26839	CHANGE_FLAG2	Cambia el flag ST # A.B (1-4) Realiza algo desconocido si A.B > 10. Enciende ST7
267EC	clkspd	cronometra la velocidad del CPU Apaga las interrupciones en la entrada y salida ⇒ A.A=velocidad/16 B.A=vueltas/16s Registros Usados: C.A D0 P CRY
26E82	CSRW5	CSR.W 5 veces
26E93	CSLW5	CSL.W 5 veces
04292	DeepSleep	Pone la calculadora en el modo "deep sleep" Modo de energía baja, La pantalla se apaga. Se enciende si se presiona la tecla ON o si se interrumpe.
267F3	makebeep	C = Duración en msec, D = Frecuencia en Hz Verifique el flag BEEP.
04929	liteslp	Pone la calculadora en el modo "deep sleep" pero con la pantalla encendida. Se enciende si se presiona cualquier tecla o se interrumpe

RSUB1	x=x+1	InP:any SB=0 ⇒ DORADDF!
RADD1	x=x-1	InP:any SB=0 ⇒ DORADDF!
DIV2	x=x/2	InP:any No inf/unf protection
CCSB1	enciende **SB** usando **D.S**	
aMODF	x=mod(x,y)	

34AAD SEMILOOP	Salta a SEMI+5
SKIP	Salta un objeto en el runstream
DupAndThen	hace DUP, Entonces ejecuta el siguiente objeto en el stream
SetISysFlag	Set ISysFlag, Loop
Setflag	Set ISysFlag

34A31 GOTO	Toma los próximos 5 nibbles del RSTK y salta ahí.
TST15	tst(x) CS si el test es verdadero P=1 <, P=2 =, P=4 >, P=13 <>
D0=ALoop	

08D66 SysPtr@	Push @C.A, Loop (Push A.A saltando un DOBINT !)

44.19 Debugging

2685C DBUG	Muestra el contenido de todos los registros. Usa un nivel del **RSTK** y #8190C para salvarlos
26863 DBUG.TOUCHE	Igual que DBUG pero congela la pantalla hasta que se presione un tecla.

45 Direcciones RAM importantes

#00100 (BITOFFSET) o (DISPIO) Bit 0 à 3 Controla el display
Solo usa un nibble (4 bits): [DON OFF2 OFF1 OFF0]
El tercer bit (DON), es usado para definir el display (ON/OFF), Revisar los comandos "DispOn" y "DispOff". El resto de bits, son usados para definir el margen izquierdo del display.
Normalmente, BITOFFSET se encuentra en #8h (1000). Con éste valor el margen izquierdo es cero: #1000b, si el margen izquierdo es cero, entonces el margen derecho también lo es, por tanto #125h contiene #000h.

Si nosotros cambiamos el margen izquierdo (#100h) a 12: #1100b, el valor del margen derecho debe restarse en 1, asi: #000h - 1 = #FFFh. En conclusión, si:

#100h = #8h ➔ #125h = #000h
#125h = #Ch ➔ #125h = #FFFh

#00101 (CONTRAST) Bit N° 0 à 4 Contraste de la pantalla
El contraste puede tomar valores entre 0 y 31, El 31 (#1F) en bits es: 0001 1111
Por tanto concluimos que el contraste sólo usa 5 bits. Modificar estos valores de 5 bits ocasiona la modificación del contraste de la pantalla. [CON3 CON2 CON1 CON0]
Los valores recomendados se establecen entre # 09h y # 18h, valores que corresponden al valor minimo y al máximo utilizados en las teclas [ON]-[-] y [ON]-[+] para regular el contraste.

EJEMPLO CONTRASTE
CODIGO PARA MASD

```
CODE
SAVE
LC 11            %Valor que queremos poner al contraste
D1= 00101        %Dirección que apunta al contraste
DAT1=C 2         %Pegamos el valor del contraste en la dirección
LOADRPL
ENDCODE
@
```

#00102 (DISPTEST)
bit alto del contraste + [VDIG LID TRIM CON4]
bits de test de la pantalla. [LRT LRTD LRTC BIN]

#00104 (CRC)
Esta entrada esta soportada, se usan 4 nibbles para CRC. Cada la memoria actualiza el CRC.

#0010B (ANNCTRL) Bit N° 0 a 3 Encendido de los indicadores
Esta entrada esta soportada.
Esta dirección es la que tiene el control de los indicadores de la pantalla.
Depende del estado de los 4 bits.
[LA4 LA3 LA2 LA1]
(Alarma )
El bit 0 enciende el indicador  .
El bit 1 enciende el indicador  .
El bit 2 enciende el indicador  .

El bit 3 enciende el indicador de sonoridad.

EJEMPLO INDICADORES:

Supongamos que queremos encender los indicadores  . Entonces los bits 0 y 1 deberán estar encendidos y los demás apagados.

CODIGO PARA MASD

```
CODE
SAVE
LC FFFFF     %Iniciamos un retardo para poder ver el resultado
{            %Iniciamos un bucle
C=C-1  B     %Vamos decrementando el contador
D1= 0010B    %Cargamos en D1 la dirección de los indicadores
LA 3         %#3h=0011 Los bits 0 y 1 se encenderán.
DAT1=A.A     %Encendemos los indicadores.
UPNC }       %Si el contador no es igual a cero continúa en el bucle
LOADRPL
ENDCODE
@
```

#0010C (ANNCTRL2) Bit N° 0,1 & 3 Encendido de los indicadores

También controla los indicadores.
[AON XTRA LA6 LA5]
(on extra io busy)
El bit 0 (LA5) enciende el indicador de ocupado (busy).
El bit 1 (LA6) enciende el indicador de transmisión. (io).

#00120 (DISPADRR) o (DISP1CTL) Dirección de la pantalla

Usa 5 nibbles que es la dirección de tu display, la dirección del GROB que permanentemente es desplegada en la pantalla se debe a esta dirección. Esta dirección es de solo escritura, usted puede escribir una nueva dirección en #00120h y entonces se modificara el grob de la pantalla.

DISP1CTL trabaja simultáneamente con BITOFFSET.
Si la dirección que contiene DISP1CTL es par, entonces BITOFFSET debe ser 8.
Si la dirección que contiene DISP1CTL es impar, entonces BITOFFSET debe ser 12.

#00125 (LINEOFFS) o (LINEBITS)

Esta dirección es de solo escritura, usa 3 nibbles para definir el margen derecho del display.
Normalmente se encuentra en 0, pero cuando la dirección que contiene **DISP1CTL** es impar, **LINEBITS** debe ser #**FFF**.

#00128 (LINECOUNT) Bit N° 0 à 3 Barrido y altura del menú

Esta dirección es de escritura y lectura, estos 4 bits controlan en lectura el número de la línea en curso de refresco y en escritura la altura de los menús, normalmente se encuentra en #37h (56-1), cuando se desea manipular todo el display, necesitamos usar las 64 líneas del display, colocando en LINECOUNT #3Fh (64-1), uno hace desaparecer la barra de menús. Para obtener el número de la línea en curso de refresco, tenemos que leer 6 bits (ejemplo: D0= 00128 A=DAT0 6). El registro **A** vale **0** cuando la última línea ha sido refrescada, de ahí su nombre LINECOUNT (Contador de líneas)

#00130 (MENUADDR) Bit N° 0 à 3 Dirección del bitmap del menú

La pantalla esta separada en dos. La dirección de la parte inferior esta fija en #00130 y corresponde al bitmap de la barra de menús. Esta dirección es de solo escritura, es igual a DISPADRR solo que con los menús.

#00108 (POWERSTATUS)

Registros de energía baja
[LB2 LB1 LB0 VLBI] solo lectura

(LowBat2 LowBat1 LowBat0 y VeryLowBatIndeed)

#00109 (POWERCTRL)
Detección de energía baja. [ELBI EVLBI GRST RST]

#0010D (BAU)
Velocidad del Serial baud [UCK BD2 BD1 BD0] UCK solo lectura
3 bits = {1200 1920 2400 3840 4800 7680 9600 15360}

#00111 (RCS)
Entrada soportada, Recibir Control/Status [RX RER RBZ RBF]

#00112 (TCS)
Entrada soportada, Transmitir Control/Status [BRK LPB TBZ TBF]

#00114 (RBR)
"Receive Buffer", esta entrada esta soportada. [byte recibe]
Leyendo borra RBF

#00116 (TBR)
"Transmit Buffer", esta entrada esta soportada. [byte being transmite]
Escribiendo enciende TBF

#00118 (SRR)
"Service Request" este registro es de solo lectura.

#0011E (SCRATCHPAD)
Registro usado por las interrupciones.

#0012E (TIMER1CTRL)
Control de TIMER1 [SRQ WKE INT XTRA]

#0012F (TIMER2CTRL)
Control de TIMER2 [SRQ WKE INT TRUN]

#00137 (TIMER1)
Esta entrada esta soportada, TIMER1 usa 4 bits (1 nibble), este es un reloj que corre o se decrementa a 16Hz (16 tiempos/s), vea pag. 77

#00138 (TIMER2)
Esta entrada esta soportada, TIMER2 usa 32 bits (8 nibbles), este es un reloj que corre o se decrementa a 8192Hz (8192 tiempos/s), vea pag 77