The development library

1 Table of contents

2 Introduction

Built in the HP49+ stands a set of function not accessible to the user by default. This library contains lots of low level development tools mainly aimed at system RPL and assembly development.
In order to enable this library, you must attach it (256 ATTACH).
Note: You can also set the flag –86. This will cause the library to attach itself on the next warmstart.
Note: When the library is attached, it appears in the APPS menu.
Note: The tools and programs in this library are extremely powerful, and misusing them may cause memory lost.

3 The tools of the development library

3.1 APEEK

Address PEEK command: Read the address stored at an address.
Example: #80711h APEEK returns the address of the home directory.

Level 1	->	Level 1
Binary integer	->	Binary integer

3.2 PEEK

Memory read: reads nibbles from a specified address in memory.
Note: Due to bank switching, the data read from address #40000h to #7FFFFh may not be accurate.

Level 2	Level 1	->	Level 1
Binary integer (address)	Binary Integer (number of nibbles to read)	->	String

3.3 POKE

Memory write command: Writes nibbles in memory.
Note: you can not write data in the Flash ROM using this command.
Note: Writing data in memory randomly will cause memory lost.

Level 1	Level 2	->
Binary integer (Address where to write)	String (Data to write)	->

3.4 PEEKARM

Memory read: reads nibbles from a specified address in memory in the ARM address space.

Level 2	Level 1	->	Level 1
Binary integer (address)	Binary Integer (number of byte to read)	->	String

3.5 POKEARM

Memory write command: Writes bytes in ARM memory address space.
Note: you can not write data in the Flash ROM using this command.
Note: Writing data in memory randomly will cause memory lost.

Level 1	Level 2	->	Level 1
Binary integer (Address where to write)	String (Data to write in hex)	->	

3.6 A→

Address out: Returns the object stored at a specific address.

Level 1	->	Level 1
Binary integer	->	Object

3.7 →A

Get Address: Returns the address of an object.

Level 1	->	Level 1
Object	->	Binary integer

3.8 →RAM

Improved NEWOB: This command makes a copy of an object in RAM, wherever the object is.

This commands allows you to copy a ROM object in RAM.

Level 1	->	Level 1
Object	->	Object located in RAM

3.9 A→H

Address to string: Returns the hex representation of an address (you can then use this with the POKE command).
The hex representation of an address is a 5 character string where the address is written backwards.

Level 1	->	Level 1
Binary integer	->	string

3.10 H→A

String to address: Returns the address represented by a 5 character string.
The hex representation of an address is a 5 character string where the address is written backwards.

Level 1	->	Level 1
String	->	binary integer

3.11 CD→

Code to hex: Returns the hex representation of a code (Assembly program) object.

Level 1	->	Level 1
Code	->	string

3.12 →CD

hex to Code: Returns the code (Assembly program) object represented by an hex string.
A hex string is a string that only contains the characters ‘0’ to ‘9’ and ‘A’ to ‘F’.

Level 1	->	Level 1
String	->	Code

3.13 →H

Object to hex: Returns the hex representation of an object.

Level 1	->	Level 1
Object	->	string

3.14 H→

hex to object : Returns the object represented by a hex string.
A hex string is a string that only contains the characters ‘0’ to ‘9’ and ‘A’ to ‘F’.
Note: if the string does not represent a valid object, this can corrupt your memory;

Level 1	->	Level 1
String	->	Object

3.15 S→H

String to hex: Returns the hex representation of the characters of a string.
Example: “A” S→H → “14”

Level 1	->	Level 1
String	->	String

3.16 H→S

hex to String : Returns the string which data are represented by a hex string.
A hex string is a string that only contains the characters '0' to '9' and 'A' to 'F'.
Example: "14" H→S → "A"

Level 1	->	Level 1
String	->	String

3.17 SREV

String reverse: gives the mirror image of a string.

Example: "14" H→S → "A"

Level 1	->	Level 1
String	->	String

3.18 MAKESTR

Create a string of the given size.

Example: 10 MAKESTR -> "ABCDEFG<cr>AB"

Level 1	->	Level 1
Real	->	String

3.19 SERIAL

Retrieve the calculator serial number

Level 1	->	Level 1
	->	String

3.20 →S2

Decompile an object in system RPL mode.

Example: << >> →S2 -> "!NO CODE !RPL :: x<< x>> ; <cr>@"

Level 1	->	Level 1
Object	->	String

3.21 XLIB~

Convert reels to an XLIB.

Level 2	Level 1	->	Level1
Real	Real	->	Xlib
Binary	Real	->	Xlib
Real	Binary	->	Xlib
Binary	Binary	->	Xlib

3.22 CRC

CRC computation: gives the CRC of a library or a string.
This command gives you the CRC of the data in a library object or string (the CRC computation starts on the size of the object and finishes 4 nibbles before the end of the object)

Level 1	->	Level 1

String/Library -> System integer

3.23 S~N

String to name conversions: This command converts strings to names and names to strings.
This command allows you to create invalid names.
Note: Do not purge or move the null directory in home. Do not modify data in this directory.

Level 1	->	Level 1
String	->	Global name
Global name	->	String

3.24 R~SB

Real to System binary conversions: This command allows to convert system binary to real and real to system binary.

Level 1	->	Level 1
Real	->	System binary
integer	->	System binary
System binary	->	Real

3.25 SB~B

Binary integer to System binary conversions: This command allows to convert system binary to binary integer and binary integer to system binary.

Level 1	->	Level 1
Binary integer	->	System binary
System binary	->	Binary integer

3.26 LR~R

Long real to real : This command allows to convert long real to real and real to long real.

Level 1	->	Level 1
Long real	->	Real
Real	->	Long real

3.27 LC~C

Long complex to complex: This command allows to convert long complex to complex and complex to long complex.

Level 1	->	Level 1
Long complex	->	Complex
Complex	->	Long complex

3.28 COMP→

Composite out: This is equivalent to the RPL LIST→ command, but it also works on Program and Symbolic objects.

Level 1	->	Level n+1..2	Level 1
List/Program/Symbolic	->	Objects	n (real)

3.29 →ALG

Create symbolic: This is equivalent to the RPL →LIST command, but it creates a symbolic object.
Note: this command will also convert a program or a list in a symbolic object.

Level n+1..2	Level 1	->	Level 1

	List/Program/Symbolic	->	Symbolic
Objects	real (n)	->	Symbolic

3.30 →PRG

Create program: This is equivalent to the RPL →LIST command, but it creates a program object.
Note: this command will also convert a symbolic or a list in a program object.

Level n+1..2	Level 1	->	Level 1
	List/Program/Symbolic	->	Program
Objects	real (n)	->	Program

3.31 →LST

Create symbolic: This is equivalent to the RPL →LIST command, but it can also convert a program or symbolic in a list.

Level n+1..2	Level 1	->	Level 1
	List/Program/Symbolic	->	List
Objects	real (n)	->	List

4 CRLIB

Create library command.
A library is one of the most complex object in the HP49. One of the basic uses of a library is to group all the files of an application.

In order to create a library, you must store in a directory all the variables that will be part of this library. Then, you must store configuration information in some special variables.

The $TITLE variable must contain a character string defining the title of the library. This string must be less than 256 characters long. The first five characters will be used for the name that is shown in the library menu.

The $ROMID variable must contain the library number or your library. This number must be in the range 769 to 1791. In order to avoid conflicts, you should go to www.hpcalc.org to check whether the number is already in use.
This variable may contain either a real or an integer.

The $CONFIG variable contains the library configuration object which is run at warmstart. The basic action that this program should perform is to attach the library to the home directory. Placing a real or an integer in the $CONFIG variable will cause the CRLIB command to generate a default CONFIG object. This Program must leave the stack intact and is not allowed to produce errors.

The $VISIBLE variable contains a list of all the variables in the current directory that you want to have visible in the library menu.

The $HIDDEN variable contains a list of all the variables in the current directory that you want to have invisible in the library. They are generally subprograms of your application.

The $EXTPRG variable contains the name of the extension program of the library. This program must be either a visible or an hidden object of the library. See the Extension program for more information.

Then, once you have specified the required variables, you can type CRLIB to create the library.

-> Level 1
-> Library

4.1 Extension program

It is possible to enhance some of the statistics menu using a user library. The HP49 does not provide every possible functions in every area, but let you customise the built in menu in order to add your functions 'as if they were built in'.

Example: customise the main statistic menu.

Go in RPL mode (MODE, +/-, ENTER) and attach the development library (256 ATTACH).

In a directory, create the following variables:

```
$ROMID          1324
$CONFIG         1
$TITLE          "Statistic enhancements"
$VISIBLE        { ABOUT }
$HIDDEN         { MessageHandler }
$EXTPRG         'MessageHandler'
ABOUT           "This library is a statistic enhancement example"
MessageHandler
        <<
                IF DUP 1 R~SB ==
                THEN
                        SWAP
```

```
                { { "7.New entry" << "My Stats" 1 DISP 7 * FREEZE >> } } +
                SWAP
        END
>>
```

Create the library (CRLIB) and store it in an extension port (0 STO)

Now, run the statistic menu (Shift 5)!

How does it work?
Each time the stat menu pops up, the HP49 executes every Extension Program of the library in the system. This extension Program takes on the stack a message number (and let it on the stack!). Each message number has a specific meaning, as described below.

Here are the I/O expected for the Extension program for different menus:

APPS menu
Input: { { "String" Action } ... } ZERO
Output: Modified list ZERO

Main Statistic menu
Input: { { "String" Action } ... } ONE
Output: Modified list ONE

Hypothesis statistic menu
Input: { { "String" Action } ... } TWO
Output: Modified list TWO

Confidence interval statistic menu
Input: { { "String" Action } ... } THREE
Output: Modified list THREE

Finance menu
Input: { { "String" Action } ... } FOUR
Output: Modified list FOUR

Numeric solver menu
Input: { { "String" Action } ... } FIVE
Output: Modified list FIVE

5 ASM

The Machine Language and system RPL Compiler (Masd)

5.1 Introduction

5.1.1 Warnings

The operating system can not control what a low level program is doing, therefore, any programming error is likely to cause the calculator to crash (with eventual memory lost). A careful developer will always save his source code in the internal flash rom or port 1 for protection before trying his programs,

This document does not intend to be a programming course, it just presents the syntax of the compiler. Ample resources are available on the web (www.hpcalc.org) to learn how to program the Saturn CPU in assembler, how to program in system RPL or how to program in ARM assembly.

With the instruction of the new ARM based series of calculator, some new things have been included that are not backward compatible with previous calculators. The careful programmer should be weary of this.

5.1.2 Starting Masd

To compile a program, put the source code on the level 1 of the stack and type `ASM` (the development library must be attached) or use the `ASM` menu of the Development library.
If you have a new version of MASD packaged as a library 259, the command to type is `asm` (note the lowercase).

5.1.3 Modes

Masd can be used to compile program in 3 different languages: Saturn ASM, ARM ASM and System RPL. Although some things are common to all modes, some are not. As a programmer, you should always know what mode you are in.
Compilation directives instruction are used to switch from one mode to another:
`!ASM` (switch to Saturn ASM mode, referred in the rest of this document as the Saturn mode)
`!RPL` (switch to System RPL mode)
`!ARM` (switches to ARM ASM mode)
In addition, in RPL mode,

```
CODE
% here we are in ASM mode
ENDCODE
```

Switches from RPL mode to Saturn mode (and generates an assembly program object)

5.1.4 Syntax

Masd expects a character string (called source) on the stack level 1.
A source is a set of instructions, comments, and separation characters and ends with a carriage return and an `@` character.
Masd is case sensitive, so be careful, as « `loop` » and « `LOOP` » are two different things for MASD.
Separation characters are those with an ASCII number below 32. They include spaces, tabs, line feed and carriage return.
In Saturn mode, some instructions need a parameter. Separation characters between an instruction and the parameter are spaces, tabs, and points. Therefore `A+B.A` can be used instead of `A+B A`.
In ARM mode, parameters for the instruction are separated by spaces and comas.
In Saturn or ARM mode, comments can be placed everywhere and begin with `%` or `;` and finish at the end of the current line.
In RPL mode, comments are delimited by '`(`' '`)`' as isolated characters and can be multi line. A line that starts with a '`*`' on the first character will also be considered a comment.
Directives change the way Masd interprets your source. Theses directives begin with a `!` and will be explained later.

5.1.5 Errors

If Masd detects one or more syntax error, it will push a list describing all errors on the stack. The `ER` command can help you make sense of that list, point you on the errors and let you correct them.
Masd will report a maximum of 16 errors before stopping compilation.
The `ER` command takes 2 objects as arguments:

The original source code (level 2)

The error list generated by MASD (level 1)

Normally, you should compile using a process similar to: `IFERR ASM THEN ER END` (this is what the `ASM2` command does BTW). Most peoples will just type the `ASM` command followed, if error by the `ER` command.

a) Format of the error list:

It's a list of at most 16 sub-lists.
Each sub-list contains 3 system-binary and 1 global-name.
The first system binary is an error message number.
The second is an extra system binary used to indicate how 'too long' a jump is.
The third one is the position in the source where the error is.
The global name is either a NULLNAME if the error was in the main source or the filename of the buggy source.

b) Error messages

`Invalid File`	The file is not a valid source or macro. (must end with a ℗)
`Too many`	You can not do this operation as you are limited to a certain amount of them (for example, you can not have more than 64 simultaneous skips)
`Unknown Instruction`	Unknown instruction
`Invalid Field`	Incorrect field
`Val betw 0-15 expected`	An integer between 0 and 15 is expected
`Val betw 1-16 expected`	An integer between 1 and 16 is expected
`Val betw 1-8 expected`	An integer between 1 and 8 is expected
`Label Expected`	A label is expected
`Hexa Expected`	An hexadecimal number is expected
`Decimal Expected`	An decimal number is expected
`Can't find`	This object can not be located
`Label already defined`	This name is already in use
`{ expected`	A { character was expected
`} expected`	A } character was expected (this can happen if you do not close all the open skips for example)
`[ or ] expected`	A [or] character was expected
`Forbidden`	This can not be done
`Bad Expression`	This expression is invalid
`Jump too long`	This jump is above the limit of the instruction (use a different type of jump)
`Matrix Error`	You can not do this thing here because you are creating a matrix object
`Define Error`	You can not do this operation in a DEFINE
`ARM register expected`	No comments.
`ARM invalid imediate`	In ARM mode, constants must be representable on 8 bit with an even number of rotation

5.1.6 Links

Links are secondary source files that MASD can be directed to compile (equivalent to the {$I} directive in PASCAL and #include in C). As there is no linking phase with MASD (like in C), a multi source project will usually have the form of a main source file that contains a certain number of links.
An example of main source would be:

```
"
'Constante_definition
'initialization
'graphic_functions
'other
@"
```

When a link call is encountered, Masd suspends compilation of the current source, compiles the new source and then continues compiling the first one.
Program and data in the final object will be in the order in which MASD encounter the links.

Syntax in ASM and ARM mode:

`'FileName` links the file called FileName.

Syntax in RPL mode:
`INCLUDE FileName` links the file called FileName.

Note 1: A link can call other links
Note 2: You can not use more than 64 links in your project
Note 3: To know how Masd looks for files, see the File search section
Note 4: Links are useful to cut projects in independent parts to allow fast and easy access to source code
Note 6: It is beneficial to place all constants definition at the beginning of the compilation process as this will speed up compilation and give more flexibility

5.1.7 Labels

A label is a marker in the program. The principal use of labels is to determine jump destinations.
A label is a set of less than 64 characters different from space, '+', '-', '*' and '/'. A label begins with a star '*' and ends with a separation character.
Syntax in ASM and ARM mode:
`*BigLoop` is the BigLoop label declaration.
Syntax in RPL mode:
`LABEL BigLoop` is the BigLoop label declaration.

Be careful about upper and lower cases!
Three types of labels can be used:

- Global labels

A global label is a label that can be used everywhere in the project, like global variables in Pascal or C.

- Local labels

A Local lab is a label that is only accessible in a local section like local variables in Pascal or C.
A local section starts at the beginning of a source, after a global label or after a link (see link section)..
A local section finishes at the end of a source, before a link or before a global label.
A local label is identified by a '.' as the first character.

- Link labels

A link label is a label that exists only in the link where it is declared, like a private clause in Object Pascal.
A link label is identified by a '_' as the first character.
Note 1: In projects, using less global labels is better because a global label takes longer to compile and because it gives a better program structure. A good habit is to use global labels to cut the program in subroutines, and to use local labels inside these subroutines.
Note 2: The command line editor is able to find labels in a source. See the `GOTO` selection in the command line `TOOL` menu.
Note 3: labels in system RPL should only be used by people who know what they are doing. They are only used for fixed address program (absolute mode) which is pretty advanced programming.
Note 4: Labels can not be given the same name as constants.

5.1.8 "extable"

"extable" is an external library that contains a list of constants.
This list can be used by the Masd as a basic list of constants and is especially useful to the System RPL programmer as most entry points are defined there (like TURNMENUOFF for example).
In addition, it also contains a set of supported constants and ASM entry points for the ASM programmer.
Please read the extable section in this document to find more information about this library.

5.1.9 Constants

Constants are a way for the user to associate a value to an alphanumerical name. This is extremely useful as it makes programs much easier to read and makes them more portable. One of the most popular ways to use constants is to represent memory address for storage of variables.
For example, instead of typing `D1=80100` every time it is needed, it is better to declare
`DC Result 80100` at the beginning of the project and then to type `D1=(5)Result` when needed (it is more portable, more readable and less likely to cause errors).

You can create a constant in ASM or ARM mode by doing:
`DC` *`CstName ExpressionHex`* or
`DEFINE` *`CstName ExpressionHex`* *or*
`EQU` *`CstName ExpressionHex`*
In RPL mode, the only valid way to define a constant is:
`EQU` *`CstName ExpressionHex`*

ExpressionHex is either an hexadecimal number or an expression (starting with a char that can not be confused with the start an hex number (0..9,A..F). An expression starting with a hexadecimal number can be typed with a leading $, an expression starting with a decimal number can be typed with a leading # character. For an expression starting with a constant that starts with a 0..9 or A..F character, you should put the constant in brackets.

Note 1: A constant cannot be given the same name as a label.
Note 2: The name of a constant follows the same rules as the name of a label.
Note 3: A constant value is stored on 16 nibbles.
Note 4: having constants starting with something that can be interpreted as a hex number, or an ARM register is not a good idea as the compiler might get confused. For example: DC SPFOO 4 MOV R4 SPFOO will generate an error on FOO as the compiler will interpret the mov as a mov from SP to R4.

Masd introduces a 'constant pointer' called CP which helps to define constants. CP is defined by:
CP=*ExpressionHex*
CP is defined on 5 nibbles, its default value is 80100 (an area of memory that can be used freely by programmers).

EQUCP *Increment ConstantName*
Declares a constant with the current CP value and then increase CP by Increment.
Note 1: in ASM and ARM mode, DCCP Increment ConstantName is also valid
Note 2: Increment is a hexadecimal value, to use a decimal value, put a leading #.
For example, if CP equals to $10
EQUCP 5 Foo
Defines a Foo constant with a value of $10 and then change the value of CP to $15.
Note:

Several constants can be defined at once using CP.
: *Inc CstName0 CstName1 ... CstNameN-1* :
Defines *N* constants CstName*x* with a value of CP+*x***Inc* and then changes the CP value to CP+*N***Inc*.
By default, *Inc* is a decimal number or an expression that can be immediately evaluated.

These features are extremely useful to define area of memory for storage of ASM program variables.

Note 1: If the entry point library (see related section) is installed on your calculator, all the values in the constant library will be available in your programs the same way than constants are.
Note 2: you can define a constant in your program to override the value of an entry in the equation library.

5.1.10 Expressions

An expression is a mathematical operation that is calculated at compilation time.
Terms of this operation are hexadecimal or decimal values, constants or labels.
An expression stops on a separation character or a '**]**'.

```
DCCP 5 @Data
...
D1=(5)@Data+$10/#2
D0=(5)$5+DUP
LC(5)"DUP"+#5
```

are correct expressions (provided that the entry point library is installed).

Notes:

- A hexadecimal value must begin with a $.
- A decimal value may begin with a # or directly a number.
- A & or (*) equals the offset of the current instruction in the program (This value has no meaning in itself, but may be used to calculate the distance between a label and the current instruction). In absolute mode, this represents the final address of the instruction.
- The value of a label is the offset of the label in the program (This value has no meaning in itself, but may be used to calculate the distance between a label and the current instruction). In absolute mode, this represents the final address of the instruction.

- Entries from the EXTABLE may be used. As the EXTABLE does not have the label names limitations with operators, in ambiguous case (`DUP+#5` may either be an addition `DUP + 5`, or an entry '`DUP+#5`'), add `""` around the word: `"DUP"+#5`.
- Calculations are done on 64 bits.
- X divide by 0 = $FFFFFFFFFFFFFFFF.
- In order to avoid wasting memory, Masd tries to compile Expressions as soon as it see them. If Masd is not able to compile an expression directly, it's compiled at the end of the compilation. In order to use less memory, it's a good idea to define your constants at the beginning of the sources so Masd can compile expression using the constants directly.
- The only operator symbols not allowed in labels are +, -, * and /, therefore, if you want to use a symbol operator after a label, you must put the symbol between " in order to 'limit' the symbol. Meaningless Example: `"DUP"<<5`.
- A label/constant with strange char may be 'protected' between " chars.
- The evaluation stack of MASD allows you to have around 10 pending computations (parenthesis, operator priority).
- Masd only works with integers, you can represent signed value using standard 2's complement, but be careful as all operators are unsigned.

Masd recognises the following operators

Operator	priority	Notes
<<	7	Left Shift: `1<<5 = $20`
>>	7	Right shift `$20>>5 = 1`
%	6	Modulo (remainder of division) X%0=0
*	5	Multiplication
/	5	Division X/0=$FFFFFFFFFFFFFFFF
+	4	Addition
-	4	Subtraction
<	3	Is smaller (true=1, false = 0)
>	3	Is greater (true=1, false = 0)
<=, ≤	3	Is smaller or equal (true=1, false = 0)
>=, ≥	3	Is greater or equal (true=1, false = 0)
=	3	Is equal (true = 1, false = 0)
#, ≠	3	Is different (true = 1, false = 0)
&	2	Logical and
!	1	Logical Or
^	1	Logical Xor

Note: throughout this documentation, you will see talks about expressions that can be "immediately" evaluated. This refers to any expression that contains only number and labels/constants that have already been declared.

5.1.11 Macros and includes

If data are to be included in a project, they can be entered in hex in a source file, using `$`.
But a simpler way is to include data from an external file, which is called a macro. The macro file must be a character string, a graphic, a code object or a list.

- In case of a string or a code, Masd includes only the data part (after the length) of the object
- In case of a graphic, only the graphic data will be included (no length, no dimensions)
- In case of a list, only the first object of the list will be included following the previous rules

The syntax in ASM or ARM mode is:

```
/FileName
```

Note: To know how Masd looks for the FileName file, see the following section.

You can also include a complete object (prologue included) using `INCLUDE` or `INCLOB`.
In ASM or ARM mode, use `INCLUDE` or `INCLOB` followed by a filename to include an object, in RPL mode, use `INCLOB`.

5.1.12 Filename conventions

Masd sometimes needs to find a file in the HP 49 memory.
The file can be found either by specifying the file name and location, or only the file name to be search in the directory search list.

The initial directory search list contains the current directory, and all parents directory up to the HOME directory.
You can add a directory in the directory search list using !PATH+ RepName where RepName identifies a directory name using filename rules.

To specify a full path, use
H/ to specify HOMEDIR as the root.
x/ where *x* is a port number, to specify a port as root (note, you can not use 3 (SD card) here.
This root is followed by a list of directories, ending with the name of the file.
2/FOO/BAR/BRA specifies the BRA file in the BAR directory, stored in the FOO backup of the port 2.
H/ME/YOU specifies the YOU file in the ME directory, in the HOMEDIR.

Note 2: You cannot have more than 16 entries in the directory search path.

5.1.13 Compilation directive

The following instruction modifies the way Masd reacts and compile things. They are valid in all modes:

!PATH+ *DirName*	Add the specified directory in the search path list.
!NO CODE	Masd will not generate a $02DCC prologue but will directly output the data. If the generated file is not a valid object, an error will be generated.
!DBGON	Masd will generate code when DISP or DISPKEY are found in the source.
!DBGOFF	Masd will not generate code when DISP or DISPKEY are found in the source.
!1-16	Switch to 1-16 mode.
!1-15	Switch to 0-15 mode.
!RPL	Switch to RPL mode.
!ASM	Switch to ASM mode.
!ARM	Switch to ARM mode.
!FL=0.*a*	Clear the *a* compilation flag.
!FL=1.*a*	Set the *a* compilation flag.
!?FL=0.*a*	Compile the end of the line if flag *a* is set.
!?FL=1.*a*	Compile the end of the line if flag *a* is clear.
!ABSOLUT *Addr*	Switch to absolute mode. The program begins at the address *Addr*. Note: Masd always consider the prolog $02DCC and code length to be the beginning of the program even if !NO CODE is set.
!ABSADR *Addr*	If in absolute mode, add whites nibbles to continue at the specified address. If not possible, errors.
!EVEN	In absolute mode, cause an error if the directive is not on an even address.
!ADR	Masd will generate a source defining all constants and labels used in the program instead of the program.
!COMPEXP	Cause Masd to calculate all previous expressions.
!STAT	Display/update compilation statistics
!DBGINF	Causes MASD to generate debugging information (see next section for more information)
!JAZZ	See local variable documentation in RPL mode
!MASD	See local variable documentation in RPL mode

a) The !DBGINF directive

If you put the !DBGINF directive into a MASD source, the assembler not only generates your compiled object, but it also returns a string (on level 1) full of debug information. The structure of this string is as follows:

5 DOCSTR
5 Length

5 Number of links (source files)

```
n*[
  2  Number of characters
  .. Name of link file
]

5  Number of symbols (labels and constants)
n*[
  2  Number of characters
  .. Name of symbol
  1  Type: 9=Label 2=Constant
  for labels:    5  Address of label
  for constants: 16 Value of constant
]

5  Number of source->code associations
n*[
  5  Offset in code (this list is sorted by offset)
  2  Number of link this instruction comes from
  5  Character offset in link where this instruction starts
]
```

Notes:

- If the source string is unnamed, i.e. in TEMPOB, the number of links is 00001 and the number of characters is 00, immediately followed by the symbol table.
- The label symbol table is, according to Cyrille, supposed to be an *offset* table. However it seems that the current (1.19-6) MASD has got a bug which makes it put *addresses* into this table. The "associations" table correctly contains offsets.

This instruction is intended for the case where someone decides to create a source level debugger.

5.2 Saturn ASM mode

This section is only applicable to the Saturn ASM mode.

5.2.1 CPU architecture

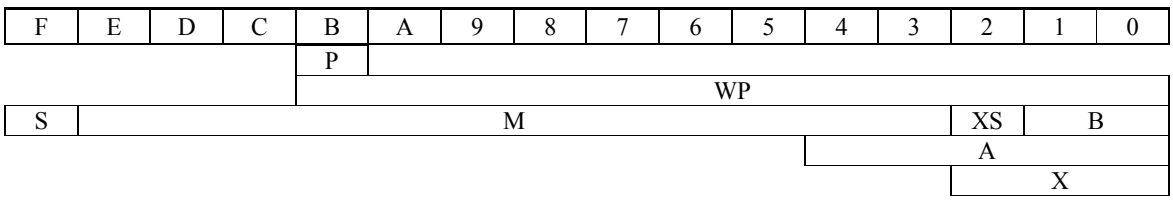
This section has for only aim to familiarise experienced ASM programmers to the Saturn architecture, not to teach anyone to program in Saturn ASM.

The Saturn CPU has 12 main registers:
A, B, C, D, R0, R1, R2, R3 and R4 are 64 bits register (see description bellow),
D0 and D1 are 20 bits pointers (you can only access memory through them, the Saturn is little endian),
PC, 20 bit program counter.

In addition, there are 16 flags ST0 to ST15 (12-15 being reserved for the system) 1 bit register accessible separately, a carry that is set when operation overflow or tests are validated and can be tested using the GOC (Go On Carry) and GONC (Go On No Carry) jump instruction, a decimal/hexadecimal mode (SETHEX and SETDEC) that affects the way + and – instructions on the A, B, C and D register works (default Is HEX), and a 8 level return stack for GOSUBs (and RTN).

a) 64 bits register

Most operations on 64 bits register will act on a specific “field”. A field is a division in a 64 bit register.
If this represents the 16 nibbles of a 64 bit register, the fields cover the register as follows:



The P field location depends of the value of the 4 bit P register (ie: you can move it), and so does the WP field.
Please look at the instruction set to see what instructions are available to the programmer.

I usually write “Rf” to indicate a register (uppercase) and a field (lowercase) as in A.m.

In addition, in the new simulated Saturn architecture, 7 new fields F1 to F7 have been introduced.
You can define the field mask by using the SETFDLn where n is a number between 1 and 7 to define the field Fn using the mask in Cw as in this example:

```
LC FF0000000000000FF SETFLD1
LA 123456789ABCDEF0
LC 0FEDCBA987654321
A=A!C.F1
```

A is now equal to:

```
1F3456789ABCDEF1
```

ie: the instruction on F fields equate to:
reg1= (reg1 & ~ mask) | ((reg1 & mask) operation (reg2 & mask))

These new fields are available for all instruction that previously used the so called ’f’ encoding and includes the following instruction:
Reg=Reg&Reg.f, Reg=Reg!Reg.f, DATx=Reg.f, Reg=DATx.f, Reg=Reg+Cte.f, Reg=Reg-Cte.f, RegSRB.f, RReg=Reg.f, Reg=RReg.f and RegRRegEX.f.

b) Other notes

You should read documentations on the internal structure of RPL objects (www.hpcalc.org has good documentation)
D0, D1, Ba and Da are used by the system (next RPL instruction pointer, RPL stack pointer (@@object on level 1 of the stack), start of free memory and free memory in 5 nibble blocks). The SAVE instruction will save these registers in dedicated memory areas, the LOADRPL instruction will restore them and continue the execution in the system.
Please consult documentation on memory map for more information.

c) New instructions

In addition to the F fields, the following new instructions have been created:

r=s.f, r=r+s.f, r=r-s.f, r=r*s.f, r=r/s.f, r=r%s.f (modulo) r=-s-1.f (logical not), r=-s.f (mathematical not), r=r<s.f (left shift), r=r>s.f (right shift), r=r^s.f (logical xor).
r=1.f (alias for r=r/r.f) has also been created.
Note 1: any combination of the A, B, C and D registers can be used (notted r and s here)
Note 2: all field (including F1-F7 fields) are valid
Note 3: Masd will always choose the shortest version of the instruction (ie: A=A+B.A will use the standard C0 encoding AND affect the carry)
Note 4: the carry is not affected by these instructions.

The following other new instructions have been added (see description in the ASM syntax section):

NATIVE? $hex	GOSLOW	REMON	CONFIGD
HST=1.x	WSCREEN	SERIAL	BIGAPP?
?HST=1.x ( )	SETTIME	OUTBYT	RESETOS
SETFLD(1-7)	SETLNED	MOVEUP	REFRESHD
OFF	SETOFFD	MOVEDN	AUTOTEST
RPL2	HSCREEN	ARMSYS	ACCESSSD
KEYDN	UNCNFGD	ARMSAT	PORTTAG?
CRTMP	GETTIME	REMOFF	MIDAPP?
BEEP2			

5.2.2 Skips

Skips are a first step from ML to a third generation language, even if they are only another way to write ASM instructions.
Skips are wonderful as they allow you to:

- structure your program
- avoid using goto’s
- make programs and piece of code that can be easily copied and past (because there is no label)

The foundation of Skips is the Block structure.
A block is enclosed in { and }, and can be nested within another block.
The following instructions deal with blocks.

SKIPS instructions	Equivalents
{ ... }	Defines a block (generates no code)
SKIP { ... }	GOTO .S ... *.S
SKIPL { ... }	GOTOL .S ... *.S
SKIPC { ... }	GOC .S ... *.S
SKC { ... }	GOC .S ... *.S
SKIPNC { ... }	GONC .S ... *.S
SKNC { ... }	GONC .S ... *.S
Test SKIPYES { ... }	Test GOYES .S ... *.S
Test { ... }	Test GOYES .S ... *.S
Test → { ... }	/Test GOYES .S ... *.S
Test -> { ... }	/Test GOYES .S ... *.S
SKUB { ... }	GOSUB .S ... *.S
SKUBL { ... }	GOSUBL .S ... *.S
STRING { ... }	$/02A2C GOIN5 *.S ... *.S (to create a character string)
CODE { ... }	$/02DCC GOIN5 *.S ... *.S (to create a code object)
STROBJ $*PROLOG* { ... }	$(5)*PROLOG* GOIN5 .S ... *.S (to create a 'prolog – length' object)

/Test is the opposite of Test. For example if Test is ?A<C.A, /Test is ?A>=C.A. The test instructions dealing with the hardware register (?HST=0, ?MP=0, ?SR=0, ?XM=0, ?SB=1, ?HST=1, ?MP=1, ?SR=1, ?XM=1 and ?SB=1) cannot be inverted.

Once blocks are defined, special instructions can be used in them. These instructions called EXIT and UP allow to jump to the end or to the beginning of a block.

These instructions	are equivalent to

{
EXIT
EXITC
EXITNC
?A=0.A EXIT
UP
UPC
UPNC
?A=0.A UP
}
	*.Beginning
	GOTO.End
	GOC.End
	GONC.End
	?A=0.A ".End
	GOTO.Beginning
	GOC.Beginning
	GONC.Beginning
	?A=0.A ".Beginning
	*.End

Note: in Saturn mode do not make confusion between EXIT and UP instructions, which are GOTOs, and EXIT and UP after a test, which are GOYES's.

EXIT and UP can jump to the beginning or to the end of an upper-level block by specifying the number of blocks to exit, after the UP or EXIT instructions.

These instructions	Are equivalent to

{
{
{
UP2
UP3
EXIT1
EXIT3
}
}
}
	*.Beg3
	*.Beg2
	*.Beg1
	GOTO.Beg2
	GOTO.Beg3
	GOTO.End1
	GOTO.End3
	*.End1
	*.End2
	*.End3

Note 1: EXIT1 is equivalent to EXIT, and UP1 is equivalent to UP.
Note 2: the same rules apply in ARM mode: EXITGE3 for example is a BGE for the exit label 3 blocks down

Using SKELSE, SKEC, SKENC, SKLSE instructions, two blocks create an IFNOT-THEN-ELSE structure.

These instructions	Are equivalent to	Or in high-level language

?A=0.A SKIPYES
{
EXIT
UP
	?A=0.A GOYES.Beg2
	*.Beg1
	GOTO.End2 % and not End1
	GOTO.Beg1
	*.End1
		IF NOT A=0 THEN
		BEGIN
		...
		...
		END

```
}                 GOTO.End2          ELSE
SKELSE            *.Beg2              BEGIN
{                   A+1.A               ...
  A+1.A             GOTO.End2           ...
  EXIT              GOTO.Beg2           ...
  UP                *.End2            END
}
```

Note: SKELSE places a GOTO between the 2 bocks, SKEC places a GOC, SKENC a GONC and SKLSE places nothing.

Notes:

UPs are compiled directly when encountered while EXITs and block opening are compiled later on. You can not have more than 64 pending EXITs and block opening simultaneously.

5.2.3 Tests

A test instruction (`?A=0.A`) may be followed by:

- A `GOYES Label`, → `Label` or `-> Label` instruction
- A -> { or → { instruction. In this case, the test is inverted and a skip block is open.
- A `RTY` or `RTNYES` instruction.
- A `SKIPYES {` or `{` instruction. In this case, a skip block is open.
- A `GOTO`, `GOTOL`, `GOVLNG`, `GOSUB`, `GOSUBL` or `GOSBVL`. In this case, the test is inverted and a proper jump instruction is generated (ie: `?A=B.A GOTO A` is compiled as `?A#B.A { GOTO A }`.
- A `EXIT` or `UP`.

5.2.4 SATURN instructions syntax

In this section:

x is a decimal number between 1 and 16. An expression can be used if its value can be determined at the first encounter.

h is a hexadecimal digit.

a is a decimal number ranging from 1 to 16 or a 0 to 15 number depending of the current mode (0-15 or 1-16). An expression can be used, if it's value can be determined at the first encounter.

f is a field A, B, X, XS, P, WP, M, S, F1, F2, F3, F4, F5, F6 or F7.

Reg is a working register A, B, C or D.

RReg is a scratch register R0, R1, R2, R3 or R4.

Exp is an expression.

Cst is a decimal constant. An expression can be used if its value can be determined at the first encounter.

DReg is a pointer register D0 or D1.

Data is memory data pointed by D0 or D1. It means `DAT0` or `DAT1`.

Note: For instructions that use two working registers, instruction using the pairs A-B, B-C, C-D and A-C are smaller and faster (if the `Fn` fields are not used).

For instructions like *Reg1=Reg1…* you can write only *Reg1…* Example: `A=A+C.A` is the same as `A+C.A`.

Syntax	Example	Notes
Reg=0.f	`A=0.M`	Sets the specific field of the register to 0
Reg=1.f	`A=1.M`	Sets the specific field of the register to 1
LC hhh..hhh LA hhh..hhh	`LC 80100` `LA #1024`	The number of nibbles loaded in the register is the number of characters necessary to write the value. So LC #12 will be equivalent to LC 00C. Note: the less significant nibble is loaded in the nibble P (as in the value of the register P) of the register, the next one into nibble p+1 mod 16, and etcetera.
LCASC(x) chrs LAASC(x) chrs	`LCASC(4) MASD` `LAASC(5) ROCKS`	Loads the hexadecimal value of *x* characters into C. *x* must be between 1 and 8. See note on LC instruction
LC(x) Exp LA(x) Exp	`LC(5)@Buf+Off`	Loads the result of an expression into C or A, using x nibbles. See note on LC instruction
Reg1=Reg2.f	`A=B.X`	Copies the value of a specific field of a register into the same field of another register
Reg1Reg2EX.f	`ABEX.W`	Exchanges the value of 2 registers on the given field. Note: this is not valid for the `Fn` fields
Reg1=Reg1+Reg2.f Reg1+Reg2.f	`A=A+B.A` `C+D.A`	Adds the value of the specific field of one register to the other register. Note: If Reg1 and Reg2 are the same, this is a multiply by 2 instruction

		Note: This instruction is affected by the DEC/HEX mode only if the field is not a F field and the registers are AB, BC, CD or AC.
Reg1=Reg1-Reg2.f Reg1-Reg2.f	A=A-B.A C-D.A	The following instructions are also available (but not on the Fn fields): A=B-A.*f* B=C-B.*f* C=A-C.*f* D=C-D.*f* see note on Reg1=Reg1+Reg2.f
Reg=Reg+Cst.f *Reg+Cst.f* *Reg=Reg-Cst.f* *Reg-Cst.f*	A=A+10.A A+10.A A=A-10.A A-FOO.A	Note 1: The Saturn processor is not able to add a constant greater than 16 to a register. If *cst* is greater than 16, Masd will generate as many instructions as needed. Note 2: Even if adding constants to a register is very useful, large values should be avoided because this generates a large program. Prefer another solution like LC(5) Cte A+C.A Note 3: Adding a constant greater than 1 to a P, WP, XS or S field is a bugged SATURN instruction (problem with carry propagation). Use these instructions with care. Note 4: After adding a constant greater than 16 to a register, the carry should not be tested (because you do not know if the last generated instruction generated the carry or not) Note 5: You can put an expression instead of the constant (Masd must be able to evaluate the expression right away). If the expression is negative, Masd will invert the addition in a subtraction and vice versa. Note 6: Be careful when using subtraction, it's easy to be misled. A-5-6.A is equivalent to A+1.A, not A-11.A because the instruction is: A-(5-6).A Note 7: If using Fn fields, be careful if non nibble bounded masks are used.
RegSR.f	ASR.W	Shift register right by 4 bit on the specified field, set SB if bits are lost. Note: this instruction is not available on the Fn fields
RegSL.f	ASL.W	Shift register left by 4 bit on the specified field, set carry if bits are lost. Note: this instruction is not available on the Fn fields
Reg1=Reg1<Reg2.f Reg1<Reg2.f	A=A<B.W	Shift register left by n bits (as defined by the value of Reg2) on the specified field
Reg1=Reg1>Reg2.f Reg1>Reg2.f	A=A>B.W	Shift register right by n bits (as defined by the value of Reg2) on the specified field
RegSRB.f	BSRB.X	Shift register right by 1 bit on the specified field, set SB if bits are lost.
RegSRC	ASRC	Circular right shift by 1 nibble
RegSLC	BSLC	Circular left shift by 1 nibble
Reg1=Reg1&Reg2.f Reg1&Reg2.f	A=A&B.X A&C.B	Logical and on the specified field
Reg1=Reg1!Reg2.f Reg1!Reg2.f	A=A!B.X A!C.B	Logical or on the specified field
Reg1=Reg1^Reg2.f Reg1^Reg2.f	A=A^B.X A^C.B	Logical xor on the specified field
Reg1=-Reg1.f	C=-C.A	Mathematical not on the specified field
Reg1=-Reg1-1.f Reg1=~Reg1.f	C=-C-1.A C=~C.A	Logical not on the specified field
RReg=Reg.f	R0=A.W	Sets the specified field of RReg to the value of the specified field of Reg Only A and C are valid for Reg. If f is W, the shorter encoding of the instruction is used
Reg=RReg.f	A=R0.A	Sets the specified field of Reg to the value of the specified field of RReg Only A and C are valid for Reg. If f is W, the shorter encoding of the instruction is used
RegRRegEX.f	AR0EX.A	Exchanged the value of the specified field of RReg with the value of the specified field of Reg Only A and C are valid for Reg. If f is W, the shorter encoding of the instruction is used
Data=Reg.f Data=Reg.x	DAT1=C.A DAT0=A.10	Write the content of the specified field of the specified register in the memory location pointed by Data register (POKE)

		Reg can only be A or C
Reg= Data.f Reg Data.x	C=DAT1.A A=DAT0.10	Read the content of the memory location pointed by Data register in the specified field of the REG register (PEEK) Reg can only be A or C
DReg=hh *DReg=hhhh* *DReg=hhhhh* *DReg*=(2)E*xp* *DReg*=(4)*Exp* *DReg*=(5)*Exp*	D0=AD D0=0100 D0=80100 D0=(2)label D0=(4)lab+$10 D1=(5)Variable	Change the first 2, 4 or all nibbles of the Data register with the given value
Dreg=Reg	D0=A	Reg can only be A or C
Dreg=RegS	D0=CS	Sets the first 4 nibbles of Dreg with the 4 first nibble of Reg Reg can only be A or C
RegDRegEX	AD0EX	Reg can only be A or C
RegDRexXS	AD1XS	Exchange the first 4 nibbles of Dreg with the 4 first nibble of Reg Reg can only be A or C
DReg=DReg+Cst *DReg+Cst* *DReg=DReg-Cst* *DReg-Cst*	D0=D0+12 D1+25 D1=D1-12 D1-5	Note 1: The Saturn processor is not able to add a constant greater than 16 to a register but if *cst* is greater than 16, Masd will generate as many instructions as needed. Note 2: Even if adding constants to a register is very useful, big constants should be avoided because this will slow down execution, and generate a big program. Note 3: After adding a constant greater than 16, the carry should not be tested. Note 4: You can put an expression instead of the constant (Masd must be able to evaluate the expression right away). If the expression is negative, Masd will invert the addition in a subtraction and vice versa. Note 5: Be careful when using subtraction, it's easy to be misled. D0-5-6.A is equivalent to D0+1.A, not D0-11.A
Please read the section on test above for information on what MUST follow a test instruction. f can NOT be a Fn field.		
?Reg1=Reg2.f	?A=C.B	
?Reg1#Reg2.f	?A#C.A	The HP special character can also be used
?Reg=0.f	?A=0.B	
?Reg#0.f	?A#0.A	The HP special character can also be used
?Reg1<Reg2.f	?A<B.X	
?Reg1>Reg2.f	?C>D.W	
?Reg1<=Reg2.f	?A<=B.X	The HP <= character can be used
?Reg1>=Reg2.f	?C>=D.W	The HP >= character can be used
?RegBIT=0.a ?RegBIT=1.a	?ABIT=0.5 ?ABIT=1.number	Test if a specific bit of A or C register is 0 or 1 Reg must be A or C
	A=PC C=PC PC=A PC=C APCEX CPCEX PC=(A) PC=(C)	Sets Aa or Ca to the address of the next instruction Set PC to the value contained in Aa or Ca Exchange the value of PC with register Aa or Ca Sets PC to the value read at the address contained in Aa or Ca
	SB=0 XM=0 SR=0 MP=0 HST=0.a ?SB=0 ?XM=0 ?SR=0 ?MP=0 ?HST=0.a	SB, XM, SR and MP are 4 bits in the HST register. They can be set to 0 by the specific instruction and tested. SB is set to 1 by RegSR and RegSRB instruction, XM by RTNSXM instruction and SR and MP should always be 0 (hardware related stuff). HST=a sets all bits set to 1 in a to 0 in the HST register. ?HST=a test that all bits set to 1 in a are 0 in the HST register
	SB=1	See above. This is only valid in emulated Saturn

	XM=1 SR=1 MP=1 HST=a ?SB=1 ?XM=1 ?SR=1 ?MP=1 ?HST=1.a	
	P=a P=P+1 P+1 P=P-1 P-1 ?P=a ?P#a P=C.a C=P.a CPEX.a C=C+P+1 C+P+1	The HP special character can be used instead of #
	GOTO label GOTOL label GOLONG Lab GOVLNG hex GOVLNG =Label GOVLNG ="COMND" GOSUB label GOSUBL label GOSBVL hex GOSBVL =Label GOSBVL ="COMND" GOC label GONC label GOTOC label GOTONC label	GOTO is limited to 1kb jumps GOTOL can jump over 16KB of code this jump to a specific address GOSUB is limited to 1kb jumps GOSUBL is limited to &6kb jumps GOSBVL jumps to a specific address GO if Carry set (limited to 64byte) GO if no carry (limited to 64 bytes) Equivalent to SKNC { GOTO label } Equivalent to SKC { GOTO label }
	RTN RTNSXM RTNCC RTNSC RTNC RTNNC RTI RTNYES RTY	Return from subroutine (GOSUB call) RTN + XM=1 RTN + set carry RTN + clear carry RTN if carry set RTN if carry not set Return from interrupt Return if test true (see test section)
	C=RSTK RSTK=C	Pop value from RSTK in Ca Push value from Ca in RSTK
	OUT=CS OUT=C A=IN C=IN	Set the first 2 nibbles of the OUT register to the value of Cb Set the OUT register to the value of C4 Copy the IN register in Ax or Cx (bugged instruction, do not use if you do not know what you are doing)
	SETDEC SETHEX UNCNGF CONFIG RESET SHUTDN INTON INTOFF RSI	Set the CPU in DECIMAL or HEXADECIMAL mode Deconfigure/Configure memory modules deconfigure ALL memory modules STOP the CPU waiting for an interrupt Enable/disable keyboard interrupts Reset interrupt system
	GOINC *label*	Equivalent to LC(5)label-&. (& is the address of the instruction)
	GOINA *label*	Equivalent to LA(5)label-& (& is the address of the instruction)
	$*hhh...hhh* *NIBHEX hhh...hh*	Includes hexadecimal data in the program. Example: $12ACD545680B.
	$/*hhhh...hhh*	Includes hexadecimal data in reverse order. Example: $/123ABC is equivalent to $CBA321.
	$(*x*)*Exp* CON(*x*)*Exp* EXP(*x*)*Exp*	Places the value of *Exp* in the code, on *x* nibbles.
	¢*Ascii* *"Ascii"*	Includes ASCII data. The end of the string is the next ¢ or carriage return. Example: ¢Hello¢. To output a ¢ character, put it twice. To put a char from its number, use **xx** where xx is an hex number. To put a \,

		put the \ chr twice.
	GOIN5 *lab* G5 *lab* GOIN4 *lab* G4 *lab* GOIN3 *lab* G3 *lab* GOIN2 *lab* G2 *lab*	Same as $(x)label-& with *x*=5, 4, 3 or 2. Useful to create a jump table.
	SAVE	Equivalent to GOSBVL SAVPTR
	LOAD	Equivalent to GOSBVL GETPTR
	RPL or LOOP	Equivalent to A=DAT0.A D0+5 PC=(A)
	LOADRPL	Equivalent to GOVLNG GETPTRLOOP
	INTOFF2	Equivalent to GOSBVL DisableIntr
	INTON2	Equivalent to GOSBVL AllowIntr
	ERROR_C	Equivalent to GOSBVL ErrjmpC
	A=IN2	Equivalent to GOSBVL AINRTN
	C=IN2	Equivalent to GOSBVL CINRTN
	OUT=C=IN	Equivalent to GOSBVL OUTCINRTN
	RES.STR	Equivalent to GOSBVL MAKE$N
	RES.ROOM	Equivalent to GOSBVL GETTEMP
	RESRAM	Equivalent to GOSBVL MAKERAM$
	SHRINK$	Equivalent to GOSBVL SHRINK$
	COPY<- COPY← COPYDN	Equivalent to GOSBVL MOVEDOWN
	COPY-> COPY→ COPYUP	Equivalent to GOSBVL MOVEUP
	DISP	Equivalent to GOSBVL DBUG (only if debug is on)
	DISPKEY	Equivalent to GOSBVL DBUG.KEY (only if debug is on)
	SRKLST	Equivalent to GOSBVL SHRINKLIST
	SCREEN	Equivalent to GOSBVL D0->Row1
	MENU	Equivalent to GOSBVL D0->Sft1
	ZEROMEM	Equivalent to GOSBVL WIPEOUT
	MULT.A	Equivalent to GOSBVL MULTBAC
	MULT	Equivalent to GOSBVL MPY
	DIV.A	Equivalent to GOSBVL IntDiv
	DIV	Equivalent to GOSBVL IDIV
	BEEP	Equivalent to GOSBVL makebeep
	NATIVE? $hex	Set carry if native function xx is undefined, clear it if defined.
	HST=1.x	Sets bits in the HST register (XM=1, SB=1, SR=1 and MP=1 are also available). Note: the program ST=0.0 SB=1 ?SB=0 (ST=1.0) will set ST0 to 0 if the calculator is non emulated and to 1 if it is emulated.
	?HST=1.x ( )	Test for HST bits. See HST=1.x comments
	SETFLD(1-7)	See section 5.2.1
	OFF	Turns the calculator off.
	RPL2	Simulates a LOOP (A=DAT0.A D0+5 PC=(A)).
	KEYDN	(C[A]) kbd peeks with immediate rtn CS if keydn. Also - Sets DOUSEALARM flag if [ON][9] sequence. Entry: P=0, HEX Mode, C[A]: #kbd peeks (loop count)
	CRTMP	Abstract: Creates a hole in the tempob area of the specified size + 6 (5 for the link field and 1 for marker nibble). Sets the link field of the "hole" to size+6 and adjusts AVMEM, RSKTOP and TEMPTOP. Entry Conditions: RPL variables in system RAM C(A) contains desired size of hole Exit Conditions: carry clear, RPL variables in system RAM D1 -> link field of hole, D0 -> object position B(A), C(A)= desired size+6

		Error Exits: Returns with carry set when there's not enough memory to create a hole of size+6.
	BEEP2	Entry: C[A]: d ;d=Beep duration (msec) D[A]: f ;f=Beep frequency (hz) P=0 Exit: CARRY:0
	REMON	Enables the remote control mode (ON+R).
	SERIAL	Copy serial number to address pointed to by D1 in Saturn memory.
	OUTBYT	Purpose: Send byte to IR printer Entry: A[B]: Byte Exit: CC, P=0, Byte Sent Alters: P:0, CARRY:0, SETHEX.
	MOVEUP	Abstract: Used to move block of memory to higher address. No check is made to ensure that the source and destination do not overlap. Code is moved from from high to low addresses. Entry Conditions: D0 -> end of source + 1 D1 -> end of destination + 1 C(A) = number of nibs to move (unsigned) Exit Conditions: HEX mode, P=0, carry clear D0 -> start of source D1 -> start of destination
	MOVEDN	Abstract: Used to move block of memory to lower address. No check is made to ensure that the source and destination do not overlap. Code is moved from lower to higher addresses. Entry Conditions: D0 -> start of source D1 -> start of destination C(A) = number of nibs to move (unsigned) Exit Conditions: P=0, carry clear D0 -> end of source + 1 D1 -> end of destination + 1
	ARMSYS	Call a function at global dword address C[0-7]&~3. The function takes should be of the form: U32 f(U32 pc, Chipset* c) { /* put your code here */ return pc; }
	ARMSAT	Call a function at Saturn address C.A&~7. The function should have the following format: U32 f(U32 pc, Chipset* c) { /* put your code here */ return pc; } In RAM asm, this means that as you enter the function, pc is in R0, @Chipset is in R1 and the return address is in LP. R2 and R3 are free to use, and R0 should normally not be modified except if you want to change the PC when exiting the function.
	REMOFF	Stops the remote control (ON+S).
	GOSLOW	Wait for (C[A]/183) ms.
	WSCREEN	Return how many columns the screen contains in Ca
	SETTIME	Sets the RTC time from C[W] in ticks.
	SETLNED	Set number of lines of disp0 from C[B], refresh display.
	SETOFFD	Set offset of display inside disp0 in bytes from C[X]&7FF.
	HSCREEN	Return how many lines the screen contains. In Ca
	UNCNFGD	Unconfigure the 4KB block containing the top 16-line header. This will refresh the header on the display.
	GETTIME	Emulates gettime function in ROM, and also updates the 8192Hz timer accordingly. Purpose:Get current time: (=NEXTIRQ)-Timer2 Return CS iff time appears corrupt. Entry: Timer2 Running Exit: Timer2 Running CC - A:NEXTIRQ (ticks) C:Time (ticks) D:Timer2 (sgn extended ticks) P:0, HEX

		CS - Same as the non-error case but the time system is corrupt because of one of: (1) TIMESUM # CRC(NEXTIRQ) -- CheckSum Error (2) TIMER2 was not running on entry. (3) Time not in range: [BegofTime, EndofTime)
	MIDAPP?	Carry=1 on HP48, 0 otherwise.
	CONFIGD	Configure a 4KB block containing the top 16-line header. C.A = Start address of the block (must be multiple of 4KB). If already configured, unconfig, refresh and re-config.
	BIGAPP?	Carry=1 on HP49, 0 otherwise.
	RESETOS	Reset the calculator (including the OS). This code doesn't return, the calculator restarts at 00000000.
	REFRESHD	Force to refresh the header on the display.
	AUTOTEST	- 003AA: AUTO_USER_TEST - 003A3: MANU_USER_TEST - 0039C: MANUFACTURE_TEST - Other: signed index, OS-specific, see OS_API.doc.
	ACCESSSD	SD Card functions (depending on P, see HP's vgeraccess for more details.
	PORTTAG?	Return port number depending on tag name. Entry: D1: name (size+chars) Exit: A[A]: port number (0-3) D1: after name Carry: clear if ok, set if wrong name

5.3 ARM mode

5.3.1 ARM architecture

For all user intents and purposes the ARM CPU has 16 32 bit registers noted R0 to R15 (R15 is also the program counter, R14 is the link register (ie: a BL (GOSUB) instruction copies the return address in R14 before jumping, a Return From Subroutine is performed by doing MOV PC, LR), and R13 is the Stack pointer).
Each instruction can be conditionally executed depending on the value of 5 flags.
Each instruction can be told to modify or not modify these 5 flags (add the S suffix to the instruction).
Please read the ARM ARM (ARM Architecture and Reference Manual for more information).
Please look at the ARMSAT Saturn instruction and the ARM mode documentation to see the instruction set and the rules of calling ARM code from Saturn code.

5.3.2 Skips

Skips are a first step from ML to a third generation language, even if they are only another way to write ASM instructions.
Skips are wonderful as they allow you to:

- structure your program
- avoid using goto's
- make programs and piece of code that can be easily copied and past (because there is no label)

The basement of Skips is the Block structure.
A block is enclosed in { and }, and can be nested within another block.
The following instructions deal with blocks.

SKIPS instructions	Equivalents
{ … }	Defines a block (generates no code)
SK<Cond> { … }	B<cond> .S … *.S
SKUB<Cond> { … }	BL<cond> .S … *.S

Once blocks are defined, special instructions can be used in them. These instructions called EXIT and UP allow to jump to the end or to the beginning of a block.

These instructions	are equivalent to

{
EXIT<Cond>
UP<Cond>
}
	*.Beginning
	B<Cond> .End
	B<Cond>.Beginning
	*.End

EXIT and UP can jump to the beginning or to the end of an upper-level block by specifying the number of blocks to exit, after the UP or EXIT instructions.

These instructions	Are equivalent to

`{`
`  {`
`  {`
`  UP<Cond>2`
`  UP<Cond>3`
`  EXIT<Cond>1`
`  EXIT<Cond>3`
`  }`
`  }`
`}`
	`*.Beg3`
	`  *.Beg2`
	`  *.Beg1`
	`  B<Cond> .Beg2`
	`  B<Cond> .Beg3`
	`  B<Cond> .End1`
	`  B<Cond> .End3`
	`  *.End1`
	`  *.End2`
	`*.End3`

Note 1: EXIT1 is equivalent to EXIT, and UP1 is equivalent to UP.

5.3.3 Instruction set

Note: for instruction names, the case does not matter.
Register names are:
R0, R1, R2, R3, R4, R5, R6, R7, R8, R9, R10, R11, R12, R13 (or SP), R14 (or LP or LR) and R15 (or PC).
Setting the S flag on an instruction causes the instruction to modify the flags.
Every instruction is evaluated ONLY of the attached condition is true. By default, the instruction is always evaluated.
Separation between arguments can be either ‘,’ or spaces.

Operation	Assembler	Action	S flags
Copy and shift	MOV{cond}{S} Rd, <Oprnd>	d:= <Oprnd>	NZCR
Not	MVN{cond}{S} Rd, <Oprnd>	d:= ~<Oprnd>	NZCR
Add	ADD{cond}{S} Rd, Rn, <Oprnd>	d:= Rn + <Oprnd>	NZCVR
Add w carry	ADC{cond}{S} Rd, Rn, <Oprnd>	d:= Rn + <Oprnd> + Carry	NZCVR
Sub	SUB{cond}{S} Rd, Rn, <Oprnd>	d:= Rn - <Oprnd>	NZCVR
Sub w carry	SBC{cond}{S} Rd, Rn, <Oprnd>	d:= Rn - <Oprnd> - Not(Carry)	NZCVR
Reverse Sub	RSB{cond}{S} Rd, Rn, <Oprnd>	d:= <Oprnd> - Rn	NZCVR
Rev sub w carry	RSC{cond}{S} Rd, Rn, <Oprnd>	d:= <Oprnd> - Rn – Not(Carry)	NZCVR
Multiply	MUL{cond}{S} Rd, Rm, Rs	d:= Rm * Rs	NZR
Multiply Add	MLA{cond}{S} Rd, Rm, Rs, Rn	d:= (Rm * Rs) + Rn	NZR
Compare	CMP{cond} Rd, <Oprnd>	flags:= Rn - <Oprnd>	NZCV
Cmp Negative	CMN{cond} Rd, <Oprnd>	flags:= Rn + <Oprnd>	NZCV
Test	TST{cond} Rn, <Oprnd>	flags:= Rn And <Oprnd>	NZC
Tst equivalence	TEQ{cond} Rn, <Oprnd>	flags:= Rn Xor <Oprnd>	NZC
And	AND{cond}{S} Rd, Rn, <Oprnd>	Rd:= Rn And <Oprnd>	NZC
Xor	EOR{cond}{S} Rd, Rn, <Oprnd>	Rd:= Rn Xor <Oprnd>	NZC
	XOR{cond}{S} Rd, Rn, <Oprnd>	Rd:= Rn Xor <Oprnd>	NZC
Or	ORR{cond}{S} Rd, Rn, <Oprnd>	Rd:= Rn Or <Oprnd>	NZC
BitClear (ÑAnd)	BIC{cond}{S} Rd, Rn, <Oprnd>	Rd:= Rn And Not <Oprnd>	NZC
Branch	B{cond} label	R15/PC:= address	
Gosub	BL{cond} label	R14:=R15/PC, R15/PC:= address	
Load Int	LDR{cond} Rd, <a_mode>	Rd:= [address]	
	LDR{cond} Rd, Label	Rd:= data at label. The label address is calculated relative to the PC. This does not work with constants	
Load Byte	LDR{cond}B Rd, <a_mode>	Rd:= [byte at address] 0 extended	
	LDRB{cond} Rd, Label	Rd:= data at label. The label address is calculated relative to the PC. This does not work with constants	
Multiple load		Stack operations (Pop)	
Inc Before	LDM{cond}IB Rd{!}, {reg list}	! sets the W bit (updates the base register after the transfer)	
Inc After	LDM{cond}IA Rd{!}, {reg list}		
Dec Before	LDM{cond}DB Rd{!}, {reg list}		
Dec After	LDM{cond}DA Rd{!}, {reg list}		
Store Int	STR{cond} Rd, <a_mode>	[address]:= Rd	
	STR{cond} Rd, Label	data at label:= Rd. The label address is calculated relative to the PC. This does not work with constants	
Store Byte	STRB{cond} Rd, <a_mode>	[address]:= byte value from Rd	
	STRB{cond} Rd, Label	data at label:= Rd. The label address is calculated relative to the PC. This does not work with constants	
Multiple Store		Stack operations (Push)	
Inc Before	STM{cond}IB Rd{!}, {reg list}	! Sets the W bit (updates the base register after the transfer	
Inc After	STM{cond}IA Rd{!}, {reg list}		
De Before	STM{cond}DB Rd{!}, {reg list}		
De After	STM{cond}DA Rd{!}, {reg list}		
multiplication	MUL rd, r1 r2	rd=r1*r2	
	MLA rd, r1, r2, r3	rd=r1*r2+r3	
	SMULL rd1, rd2, r1, r2	Signed mul rd1=low r1*r2, rd2=high r1*r2	
	SMLAL rd1, rd2, r1, r2	Signed mul add rd1+=low r1*r2, rd2+=high r1*r2	
	UMULL rd1, rd2, r1, r2	rd1=low r1*r2, rd2=high r1*r2	
	UMLAL rd1, rd2, r1, r2	mul add rd1+=low r1*r2, rd2+=high r1*r2	
*labelName	Creates a label		
$	See $ in ASM mode		
ε, ¢	See ASM mode		

Where cond can be any of:

EQ	NE	CS HS	CC LO	MI	PL	VS	VC	HI	LS	GE	LT	GT	LE
equal	Non equal	Carry set, unsigned >=	< Carry clear	Negative	Positive or 0	Overflow	No overflow	Unsigned >	Unsigned <=	>=	<	>	<=

Oprnd can be of the form:

Immediate value	Cte Note, cte is encoded on 8 bits + a rotation right encoded on 4 bits. This means that not every value is possible.
Logical shift left	Rm LSL Cte Rm < Cte
Logical shift right	Rm LSR Cte Rm > Cte
Arithmetic shift right	Rm ASR Cte Rm >> Cte
Rotate right	Rm ROR Cte Rm >>> Cte
Register	Rm
Logical shift left	Rm LSL Rs Rm < Rs
Logical shift right	Rm LSR Rs Rm > Rs
Arithmetic shift right	Rm ASR Rs Rm >> Rs
Rotate right	Rm ROR Rs Rm >>> Rs

In all cases, Cte must be a decimal value or an expression that can be evaluated immediately.

A_mode can be:

[Rn +/-Cte]	Value of rn + or - constant
[Rn +/-Rm]	Value of rn + or – value of rm
[Rn +/-Rm LSL Cte] [Rn +/-Rm < Cte]	Value of rn + or – value of rm shifted left
[Rn +/-Rm LSR Cte] [Rn +/-Rm > Cte]	Value of rn + or – value of rm shifted right
[Rn +/-Rm ASR Cte] [Rn +/-Rm >> Cte]	Value of rn + or – value of rm shifted arithmetically right
[Rn +/-Rm ROR Cte] [Rn +/-Rm >>> Cte]	Value of rn + or – value of rm rotated right
[Rn +/-Cte]!	Value of rn + or – constante Rn is updated with that value
[Rn +/-Rm]!	Value of rn + or – value of rm Rn is updated with that value
[Rn +/-Rm LSL Cte]! [Rn +/-Rm < Cte]!	Value of rn + or – value of rm shifted left Rn is updated with that value
[Rn +/-Rm LSR Cte]! [Rn +/-Rm > Cte]!	Value of rn + or – value of rm shifted right Rn is updated with that value
[Rn +/-Rm ASR Cte]! [Rn +/-Rm >> Cte]!	Value of rn + or – value of rm shifted arithmetically right Rn is updated with that value
[Rn +/-Rm ROR Cte]! [Rn +/-Rm >>> Cte]!	Value of rn + or – value of rm rotated right Rn is updated with that value
[Rn] +/-Cte	The value used is the value of rn, but rn is then updated with Value of rn + or – constante
[Rn] +/-Rm	The value used is the value of rn, but rn is then updated with Value of rn + or – value of rm
[Rn] +/-Rm LSL Cte	The value used is the value of rn, but rn is then updated with Value of rn

[Rn] +/-Rm < Cte	+ or – value of rm shifted left
[Rn] +/-Rm LSR Cte [Rn] +/-Rm > Cte	The value used is the value of rn, but rn is then updated with Value of rn + or – value of rm shifted right
[Rn] +/-Rm ASR Cte [Rn] +/-Rm >> Cte	The value used is the value of rn, but rn is then updated with Value of rn + or – value of rm shifted arithmetically right
[Rn] +/-Rm ROR Cte [Rn] +/-Rm >>> Cte	Value of rn + or – value of rm rotated right Rn is updated with that value

5.3.4 ARMSAT instruction

When using the ARMSAT instruction, the Saturn pc is in register r0 and the address chipset structure that contains the state of the Saturn CPU is in r1.
That structure has the following elements at the following offsets:

0	P_U32 read_map[256+1]; read_map[x] points on the 2Kb of Saturn address space at Saturn address x<<12
1028	P_U32 write_map[256+1]; read_map[x] points on the 2Kb of Saturn address space at Saturn address x<<12 for write purpose (write_map[x]=0 if x points on some non readable memory)
2056	enum ModulePriority top_map[256+1]; // Type of block on top, to know if new configured block takes over
2316	REG A;
2324	REG B;
2332	REG C;
2340	REG D;
2348	REG R0;
2356	REG R1;
2364	REG R2;
2372	REG R3;
2380	REG R4;
2388	U32 D0
2392	U32 D1;
2396	U32 P, P4, P4_32; // P4 = 4*P, P4_32 = 4*P-32, use setP() to modify P.
2408	U32 ST;
2412	U32 HST;
2416	U32 carry; // 0 or !0
3420	BOOL dec; // 0->hex or 1->dec
	U32 RSTK[NB_RSTK];
	U32 RSTK_i; // Index for next push.
	REG FIELD[32]; // Field masks.
	U32 FIELD_START[32]; // Lowest nibble of the field.
	U32 FIELD_LENGTH[32]; // Length of the field.

Therefore, LDR R2 [R1 2316] allows you to read the lower 32 bits of the Saturn register A.

```
LDR R2 [R1 1]
LDR R3 [R2 1]
```

allows you to read the first 8 nibbles at Saturn address 01008
The following file can be used to declare your saturn chipset structure.

```
"!ASM
CP=0
DCCP #1028 SREAD
DCCP #1028 SWRITE
DCCP #260 SPRIORITY
DCCP 8 SRA
DCCP 8 SRB
DCCP 8 SRC
DCCP 8 SRD
DCCP 8 SR0
DCCP 8 SR1
DCCP 8 SR2
DCCP 8 SR3
DCCP 8 SR4
DCCP 4 SD0
DCCP 4 SD1
DCCP 4 SRP
DCCP 4 SRP4
```

```
DCCP 4 SRP32
DCCP 4 SST
DCCP 4 SHST
DCCP 4 SCARRY
DCCP 4 SDEC
DCCP #32 SRSCK
DCCP 4 SRSTKP
DCCP #256 SFMASK
DCCP #128 SFSTART
DCCP #128 SFLENGTH
@"
```

5.4 System RPL mode

Masd can also compile SysRPL programs (you should read the book "An Introduction to System RPL" before trying to write system RPL programs).
The `!RPL` directive will switch MASD in RPL mode.
Note: if the Flag –92 is Set, Masd starts in `!RPL` and `!NO CODE` mode.

5.4.1 Instructions

In RPL mode, Masd interprets instructions/tokens in the following order.

a) Reals and system binary

If the instruction is a decimal number, a system binary is created (MASD will try, if possible to use the internally defined system binary)
It that number has a decimal point (in the middle, or starts with the decimal point), a real number is created.

b) Unnamed local variables

If the instruction is a recall or a set of a local variable defined by `<<` the correct instruction is generated.

A local environment is created using:
`<< var1 var2 ... varN >>` with $N<23$
These variables have names during compile time, but they are implemented as unnamed local variables, which are faster to access than named local variables.
A local variable is recalled by typing its name or with an optional ending `@`.
Data can be stored in a local variable by typing its name, with a leading or ending `!` or a leading `=`.
Note 1: Local variable are available until the next local definition.
Note 2: The local environment is not closed automatically, use `ABND` or other provided words.
Example:
`<< label1 label2 .. labelN >>` will become:
`' NULLLAM <#N> NDUPN DOBIND` (or `1LAMBIND` if there is only one variable)
And:

```
label1 → 1GETLAM
label1@ → 1GETLAM
=label1 → 1PUTLAM
!label1 → 1PUTLAM
label1! → 1PUTLAM
```

Program example

`::`
`<< A B >>`
`B A!`
`ABND`
`;`
	`::`
	`' NULLLAM TWO NDUPN DOBIND`
	`2GETLAM 1PUTLAM`
	`ABND`
	`;`

Note that it is your responsibility to destroy the local environment using ABND and that Masd does not handle multiple level of definition of local variables, nor it is destroying the current environment, even if ABND is used. Variables defined this way will be valid until a new set of variables are defined.

c) Defines

If the instruction matches a define, the correct code is inserted (see the `DEFINE` instruction)

d) Labels

If the instruction matches the name of a constant or a label, the value of the said constant or label is inserted (if you insert a label, be sure to know what you are doing and to be in absolute mode).

e) extable

If the instruction matches an entry in the extable (see appropriate section at the end of this document) the value associated with this entry is used.

```
DUP
```

Will produce

```
88130
```

Note: Using an external table is much faster than using constants. On the other hand, constants are project dependant, which is not the case of an external table.

f) Tokens

Then, the following instruction are tested:

::	Program prologue $02D9D
; or END	List, Program or Algebraic end $0312B
«	List prologue $02A74
»	List end $0312B
MATRIX	Algebraic matrix object
SYMBOLIC	Algebraic prologue $02AB8
UNIT	Unit prolog $02ADA
FPTR2 ^constant	flash pointer from constant
FPTR bank value	flash pointer from value
# *cst*	System Binary of *cst* value, given in hexadecimal. If there is no spaces between the # and the cst, MASD will try, if possible to use the internally defined system binary
PTR *cst*	Address. PTR 4E2CF generates FC2E4.
ACPTR *cst1 cst2*	Extended pointer with given hexadecimal values for the address and switch address
ROMPTR2 ~*xlib_name*	XLIB object from constant
ROMPTR *LibN ObjN*	XLIB object from value
% *real*	Real number
%% *real*	Long real number
C% *real1 real2*	Complex number
C%% *real1 real2*	Long complex number
"..."	Character string. Special characters can be included by typing \ and the ASCII value on two hexadecimal characters. \ can be inserted by typing \\
ZINT decimalvalue	Integer
ID *name*	Global name (see " for info on character encoding)
LAM *name*	Local name (see " for info on character encoding)
TAG *chrs*	Tagged object
X*xlibName*	XLIB identified by its name. If it is a standard HP48 command (like xDUP), the address is used instead of an XLIB object.
HXS *Size Data*	Binary integer ($02A4E), Size is in hexadecimal and Data is a set of hexadecimal characters. Example: HXS 5 FFA21
GROB *Size Data*	GROB ($02B1E).
LIBDAT *Size Data*	Library data ($02B88).
BAK *Size Data*	Backup ($02B62).
LIB *Size Data*	Library ($02B40).
EXT3 *Size Data*	Extended3 ($02BEE).
ARRAY *Size Data*	Array ($029E8).
LNKARRAY *Size Data*	Linked Array ($02A0A).
MINIFON *Size Data*	Minifont object
ARRY2 *Size Data*	Array object
ARRY [...] ARRY [[.] [.]]	Array object, they can have 1 or 2 dimension. All objects in the array must be of same type
xRplName	If RplName is a RPL instruction, compiles the RPL instruction (or xlib depending)
CHR character	Character object. See rules on " for more information
LABEL labelname	Creates a label at this position. Use carefully
EXTERNAL name xlib name	Equivalent to DEFINE name ROMPTR2 xlibname
FEXTERNAL name fpt	Equivalent to DEFINE name FPTR2 fptrname

rname	
CODE *Size Data*	Code object ($02DCC).
CODE Assembly stuff ENDCODE	Include a code object, change to ASM mode and closed the code object on the next ENDCODE.
NIBB *Size Data* or NIBHEX *Data* or NIBBHEX Data or CON(*Size*) *Expr*	Includes directly hexadecimal data (no prolog).
INCLOB *FileName*	Includes the content of the file *FileName*.
INCLUDE *FileName*	Includes the source of the file *FileName* to be compiled (Like **'** in ASM mode).
LABEL *label*	Defines a label (like ***** in ASM mode).
EQU *CstName ExpHex*	Defines a constant (Like **DC** in ASM mode).
EQUCP *Interleave CstName*	Defines a constant (Like **DCCP** in ASM mode).
DEFINE name ...	Associate the data compiled between the name and the end of the line with the name. After that, if the name is used again, the associated data is placed in the compiled file
DIR VARNAME name1 obj1 VARNAME name2 obj2 ... ENDDIR	Creates a directory containing the objects in the given variables.

5.5 Examples of program using the MASD compiler

```
"!NO CODE !RPL
* This program display a 131*64 graphic in a pretty way :-)
* DO LCD->, run it, and enjoy!
* This program has been created by Philippe Pamart
::
* remove the menu and test for a grob
CK1&Dispatch grob
::
TURNMENUOFF
CODE

% R0a: X
% R1a: Y
% R2a: @ grob

SAVE GOSBVL DisableIntr              % No interrupts
A=DAT1.A D0=A LC 00014 A+C.A R2=A.A  % adr 1st pixels of the grob
D0+10 A=0.W A=DAT0.10 C=0.W LC 8300040 ?A=C.W % test the size
( *.End GOSBVL AllowIntr LOADRPL )   % if not ok, return to RPL

GOSBVL "D0->Row1" D1=A D0-15 C=DAT0.A C-15.A GOSBVL WIPEOUT % erase s
creen
LC 0003F R1=C.W                  % initial position in Z

{
 LC 00082                       % we are ready to scan right to left
 {
  R0=C.A                        % save the counter
  LC 001 GOSBVL OUTCINRTN ?CBIT=1.6 -> .End % If backspace, then stop
  GOSUB .PointAndLine             % test the current point
  C=R0.A C-1.A UPNC              % go one pixel on the right
 }
 A=R1.W A-1.A R1=A.A              % go one line higher
 {                              % ready to scan from right to left
  LC 001 GOSBVL OUTCINRTN ?CBIT=1.6 -> .End % If backspace, then stop
  GOSUB .PointAndLine            % test the current point
  A=R0.A A+1.A R0=A.A LC 83 ?A#C.B UP % go one pixel on the left
 }
 A=R1.A A-1.A R1=A.A UPNC         % go one line higher (if not finish)
}
```

```
GOTO .End

*.PointAndLine                  % This test the current pix, returns
                               % if the pixel is white, draw a line
                               % if it is black
A=R1.A A+A.A C=R2.A C+A.A ASL.A A+C.A % Aa: @ line of pixel in the grob
C=R0.A P=C.0 CSRB.A CSRB.A A+C.A D0=A % D0: point on the pixel to test
,
                               % P = number of the pixel to test in
                               % nibble (in Z/4Z)
LC 2481248124812481 P=0          % Cp: pixel mask
A=DAT0.B A&C.P ?A=0.P RTY        % test the pixel. if white, return
GOSUB LIGNE GOSUB LIGNE         % else, draw line twice in Xor mode
GOSBVL "D0->Row1" D0-20          % and draw the pixel in black.
A=R0.A C=R1.A GOVLNG aPixonB

*LIGNE
GOSBVL "D0->Row1" D0-20          % D0 point on the screen
A=R0.A B=A.A LA 00041           % A/B: X coordinates
C=R1.A D=C.A C=0.A              % C/D: Y coordinates
GOVLNG aLineXor                 % draw the line!

ENDCODE
;
;
@"



"!NO CODE !RPL                   ( turn into RPL mode)
::                              ( open a RPL program )
TURNMENUOFF                     ( remove the menu line )

CODE
                                % open an assembly program

% this program takes control of the screen and
% display a mandelbrot set using the standard algorythm
% ie: for each point from x=-1.5 to 0.5,
%      for each point from y=-1 to 1
%       if any an, n<256 in the serie
%         a0=x+iy (complex number), an+1=a0+an²
%         has an absolute value > 2, the point is not part of the set
% the numbers are stored on 32 bits.
% the numbers are shifted by 12 bits, the lower 12 bits representing
% the decimal part of the number (in 1/4096)

SAVE                            % save the RPL pointers
INTOFF                          % disable keyboard interrupts
SKUB (                          % jump over the ARM code
*start

!ARM                            % switch to ARM mode
STMDB sp! (R4 R5 R6 R7 R8 LP)   % save registers in the stack

LDR R2, [R1, #2324]             % load R2=x (content of saturn
                                % reg B, nibbles 0-7)
LDR R3, [R1, #2340]             % load R3=y (content of saturn
                                % reg B, nibbles 0-7)

MOV R7 R2                       % copy X in r7
MOV R8 R3                       % copy Y in r8
MOV R6 256                      % copy 256 in R6

(
 MUL R4, R2, R2                 % r4= x² << 12
 MOV R4 R4 >> 12                % r4= x²
 MUL R5, R3, R3
 MOV R5 R5 >> 12                % r5= y²
```

```
  ADD LP R4 R5                      % LP = x² + y²
  CMP LP $4000                      % if abs² an > 4
  EXITGT                            % exit

  SUB R4 R4 R5                      % r4= x²-y²

  MUL R3 R2 R3                      % R3= X*Y

  ADD R2 R7 R4                      % r2= X + x²-y² = new x

  MOV R3 R3 >> 11                   % r3= x*y*2
  ADD R3 R8 R3                      % r3= Y+2*x*y = new Y

  SUBS R6 R6 1                      % decrement loop counter
  UPNE                              % up if not 0

  % we have looped 256 times and abs(An)<2, the point is in the set!
  LDRB R6 [R1 2408]                 % clear the flag ST0
  BIC R6 R6 1
  STRB R6 [R1 2408]
  LDMIA sp! {R4 R5 R6 R7 R8 PC}     % restore all registers and return

}

% we have reached a An where abs(An)>2,the point is out of the set
LDRB R6 [R1 2408]                   % set the flag ST0
ORR R6 R6 1
STRB R6 [R1 2408]
LDMIA sp! {R4 R5 R6 R7 R8 PC}       % restore all registers and return

!ASM                                % back in ASM mode
*end
}
C=RSTK D0=C                         % D0 points on ARM instruction
D1=80100                            % D1 points at a place where
                                    % I can copy the program
LC(5) end-start MOVEDN              % copy n nibbles

C=0.B SETLNED                       % hide the header

D1=8229E                            % point on 2Kb free memory
LC A9 A=0.W A-1.W { DAT1=A.W D1+16 C-1.B UPNC } % paint it in black
D0=00120 LC 8229E DAT0=C.A          % point the screen to that memory
D0=C                                % D0 point on that memory

LC FFFFEFFF D=C.W                   % D=Y=-1
LC 4F R3=C.B                        % loop 80 times
{
  C=0.W LC 1800 C=-C.W B=C.W        % B=X=-1.5
  LC 82                             % loop 131 times
  A=0.S A+1.S                       % set bit 0 in As
  {
    RSTK=C                          % save loop counter in RSTK
    LC 80100 ARMSAT                 % evaluate the ARM code
    ?ST=0.0                         % if point is in the set, do nothing
    {
      C=DAT0.S C-A.S DAT0=C.S       % else, turn the pixel off
    }

    A+A.S SKNC { D0+1 A+1.S }       % next pixel

    C=0.W LC 40 B+C.W               % increment X
    C=RSTK C-1.B UPNC               % count down and loop
  }
  D0+2                              % next graphic line
  C=0.W LC 66  D+C.W                % increment Y
  C=R3.W C-1.B R3=C.W UPNC          % count down and loop
}
LC FF OUT=C { C=IN2 ?C=0.B UP }     % wait for no key down
```

```
( C=IN2 ?C#0.B UP )                 % Wait for 1 key down
INTON                               % restore the keyboard interrupt
LC 10 SETLNED                       % restore the header size
SCREEN CD0EX D0=00120 DAT0=C.A      % restore the screen pointer
LOADRPL                             % return to RPL
ENDCODE                             ( end of ASM program )
;                                   ( end of RPL program )

@"
```

6 ASM→

The disassembler converts Saturn assembly into a source string.

The syntax used is Masd syntax, in mode 0-15.
Each line contains an address and an instruction.
If the system flag -71 is set (with `-71 SF`), addresses are not shown, except for the destinations of jumps. In this case, the resulting source may be then reassembled if needed.
ASM-> can either disassemble a CODE object or the memory area between 2 given addresses (as binary integer)
Example:

`-71 CF2` (default)	`-71 SF2`

`AE734 GOSBVL 0679B`
`AE73B LC 01000`
`AE742 C-A A`
`AE744 GONC AE742`
`AE747 GOVLNG 138B9`
	`GOSBVL 0679B`
	`LC 01000`
	`*AE742`
	`C-A A`
	`GONC AE742`
	`GOVLNG 138B9`

Level 2	Level 1	->	Level 1
Binary integer (start address of the memory area to disassemble)	Binary integer end address	->	String (disassemble between the 2 address)
	Code object	->	String

7 ARM→

The disassembler converts ARM assembly into a source string.

Each line contains an address and an instruction.
If the system flag -71 is set (with `-71 SF`), addresses are not shown, except for the destinations of jumps. In this case, the resulting source may be then reassembled if needed.
ARM-> can either disassemble a CODE object (which does not make much sense at this point in time) or the memory area between 2 given Saturn addresses (as binary integer)
Example:

`-71 CF2` (default)	`-71 SF2`

`874FF LDMGEIA R0 ! { R5 R6 R7 LP PC }`
`87507 STMDB R5 { R5 R12 PC }`
`8750F BL 87527`
`87517 BLGE 87527`
`8751F BGE 87527`
`87527 ADD R0 R0 R0`
`8752F MOV R0 R1`
`87537 TST R4 R5`
	`LDMGEIA R0 ! { R5 R6 R7 LP PC }`
	`STMDB R5 { R5 R12 PC }`
	`BL 876E3`
	`BLGE 876E3`
	`BGE 876E3`
	`*876E3`
	`ADD R0 R0 R0`
	`MOV R0 R1`
	`TST R4 R5`
	`@`

Level 2	Level 1	->	Level 1
Binary integer (start address of the memory area to disassemble)	Binary integer end address	->	String (disassemble between the 2 address)
	Code object	->	String

8 The Entry point library: extable

The entry points library is an external library (you can get it from the HP web site) that contains a table of entry points names and address. This is used by the MASD compiler to get the value of system RPL entry points or assembler constants (like TURNMENUOFF for example).

This library should be stored in port 0, 1 or 2.
If you want to program in system-RPL, you must install this library.

This library also contains 4 functions:

8.1 nop

This function is here for internal purpose and should not be used.
Running this function does nothing

8.2 GETNAME

Lookup for the name of an entry from its address.
Example: #054AFh GETNAME -> “INNERCOMP”
Note: as multiple entries might have the same address, GETNAME is not a bijective (one-to-one) function

Level 1	->	Level 1
Binary integer	->	String

8.3 GETADR

Lookup for the address of an entry.
Example: “INNERCOMP” GETADR -> #054Afh

Level 1	->	Level 1
String	->	Binary integer

8.4 GETNAMES

Find all the entry which name starts by a specific string.
Note: giving a null string as an input will return a list of all the entry points.

Level 1	->	Level 1
String	->	List of entry names.