HP49 Assembly Language Examples

By
Peter Geelhoed

HP49 Assembly Language Examples

This document is aimed at System RPL programmers who wish to learn Assembly. We will assume MASD syntax in SysRPL mode on a 49 for the documentation of MASD look here: http://www.hpcalc.org/details.php?id=2986

The following programs are very useful for programming, they are virtually indispensable. SysRPL programmers probably have these already.

ROM version 1.19-6, Other ROM's may cause problems with the following programs
http://www.hpcalc.org/details.php?id=3240

For programming itself you should of course have the **extable** library.
http://www.epita.fr/~avenar_j/hp/49.html

Emacs is *the* programming environment for the 49, get it!
http://www.hpcalc.org/details.php?id=3940

Nosy can be used to look into the code of Rom routines and other programmes, which is very educational.
http://www.hpcalc.org/details.php?id=4323

It is a good idea to step through some of these programs to see how they work. It is the way I learned Assembly. I recommend **Jazz6.8e** 's DB.
http://www.hpcalc.org/details.php?id=4700

If you want to know everything about assembly you should read:
"**Introduction to Saturn Assembly Language**" available at
http://www.hpcalc.org/details.php?id=1695.

For a description of romroutines you can take a look at the ML part of **entries.srt** But remember it is a 48 document so the addresses will be different
http://www.hpcalc.org/details.php?id=1782

Carsten Dominik's **Entry Database** also is a great source of romroutine descriptions. It contains entries.srt and more.
http://zon.astro.uva.nl/~dominik/hpcalc/entries/

For a complete description of the 48 RAM you can study **rammap.a**
http://www.hpcalc.org/details.php?id=3231

Contents

0. Introduction

Although these examples will not crash your 49, it is probably a good idea to backup your memory. Typos are easily made! You can also upload the file examples.bz to your 49, it is a selfextracting directory containing all the examples. It will save you loads of work typing everything in.

The last four chapters have a few advanced examples, you might want to postpone examining them until you have taken a look at the other examples.

We will learn to use the registers of the SATURN in due course but it is useful that you know them now:

Registers	Description	Communicates with:	Width in nibbles
A	working register	All, except RSTK, ST, P	16
B, D	working register	C, A	16
C	working register	All	16
R0...R4	scratch registers	C, A	16
D0, D1	memory pointer	C, A	5
DAT0, DAT1	memory itself	C, A	2^20
RSTK	return stack	C	5
PC	program counter	C, A	5
IN	IN register	C, A	4
OUT	OUT register	C, A	3
P	field selector	C	1
ST	status bits	C	4

To read from memory you set D0 (or D1) to the correct value and then do A=DAT0 f. f Is a field selector

The working and scratch registers are 16 nibbles wide and operations can work on different fields.

Field Selection

```
P  [P-P]    -----------------------------------------------
WP [0-P]    15:14:13:12:11:10: 9: 8: 7: 6: 5: 4: 3: 2: 1: 0
XS [2-2]    -----------------------------------------------
X  [0-2]    |S|                                  |XS|<-B->|
S  [15-15]                                 |<---- A ----->|
M  [3-14]     |<--------------- M ---------------->|<-- X ->|
B  [0-1]    |<--------------------- W -------------------->|
W  [0-15]
A  [0-4]
```

We will use the A field very frequently, it is 5 nibbles wide and the A stands for Address. As the address space of the SATURN is 5 nibbles. The B register is the Byte register, it is particularly useful when you are working with strings.

Some examples of instructions

A=C A	this speaks for itself; copy the contents of field A of C to A
A=DAT0 B	copy from memory location D0 into A(B)
AR1EX W	exchange the contents of A and R1
LC 123	load C with #123h
GOSBVL romroutine	calls romroutine
GOSUB subroutine	calls subroutine

The code is set in blue font for better readability.

1. Basics

Generally, ML objects are called from RPL (User/Sys) and when they finish they need to return to RPL again. This means that certain registers must contain the right values.

D (A) the amount of free mem (in chunks of 5 nibbles)
D1 points to top of the stack (level 1)
D0 points to the runstream
B (A) pointer to top of return stack

It is ***IMPERATIVE*** that you set these correctly before you return to RPL. You may crash your calculator if they are wrong. Also the field selector register P must be 0.

EXAMPLE 1.1 From ASM to RPL

When finishing a ML program you need to set the Program Counter (PC) to the next object in the runstream.

D0 contains an address which has the address of the next object in the runstream.

```
"CODE
A=DAT0 A                 % recall next object in runstream
D0+5                     % set D0 to "nextnext" object in runstream
PC=(A)                   % set the program counter to the address A
                         % is pointing to
ENDCODE
@"
```

EXAMPLE 1.2 Calling Romroutines

to call a romroutine there are two commands
GOSBVL which means **GOS**u**BV**ery**L**ong, and
GOVLNG which stands for **GOV**ery**L**o**NG**.

Remember that GOVLNG does not set the ReturnStack so when you call a routine with GOVLNG your code will not continue after the romroutine. It is mainly used for routines that return you to RPL.

Example 1.1 can also be written as

```
"CODE
GOVLNG Loop
ENDCODE
@"
```

Loop is a subroutine that does exactly what Example 1.1 does
We use GOVLNG because you do not need to continue after Loop

EXAMPLE 1.3 Saving RPL pointers

The previous examples do not do very much, in fact they do nothing. If we want to do more we will probably need to use more registers. But remember? We need to set them back to the right values. Fortunately there are romroutines that do this for you: SAVPTR saves the RPL pointers and GETPTR recalls them

```
"CODE
GOSBVL SAVPTR          % save RPL pointers, use GOSBVL
                       % because we want to
                       % continue after this
GOSBVL GETPTR          % get the pointers back
GOVLNG Loop            % return to RPL
ENDCODE
@"
```

There is also a subroutine that does the last two lines in one;
romroutine GETPTRLOOP.

EXAMPLE 1.4 DISPKEY
I want to mention here that there is a romroutine which displays the contents of all the registers and waits for you to press a key: DISPKEY, you can also call it with GOSBVL DBUG.TOUCHE

```
"CODE
GOSBVL SAVPTR          % save RPL pointers
DISPKEY                % display the registers
GOVLNG GETPTRLOOP      % get the pointers back and return
                       % to RPL
ENDCODE
@”
```

These programs still do not do anything, but it is important that you understand how they work.

EXAMPLE 1.5 Loading a register with a constant

Quite often you will need to load a constant into a register. You can do this with P, D0, D1 and A and C. The following program shows how, please step through it with Jazz or insert `DISPKEY`’s.

```
"CODE
GOSBVL SAVPTR          % save RPL pointers
D0= 12345              % load hex digits 12345 in D0
D0= 12                 % load 12 into the two least
                       % significant digits of D0
                       % D0 now has 12312
LA 1234567890ABCDEF    % fill A with constant

D=0 W                  % 0 is the only constant that will also
                       % work with B and D
C=0 W                  % clear C to see the next one more
                       % clearly
P=3                    % load the filed selector with 3
LC 123                 % load 123 into C but start at digit
                       % three, C will now contain
                       % 0000000000123000
P=0                    % reset P to 0, or else RPL will crash
GOVLNG GETPTRLOOP      % get the pointers back and return
                       % to RPL
ENDCODE
@”
```

2. The Stack

Quite often you will need your code to work on an object on the stack. D1 points to the first stack level so we can get the address of the object on the stack there.

EXAMPLE 2.1 Doubling a binary integer
Let's double the binary integer on the stack. We need to make sure there is an integer on the stack, we use SysRpl to put one there

A Bint (binary integer) is an object with a prologue #02911h and a body of 5 nibbles:

```
BINT 1        = 1192010000
BINT 12F45    = 1192054F21
```

A good way to explore the syntax of objects is the command ->H.

```
::
BINT1                  ( put bint on the stack)
CODE
A=DAT1 A               % read address of bint
D1+5                   % point to next item on stack
                       % (basically removing level 1)
D+1 A                  % free mem has increased by 5 nibbles
SAVE                   % MASD syntax for GOSBVL SAVPTR
D0=A                   % D0 now points to the bint in the
                       % memory
D0+5                   % skip the bint prologue
A=DAT0 A               % read the value of the bint
A+A A                  % double the value
GOVLNG PUSH#ALOOP      % does GETPTR, pushes the value of A(A)
                       % to the stack as a bint and then
                       % goes to RPL
ENDCODE
;
@"
```

There is also a subroutine that gets the address of the object on the stack and saves the RPL pointers: "PopASavptr" so we could replace

"A=DAT1 A
D1+5
D+1 A
SAVE"
in example 2.1 with
"GOSBVL PopASavptr"

Sometimes you just want the value of the BINT on the stack. POP# gets it for you. It puts the value of the bint in A(A) but be sure to have the RPL pointers intact, use it before you do GOSBVL SAVPTR!
Now that we know what binary integers are and how to pop and push them onto the stack, we can take a look at dividing and multiplying. MULTBAC multiplies A(A) and C(A) and puts the result in B(A)

EXAMPLE 2.2 Multiplying

```
"::
BINT10                  ( put two bints on the stack )
BINT3
CODE
GOSBVL POP#             % pop bint 3
R4=A A                  % save in R4
GOSBVL POP#             % pop bint 10 to A(A)
SAVE                    % save RPL pointers
C=R4 A                  % C is 3, A is still 10
GOSBVL MULTBAC          % multiply A and C, result in B
A=B A                   % can't push B so put it in A
GOVLNG PUSH#ALOOP       % push A to the stack
ENDCODE
;
@"
```

To divide two integers we have IntDiv, it divides A(A) by C(A) and puts the result in C the remainder will be in A.

EXAMPLE 2.3 Dividing

```
":: BINT10  BINT3       ( put two bints on the stack)
CODE
GOSBVL POP#
R4=A A                  % save bint 3 in R4
GOSBVL POP#             % bint 10 in A
SAVE                    % save pointers now
C=R4 A                  % C has bint 3 and A still has bint 10
GOSBVL IntDiv           % devide A by C
R0=C A                  % put the values in R0 and R1
R1=A A                  % because PUSH2# pushes those
GOSBVL GETPTR           % get RPL pointers
GOSBVL PUSH2#           % and push R0 and R1
GOVLNG Loop             % return to RPL
ENDCODE
;
@"
```

EXAMPLE 2.4 Editing a string

We will now make a string which contains "HP49G"
Because we will get to making objects from scratch in a later stage we will assume that there is a 5 character string on level 1 of the stack.
strings have the following structure:
prologue, #02A2Ch
length field, 5 nibbles. It has the size of the string in nibbles, including the size of the length field itself but excluding the prologue. For a 5 character string it is therefore: 5 nibbles for the length field and 10 nibbles for the 5 characters which comes to 15 nibbles.
body, the characters

```
"::
"AAAAA"
CODE
GOSBVL PopASavptr      % save RPL and get addr of string
R0=A A                 % save the addr of the string in R0
D0=A                   % point to string prologue
D0+5                   % skip prologue
D0+5                   % skip length
LC 48                  % load C register with #48h which is
                       % the character number of "H"
DAT0=C B               % write one byte (2 nibbles) to memory
D0+2                   % point to next char
LC 50                  % character P
DAT0=C B               % write P
D0+2                   % point to next char
LC 34                  % character 4
DAT0=C B               % write 4
D0+2                   % point to next char
LC 39                  % character 9
DAT0=C B               % write 9
D0+2                   % point to next char
LC 47                  % character G
DAT0=C B               % write G
GOVLNG GPPushR0Lp      % get pointers, push address in R0,
                       % return to RPL
ENDCODE
;
@"
```

We can condense this significantly by not loading every character separately but all in one go:

```
"::
"AAAAA"
CODE
GOSBVL PopASavptr       % save RPL pointers and get addr of
                        % string
R0=A A                  % save the addr of the string in R0
D0=A                    % point to string prologue
D0+10                   % skip prologue and prologue
LC 4739345048           % load "HP49G" in C
DAT0=C 10               % write string
GOVLNG GPPushR0Lp       % get pointers, push address R0, return % to RPL
ENDCODE
;
@"
```

3. Tests & loops

Sometimes we need to test something and jump to a another point in the code (like an IF THEN ELSE if you like). The points to jump to are called labels. The assembler expects them after a "*" on a new line. The Saturn can perform many tests on its working registers.

EXAMPLE 3.1 Comparing registers

Let's see if we can make a program that returns a TRUE flag if the value of the bint on the stack is #6FEh

```
“::
#6FE
CODE
GOSBVL POP#            % read a bint from the stack to A(A)
SAVE                   % save RPL pointers
LC 006FE               % this is the number to check
?C=A A -> Equal        % if they match jump to equal
GOVLNG GPPushFLoop     % if not get ptrs, push FALSE and loop
*Equal
GOVLNG GPPushTLoop     % if match push TRUE
ENDCODE
;
@”
```

Another useful thing to test for is the Carry. It is set when an overflow (or underflow) of a register occurs. So if you subtract something from 0 a carry will be set. This is extremely useful if you want to do something a number of times.

EXAMPLE 3.2 Loops

```
"CODE
SAVE                   % we know this by now
LC 0000A               % load 10 in C
*Label1                % you can use any name for a label
C-1 A                  % subtract one from C
                       % usually the A field is used as a
                       % counter, although here we could have
                       % used the B field
GONC Label1            % Go to Label1 if there is no carry
LOADRPL                % MASD speak for GOVLNG GETPTRLOOP
ENDCODE
@"
```

This code will continue to subtract one from C until a carry is set.
Question: How many times will this loop?

No, it doesn't loop 10 times. Let's count the number of times the code passes Label1

#	Contents of C, after C-1 A	Carry
1	00009	No
2	00008	No
3	00007	No
4	00006	No
5	00005	No
6	00004	No
7	00003	No
8	00002	No
9	00001	No
10	00000	No
11	FFFFF	Yes

It loops 11 times, so that is one more than what you start with.

EXAMPLE 3.2A Masd syntax loop
The masd syntax can be used to write this up a bit shorter. It compiles to exactly the same as example 3.2 Personally I do not use it, but that is because I learned Jazz syntax first. See the masd documentation for the full masd syntax.

```
"CODE
SAVE                    % save RPL pointers
LC 0000A
{ C-1 A UPNC }          % loop 11 times
LOADRPL
ENDCODE
@"
```

EXAMPLE 3.3 Status bits

The status bits are 16 bits that can be set and tested for easily, they are useful in keeping one bit data. You can use bits 0 thru 11, 12 thru 15 are used by the system, don't set them if you do not know what you are doing!

```
"CODE
SAVE                    % save RPL
CLRST                   % clear status bits 0 thru 11
ST=1 9
?ST=0 9 -> Label2       % if bit 9 is not set jump to Label2
LOADRPL                 % go back to RPL
*Label2
LOADRPL                 % go to RPL after jump
ENDCODE
@"
```

EXAMPLE 3.4 Timer

We will now discuss the built in timer, it decreases 8192 times per second and is 8 nibbles wide. It is located at the address TIMER2. There is another timer called TIMER1, which decreases 16 times per second and is one nibble wide. We will use TIMER2 to show you an application of the P register. It takes a bint from the stack and waits that amount of ticks of the TIMER2.

```
“::
10000                   ( put bint on the stack)
CODE
A=0 W                   % clear A because we need 8 nibbles
GOSBVL POP#             % read bint from stack
SAVE                    % save pointers
D0=(5) TIMER2           % point to TIMER2
P=7                     % load P with 7
C=DAT0 WP               % read nibbles 0 true 7
C-A WP                  % subtract the bint
*Wait
A=DAT0 WP               % read timer
?A>C WP -> Wait         % wait until A is equal to C
                        % or less than C if the moment has
                        % passed
P=0                     % reset P to zero or else RPL will
                        % crash
LOADRPL                 % get pointers and return to RPL
ENDCODE
;
@”
```

You can use this code with TEVAL but be sure to put a BINT on the stack.

4. Subroutines and the Return Stack

We have seen in example 1.3 that there are some very useful subroutines already in ROM, but you can make your own. They are called with GOSUB which behaves the same as GOSBVL but you need to make your own routine

EXAMPLE 4.1 Calling your subroutine

```
“CODE
SAVE                  % save RPL pointers
GOSUB Delay           % call the subroutine Delay
                      % put adress of next instruction
                      % (GOSUB Delay) on return stack
GOSUB Delay           % do the delay routine again
                      % and put the addr of LOADRPL in RSTK
LOADRPL               % return to RPL

*Delay                % Delay subroutine, it simply loops a
                      % number of times before returning
LC 03000              % loop #3001h times, can be any number
*DelayLoop
C-1 A                 % subtract one
GONC DelayLoop        % goto DelayLoop until Carry is set
RTN                   % jump to the adress in the return stack
ENDCODE
@”
```

The return stack is a 8 level register that is a LIFO (last in first out) stack. It holds the address at which your code will continue after the subroutine. It can also be used as a place to save 5 nibbles. Be sure to remove them, otherwise the code may jump to that address!
The instructions to manipulate the return stack are limited:
RSTK=C puts the A field of the C register onto the returnstack
C=RSTK reads the address on the return stack into C(A) and removes it from the return stack.
Also all return commands pop one address from the return stack, and jump to it

EXAMPLE 4.2 Using the return stack

This code is not very useful but is shows how one can use the return stack to save some data. For example when you do not want to alter any of the other registers.

```
“CODE
C=DAT1 A               % read the adress of the object on the
                       % stack, if this is 0 the stack is empty
CD1EX                  % exchange the registers D1 and C
                       % so that C has the stack pointer and
                       % D1 the object from the stack
RSTK=C                 % save the stack pointer in RSTK because you
                       % will need it to return to RPL at the end
                       % of the code
C=DAT1 A               % read the prologue, you could do other
                       % things here
C=RSTK                 % retrieve the stack pointer from the return
                       % stack
D1=C                   % reset D1 to the stack
A=DAT0 A               % recall next object in runstream
D0+5                   % set D0 to "nextnext" object in runstream
PC=(A)                 % set the program counter to the address A
                       % is pointing to
ENDCODE
@”
```

EXAMPLE 4.3 Data inside your code

You can use a combination of a GOSUB and C=RSTK to get an address inside your code. This can be useful when you have some fixed data which you need in your code.

```
“CODE
SAVE                   % save rpl pointers
GOSUB Data             % jump to Data and save the address of the
                       % next instruction (NIBHEX) in RSTK
NIBHEX C2A20B10008454C4C4F40275F425C444
                       % NIBHEX is a MASD directive that puts raw
                       % hex in your code
*Data
C=RSTK                 % read the address of the data
A=C A
GOVLNG GPPushALp       % push the data address to the stack
ENDCODE
@”
```

We have pushed a string to the stack always be sure that you push good objects to the stack. A corrupt object may crash the calc.

5. Tempob

When you want to make new objects you need to reserve an amount of memory. The easiest way to do this is to MAKE$. It will reserve the memory and make a string prologue and length. The address of the string will be in R0(A) and D0 will point to the first character in the string.
TempOb must always contain objects, if you put raw hex in it the next garbage collection will screw up your memory. MAKE$ takes care of this, if you use CREATETEMP you will have to do it yourself see chapter 8 for this.
MAKE$ will however do a GARBAGE collection if there is not enough memory. Therefore you should not run any programs containing MAKE$ directly after compilation, instead store the code in a variable before running it. Also see chapter 8 for more information on this.

EXAMPLE 5.1 Making a string
We will make a string of 10 characters and make the first one a "A"

```
"CODE
SAVE                    % save pointers
LC 0000A                % number of characters
GOSBVL MAKE$            % make the string
LC 41                   % character code for "A"
DAT0=C B                % write the first character
GOSBVL GPPushR0Lp       % get pointers and push string to stack
ENDCODE
@"
```

You will notice that the first character is an "A" but the rest of the string looks like garbage. This is because MAKE$ does reserve the memory but it will contain the leftovers from previous use. You will have to fill it in yourself.

EXAMPLE 5.2 Writing in ML
Let’s fill the string with the ten numbers 0 to 9. We shall now use P as a counter, it is only one nibble wide so we can use it only for loops that loop 16 times or less

```
“CODE
SAVE                  % save pointers
LC 0000A              % number of characters
GOSBVL MAKE$          % make the string
LC 30                 % character “0”
P=6                   % 16 minus the number of loops
*Write
DAT0=C B              % write the character
C+1 B                 % next character
D0+2                  % point to next character
P+1                   % increase until P= zero again
GONC Write            % loop 10 times
GOSBVL GPPushR0Lp     % get pointers and push string to stack
ENDCODE
@”
```
It is important that P is reset to zero, because returning to RPL requires it.

EXAMPLE 5.3 Shrinking the string

Sometimes you do not know how big an object will be when it is finished, so you need to reserve sufficient memory and free the remainder when you are finished. We can use a routine called Shrink$ that does just that. In the following code we do know how big the object will be but that is beside the point. We will put the values of B(A) and D(A) in a list and push it to the stack.

```
“CODE
SAVE
LC 01000                % reserve 1000 bytes
GOSBVL MAKE$
LC 02A74                % DOLIST prologue
DAT0=C A                % write it
D0+5                    % point to next addr
LC 02911                % DOBINT prologue
DAT0=C A                % write it
D0+5                    % and point to the next addr again
C=B A
DAT0=C A                % write value of B(A)
D0+5
LC 02911                % do all this again for D(A)
DAT0=C A
D0+5
C=D A
DAT0=C A
D0+5
LC 0312B                % load SEMI
DAT0=C A                % and write it to terminate the list
D0+5
GOSBVL Shrink$          % Shrink$ will shrink the string to the
                        % current D0 with the addr of the strin
                        % prologue in R0
A=R0 A                  % R0 is the string prologue
A+10 A                  % the list prologue is 10 nibbles down
GOVLNG GPPushALp        % pust it to the stack
ENDCODE
@”
```

6. The Screen

Because ML is so fast it is very useful for displaying graphics on the screen. We will start with *writing* something to the screen.

EXAMPLE 6.1 The Screen

Let's see if we can put some pixels on the screen, don't be afraid it only **looks** like a crash!

```
"CODE
SAVE
GOSBVL "D0->Row1"       % this sets D0 to the first
                        % nibble of the screen buffer
D1=A                    % A also contains the address
D0=00000                % point D0 to an address in mem
LC(5) 34*56             % load 5 nibbles of C with 34*56
                        % 34 nibbles per line 56 lines
GOSBVL MOVEDOWN         % copy C nibbles from D0 to D1
C=0 A                   % make a little loop to allow
*Wait                   % some time for viewing the text
C+1 A
GONC Wait
LOADRPL                 % return to RPL
ENDCODE
@"
```

EXAMPLE 6.2 Writing to the Screen

While writing nibbles to the screen can be entertaining it may be more useful to write readable text.

```
"CODE
SAVE
GOSBVL "D0->Row1"        % point D0 to screen
GOSUB Data
NIBHEX 8454C4C4F40275F425C444
*Data
C=RSTK                   % remember this from EXAMPLE 4.3?
D1=C                     % point to characters
LC 00005
B=C A                    % B is the offset of the text in nibbles
LC 00022
D=C A                    % D is the width of the screen
                         % usually it is 34 (#22h)
LC 0000B                 % C is the number of characters
GOSBVL "$5x7"            % display the text in the screen
C=0 A                    % make that loop again
*Wait
C+1 A
GONC Wait
LOADRPL                  % return to RPL
ENDCODE
@"
```

And there we have the program you learn in every language!

EXAMPLE 6.3 DISPADDR

There is a very handy address called DISPADDR at #00120h you can write an address at this position and the system will start displaying the data at that address. DISPADDR is ***WRITE*** only

So we can rewrite example 6.1 as

```
"CODE
SAVE
D0= 00120            % DISPADDR
A=0 A                % the address of the new screen
DAT0=A A
LOADRPL              % return to RPL
ENDCODE
@"
```

You noticed that there is no wait loop, this is because the DISPADDR pointer is not updated after the code is finished. To update the screen you could generate an error or reset the original pointer.

EXAMPLE 6.4 Greyscales [ADVANCED]

Greyscales are basically very simple. Since the pixels on the screen have only two states (on/off) you need to turn a pixel on and off really quick to get a grey pixel. The 49 has a grey grob type, which we shall not use. It is more instructional to use the old method of two 131*64 grobs, one beneath the other in a single 131*128 grob. This is also the format used by most PC based conversion programs. (I think XNView is very good). It means that with two grobs you can have 4 colours. The “heaviest” grob is the top one

First we need to create a greyscale grob. We do this in UserRPL.

```
<<
#131d #128d BLANK {#0d #0d} #131d #32d BLANK NEG REPL {#0d #64d}
#131d #16d BLANK NEG REPL {#0d #96d} #131d #16d BLANK NEG REPL
>>
```

The code will take the resulting grob as an argument on level 1 on the stack
MAKE SURE there is a 131*128 grob on the stack or it may crash

```
“CODE
ST=0 15               % this turns off some interrupts
                      % see below for further explanation
GOSBVL =PopASavptr    % get the address of the grob
A+20 A                % point to grob body of heaviest grob
B=A A                 % save addr in B
GOSBVL "D0->Row1"     % get the addr of the current screen
R0=A A                % save it in R0
LC(5) 64*34           % 64 lines and 34 nibbles per line
C+B A                 % Add to addr of first grob so that
A=C A                 % A has the addr of grob2
D0= 00100             % #00100h is BITOFFSET you can move the
C=DAT0 X              % screen pixel by pixel by altering it
RSTK=C                % save current bitoffset in RSTK
?ABIT=0 0 -> EVEN     % if addr of grob is even no need to
                      % shift
LC C                  % shift 4 bits left (-4=C)
DAT0=C 1              % write new bitoffset
D0= 00125             % LINENIBS contains the number of
                      % nibbles per line
LC FFF                % it has to be decreased because of the
DAT0=C X              % new bitoffset
```

```
*EVEN
D0= 00120               % DISPADDR
D1= 00128               % in LINECOUNT you can write the number
                        % of lines to be displayed
LC 3F                   % 64 lines including line 0 so #63d
DAT1=C  B

*MAIN
GOSUB PAINT             % subroutine that displays grob in A
ABEX   A                % switch grob1 and grob2
GOSUB PAINT             % display grob1 twice
GOSUB PAINT
ABEX   A                % A has grob2 and B grob1 again

GOSBVL OnKeyDown?       % OnKeyDown? returns a carry if the
                        % On Key is being pressed. If ST 15 is
                        % clear, holding down ON will halt the
                        % code until you release it, and the
                        % code will simply continue

GONC MAIN               % keep looping until ON key is pressed
ST=1 15                 % Allow the interrupts again
A=R0 A                  % Get the old display address
DAT0=A A                % reset it
LC 37                   % set the linecount back to 55
DAT1=C B
D1-3                    % 00128 – 3 = 00125 LINENIBS
C=0 A
DAT1=C X                % set the LINENIBS to 0
D0= 00                  % load the last two digits of D0 with
                        % 00 (00120 -> 00100 =BITOFFSET)
C=RSTK                  % get the old bitoffset
DAT0=C X
LOADRPL                 % return to RPL

*PAINT                  % subroutine that waits until the
                        % display refresh is at line 0 then
                        % displays the grob in A
                        % D1= LINECOUNT, when reading it, it
                        % has the current display line
                        % D0= DISPADDR

C=DAT1  B               % read current display line
```

```
?C#0  B  ->  PAINT % until it is 0
DAT0=A  A                     % write addr of grob
*WAIT
C=DAT1 B                      % wait until line is no longer 0
?C=0 B -> WAIT                % or the screen may flicker
RTN                           % return from subroutine
ENDCODE
@”
```

7. The Keyboard

The keyboard can be read from the IN register, it must be run from a even address. This is why we use the romroutine “CINRTN” or “AINRTN”
The IN register is 4 nibbles wide and the bits 0 to 8 are used for the keys, bit 11 is set if ON is pressed. It more or less tells you which row the key is on.
OUT is a three nibble wide register which is used to determine the “column” of the key.

EXAMPLE 7.1 Waiting for a key
This program waits for a key to be pressed and then returns the value of the IN register.

```
“CODE
SAVE
LC 1FF                    % we need to set the out register to
                          % nine ones
OUT=C
A=0 A                     % clear 5 nibbles of A because the IN
                          % register is only 4 nibbles wide
*MAIN
GOSBVL AINRTN             % read the IN register
?A=0 X -> MAIN            % if no key then zero don’t check for ON
GOSBVL Flush              % flush the key, or else it will be
                          % executed after the code
GOVLNG PUSH#ALOOP         % push the number to the stack
ENDCODE
@”
```

EXAMPLE 7.2 Reading the key code

To read a key you need to set a mask in the out register, if you set one bit only keys from that column will result in an IN value.
This program waits for a key press and return the IN and OUT registers to the stack.

```
“CODE
SAVE
ST=0 15                  % turn off interrupts
C=0 A                    % load C(A) with 1
C+1 A                    % this is shorter than LC 00001
A=0 A                    % clear A

*MAIN
P=0                      % set P to zero, it may be three
                         % after the test for IN
C+C A                    % shift C left one bit
?CBIT=0 9 -> RESET       % OUT uses only 9 bits
C=0 A                    % so set bit 0 again
C+1 A
*RESET
OUT=C                    % set the OUT mask
GOSBVL AINRTN            % read the IN register
P=3                      % test only 4 nibbles
?A=0 WP -> MAIN          % if not zero then key pressed from this
                         % column
P=0                      % reset P to zero
R0=A A                   % R0 is the IN value
R1=C A                   % and R1 has OUT
ST=1 15                  % allow interrupts again
GOSBVL Flush             % flush keys
GOSBVL GETPTR            % Get the RPL pointers
GOSBVL PUSH2#            % push R0 and R1 to the stack
GOVLNG LOOP              % back to RPL
ENDCODE
@”
```

EXAMPLE 7.3 Beeping in ML [ADVANCED]
The out register can also be used to make beeps, the process is complicated but luckily we have the “makebeep” romroutine. It takes the pitch in Hz in D(A) and the length in msec in C(A). We shall now look at a piece of code that plays a hxsstring. The format of this string must be:

Hxsprologue, hxslength, pitch1, len1, pitch2, len2, ... ,pitch#n, len#n.
With pitch and len in three nibbles.
EG HXS 0000C 8B1046601046, this would play an A and a C of 1,6 seconds
To easily create these strings take a look at my NOKIA program http://www.hpcalc.org/details.php?id=4698

I've added a song already, you may have to remove the linefeeds in the hxsstring

```
“::
HXS 000FC
C024B0000500C024B0C0286188186149 24B00005004924B0492861C02861C0286
```

```
“::
HXS 000FC
C024B0000500C024B0C02861881861 4924B00005004924B0492861C02861C0286
14924B00138610134B00005000134B0AB24B00005004924B0C42861C424B0000B
00C424B00005004924B0AB2861AB28614924B0000500C424B0492861C02861C0
24B00005004924B0C428610005008814B0EE14B0000500C424B0C02861

CODE
GOSBVL =PopASavptr      % get the addr of the hxs string
D1=A
D1+5                    % point to length of hxsstring
C=DAT1  A               % read length
C-11 A                  % subtract 5 for length and 6 for
                        % predecrease
GONC NULL               % if hxs smaller than 6 nibbles
LC 00203                % error out with “bad argument value”
GOVLNG GPErrjmpC
*NULL
RSTK=C                  % save length in RSTK
A+10 A                  % A has addr of first pitch
R4=A  A                 % save in R4, make beep changes almost
*MAIN                   % every register
A=R4  A                 % get addr of pitch
D1=A                    % point to it
C=0    A                % clear C, since makebeep uses 5 nibbles
C=DAT1  X               % read pitch
D=C    A                % D(A) must have the pitch
D1+3                    % point to length of beep
C=DAT1  X               % read it
D1+3                    % point to next pitch
AD1EX
R4=A  A                 % save it in R4
?D#0 X -> NoPause       % if pitch is 0 then it is a pause
GOSUB WAIT              % do the wait subroutine
GOTO  SKIP              % and skip the beep
*NoPause
```

```
GOSBVL =makebeep        % beep at pitch D(A) for C(A) msec
*SKIP
C=RSTK                  % get the remaining length of the hxs
C-6 A                   % decrease by 6 nibbles per beep
GONC END                % and go to RPL if end is reached
LOADRPL                 % return to RPL
*END
RSTK=C                  % save count in RSTK
GOTO  MAIN              % and return do next beep

*WAIT                   % Wait subroutine, waits for C(A) msec
                        % we need to calculate the number of
                        % ticks you have to wait, so multiply C
                        % with 8.192 ticks per msec
                        % 1/8 + 1/16 + 1/256 + 1/2048 = 0.19189
D0=(5) TIMER2           % point to 8 nibble counter
A=0 W                   % clear A
A=C A                   % get msec's
A+A A
A+A A
A+A A                   % multiply by 8
CSRB A
CSRB A
CSRB A                  % add 1/8 msec's
A+C A
CSRB A                  % C= 1/16 msec's
A+C A
CSR A                   % C= 1/256 msec's
A+C A
CSRB A
CSRB A
CSRB A                  % C= 1/2048 msec's
A+C A
P= 7                    % 8 nibbles
C=DAT0 WP               % read counter
C-A WP                  % subtract number of ticks
*WLOOP
A=DAT0 WP               % read counter
?C<A WP -> WLOOP        % loop until it is time
P=0                     % reset P
RTN                     % and return from subroutine
ENDCODE
;
@”
```

8. Garbage collections

For certain programs you will need a lot of memory, then a garbage collection may be necessary. This code cannot be run from port 1 or 2 or TempOb, the garbage collection might move the code itself and the PC (program counter) would not point to the correct address. After compilation store it in a variable in your directory.
Let's see how it is done.

EXAMPLE 8.1 Garbage collections

```
"CODE
SAVE                    % save rpl pointers
LC 0F000                % 30 kB of room, you can change the
                        % amount to see how the garbage
                        % collections work
R4=C A                  % save it in R4
GOSBVL CREATETEMP       % reserve the mem
GONC MemOk              % if no carry, it all worked
                        % if carry was set there is to little
                        % free mem
GOSBVL GARBAGECOL       % and we need to do a garbagecollection
C=R4 A                  % get the 30 kB again
GOSBVL CREATETEMP       % and try again to reserve the room
GONC MemOk              % if no carry goto MemOk
GOVLNG GPMEMERR         % if still not enough room, error out
                        % with a "not enough memory" error
*MemOk
  AD0EX
  D0=A                  % get the addr of tempob
  LC(5)DOCSTR           % load the string prologue
  DAT0=C A              % write it at the start of tempob
  D0+5                  % point to length of string
  C=R4 A                % size of tempob
  C-5 A                 % subtract 5 nibbles for the prologue
  DAT0=C A              % write length
  GOVLNG GPPushALp      % push it to the stack
ENDCODE
@"
```

To make programs that can run from port 1 or 2 we need to use a little trick.
We will do the garbage collection in SysRPL. It will move the code but the PC will be correct.

EXAMPLE 8.2 Garbage collections from TempOb

It is important that, if your code needs arguments from the stack, you do not change the stack before you test the amount of memory. The code should be "restartable" after the GPMEMERR

```
"::
ERRSET                    ( start the errortrapping structure)
  CODE
  GOTO Start              % goto the Start label in the next CODE
                          % object
  ENDCODE
ERRTRAP                   ( if an error is found do the )
                          ( garbagecollection and start the code)
                          ( again )
::
  GARBAGE                 ( This is only done after an error)
  CODE
*Start                    % Label to jump to from the first CODE
                          % object
  SAVE
  LC 0F000                % reserve 30 kB
  R4=C A                  % save for length
  GOSBVL CREATETEMP
  GONC MemOk              % error if not enough memory
  GOVLNG GPMEMERR
*MemOk
  AD0EX
  D0=A                    % get the addr of tempob
  LC(5)DOCSTR             % load the string prologue
  DAT0=C A                % write it at the start of tempob
  D0+5                    % point to length of string
  C=R4 A                  % size of tempob
  C-5 A                   % subtract 5 nibbles for the prologue
  DAT0=C A                % write length
  GOVLNG GPPushALp        % push the string to the stack
  ENDCODE
;
;
@"
```

If there is not enough memory after the garbage collection you will get the "insufficient memory" error

EXAMPLE 8.3 Reserving all memory [ADVANCED]
There is a romroutine that reserves all the possible memory in a string, but leave room for pushing it to the stack. This is particularly useful when you do not know how much memory you will need. The romroutine is MAKERAM$. It will return the size of the string in D(A). You will have to make sure that you do not write anything outside of the string, so we will use D as a counter. We will now discuss a complicated example. It puts all the words in a string separately in a list. You cannot know how much memory you will need so we will reserve all the memory. We will also show how to leave the string on the stack in case you want to use the GARBAGE trick of example 8.2

```
"::
"word1 word2                ( sample string, you can also use)
word3 word4"                ( the source itself)

CODE
A=DAT1 A                    % get the address but leave it on the
R4=A A                      % stack and save it in R4
SAVE
GOSBVL MAKERAM$             % reserve all memory
A=R4 A                      % get the string address
D1=A                        % point to string prologue
D-10 A                      % D has the remaining nibbles in the
                            % TEMPOB string, decrease 10 nibbles
                            % for the DOLIST prologue and the SEMI
GOC MEMERR                  % if carry then not enough memory so
                            % error out
LC(5)DOLIST
DAT0=C A                    % write DOLIST prologue
D0+5                        % skip it
D1+5                        % skip sample string prologue
C=DAT1 A                    % read size of string
C-7 A                       % subtract 5 and 2 for predecrease
                            % it now has the number of nibbles
                            % minus two of the sample string
GOC NULL                    % if carry then null$
CSRB A                      % number of characters minus one
B=C A                       % use B as a character counter
D1+5                        % skip length
*START                      %
LC 20                       % use any character under #21h as a
                            % separation character
*MAIN
A=DAT1 B                    % read character
```

```
?C<A B -> BLACK         % if A is greater than 20 it is a
                        % character and therefore a word
                        % we have to make a new string in
                        % TEMPOB
D1+2                    % if not it is a separation character
                        % so skip it
B-1 A                   % decrease counter in B
GONC MAIN               % if carry then end of string
*NULL
LC(5)SEMI               % load SEMI
DAT0=C A                % write it, to terminate the string
D0+5                    % skip the SEMI
GOSBVL Shrink$          % Shrink the string
A=R0 A                  % read the address of the TEMPOB string
A+10 A                  % point to the address of the list
GOVLNG GPOverWrALp      % overwrite the string on the stack
                        % and push the list, return to RPL

*MEMERR
GOVLNG GPMEMERR         % jump here if there is to little mem

*BLACK                  % jump here if you find a new word
D-10 A                  % you need 5 nibbles for the prologue
                        % and 5 for the size
GOC MEMERR              % memory error if carry
CD0EX
D0=C                    % get the address of the string in C
R1=C A                  % and R1
LC(5)DOCSTR             % string prologue
DAT0=C A
D0+10                   % skip prologue and length because we
                        % do not know the size of the word yet
A=0 M                   % clear nibbles 3 thru 14 of A
                        % we will use it as a counter, since we
                        % will use only the B field of A for
                        % the characters
LC 21                   % load character 33 any character
                        % smaller than that is a separation
                        % character
*CLOOP
A=DAT1 B                % read character
?A<C B -> WHITE         % check for separation character
D-2 A                   % decrease memory for one character
GOC MEMERR              % error out if not enough memory
```

```
DAT0=A B                % write the character
A+1 M                   % add one to word size counter
D1+2                    % skip character in sample string
D0+2                    % also skip it in the word
B-1 A                   % decrease sample string char counter
GONC CLOOP              % end only if sample string ends
B+1 A                   % then add one to B

*WHITE                  % when word ends jump here
ASR W                   % shift A(M) to A(A)
ASR W
ASR W
A+A A                   % double to get size in nibbles
A+5 A                   % add five for length field
C=R1 A                  % get address of word prologue
CD0EX
D0+5                    % point to string length field
DAT0=A A                % write it
D0=C                    % and put D0 back to addr after word
?B#0 A -> AGAIN         % if B=0 then the sample string ends
GOTO NULL               % we need a GOTO here because
                        % GONC can only jump 256 nibbles
*AGAIN
GOTO START              % start a new word
ENDCODE
;
@"
```

9. Memory Banks

The ports of the 49 are a very good place to keep your libraries and backup data. Sometimes you need to access one of the banks in a port or perhaps in ROM itself. You can do this with ACCESSBank0, ACCESSBank1 to ACCESSBank15. They “open” the bank if P=0 and they “close” it if P=1. Other values for P have different operations but that goes to far here.

EXAMPLE 9.1 Reading the serial number
Let’s read the serial number of your 49, it is in Bank0.

```
“CODE
SAVE
LC 0000A                % it is a string of 10 characters
GOSBVL MAKE$            % so make one!
AD0EX
D1=A                    % get the addr of the body of the
                        % string in D1 and A
P=0
GOSBVL ACCESSBank0      % select the bank0
D0= 40130               % the ID is at this address
LC 00014                % copy 20 nibbles
GOSBVL MOVEDOWN
P=1                     % P=1 means return from that bank
GOSBVL ACCESSBank0
GOVLNG GPPushR0Lp       % push the string to the stack
ENDCODE
@”
```

To work with libraries you should know how to access them in ML. There is a librarytable in memory which has the three nibble library ID and the address of the library itself, as well as the address of the romroutine that you have to call to access the bank in which the library is located. The address of the lib points to the **libid** in the library itself and not to the prologue!
The access routine should be called with a little trick you will see that in the example.

EXAMPLE 9.2 Reading a library title from a port [ADVANCED]

This is an advanced example and you need to know a little about libraries to really understand it.

```
"::
1790                    ( library id of Emacs, you can change this)
CODE
GOSBVL POP#             % get the id
SAVE
LC 101
?A>C X -> LibOk         % check if it is larger than 257
                        % test only last three digits because
                        % that is what you will be working with
LC 00203                % bad argument value error
GOVLNG GPErrjmpC
*LibOk
D0= 8611D               % addr of libcount (number of libs)
C=DAT0 X
B=C X                   % store it in B(X) as a counter
D0-13                   % +3 –16 skip lib count but
                        % "predecrease" because you do a D0+16
                        % next
*Find
D0+16                   % point to next libid
B-1 X
GONC END
GOVLNG GPPushFLoop      % push False if end of libtable, so
                        % that lib is not there
*END
C=DAT0 X                % read lib id
?A#C X -> Find          % if not same then next one
D0+3                    % skip lib id
A=DAT0 A                % read addr of lib
R1=A A                  % save in R0
D0+5                    % point to access routine
C=DAT0 A
R2=C A                  % save it in R2
GOSUB CallBank          % call the access routine
D0=A                    % point to libid in lib
D0-2                    % point to second title length field
                        % title itself will be in front of it
C=0 A                   % clear C
C=DAT0 B                % read title length in bytes
C+C A                   % title length in nibbles
C+10 A                  % add 10 nibbles for prologue and
```

```
                                % length
GOSBVL CREATETEMP               % reserve the room
GONC MemOk                      % if not enough room
GOVLNG GPMEMERR                 % error out
*MemOk
A=R1 A                          % get addr of libid again
AD0EX                           % put it in D0
R0=A A
D1=A                            % and point D1 to TEMPOB
LC(5)DOCSTR                     % string prologue
DAT1=C A                        % write it
D1+5                            % skip it
D0-2                            % point to title length
C=0 A
C=DAT0 B
C+C A                           % length in nibbles
A=C A                           % also store it in A
C+5 A                           % add 5 for length of stringlength
DAT1=C A
D1+5                            % point to body of string
CD0EX                           % get addr of title length field
C-A A                           % subtract length
CD0EX                           % point D0 to beginning of title
C=A A                           % copy C nibbles
GOSBVL MOVEDOWN
P=1                             % P=1 to return from the bank
GOSUB CallBank                  % call the access routine again
P=0                             % P could be 1 if library is in RAM
A=R0 A                          % get the addr of TEMPOB
GOSBVL GPPushA                  % push it to the stack
GOVlNG PushTLoop                % push true and go to RPL

*CallBank                       % routine to access a bank
                                % before calling it R2 should contain
                                % the address of the accessroutine from
                                % the librarytable
C=R2 A                          % get the accessroutine address
?C=0 A RTNYES                   % return if C=0, which means that the
                                % library is in RAM
PC=C                            % point the programcounter to the
                                % access routine
ENDCODE
;
@"
```