JAZZ v6.5
System RPL and Machine Language Development Library

©1996 Mika Heiskanen   Jan Brittenson

November 25, 1996

Contents

List of Tables

1 Copyright & Acknowledgements

1.1 Copyright

1.2 Extra credits & Acknowledgements

Jan Brittenson	Original author of the machine language debugger. Jan has graciously given permission to modify the program and to include it in *JAZZ*.
Jens Kerle	Bug fixes.
Dan Kirkland	The rewrite of the keyhandler subroutines used in ED and EC. The first sensible sort of the entries tables.
Will Laughlin	Backward searching in ED and calling EC from ED.
Christophe Meynard	Alphabetic sort and alphabetic searching in EC. Fill-key and parametric calls to EC in ED. Screen reformatting and register editing in DB.
Mario Mikocevic	Basis for the machine language instruction disassembler. Author of GNU Tools which were used to develop *JAZZ* and to test new ideas.
Detlef Mueller and Raymond Hellstern	Inspiration through RPL48 package.
Rick Grevelle	Medium font.
Davor Jadricevic	Original small font.
Al Arduengo	Improved small font.
Dominique Rodriguez	First version of the LaTeX documentation.
Cary McCallister	For answering.

Betatesting & suggestions:

Seth Arnold	Douglas Cannon	Carlos Ferraro
Rick Grevelle	Joe Horn	Boris Ivanovich
Jens Kerle	Dan Kirkland	Jeoff Krontz
Will Laughlin	Bill Levenson	Tom van Migem
Mario Mikocevic	Detlef Mueller	Richard Steventon
Kurt Vercauteren	Vladimir Vukicevic	Christ van Willegen
	Stefan Wolfrum	

2 Introduction

2.1 Overview

JAZZ library provides commands which enable assembling, disassembling and debugging both system RPL and machine language on a HP48 calculator. In the development of the library the syntax used by HP was kept closely in mind, thus minimizing the learning time and the work required in transporting source code between *JAZZ*, HP Tools and GNU Tools.

As opposed to the computer based development tools, *JAZZ* is very tightly integrated. Thus for example instead of having four separate phases in library assembly

1. RPLCOMP to compile system RPL to assembly language.
2. MAKEROM to create library header files and tables.
3. SASM to assemble the sources.
4. SLOAD to link the object files.

one can do the same with a single *JAZZ* command. Another example is the string editor ED from which one can for example

- assemble the source code being edited
- study the disassembly of any entry used in the source code
- visit the entries catalog
- visit the regular stack

Below is a brief overview of the commands available in *JAZZ* in the order they appear in the library menu.

Command	Description
ASS	Assembler
DIS	Disassembler
DISXY	Memory area disassembler
DOB	Disassembler with guessing
DISN	Machine language disassembler
DB	Machine language debugger
SDB	SRPL debugger
SHALT	SDB HALT command
EA	Entry name conversion utility
RTAB	RPL.TAB recall command
DTAB	DIS.TAB recall command
RTB→	RPL.TAB to text conversion
→RTB	Text to RPL.TAB conversion
→DTB	Creates a new DIS.TAB
EC	Entries catalog
ED	String editor
TED	User RPL editor
VV	String and grob viewer
SSTK	SRPL stack display
JAZZ	Uncover command for *JAZZ*

2.2 Installing and Deleting *JAZZ*

JAZZ is a regular auto-attaching library (library number 992). *JAZZ* takes approximately 70Kb of memory without entry tables. *JAZZ* does not work from a covered port, thus GX banks 2–33 are not allowed storage locations for *JAZZ*. Thus to install it one needs

a) HP48GX with sufficient RAM or a with a RAM card in slot 1.

b) HP48S/G with sufficient expanded RAM.

c) HP48SX with a RAM card in slot 1 or 2.

To install *JAZZ*

1. Download the file `jazz.lib` into your calculator in binary mode.
2. Put the content of the created variable `jazz.lib` on the stack.
3. Delete the variable `jazz.lib`.
4. Store it in the port of your choice (for example with 0 STO.)
5. Power-cycle the calculator (OFF ON or ON-C).

To delete *JAZZ*

1. If *JAZZ* is stored in a card turn off the write-protection of that card.
2. Exit any running *JAZZ* programs.
3. Detach the library with :0:992 DETACH (1,2 for the cards).
4. Purge the library with :0:992 PURGE (1,2 for the cards). If you get an error message 'Object in use' then do ON-C and start again from 3.

2.3 Installing the Entry Tables

To provide symbolic names for HP48 ROM subroutines *JAZZ* uses preformatted entry tables

RPL.TAB contains entries sorted by their names.

DIS.TAB contains data providing an address sort for RPL.TAB.

To install the tables download the file `hptabs.lib` into your calculator and proceed as in *JAZZ* installation. The tables can be stored in any port, but storing them in a covered port will slow down most *JAZZ* commands. I personally use port 2 anyway.

The purpose of the tables is the same as that of `entries.o` for HP and GNU Tools. Without the tables one would have to use addresses instead of symbolic names whenever using ROM entries, or one would have to write the equivalent equates by hand (as in `entries.a`). In either case programming would be extremely tenuous, and thus even though the tables are not obligatory using them is highly recommended.

The tables provided in the *JAZZ* package were built from HP's latest supported entries table, with some minor modifications which I thought necessary. The tables are stored inside a library for easy storage and access. The library provides both tables, but one can delete DIS.TAB to save memory. Once again though, DIS.TAB provides an address sort to the entry table and thus provides a crucial speed increase for the disassembler, so keeping both the tables is recommended.

The entries library was built from the `hpentr.a` file provided along with the package using a modified version of Deltef Mueller's `gentab` program. The C-source for the program is also included in the package, to use it simply

```
gentab2 < hpentr.a > table.a
```

and assemble `table.a` as instructed with SASM.

2.4 Installing the Universal Font Library

JAZZ requires the universal font library containing the small and medium fonts to be installed in order to work properly. For the user's convenience the proper library is provided with the package.

To install the UFL download the file `ufl3.lib` into your calculator and proceed with the usual library installation procedure. Note however that UFL should be installed in the same port as *JAZZ* itself, otherwise some commands may display garbage due to the very tricky nature some commands have to use to fetch the fonts into use (such as the machine language debugger entry point).

2.5 *JAZZ* homepage

JAZZ homepage is at `http://www.hut.fi/ mheiskan`. The page always contains a link to the latest published version of *JAZZ* as well as links to the very latest beta version and the respective source code. For conveniece the documentation is also available in regular text, hopefully in the future also in LaTeX and related forms.

The page also contains other material available from the author.

2.6 Related Material Available via FTP

This document describes only the provided tools, not the languages themselves. For information on the latter please refer to the tools package published by HP.

The following files should prove useful for any *JAZZ* user:

`hpcvbbs.external.hp.com/dist/ms-dos/tools.exe`

> This is a self-extracting zip-file containing the HP Tools binaries and their documentation. Studying `RPLMAN.DOC`, `RPLCOMP.DOC` and `SASM.DOC` is recommended.

`fly.cc.fer.hr/pub/hp48/gtools/`

> This directory contains the latest version of GNU Tools. GNU Tools provide the full functionality of HP Tools and some more or less significant improvements. As opposed to HP Tools the source code is also available, and the tools have been tested to compile on several platforms. *JAZZ* was developed using the GNU Tools.

`hpcvbbs.external.hp.com/dist/hp48g/programming/entries/ent_srt.zip`

> This file is a subject sorted entry list. As opposed to `RPLMAN.DOC` it also describes assembly language subroutines and unsupported entries. For non-fixed entries both SX and GX addresses are given.

`hpcvbbs.external.hp.com/dist/hp48s/programming/entries/entries.zip`

> This file contains a entry lists sorted by subject and by address. The lists were made for HP48 S/SX models only.

`hpcvbbs.external.hp.com/dist/unix/sadhp105.zip`

> SAD is a very powerful Saturn dissasembler which runs in Unix. The package contains extensive symbols files for both S and G models so one can check whether a given unsupported entry is fixed or not. When disassembling one can ask SAD to do the testing for you—for every entry used in the program.

3 Assembler

3.1 The Assembler Command

Command:	ASS
Description:	Assemble source string
Stack:	(\$ → ob)
	(\$ → \$ %errorpos)
Keys:	`ON` key aborts assembly.
User flags:	1 – Report mode on when set.
	7 – Entry tables off when set.

As opposed to the HP Tools *JAZZ* provides only one command to assemble source code. Instead of detecting special tokens in separate phases the *JAZZ* assembler merely changes the internal modes to expect the tokens suitable for the various modes. Thus the assembler manages to combine RPLCOMP, MAKEROM, SASM and SLOAD programs.

3.1.1 Report Mode

When report mode is on, ASS will show how the assembly proceeds in the status area as follows

```
PC:[pc] Free:[free] Pass:[p][c]
[ Current tokens or source line ]
```

where

pc is the current output location in nibbles

free is the amount of remaining free memory in nibbles

p is the current assembler pass (1-2)

c is "+" if pass 1 is active and pass 2 is known to be needed

3.1.2 Errors

As opposed to HP Tools and GNU Tools, ASS will stop at the first error it encounters. To make the error message as informative as possible, the internal error handling code is never executed, the behaviour is only emulated as far as necessary. The resulting error display is then shown in the status area as follows

```
[Error message] [line][position]
[ Current tokens or source line ]
```

ASS emulates only the standard internal error trap (`SysErrorTrap`). If the trap found by ASS is not `SysErrorTrap` then by default ASS must assume the error trap may be crucial to the program which called the assembler, and so ASS lets it run regularly. Since this also implies that no extended error messages can be shown, ASS provides a way around it by enabling the display if the trap starts with the SRPL `NOP` command. This should be useful for example in internal SOL replacements, and for example SSTK uses the feature.

All the assembler error messages are listed in Appendix A.

3.2 System RPL Assembly

3.2.1 System RPL Tokens

The assembler always starts in SRPL mode. The tokens recognized by ASS are in tables 1 and 3 (page 12).

Tokens	Description
NIBB <len> <hexbody>	Hexadecimal data.
PTR <hex>	Pointer to the given address.
TITLE <text>	Shows <text> on line1 . Line 2 is cleared.
STITLE <text>	Shows <text> on line 2.
INCLUDE <name>	Includes source from named variable.
INCLOB <name>	Includes object from named variable.
DEFINE <label> <text>	Defines replacement text for label.
LABEL <label>	Defines location of a label.
{{ <labels> }}	Lambda name generation.

Table 1: Miscellaneous SRPL Tokens

3.2.2 System RPL Comments

Any line starting with "*" is considered a comment in all modes. In SRPL also anything surrounded by parenthesis is considered a comment. Note that RPLCOMP also requires whitespace after the leading open parenthesis.

3.2.3 Lambda Variable Generation

To ease the use of NULLLAM lambda variables *JAZZ* implements localized lambda bindings as follows

```
{{ label1 .. labelN }}  →  ' NULLLAM <#N> NDUPN DOBIND

           {{ label }}  →  1LAMBIND
```

The maximum number of variables that can be bound is 22 as that is the maximum NULLLAM which has direct PUTLAM and GETLAM ROM entries available. After the bindings the labels can be used as follows

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

For example the following programs compiles to

```
::                     ::
  {{ A B }}              ' NULLLAM TWO NDUPN DOBIND
  B A!          ⟶        2GETLAM 1PUTLAM
  ABND                   ABND
;                      ;
```

Abbrev.	Description
<decint>	Decimal integer.
<dec>	Signed floating point number or 'Inf' '-Inf' or 'NaN'.
<hex>	Hexadecimal integer.
<len>	Hexadecimal integer identifying the length of a <hexbody>.
<hexbody>	Sequence of hexadecimal digits.
<chr>	Character.
<chrbody>	Sequence of characters.
<text>	Sequence of characters until newline.
<label>	A Label.
<labels>	A sequence of labels.

Table 2: RPL Token Abbreviations

Prolog	Address	Object	Tokens
DOBINT	#02911h	Binary Integer	# <hex>
			#<hex>[1]
			<decint>[1]
DOREAL	#02933h	Real Number	<dec>[2]
			% <dec>[2]
DOEREL	#02955h	Extended Real	%% <dec>[2]
DOCMP	#02977h	Complex Number	C% <dec> <dec>[2]
DOECMP	#0299Dh	Extended Complex	C%% <dec> <dec>[2]
DOCHAR	#029BFh	Character	CHR <chr>
DOARRY	#029E8h	Array	ARRY <len> <hexbody>[3]
DOLNKARRY	#02A0Ah	Linked Array	LNKARRY <len> <hexbody>
DOCSTR	#02A2Ch	Character String	"<chrbody>"
			$ "<chrbody>"
DOHSTR	#02A4Eh	Hex String	HXS <len> <hexbody>
DOLIST	#02A74h	List	{ ... }
DORRP	#02A96h	Directory	—[4]
DOSYMB	#02AB8h	Symbolic	SYMBOL ... ;
DOEXT	#02ADAh	Unit	UNIT ... ;
DOTAG	#02AFCh	Tagged	TAG <chrbody> ...
DOGROB	#02B1Eh	Graphics	GROB <len> <hexbody>
DOLIB	#02B40h	Library	LIB <len> <hexbody>
DOBAK	#02B52h	Backup	BAK <len> <hexbody>
DOEXT0	#02B88h	Library Data	LIBDAT <len> <hexbody>
DOACPTR	#02BAAh	Access Pointer	ACPTR <hex> <hex>[5]
DOEXT1	#02BAAh	External 1	EXT1 <len> <hexbody>[5]
DOEXT2	#02BCCh	External 2	EXT2 <len> <hexbody>
DOEXT3	#02BEEh	External 2	EXT3 <len> <hexbody>
DOEXT4	#02C10h	External 4	EXT4 <len> <hexbody>
DOCOL	#02D9Dh	Program	:: ... ;
DOCODE	#02DCCh	Code	CODE <len> <hexbody>
DOIDNT	#02E48h	Identifier	ID <chrbody>
DOLAM	#02E6Dh	Temporary Identifer	LAM <chrbody>
DOROMP	#02E92h	ROM Pointer	ROMPTR <hex> <hex>

[a] Number searched from ROM first.
[b] <dec> can be substituted by 'Inf', '-Inf' or 'NaN'.
[c] Syntax specified in RPLCOMP.DOC is not supported.
[d] Directory assembly is not supported in any tools.
[e] ACPTR can be compiled only in G/GX, EXT1 only in S/SX models.

Table 3: Object Generating Tokens

3.2.4 Escape Sequences

To allow editing source files with no access to special characters, the assembler supports the character escape sequences summerized in table 4. The escape sequences are allowed in

- Character strings (`$ ".."`)
- Identifiers (`ID ..`)
- Temporary identifiers (`LAM ..`)
- Library titles (`xTITLE ..`)

Note that HP Tools and GNU Tools

- may not support all the common escape codes.
- assume character codes are decimal, and use `\xhh` for hexadecimal character codes.
- allow variable width values (prone to error)

Code	Value	Description
`\\`	92	Backslash
`\a`	7	BEL (alert character)
`\b`	8	BS (backspace)
`\t`	9	HT (tabulator)
`\n`	10	LF (linefeed)
`\f`	12	FF (formfeed)
`\r`	13	CR (carriage return)
`\hh`	`<hex>`	Character with hex value hh.

Table 4: Character escape sequences

3.3 Machine Language Assembly

3.3.1 Machine Language Tokens

The assembler always starts in SRPL mode. Machine language can only be assembled when appropriately embedded inside the following tokens

```
CODE      ASSEMBLE
...       ...
ENDCODE   RPL
```

The first pair generates a code object while the latter merely switches to machine language assembly. Note that since the assembler checks the final result for structural validity the latter pair can only be used when not generating any code or when the code will be embedded in a larger object with no strict limitations on the internal structure (such as libraries).

The mnemonics recognized by the assembler are documented by `SASM.DOC` in the HP Tools package, so this manual will not go into depth in explaining them. Instead only the main changes are summerized in the following tables. The following mnemonics are recognized, but are simply ignored:

`EJECT`, `REL`, `LIST`, `LISTM`, `LISTALL`, `UNLIST`

Tokens	Description
D0=D0+ <expr>	Allows values between 1-256.
D0=D0- <expr>	Allows values between 1-256.
D1=D1+ <expr>	Allows values between 1-256.
D1=D1- <expr>	Allows values between 1-256.
TITLE <text>	Text shown on line 1, line 2 cleared.
STITLE <text>	Text shown on line 2.
MESSAGE <text>	Text shown on line 1, line 2 cleared.
SETFLAG <label>	Is interpreted as “label = 1”.
CLRFLAG <label>	Is interpreted as “label = 0”.

Table 5: Mnemonics with changed behaviour.

Tokens	Description
EXITM	Exit macro description.
ABS <expr>	Absolute compile address.
RDSYMB <file>	Reads symbols from an object file.
CHARMAP <file>	Alternative character set.
Dn=HEX <hex>	Loads D0/D1 with a hex number of unspecified width.
GOSHORT <label>	Short jump generation based on carry.
JUMP <label>	Alias for GOSHORT.
INC(n) <label>	Incremental m-nibble reference to label.
LINK <label>	Generate offset to next LINK reference to label.
SLINK <label>	Generate offset to 1st LINK reference to label.

Table 6: Not implemented mnemonics.

Tokens	Description
LCSTR <ascii>	Reversed LCASC.
LASTR <ascii>	Reversed LAASC.
CSTRING <ascii>	NIBASC with a 0-byte terminator (C-style).
ABASE <expr>	Set allocation counter to given address.
<label> ALLOC <expr>	Allocate given amount of nibbles for label.
Dn=Dn+r	f.ex `D0=D0+A` is expanded to `CD0EX C=C+A CD0EX`
Dn=Dn-r	f.ex `D1=D1-C` is expanded to `AD1EX A=A-C AD1EX`
Dn=Dn+P	f.ex `D0=D0+P` is expanded to `CD0EX C+P+1 CD0EX`
Dn=Dn-P	f.ex `D0=D0-P` is expanded to `CD0EX C=-C C+P+1 C=-C CD0EX`
r=Dn	f.ex `A=D0` is expanded to `AD0EX D0=A`
	f.ex `B=D0` is expanded to `CD0EX B=C CD0EX`

Table 7: Implemented GNU Tools mnemonics.

Tokens	Description
NIBBIN <binary>	Binary version of NIBHEX.
NIBGRB <binary>	Grob data (binary) version of NIBHEX.
HEX(n) <hex>	Hexadecimal data with size field of width n.
HEXM(n) <hex>	Hexadecimal data with decremented size field of width n.
ASC(n) <ascii>	Ascii data with size field of width n.
ASCM(n) <ascii>	Ascii data with decremented size field of width n.

Table 8: Not implemented GNU Tools mnemonics.

3.3.2 Machine Language Comments

Any line starting with "*" is considered a comment (in all modes). In addition anything written on a single line after the expected amount of arguments for the instruction on that line is considered a comment.

3.3.3 Object inclusion in machine language

In SRPL mode the `INCLOB` token includes an object into the assembly as is. In machine language mode one usually wants only to include the actual data contained by the object though, thus the objects are stripped from the start based on their type as summarized in the table below.

Prolog	Skip
DOARRY	5
DOCODE	10
DOCSTR	10
DOHSTR	10
DOEXT0	10
DOEXT2	10
DOEXT3	10
DOEXT4	10
DOGROB	20

Table 9: Object stripping in machine language.

3.3.4 Macros

The assembler does not implement macros as described in `SASM.DOC`, in particular there is no argument substitution.
The following is used to define macros:

```
<label> MACRO
        <line1>
        ..
        <lineN>
<label> ENDM
```

After the definition all occurrances of `<label>` are replaced by the source lines in the macro definition. When the macro definition takes only a single line, the following can be used instead:

```
<label> MICRO   <line1>
```

As there are some obvious uses for micros in terms of register assignments, the assembler provides compact way to assign a symbolic name for a scratch register:

```
<label> REG     <scratch register name>
```

For example the following line

```
X       REG     R2
```

is equivalent to the following four micro definitions

```
A=X     MICRO   A=R2
C=X     MICRO   C=R2
X=A     MICRO   R2=A
X=C     MICRO   R2=C
```

Note that as the assembler does not use mnemonic tables for speed, it cannot do full reserved word tests in assembly. Thus the user might mistakingly declare a scratch register name to be P, which of course causes problems if the user intends to use the `C=P` mnemonic elsewhere in the source code. Caution is thus adviced in the use of symbolic names.

3.3.5 Conditional Assembly

JAZZ does not use the unnecessarily complicated condtional assembly opcodes used both in GNU Tools and HP Tools. Instead *JAZZ* enables comparison operators in expressions and thus testing any condition reduces to a single test for a non-zero expression. Thus the only needed opcodes in *JAZZ* are `IF`, `ELSE` and `ENDIF`.

The opcodes are implemented label independantly, thus the label convention described in `RPLMAN.DOC` should not be used either – it would result in duplicate label errors. *JAZZ* automatically follows the matching of the opcode pairs, but due to the simple approach used extra `ELSE`'s might be missed if the nested inside another conditional segment. This causes no ill-effects since it can happen only when `ELSE` code shouldn't be assembled anyway.

The maximum nesting depth for conditional assembly is 64. The matching opcodes must be located in the same source file.

The opcodes are currently implemented only in assembly mode, but the implementation allows using them for SRPL too as long as the conditional assembly opcodes are embedded inside `ASSEMBLE-RPL` tokens. Proper tokens for doing the same directly in SRPL mode are planned in future *JAZZ* versions though via the `#IF`, `#ELSE` and `#ENDIF` tokens.

3.3.6 Expressions

Expressions are evaluated using 64-bit signed integer math, except that the comparison operators use unsigned math. If a value does not fit within 64 bits, the most significant bits are lost.

Term	Example
decimal constant	`123456`
hexadecimal constant	`#123ABC`
binary constant	`%10110001`
global symbol	`=symbol`
local symbol	`symbol`
	`:symbol`
subexpression	`(expr)`
next local label	`+ ++`
previous local label	`- --`

Op.	Pri.	Description
$\wedge$	9	exponentiation
$*$	8	multiplication
`/`	8	devision
`%`	8	modulo
`+`	7	addition
`-`	7	substraction
`&`	5	bitwise AND
`!`	4	bitwise OR
< <=	2	less than (or equal)
> >=	2	greater than (or equal)
== <>	2	equal, not equal

Table 10: Components in expressions.

3.3.7 Label Generation

Sometimes assigning names for insignificant branches can be tiresome. To ease programming the assembler provides local label generation as follows

- `+` refers to the next `+` label
- `++` refers to the next `++` label
- `-` refers to the previous `-` label
- `--` refers to the previous `--` label

For example to search for a newline character one might use

```
        LCASC   '\n'
-       A=DAT0  B       <--+
        D0=D0+  2          |
        ?A#C    B          |
        GOYES   -       ---+
```

Note that overuse can easily make the source code unreadable or cause mistakes if used a too much in a short range.

3.3.8 Debugging

The assembler will recognize `DEBUG` as a special mnemonic and will output a `GOSBVL` call into the proper entry point inside the DB debugger. For more information see the DB documentation on the `=DBGX` and `=DBSX` entry points.

3.4 Library Assembly

The assembler starts library creation mode when the library number (ROMID) has been declared. The assembler cannot produce libraries embedded in other objects, so the declaration has to occur at the start of the source (or after equates). Once in the library mode the assembler expects the optional library title declaration next. Although not obligatory, it is good practice to group all the other library specific declarations at the start too, thus a source code might start like this

```
ASSEMBLE
=symbol EQU     #value
...
RPL
xROMID   <library number>
xTITLE   <library title>
xCONFIG  <configuration object name>
xMESSAGE <message table name>
EXTERNAL <command 1 name>
...
EXTERNAL <command N name>
```

The user should note that HP Tools and GNU Tools use a separate MAKEROM phase to build library specific tables and headers. This is accomplished by using separate loader files, thus the only tokens they use are xROMID and EXTERNAL. In fact in *JAZZ* EXTERNAL is (since v5.6) only provided for the possibility of assigning the command numbers based on the order of introduction. If the declarations are omitted *JAZZ* will assign the numbers based on the order of the location declarations instead.

The assembler will include all that follows the library declarations in the library itself, whatever data there may be. To declare some objects to be actual commands in the library one would then use the following tokens

`xNAME <label>` This is the common way to declare a visible command. The command will be accessible to the user by typing "label", while in the source code the object is referred to as "xlabel".

`NULLNAME <label>` This is used to declare a subroutine which will not be accessible to the user.

`sNAME <label> <hash>` This is a variant of xNAME which eases using special characters in the command name. While one uses "label" in the source code, the user will execute the command by typing "hash".

`hNAME <label>` This is a variant of sNAME where the hash name is declared to be empty. This means that the user cannot execute the command by typing a name. Also, since the library menu stops displaying as soon as a command with no name is encountered, hNAME will 'hide' any command declared after it. This is quite rarely used though.

In addition one can declare a command to have a second (or several if used again) typable name by using

```
tNAME <label> <hash>
```

This can be used to assign names to NULLNAME's too, and is thus a rather useful feature if one wishes to gain access to the low level subroutines. One should note though that the secondary names are only typable, whether the command will have a name that is displayed by the internal decompiler or not is determined by the actual declaration of the command by `xNAME`, `NULLNAME`, `sNAME` or `hNAME`.

The user should note that all visible commands are required to have a property field in front of them. The property field is a collection of flags (bits) which declare the command to have certain properties. The body of the command may then be required to be followed by a sequence of objects associated to the flags in the property field. This document will not explain the properties in detail[1], suffice it to mention that when the library number is below 1792

`CON(1) 8` indicates a *command* with no special properties.

`CON(3) 0` indicates a *function* with no special properties.

For a sample source code which can be assembled with *JAZZ* see Appendix B.

3.5 Differences to HP Tools and GNU Tools

Besides the obvious missing or additional mnemonics, there are several differences to HP Tools by design, of which the most important ones are described next.

1. HP Tools MAKEROM is a separate library builder, thus RPLCOMP does not support any library title, configuration object or message table creation tokens.

2. HP Tools does not support extended precision floating point numbers, nor the special values "Inf", "-Inf" or "NaN".

[1] See Appendix E in the entries.srt document. (Section 2.6, reference ent_srt.zip.)

3. *JAZZ* does not support the ARRY token as defined by HP Tools.
4. *JAZZ* does not support macro argument substition.
5. *JAZZ* implements conditional assembly and the associated `SETFLAG` and `CLRFLAG` opcodes differently.
6. Since the assembly is performed on the calculator itself, there is no need to include the download header line (`NIBASC 'HPHP48-X'`) in the source code.
7. RPLCOMP automatically inserts "::" and ";" at proper places between a `WHILE-REPEAT` structure because `WHILE` expects only a single object before the `REPEAT` command. However HP Tools will also introduce the delimiters when not desired, for example in "`WHILE DROP REPEAT`" where there already is only a single object in between. Thus HP Tools sometimes unnecessarily slows down the code as well as increases the code size.
8. *JAZZ* does not support the special *256+ operator, while HP Tools does not support binary terms nor comparison operators in expressions.
9. Symbol handling allows more than SASM. For example the following would cause an error in HP Tools if used in an expression:

   ```
   (=GETPTR)-(=SAVPTR)
   ```

   Absolutely necessary error checks are done though, for example each expression value is determined to be absolute or relative, and accordingly some mnemonics would error if the type is wrong.

`D0=D0+ label`	→	Error
`D0=D0+ (label2)-(label1)`	→	Valid

10. The Saturn processor contains bugged opcodes, and accordingly SASM will error for example if `A=A+CON fs,expr` is used with a single nibble field selector. *JAZZ* assembler will allow the mnemonics to be used, since although they are bugged they do show predictable behaviour. For those interested, the bug causes the addition (substraction) to extend to the full 64-bit register instead of the single nibble. The extend occurs so that the addition starts from the specified nibble and wraps around to the first nibble. Also the carry is not properly set. To get an idea of the bug one should try DB on the following example

    ```
    CODE
            A=0     W       Clear A[W]
            A=A+CON S,15    Works fine since there is no overflow
            A=A+CON S,2     Just see what happens..
            LOOP
    ENDCODE
    ```

11. As there is no specific linker in *JAZZ*, it must expect some symbols to be resolvable when introduced. Such symbols are those defined by the equate mnemonics (`=, EQU, ALLOC, ABASE`).
12. The `EXTERNAL` token is not implemented as in RPLCOMP. In particular it cannot be used to declare "˜symbol" to be external so that "symbol" would compile to a `ROMPTR`. However using a `DEFINE` in this case would of course be much simpler.

4 Disassembler

4.1 The Disassembler Commands

Command: DIS
Description: Disassemble object.
Stack: (ob → $)
Keys: `ON` key aborts disassembly.
User flags: 2 – Guess mode disabled when set.
4 – Machine language disassembly disabled when set.
5 – Tabulator disabled when set.
6 – Put labels on own rows when set.
7 – Entry tables disabled when set.

Command: DOB
Description: Disassemble object or machine language.
Stack: (ob → $ hxs_end)
(#address → $ hxs_end)
(hxs_start → $ hxs_end)
(“entry” → $ hxs_end)
Keys: `ON` key aborts disassembly.
User flags: 2 – Guess mode disabled when set.
5 – Tabulator disabled when set.
6 – Put labels on own rows when set.
7 – Entry tables disabled when set.

Command: DISXY
Description: Disassemble memory area with guessing.
Stack: (hxs_start hxs_end → $)
Keys: `ON` key aborts disassembly.
User flags: 2 – Guess mode disabled when set.
5 – Tabulator disabled when set.
6 – Put labels on own rows when set.
7 – Entry tables disabled when set.

Command: DISN
Description: Disassembles N machine language instructions.
Stack: (hxs_start %N → $)
Keys: `ON` key aborts disassembly.
User flags: 5 – Tabulator disabled when set.
6 – Put labels on own rows when set.
7 – Entry tables disabled when set.

4.2 Guess Mode

In guess mode the disassembler will try to guess data structures embedded in machine language. Currently only the following types are recognized:

```
        GOSUB   +
        REL(5)  +               Optional leading size indicator
        BSS     <expr>          Data is all zeros
+       C=RSTK
```

```
        GOSUB   +
        REL(5)  +               Optional leading size indicator
        NIBASC  'ASCII'         String data in various formats
        CSTRING 'ASCII'         and possibly spanning multiple lines
+       C=RSTK

        GOSUB   +
        REL(5)  +               Optional leading size indicator
        NIBHEX  <hex>           Miscellaneous data
+       C=RSTK
```

The sufficient condition for an ASCII guess to be succesful is that the area should consist mostly (75%) of common ASCII characters. The ASCII lines are splitted to `CSTRING`'s or by newline characters so that the maximum lenght of each will be 40 characters.

Should the beginning machine language programmer wonder why the guess mode is necessary, suffice it to say that as machine language is not a sturctured high-level language, there is no way of telling when machine language ends, or even when it starts. In particular, there's no telling when somebody has embedded for example a data table inside machine language. A call to a `C=RSTK` instruction is a common way to pass the address of such a table, but there are other ways too which the disassembler cannot reasonably be expected to guess. Thus there are cases in which the disassembler is bound to fail.

4.3 DOB End Address Guessing

When given only a start address DOB will try to guess the end address of disassembly as follows

- A leading `DupAndThen` is skipped.
- For skippable RPL objects the end address is the end address of the object.
- For primitive code objects the first 5 nibbles are skipped, then machine language is skipped until a new PCO or an RPL object is found.
- In machine language
  - all normal instructions are included.
  - if a forward branch forward is found then scanning continues at the target address of the branch.
  - if a short jump forward is found then scanning continues at the target address of the jump.
  - if a backward jump to an address smaller than the start address is found then the the jump terminates the scan.
  - a double backward branch terminates scan.
  - scanning is terminated if one of the following instructions is encountered:

    `GOVLNG`, `GOLONG`, `PC=(A)`, `PC=(C)`, `PC=A`, `PC=C`, `APCEX`, `CPCEX`, `RTNSXM`, `RTN`, `RTNSC`, `RTNCC`, `RTI`.

4.4 ROMPTR Name Hook

The entry tables contain only ROM entry addresses, anything else such as `ROMPTR`'s cannot be stored in it. For convenience all disassembler commands check for variable "Romps" in home director which can contain names for `ROMPTR` objects. This feature should be especially useful when debugging libraries, as then SDB can show the names of the subroutines one is debugging. The variable should contain a list of the form

```
{
  ROMPTR <hex> <hex>   ID name1
  ROMPTR <hex> <hex>   ID name2
  ...
}
```

Just to advertise another product by the author, the Profiler library which performs execution time and crash analysis on libraries also provides a command which creates the "Romps" variable given a text file containing the names of the various `ROMPTR`'s.

4.5 Warnings

DOB, DISXY and DISN disassemble areas of memory instead of well defined objects. Using these commands to disassemble memory in the temporary object area is dangerous since a possible grabage collection during disassembly can move the memory area being disassembled. Use only DIS to disassemble objects in temporary object area!

Composite type is tracked up to 64 levels. If the limit is exceeded a ";" may be output when a "}" is due.

As opposed to the assembler the disassembler cannot handle hidden hash tables, thus some named `ROMPTR`'s will be disassembled to "`ROMPTR <hex> <hex>`".

Label values are guessed for `Dn=(2)` and `Dn=(4)` instructions if

- `Dn=(2)` is likely to refer to the control device.
- `Dn=(4)` is likely to refer to a RAM variable.

This works well for disassembling ROM but does badly when disassembling a program which uses the even-page method in it's data allocation. The benefits are clearly greater though.

5 System RPL Debugger

5.1 SDB Command

Command:	SDB
Description:	Debug system RPL program. If the debugger is already running then only shows the SDB menu.
Stack:	(seco → ?)
	(id → ?)
	(lam → ?)
	(romptr → ?)

SDB is a system RPL debugger, not a user RPL debugger. There is no special code to debug user RPL commands, consequently some of them will not be single stepped correctly. One example is `xHALT`, for which the substitute `xSHALT` command is provided.

Debugging SRPL is a very complicated task and undoubtedly SDB cannot debug some lesser known commands correctly. If such commands are encountered SDB can even cause a crash and memory loss. This is unfortunately unavoidable since there really is too much code in ROM to worry about. SDB should manage to debug most normal programs though.

SDB either enters into programs or executes them (by emulation for known tricky programs). None of the interactive commands in HP48 ROM are emulated, in particular `ParOuterLoop`. To emulate it one must insert SHALT commands into the `ParOuterLoop` parameter programs, then start SDB and use `CONT` to reach the points where the SHALT commands are.

5.2 SDB Menu

When started SDB will show a menu of the following commands. If lost during debugging the menu can always be recovered by executing SDB again.

→SST	Single step next command in RPL stream. If right-shifted then single-steps rest of the stream as a single unit.
→IN	If possible then enter the program referred to by the next command, else single step it.
SNXT	Show next commands on status area. If pressed for a second time shows the RPL return stack.
SST→	Starts continuous →`SST` mode, subsequent presses toggle slow/fast mode, e.g. whether stack display is updated after each step or not. Any other key aborts continuous evaluation.
IN→	Starts continuous →`IN` mode.
DB	Start DB on next code or PCO object in stream.
KILL	The HP48 `KILL` command.
SKIP	Skip next command. If right-shifted then skips rest of the current stream, e.g. executes a `SEMI`.
SEXEC	Excecute program on stack level 1 as the "next" command.
SBRK	Set object on stack level 1 to be a breakpoint object. If right-shifted clears breakpoint object.
LOOPS	Browse loop environments. Up/down keys scroll display, any other key exits. If right-shifted dumps the topmost loop environment to the stack.
LAMS	Browse lambda environments. Up/down keys scroll display, left/right keys decrease/increase environment depth, any other key exits. If right-shifted dumps topmost lambda environment to the stack.
IN?	Toggles →`IN` mode to never enter into secondaries, only into `ID`'s, `LAM`'s and `ROMPTR`'s when allowed. Prevents the debugger from entering into ROM subroutines during continous debugging.

6 Machine Language Debugger

6.1 DB Command

Command:	DB
Description:	Debug machine language.
Stack:	(id → ?)
	(romptr → ?)
	($entry → ?)
	(#address → ?)
	(hxs_address → ?)
	(code → ?)

DB is a machine language debugger which permits one to single-step machine language as well as examine register and memory contents. Since it single-steps machine language only, it is not generally useful for debugging RPL, unless one wishes to follow RPL execution on machine language level.

When invoked DB expects an argument in level one which directly refers to machine language either by address, type (code object) or an object whose content is a code object. If no such reference is found the debugger returns immediately.

6.2 MLDLpar Variable

The debugger stores all it's internal variables into a variable called "MLDLpar". The variable will be created automatically when DB is executed. The size of the variable is 758 bytes.

6.3 DB Entry Hooks

Whenever DB is started it also initializes special entry hooks to enable starting DB directly from machine language. To call these hooks one would then use

`GOSBVL #70520` on HP48 S/SX models

`GOSBVL #806A0` on HP48 G/GX models

The assembler provides a *JAZZ* specific token named `DEBUG` which will automaticall be assembler to the correct model-specific call. The disassembler will output "`DEBUG`" whenever it encounters these special calls.

One should note that the hooks are initialized only by DB command itself, thus one cannot just blindly call the hooks, one must make sure the hooks have been initialized. A convenient and easy way to do this is to execute "0 DB" which initializes the hooks, sets up the proper display size for DB and finally aborts as the argument is invalid.

However the hooks cannot do all the initializations like in regular DB command execution, in particular

- The entry tables cannot be searched for.
- The display size cannot be initialized normally.

However since lambda variable search is very easy to do in machine language, the hooks will try to initialize the tables for DB by searching for a special lambda variable used by SSTK to store the tables. Thus starting SSTK (or SDB) before DB ensures the tables will always be available.

The text display grob which is used to display DB screens is usually 7 lines high, while DB uses the expanded 8 line size. The expansion cannot be done in machine language without violating the CPU register contents (which the hook saves), so one should still resort to using "0 DB" for the initialization. If the display size is invalid on entry, the graphics grob might be corrupted and memory could be lost.

User flag 1 disables the entry hooks so that they merely return back to the calling place instead of starting DB. This enables keeping the `DEBUG` tokens in source code while trying out regular runs in development phase.

6.4 DB Screens

To get an idea of what is possible with the debugger we start by first describing the screens in DB. The user should not feel obliged to go through the details for each screen yet, one can always come back to this section when needed.

The sample screens in this section can be reasonably reproduced by starting DB with

```
"#>HXS" DB (or #59CCh DB)
```

and then by pressing the key assigned for each screen.

Some people have found the small font screens too hard to read (or just too hard to get used to), therefore since *JAZZ* version 5.9 some screens have alternatives which are displayed with the medium font instead. Toggling the font size is done with the [SPC]-key.

At the moment sample screens for the medium font screens are not available. The comments for each screen will mention whether a medium font alternative exists.

6.4.1 Screen 1 – General CPU State ([A])

Mnemonic	`GOSUBL  SAVPTR`
Opcode	` 8E4CD0`
PC, P, CRY, HEX/DEC mode, ST	`@059C1 P:0 CH ST:B318`
A[A], C[A], MP, SR	`A:059CC C:B1CB7 MP SR`
B[A], D[A], SB, XM	`B:A90E1 D:01BF8 SB XM`
D0 and 6 bytes @D0	`D0:2345C/6B436F6A04B2`
D1 and 6 bytes @D1	`D1:B1CBC/000000000000`
Top 3 levels of RSTK	`RST:00000:00000:00000`

- HST-bit names are shown in inverse when set in all screens.

6.4.2 Screen 2 – General CPU State II ([B])

PC, Opcode, Mnemonic	`059D1 8E5CD0 GOSUBL SAVPTR`
A[W], P, ST	`A:6A51F4941A9059CC P:0 ST:B318`
B[W], ST[X] bits	`B:00000000000A90E1 0011 0001 1000`
C[W], CRY, HEX/DEC, HST	`C:B0000000077B1CB7 CH MPSRSBXM`
D[W], D0 and 5 nibbles @D0	`D:9800000000001BF8 D0:2345C/6B436`
R0[W], D1 and 5 nibbles @D1	`0:00000000000A51E9 D1:B1CBC/00000`
R1[W], RSTK levels 1 and 5	`1:6A51F4941A9059D1  00000 00000`
R2[W], RSTK levels 2 and 6	`2:6A51F4941A9CEF94  00000 00000`
R3[W], RSTK levels 3 and 7	`3:6A51F4941A900000  00000 00000`
R4[W], RSTK levels 4 and 8	`4:00000000000B2302  00000 00000`

- Has medium font alternative which shows registers A[W]-D[W], D0 and D1.

6.4.3 Screen 3 – CPU State & Instruction Stream ([C])

```
A[A], C[A], D0, @D0 ............... A:059CC C:B1CB7 D0:2345C/6B436F6A
B[A], D[A], D1, @D1 ............... B:A90E1 D:01BF8 D1:B1CBC/00000000
Instruction 1, P, CRY, HEX/DEC ...  059D1 GOSUBL  SAVPTR    |P:0  CH
Instruction 2, ST ................  059D7 GOSUBL  POP#      |ST:B318
Instruction 3, HST ...............  059DD R1=A              |MPSRSBXM
Instruction 4, R0[A] .............  059E0 C=0     A         |R0:A51E9
Instruction 5, R1[A] .............  059E2 LC(1)   5         |R1:059D1
Instruction 6, R2[A] .............  059E5 GOSUB   MAKE$N    |R2:CEF94
Instruction 7, R3[A] .............  059E9 C=R1              |R3:00000
Instruction 8, R4[A] .............  059EC DAT0=C  A         |R4:B2302
```

- Has medium font alternative which shows registers R0[W]-R4[W] and RSTK.

6.4.4 Screen 4.1 – Data Stream ([D])

```
D0 Memory Dump .................... D0:409C1/9CB0428A0459A04C4A040961
D0 Ascii Stream ...................  40981:.@+1.......a...C#..$.@+1
...................................  409B1:..........$.@....@'.v..H
...................................  409E1:.'.].DU$...'.6"..+....-.
...................................  40A11:..5@...+@....?...+10"..o
D1 Memory Dump .................... D1:B5CA1/000000000000000000000000
D1 Ascii Stream ...................  B5C51:..........p....@....:...
...................................  B5C81:.?p........(.......zz...
...................................  B5CB1:......................a.
...................................  B5CE1:..a>...............h..p.
```

- Current D0/D1 position is shown in inverse.
- [√] will toggle displaying only D0,D1 or both.
- [D] switches to screen 4.2

6.4.5 Screen 4.2 – RPL Stream ([D][D])

```
D0 RPL Stream ..................... D0:409C1 AtUserStack
...................................    409C6 KeyOb@
...................................    409CB KeyOb0
...................................    409D0 Custom?
...................................    409D5 NOTcaseDROP
D1 RPL Stream ..................... D1:B5CA1 PTR 00000
...................................    B5CA6 PTR 00000
...................................    B5CAB PTR 00000
...................................    B5CB0 PTR 00000
...................................    B5CB5 PTR 00000
```

- [√] will toggle displaying only D0,D1 or both.
- [∧] [0-8] will show stack level N instead of D0.
- [D] switches to screen 4.3

6.4.6 Screen 4.3 – Data Table Stream ([D][D][D])

D0 Table Stream	D0:409C1 CON(5) #40BC9
	409C6 REL(5) L81448
	409CB CON(5) #40A95
	409D0 REL(5) L8141C
	409D5 CON(5) #61960
D1 Table Stream	D1:B5CA1 CON(5) #00000
	B5CA6 REL(5) LB5CA6
	B5CAB CON(5) #00000
	B5CB0 REL(5) LB5CB0
	B5CB5 CON(5) #00000

- [ARG] [√] will toggle the table format.
- [√] will toggle displaying only D0,D1 or both.
- [∧] [0-8] will show stack level N instead of D0.
- [CST] [3] toggles ascii/hexadecimal constants.
- [D] switches to screen 4.1

6.4.7 Screen 5 – Memory Dump ([E])

Locations 5990-589F	05990:3AE680D0F40D4F01 .n..O...
Locations 59A0-58AF	059A0:56113680913420CC e.c..C..
Locations 59B0-58BF	059B0:4E0156716FCC56FD ..e...e.
Locations 59C0-58CF	059C0:015B38D5E0101D95 ...]...Y
Locations 59D0-58DF	059D0:08E4CD08E46C0101 .N..N...
Locations 59E0-58EF	059E0:D230574911191443 -.u...A4
Locations 59F0-58FF	059F0:4E4A201101311456 ......Ae
Locations 5A00-5A0F	05A00:12280A50143174E7 !...A.G˜
Locations 5A10-5A1F	05A10:8E58D01311741431 ...1.GA.
Locations 5A20-5A2F	05A20:743450000EA248BE GC...*..

- Current location is indicated by an inverse character.
- Has medium font alternative which shows hexadecimal data only.

6.4.8 Screen 6 – ML Instruction Stream ([F])

Instruction 1	059D1 GOSUBL SAVPTR
Instruction 2	059D7 GOSUBL POP#
Instruction 3	059DD R1=A
Instruction 4	059E0 C=0 A
Instruction 5	059E2 LC(1) 5
Instruction 6	059E5 GOSUB MAKE$N
Instruction 7	059E9 C=R1
Instruction 8	059EC DAT0=C A
Instruction 9	059EF LC(5) =DOHSTR
Instruction 10	059F6 A=R0

- Has medium font alternative which shows 8 lines of disassembly.

6.4.9 Screen 7 – General CPU State & Breakpoints (MTH)

Mnemonic *BRK1*	`GOSUBL  SAVPTR       |1:12345 -70`
Opcode *BRK2*	` 8E4CD0              |2:00000  00`
Cycles/Total Cycles *BRK3*	`14                0  |3:00000  00`
PC, P, CRY, HEX/DEC, ST *BRK4*	`@059D1 P:0 CH ST:B218|4:00000  00`
A[A] C[A] MP SR *BRK5*	`A:059CC C:B5C9C MP SR|5:00000  00`
B[A] D[A] SB XM *BRK6*	`B:A9634 D:027AF SB XM|6:00000  00`
D0 and 6 bytes @D0 .... *BRK7*	`D0:409C1/9CB0428A0459|7:00000  00`
D1 and 6 bytes @D1 .... *BRK8*	`D1:B5CA1/000000000000|8:00000  00`
Top 3 levels of RSTK, R0[A]	`RST:00000:00000:00000   R0:A56F6`
R1[A] R2[A] R3[A] R4[A]	`R1:059D1 2:DE77D: 3:00000 4:BFA21`

- Breakpoints at the current PC are displayed in inverse.

6.4.10 Screen 8 – Watchpoints (VAR)

Watchpoint 1	`1:00000:2369B108DADF1008 2.......`
Watchpoint 2	`2:00000:2369B108DADF1008 2.......`
Watchpoint 3	`3:00000:2369B108DADF1008 2.......`
Watchpoint 4	`4.00000:2369B108DADF1008 2.......`
Watchpoint 5	`5:80100:3707020403047020 sp @0@..`
Watchpoint 6	`6:80319:A1A1A1A1A1A1A1A1 ........`
Tempob Bottom Watch	`  817CC:000000E4A20DB000 ...N*...`
Tempob Top Watch	`  A960C:00000895836B5834 ...Y8..C`
RPL Return Stack Watch	`  A963E:5A5E3A534F4C4147 ...5...t`
RPL Data Stack Watch	`  B5C7E:1075A22549FEF303 .W*R..?0`

- Watchpoints 1-6 are changeable, but are initialized by DB to

  1-4: Zero.

  5: IRAMBUFF base address (SX:#70100 GX:#80100)

  6: uart_buffer address (SX:#701FC GX:#80319)

- Watchpoints 7-10 are not changeable, but are updated continuously by reading the contents of the assigned RAM variables TEMPOB, TEMPTOP, RSKTOP and DSKTOP.

- Has medium font alternative which shows limited information.

6.5 Debugging with DB

At the lowest level the debugger works by storing the current instruction into an execution buffer and storing a jump back into DB after it. When stepping the current registers are then restored from MLDLPAR, a jump to the execution buffer is made, the jump back then causes the possibly changed registers to be saved back into MLDLPAR. Due to the algorithm the debugger manages to emulate for example the hardware bug in the `A=A+CON` and similar instructions.

Obviously the method does not work for opcodes which for example change the PC, thus such instructions are merely emulated separately by special code for each instruction. The end result will of course be exactly what the user expects, the CPU status is correctly changed.

For actual debugging DB offers various methods to advance the machine language execution

- The “StepIn”-key executes a single opcode.
- The “Step”-key is similar to the “StepIn”-key except that subroutine calls are executed as if they were a single opcode.
- The “RunIn”-key is similar to “StepIn”-key, but the debugger keeps running with display updates until aborted with the ON-key or until a specified number (ARG) instructions have been executed.
- The “Run”-key is a similar variant for the “Step”-key.
- The “Eval”-key returns full control to the program by letting it run freely. This differs from the above debugging keys so that only breakpoints set to RAM are recognized. The stepping keys compare the current address against the set breakpoints after each step and thus can catch breakpoints even in ROM. However since here the full control is returned to the program itself only breakpoints directly in the program can be found, this naturally means the program must be in RAM so that the breakpoints can be stored. If no breakpoints are encountered in the evaluation the control returns to whatever program called the debugger, be it for example the internal system loop or the SDB debugger.

To exit the debugger one can use the “Eval”-key to let the program finish, or one might use one of the direct exit keys

- The “ExitNow”-key assumes the current CPU status is fine and executes the `LOOP` sequence to exit to RPL. Obviously this can be dangerous if the RPL status registers are not in place, or if the program hasn't completed any possible environment changes it has been doing.
- The “Exit”-key restores the RPL status registers to the values which they had at the start of DB, and then executes the `LOOP` sequence to exit to RPL as if nothing has happened. If the program hasn't changed the RPL environment in any way, for example by allocating an object, then this is a fast and clean way to exit the debugger. However if the environment has changed then this, like the previous method, can cause crash.
- If the user knows the debugged program has made a fatal error then he should not let the control pass back to RPL again. To avoid memory loss one should then press the “Reset”-key which causes a warmstart. To prevent accidents though this key must be pressed twice, any other key would abort the reset. Note that ON-C can also be used unless it is specifically disabled by the program.

6.6 Movement in DB

Backward scrolling in machine language streams is a very complex task, and at best very prone to errors. Thus movement with DB is limited to quite basic tasks

- The cursor keys, with an optional ARG multiplier, modify the PC by a given amount. For example the down-arrow does not scroll down to the next instruction, it merely adds 16 to the current PC.
- Forward skipping is done with the “Skip”-key.
- “SetMark” key sets a mark at PC (or ARG), the PC and the mark can then be swapped by using the “mark↔pc”-key.

6.7 DB Regular Mode Keyboard

Screen 1	Screen 2	Screen 3	Screen 4	Screen 5	Screen 6
Screen 7	SetBrkHere	SetOpt:	Screen 8	pc -= 16	Skip
EditRegs		Eval	pc -= 1	pc += 16	pc += 1
SaveRegs	SwapRegs			SetStk:	Reset
		mark↔pc	ViewPict	Exit	ExitNow

Help				
				Run
				RunIn
				Step
	InputArg	SetMark	Redraw	StepIn

6.8 Register Save Buffer

When started DB saves all the registers into a special register save buffer. New register contents can be saved at any time, or the contents of the save area can be swapped with the current CPU registers at any time, thus providing a primitive "undo"-feature. One should note though that only the CPU status can be saved, not the RAM variables or anything from calculator memory for that matter. Thus one cannot undo any changes in the RPL environment, using the undo-key for example after a data allocation could cause a crash if debugging is continued as if nothing has happened.

The feature is useful for studying ROM subroutine register usage. One can save the registers ("SaveRegs"-key) before single-stepping a subroutine, and then press the "SwapRegs"-key repeatedly to see what registers have been changed. Naturally one cannot count on this analysis too much, disassembling the subroutine is always a more accurate method for obtaining this information.

6.9 Breakpoints

With DB one can set up to eight breakpoint addresses. The breakpoints are stored in a separate table in MLDLpar, and whenever the debugger is about to single step something it will first check if a breakpoint is set to the current PC. If so, a breakpoint message is displayed and debugging is aborted.

The table-comparison method does not work though when a breakpoint has been set into a subroutine which is stepped as a single entity with the "Step" or "Run" instructions. Thus DB stores a special breakpoint jump into DB at the breakpoint addresses. Naturally this works only when the debugged program is in RAM.

Breakpoints are set as follows

- “SetBrkHere” sets a breakpoint at the current PC.
- “SetBrkAddr” sets a breakpoint at the address given as ARG.
- “SetBrkCntr” sets a breakpoint counter to zero (or ARG). The breakpoint number is given by pressing $\boxed{\text{1-8}}$.

6.10 DB Arguments

Several of the DB keys accept an optional argument which can serve as a repeat count, or will alter the behaviour of the key completely. Argument editing is started by pressing the $\boxed{0}$-key, after which the following keyboard is in effect.

A	B	C	D	E	F
	SetBrkAddr	ShowBinary		pc -= 16n	SkipN
EditAt	SetBrkCntr		pc -= n	pc += 16n	pc += n
SetCycles	SetTotal		SetFormat		
SetPC		Negate	AddPC	Abort	BackSpace

	7	8	9	Run N
	4	5	6	RunIn N
	1	2	3	Step N
Abort	0	SetMarkAtN		StepIn N

6.11 Cycle Counters

The debugger maintains two cycle counters which can be set to any value at any time. The counters are named after the purpose they were designed for

Cycles This counter is intended for measuring cycle counts for individual code segments or subroutines. “SetCycles”-key can be used to set a value to it, in particular using a zero-ARG will initialize it to zero.

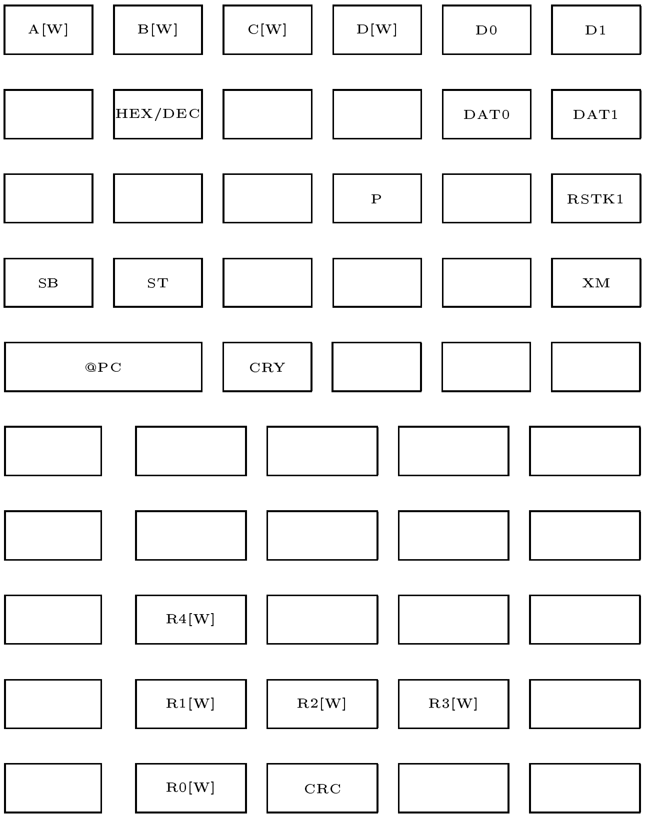
Total This counter is intended for measuring cycle counts for larger entities, such as a collection of subroutines. “SetTotal”-key can be used to set a value to it.

The cycles for each opcode are not those given by `SASM.DOC` but were measured with a special program which is able to determine the effects of even and odd addresses. The counts have been scaled by four to avoid units of 0.25 cycles in the display.

If the cycle count for some opcode couldn't be determined experimentally DB will use zero cycles.

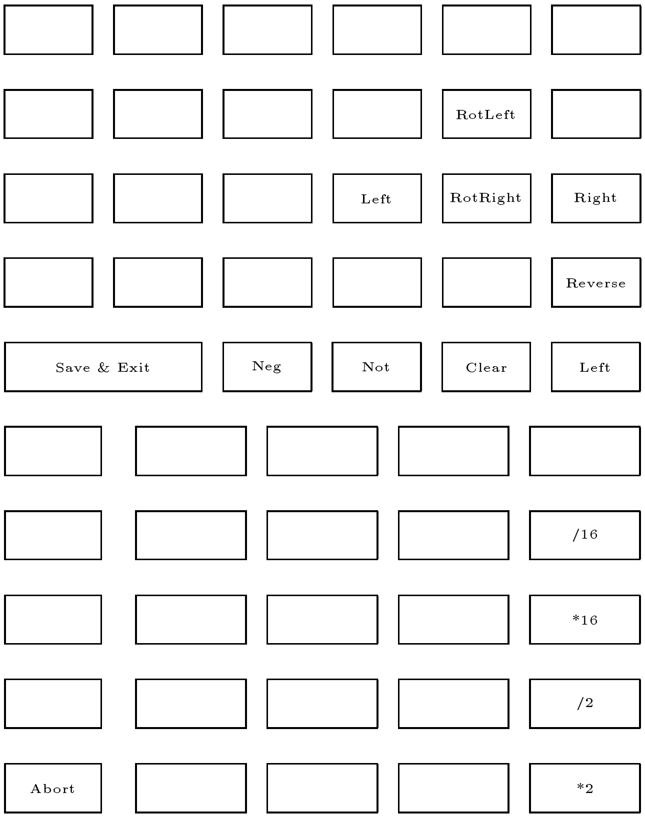
6.12 Register Editing

Register editing in DB is started by pressing the "EditRegs"-key ([']), after which the debugger expects a further keypress to choose the register for editing. The possible choices are summarized below, any other key aborts the register editing.



Some of the choices contain merely a flag, in which case the flag is merely toggled ([HEX/DEC], [CRY], [SB] and [XM]). Another special choice is [CRC] which does not imply the hardware CRC, but a library CRC. Based on the current PC the debugger will try to find the library the user is debugging, if one is found then one can edit the CRC. The debugger automatically prompts the user with a calculated value, this should be especially useful when poking simple fixes in libraries.

Once a register has been chosen the following keys can be used to edit the register contents.



7 SRPL Stack Display

Command:	SSTK
Description:	System RPL stack display
Stack:	($\rightarrow$)
Flags:	3 – When set uses internal decompiler instead.

The SSTK command starts a new loop to replace the built-in stack display with one which uses the *JAZZ* disassembler to display the lines. Most of the functionality remains exactly the same as in the internal loop, however multi-line displays are no longer supported. To exit SSTK simply execute it again.

8 Entries Catalog

Command:	EC
Description:	Entries table browser.
Stack:	(→)

The EC command enables browsing the entry tables RPL.TAB and DIS.TAB which are usually in unreadable form. As opposed to the assembler and disassembler EC does require both tables to be present, if they one or both are missing an error is generated.

For each entry the browser shows the entry address, the entry name and the entry type based on the table 12. The entries themselves can be sorted alphabetically or by address, on entry address sort is chosen by default.

For all commands the entry being selected is always the one topmost in the display.

Key	Description
△	Move up one entry.
▽	Move down one entry.
LS △	Move up one page.
LS ▽	Move down one page.
RS △	Move up to first entry.
RS ▽	Move down to last entry.
▷	If the entry is a grob then VV is called to view the grob, else the entry is disassembled and ED is called to view the disassembly.
A	Toggle alphabetical and address sorts.
B	Toggle substring and exact match modes.
F	Input string and find match in current match mode.
α	Input string and find match in current match mode.
NXT	Find next match in current match mode.
LS NXT	Find previous match in current match mode.
EEX	Toggle normal and grep modes. In grep mode only matches to the last search string are displayed.
ENTER	Push entry address to the stack tagged with the name.
LS ENTER	Push entry address to the stack.
RS ENTER	Push entry name to the stack.
0-9	Find entry starting with input address. Use 0-9 to specify more digits in the address.
+/-	Toggle beep on/off.
ON	Exit the entries browser.

Table 11: Entries catalog keys

Type	Prolog	Display
Binary Integer	DOBINT	#
Real Number	DOREAL	%
Extended Real	DOEREL	%%
Complex Number	DOCMP	C%
Extended Complex	DOECMP	C%%
Character	DOCHAR	CHR
Array	DOARRY	[]
Linked Array	DOLNKARRY	[L]
Character String	DOCSTR	$
Hex String	DOHSTR	HXS
List	DOLIST	{}
Directory	DORRP	RRP
Symbolic	DOSYMB	SYMB
Unit	DOEXT	EXT
Tagged	DOTAG	TAG
Graphics	DOGROB	GROB
Library	DOLIB	LIB
Backup	DOBAK	BAK
Libdary Data	DOEXT0	EXT0
External 1	DOEXT1	EXT1
External 2	DOEXT2	EXT2
External 3	DOEXT3	EXT3
External 4	DOEXT4	EXT4
Program	DOCOL	::
Code	DOCODE	CODE
Identifier	DOIDNT	ID
Temporary Identifier	DOLAM	LAM
ROM Pointer	DOROMP	ROMP
Primitive Code Object	(*)+5	PCO
Machine Language		ML
Other		ML

Table 12: Displayed entry types in EC

9 Editor

Command: ED
Description: String editor.
Stack:
($ → $')
($ %position → $')
(ob → ob')

Command: TED
Description: User RPL editor.
Stack:
($ → $')
(ob → ob')

ED is an editor intended for editing system RPL and machine language source code and thus has many keys for this special purpose. The editor can nevertheless easily be used as a general purpose editor.

ED makes no duplicate of the string being edited if it is in temporary object area, thus enabling editing very large strings, almost the size of available RAM. Note that this implies any changes made to the string cannot be recovered, if one for example has multiple copies of the string on stack they all change at the same time. If one wishes to keep a copy of the original one must store it for example in the HOME directory.

If the input is not one of the standard string input combinations *JAZZ* will automatically assume stack one has to be decompiled for editing, and will then be assembled when the editor exits. Any error generated in the assembly phase will cause the editor to restart at the error position. TED is an alternate interface to the editor itself which assumes the object is user level instead of system level – meaning the built-in decompiler and compiler will be used instead of the *JAZZ* disassembler and assembler.

ED is very fast but since it supports the tabulator character it has to do special display adjustment calculations whenever the display is scrolled. Thus scrolling the display when very long lines can be quite slow.

The optional cursor position argument is compatible with the error position output of the ASS assembler. Thus if the assembler generates an error and its position starting ED will automatically put the cursor at the error location.

9.1 Editor Mode Keys

Key	Acronym	Description
RS NXT	Status	Display editor status page.
EEX	TogBeep	Toggle beep on/off.
LS EEX	TogCase	Toggle upper/lower case.
RS EEX	TogOver	Toggle insert/overwrite mode.
+/-	TogSpeed	Toggle normal/fast mode.

9.1.1 Sample Status Display

```
          Editor Status

Text size   :   2998   Insert
Clip size   :      0   Fast: OFF
Free memory : 104387   Case: OFF
Work memory : 107385   Beep: ON

Cursor pos  :   2998
Cursor X    :      1
Cursor line :    271
```

9.2 Cursor Movement Keys

Key	Acronym	Description
◁	Left	Move cursor left.
▷	Right	Move cursor right.
△	Up	Move cursor up.
▽	Down	Move cursor down.
LS ◁	WordLeft	Skip word to the left.
LS ▷	WordRight	Skip word to the right.
RS ◁	LineStart	Jump to start of line.
RS ▷	LineEnd	Jump to end of line.
LS △	PageUp	Jump up one page.
LS ▽	PageDown	Jump down one page.
RS △	TextStart	Jump to start of text.
RS ▽	TextEnd	Jump to end of text.

9.3 Editing Keys

Key	Acronym	Description
DEL	Del	Delete character under cursor.
⇐	BackSpc	Delete previous character.
LS DEL	CutLine	Cut current line.
RS DEL	CutRight	Cut rest of current line.
LS ⇐	CutWordLeft	Cut word to the left.
RS ⇐	CutWordRight	Cut word to the right.

9.4 Block Keys

Block is a segment marked by two block delimiters. When both delimiters are set and the delimiters are properly ordered so that the end delimiter comes after the start delimiter the editor will show the block in inverse video.

The *clip* is a saved segment of text which is not directly visible. *Cutting* refers to deletion where the deleted area of text is saved in the clip.

When a command refers to block/clip the block is always selected if one has been marked.

Key	Acronym	Description
A	BlockStart	Mark start of block at cursor.
B	BlockEnd	Mark end of block at cursor.
C	BlockCopy	Copy block/clip to cursor position.
D	BlockCut	Copy block to clip and then delete it.
RS A	BlockUp	Mark end of block at cursor, start of block at start of text.
RS B	BlockDown	Mark start of block at cursor, end of block at end of text.
RS C	BlockSwap	Swap clip and text.
LS D	BlockDel	Delete block.
RS D	BlockKeep	Delete all but block.
LS STO	BlockPush	Push block/clip to the stack.
RS STO	BlockPop	Pop string from stack to cursor position.
VAR	BlockRev	Reverse characters in block.
LS VAR	BlockToHex	Convert block to hexadecimal.
RS VAR	BlockToAsc	Convert block to ascii.

9.5 Searching and Replacing Keys

All find keys are incremental, e.g. the display a new match is sought as soon as the find string is changed. All find and replace keys act forward unless otherwise mentioned. The replace keys asking for verification will display the current match in inverse video and offer a menu of choices. The choices are

`YES`	Replace the match.
`ALL`	Replace this and all following matches.
`NONE`	Do not replace any matches.
`NO`	Do not replace current match.

The global replace keys replace matches in the entire text and return to the starting cursor position when done.

Key	Acronym	Description
F	Find	Find.
NXT	FindNext	Find next match.
LS NXT	FindPrev	Find previous match.
LS F	Repl?	Find/replace with verification.
RS F	ReplAll	Find/replace all.
LS E	GlobalRepl?	Global find/replace with verification.
RS E	GlobalRepl	Global find/replace.

9.6 Marking Keys

The editor offers nine marks which can be set at any text position. In addition "mark 0" is used to indicate the cursor position before the last operation.

Set a mark:	Press left-shift followed by a digit 1-9.
Jump to a mark:	Press right-shift followed by a digit 1-9.
Jump to last position:	Press right-shift followed by 0.

9.7 Editor Macros

The editor allows defining a single macro which can be up to 50 keys long. Macro definition is started by pressing the "MacroStart" key and is ended by pressing the "MacroDo" or "MacroEnd" keys. If an error occurs during the macro definition the macro save is aborted. If an error occurs during macro evaluation the macro execution is aborted.

Key	Acronym	Description
CST	MacroDo	Execute macro.
LS CST	MacroStart	Start macro definition.
RS CST	MacroEnd	End macro definition.

9.8 Editor Counter Keys

Key	Acronym	Description
LS ∧	CntrInit	Initialize counter value, range and type. If the input-line starts with the character "#" the counter type will be hexadecimal, else decimal. The number of digits determines the range of the counter so that when the maximum value is exceeded the counter goes to zero. The digits themselves form the initial value of the counter.
RS ∧	CntrOut	Inserts the current counter into text in the current format and increments the counter.

For example to produce line numbers for a text one could do

TextStart	*Jump to start of text*
CntrInit 001 ENTER	*Assuming there are less than 999 lines*
MacroStart	*Start defining a macro*
CntrOut	*Output the first value*
Space	*Press space to separate the value from text*
LineStart	*Back to start of line*
Down	*To next line*
MacroEnd	*End macro*
Arg? 999 ENTER	*Input repeat count*
MacroDo	*And repeat the macro*

Assuming there are less than 999 lines the macro execution would cause an error at the bottom of text (the cursor cannot be moved down) and the macro execution would automatically stop. Alternatively the key-repeater and macro execution can both be abortered with the ON-key.

9.9 Character Catalog

Not all characters have an assigned key in the keyboard, thus the editor provides a character catalog which can be invoked both from regular mode and during input line editing.

The catalog will show the full character set, 32 characters in each row. Below the character set display is the character information line which shows

- the possible key to which the character is assigned
- the character itself
- the character code in hexadecimal, decimal and binary

Key	Acronym	Description
◁	CatLeft	Move cursor left.
▷	CatRight	Move cursor right.
△	CatUp	Move cursor up.
▽	CatDown	Move cursor down.
RS ◁	CatFarLeft	Move cursor to start of row.
RS ▷	CatFarRight	Move cursor to end of row.
RS △	CatFarUp	Move cursor to far toprow.
RS ▽	CatFarDown	Move cursor to bottom row.
ENTER	CatEnter	Echo character to text/input line.
ON	CatExit	Exit character catalog.

9.10 Editor Inputline

The following keys all input information from the user before doing their specific operations

Key	Acronym	Description
MTH	Arg?	Repeat next key N times.
LS MTH	Row?	Goto-row
RS MTH	Pos?	Goto-cursor-position
LS PRG	Char?	Echo character code.

During the input line the following keys are active

Key	Acronym	Description
◁	InputLeft	Move cursor to left.
▷	InputRight	Move cursor to right.
RS ◁	InputFarLeft	Move cursor to start of inputline.
RS ▷	InputFarRight	Move cursor to end of inputline.
DEL	InputDel	Delete character under cursor.
⇐	InputBackSpace	Delete previous character.
RS PRG	InputCatalog	Start character catalog.
▽	InputWord	Initialize inputline to word under cursor.
NXT	InputNext	During search finds next match.
LS NXT	InputPrev	During search finds previous match.
ENTER	InputExit	Accept inputline.
ON	InputAbort	Abort inputline.

9.11 Editor Programming Keys

The editor defines several keys which are were designed to make the editor especially useful for programming. These include simple keys for some often uses phrases, such as assembly language calls and jumps and RPL delimiters, keys for visiting the entries catalog, filling up entry names from the entry tables, temporarily visiting the stack environment, assembling the text, disassembling entries in the text and studying them in a new editor etc.

Key	Acronym	Description
SIN	"GOTO"	Insert GOTO into text.
LS SIN	"GOLONG"	Insert GOLONG into text.
RS SIN	"GOVLNG"	Insert GOVLNG into text.
COS	"GOSUB"	Insert GOSUB into text.
LS COS	"GOSUBL"	Insert GOSUBL into text.
RS COS	"GOSBVL"	Insert GOSBVL into text.
TAN	"GOYES"	Insert GOYES into text.
LS TAN	"GONC"	Insert GONC into text.
RS TAN	"GOC"	Insert GOC into text.
RS -	"ASS-RPL"	Insert ASSEMBLE-RPL into text.
α LS -	":: ;"	Insert :: ; into text.
α RS -	"$ """	Insert $ "" into text.
α LS +	"{}"	Insert {} into text.
α RS +	"CODE-END"	Insert CODE-ENDCODE into text.

Key	Acronym	Description
LS +/−	ASS	Assemble text. If assembly succeeds ED exits, else the error message is shown until a key press and the cursor is moved to the error location.
RS +/−	DOB	Disassemble object under cursor and view the disassembly in a new editor. To come back simply exit the new editor.
PRG	ECCAT	Start entries catalog in alphabetical mode. Any data pushed from the catalog is is inserted into the cursor position.
α RS DEL	ECCAT	This is an alternative key which can be used in α-mode.
α RS $\Leftarrow$	ECFILL	This key will attempt to fill the word under cursor with the best match from the entries tables. If there is only one possible match ED will just fill-up the word. If there are more than one possible expansions then the word is expanded to longest possible length and a single beep is given. If there are no possible matches at all then two beeps are given.
STO	STK	Start internal system loop. During this loop the HP48 is fully functional, including the possibility of starting a new editor. To come back to the editor press the CONT-key.
1/x	DFIND	Find matching delimiter for the delimiter under cursor. This works for the usual SRPL composite delimiters, the user RPL $\ll$ and $\gg$ delimiters as well as CODE-ENDCODE, DIR-ENDDIR and ASSEMBLE-RPL pairs.

The special tokens understood by the DOB-key are

Tokens	Description
#<hex>	The disassembly of the address is generated.
L<hex>	The disassembly of the address is generated.
PTR <hex>	The disassembly of the address is generated.
ROMPTR <hex> <hex>	The disassembly of the ROMPTR is generated.
ID <name>	The contents are disassembled, edited, assembled and stored back.
INCLOB <name>	The contents are disassembled, edited, assembled and stored back.
INCLUDE <name>	The text contents are edited and stored back.
GROB <hex> <hexbody>	The grob is viewed on top of the text.

BlockStart	A	BlockEnd	B	BlockCopy	C	BlockCut	D		E	Find	F
	a		b		c	BlockDel	d	GlobalRepl?	e	Repl?	f
BlockUp	α	BlockDown	β	BlockSwap	Δ	BlockKeep	δ	GlobalRepl	ϵ	ReplAll	θ
Arg?	G	ECCAT	H	MacroDo	I	BlockRev	J	Up	K	Next	L
Row?	g	Chr?	h	MacroStart	i	BlockToHex	j	PageUp	k	Prev	l
Pos?	γ	CharCat	η	MacroEnd	∞	BlockToAsc	|	TextStart	$\uparrow$	Status	λ
”	M	STK	N		O	Left	P	Down	Q	Right	R
	m	BlockPush	n		o	WordLeft	p	PageDown	q	WordRight	r
	’	BlockPop	μ		Ω	LineStart	$\leftarrow$	TextEnd	$\downarrow$	LineEnd	ρ
“GOTO”	S	“GOSUB”	T	“GOYES”	U		V	$\wedge$	W	DFIND	X
“GOLONG”	s	“GOSUBL”	t	“GONC”	u		v	CntrInit	w		x
“GOVLNG”	σ	“GOSBVL”	τ	“GOC”	%		$\sim$	CntrOut	ω		$\overline{x}$
Exit			IndEnter	TogSpeed	Y	TogBeep	Z	Del	Del	BackSpc	BackSpc
			&	ASS	y	TogCase	z	CutLine	!	CutWordLeft	?
			@	DOB	$\pm$	TogOver	Π	CutRight	ECCAT	CurWordRight	ECFILL

AlphaOn	AlphaOff	7	7	8	8	9	9	/	/
LeftShiftOn	TogCase	SetMark7	$\overline{X}$	SetMark8	$\int$	SetMark9	$\sum$	()	()
LeftShiftOn		GoMark7	$\sim$	GoMark8	|	GoMark9	\	#	#
LeftShiftOn	LeftShiftOn	4	4	5	5	6	6	*	*
LeftShiftOff	LeftShiftOff	SetMark4	$	SetMark5	£	SetMark6	Ψ	[ ]	[ ]
LeftShiftOn	LeftShitfOn	GoMark4	Γ	GoMark5	$\exists$	GoMark6	$\circ$	“ASS-RPL”	_
RightShiftOn	RightShiftOn	1	1	2	2	3	3	-	-
RightShiftOn	RightShiftOn	SetMark1	==	SetMark2	<	SetMark3	$\leq$	“<<>>”	“:: ;”
RightShiftOff	RightShiftOff	GoMark1	$\neq$	GoMark2	>	GoMark3	$\geq$	””	“$ ”””
ON		0	0	.	.	Space	Space	+	+
		=	=	,	;	Π	π	{}	“{ }”
OFF		LastPos	$\rightarrow$	NewLine	NewLine	Tabulator	Tabulator	::	“CODE-END”

Table 13: ED Keyboard

10 String and Grob Viewers

Command:	VV
Description:	String and grob viewer.
Stack:	(\$ → \$)
	(grob → grob)
	(ob → ob)

The string viewer is quite obsolete with ED present.

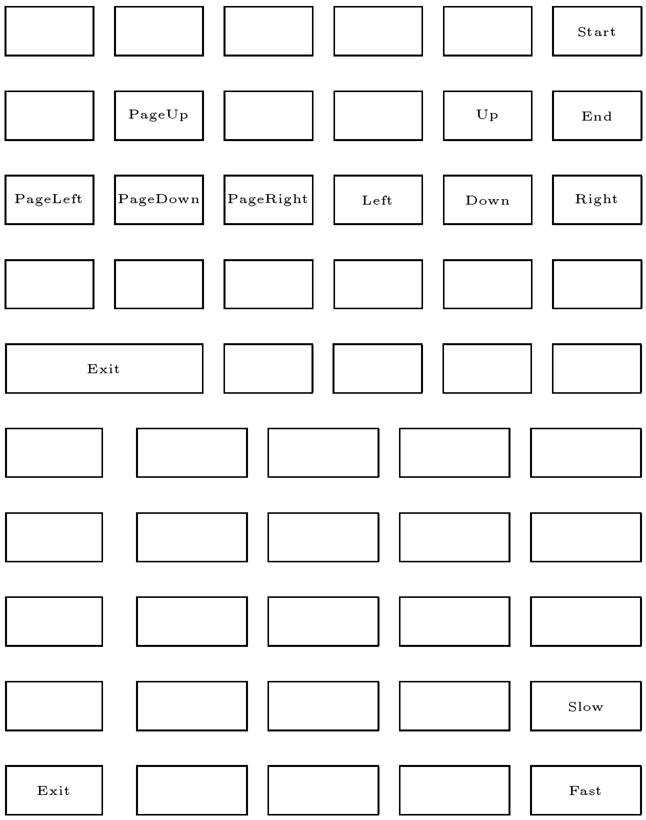
The grob viewer uses a "grob!" replacement with automatic cutting at the display borders. Masking grobs less than 4 bits wide is not promperly implement yet, thus grobs that narrow may appear strange or even vanish from the display.

The grob viewing is done on the internal text grob, thus the following program will create a weird effect.

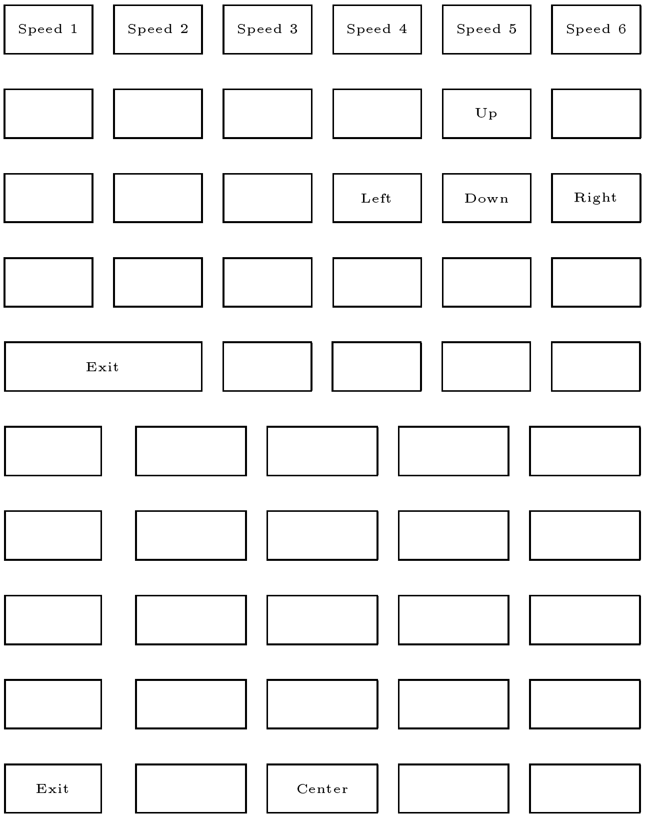
```
:: ABUFF xVV ;
```

As a special feature VV recognizes the AGROB fancy print decompiler which exists for example in EQSTK and JAVA libraries. This means that for example algebraics can be viewed in the form commonly written on paper.

10.1 String Viewer Keys



10.2 Grob Viewer Keys



11 Entries Table Utilities

Command: EA
Stack: ($entry → hxs_addr)
(hxs_addr → $entry)
(ob → hxs_addr)
Description: Convert between entry addresses and entry names.

Command: RTAB
Stack: (→ $)
Description: Recalls RPL.TAB to stack.

Command: DTAB
Stack: (→ $)
Description: Recalls DIS.TAB to stack.

Command: RTB→
Stack: (→ $)
Description: Converts RPL.TAB into readable form.

Command: →RTB
Stack: ($ → $')
Description: Converts an entry list into RPL.TAB form. The converter accepts lines of the form

[`=`]*name* [`EQU #`]*address*

where the brackets indicate optional parts. Note that the input should be sorted alphabetically, no checks for that are done. Also for programming simplicity the program errors if the last character of the input is not a newline character.

Command: →DTB
Stack: (→)
Description: Creates DIS.TAB based on RPL.TAB and stores the result in HOME directory.

11.1 Modifying the Entries Tables

The preferred method of creating modified entries tables is to edit the `hpentr.a` entries file distributed with the package and to use `gentab2` to produce a new tables library.

If using the entries table utilities is the only option, then changing the entries tables can be done as follows

1. Use RTB→ to get a readable entries list.
2. Use ED to edit the list. Note that new entries must be added to their correct alphabetical location.
3. Use →RTB to convert the the string into RPL.TAB format.
4. Store the result to variable `RPL.TAB`.
5. Delete the old entries tables (both RPL.TAB and DIS.TAB)
6. Use →DTB to get a new DIS.TAB.
7. Use a library builder to build a new entries library.

```
  $ROMID   993
 $CONFIG   ::  993 TOSRRP ;
$VISIBLE   { RPL.TAB DIS.TAB }
  $TITLE   ""
```

A Error Messages

A.1 General Error Messages

Invalid DBpar

DBpar variable is too short or corrupt. Purge it and restart DB

Lib Not Fixed

JAZZ does not work from a covered port.

RPL.TAB Missing

Command requires an existing RPL.TAB.

A.2 Assembler Error Messages

ACPTR is a G Object

Access pointers cannot be compiled on a G/GX.

Argument 0-15 Expected

Argument must be in the range 0-15.

Argument 1-16 Expected

Argument must be in the range 1-16.

Argument 1-256 Expected

`D1=D1+` etc. expect an argument in the range 1-256.

Argument Field Expected

Instruction requires an argument.

Body Len Not Hex

Body length field must be hexadecimal.

Body Len Too Big

Body length field is too big.

Body Too Long

Body is longer than indicated by the length field.

Body Too Short

Body is shorter than indicated by the length field.

Branch Too Long

Branch target is too far for the instruction. May have to reverse the test and use a jump instead.

Can't INCLOB

Object was not found from the current path.

Can't INCLUDE

Source string was not found from the current path.

Cannot Redefine Value

A symbol value assigned by "=" can only be changed by "=".

DEFINE Depth Overflow

Define stack depth overflow, possible recursion.

DEFINE String Missing

The defined text for symbol is missing.

Division By Zero

Expression contains division by zero.

Duplicate Config

Configuration object label has already been declared.

Duplicate Label

Label has already been used.

Duplicate Message Table

Message table label has already been declared.

Duplicate Name

Library command with same name already exists.

Embedding Not Allowed

`xROMID` must be used before any code is assembled.

Empty Label

A label is expected after "=" and ":" in column 1.

Empty Macro

Macro contains no text.

ENDCODE Expected

`ENDCODE` is expected before end of source for the matching `CODE`.

ENDCODE Not Expected

Use RPL as matching delimiter for `ASSEMBLE`.

ENDM Missing

`MACRO` was used without a `ENDM`.

Expecting Hex Size

`CODE` must be followed by blank or a hexadecimal size field.

Expr Buffer Overflow

Expression is too complex for the parser.

EXT1 is a S Object

`EXT1` objects cannot be compiled on a S/SX.

Field Selector Expected

A field selector is expected.

GOYES Expected

Previous instruction was a test, `GOYES` must be used next.

GOYES Without Test

Previous instruction was not a test.

Invalid #

Token is not a hexadecimal number.

INCLUDE Depth Overflow

Include stack depth overflow, possible recursion.

INCLUDE Ob Not String

Included variable must contain as string.

Invalid %

Invalid real number.

Invalid %%

Invalid extended real number.

Invalid APda

Hexadecimal access address is expected.

Invalid APaa

Hexadecimal access handler address is expected.

Invalid Asc Field

Argument cannot be decoded, probably due to an invalid escape sequence.

Invalid Binary Number

Invalid binary (base 2) number.

Invalid Body

Body contains non-hexadecimal characters.

Invalid C%

Invalid complex number.

Invalid C%%

Invalid extended complex number.

Invalid CHR

Invalid escape sequence used for the value of a character object.

Invalid Decimal Number

(not used)

Invalid Expression

Invalid expression. (Missing argument, ASCII not allowed etc.)

Invalid Hash Assignment

There is no library command for which to assign the secondary hash, or the label is not used by a library command.

Invalid Hex Number

Invalid hexadecimal number.

Invalid Field Selector

Field selector must be one of P,WP,XS,X,S,M,B,W,A.

Invalid ID/LAM

Name part was not decoded correctly, probably due to an invalid escape sequence.

Invalid LibID

Library number must be a hexadecimal number.

Invalid Mnemonic

Unknown mnemonic.

Invalid (N) Field

(N) field is non-decimal or is too big.

Invalid NIBB Length

Data segment length must be hexadecimal.

Invalid Operator

Expression is missing an operator.

Invalid PTR

A hexadecimal address is expected.

Invalid Reg Combination

The CPU has no instruction for the used register combination.

Invalid ROMID

Library number must be in the range #0-#7FFh.

Invalid RomWd

Command number must be a hexadecimal number.

Invalid Scratch Reg

Scratch registers are R0,R1,R2,R3,R4.

Invalid String

Body of string was not decoded correctly, probably due to an invalid escape sequence.

Invalid TITLE

Library title contains bad escape sequence.

Invalid Token

SRPL word was not recognized.

Invalid Use of Symbol

Symbol cannot be used in machine language.

Label Already External

Symbol is already defined by the entries tables.

Label Already Defined

Symbol has already been used for a `DEFINE`.

Label Already Romptr

Label has already been used for a library command.

Label Expected

Instruction expects a label as argument.

Label Reserved

Symbol is a reserved word.

Length Too Big

Data segment length field is too big.

Macro Already Exists

Symbol has already been used for a `MACRO`.

MACRO Missing

`ENDM` was used without a `MACRO`.

Missing Body

A hexadecimal body is expected.

Missing String

Body of string object is expected.

Missing TagOb

A tagged object must have an object to tag.

Missplaced ; or }

The composite depth counter couldn't match all delimiters, one (or more) is probably missing.

More Tokens Expected

Multi-token segment requires more tokens.

Need Hex Field

A hexadecimal number is expected next.

No Program

No code was assembled.

Not Implemented

The token is not implemented.

Not in MAKEROM Mode

`xROMID` must be used before any other library tokens.

Relative Value

Argument must have absolute value, it cannot be a label.

ROMPTR LibID > FFF

Library number must be in the range #0-#FFFh.

ROMPTR RomWd > FFF

Command number must be in the range #0-#FFFh.

Too Big #

Integers should be in the range #0-#FFFFFh.

Too Big Exponent

Expression has an exponent larger than 48.

Too Big PTR

Address should be in the range #0-#FFFFFh.

Too Long Hex Field

`LCHEX` argument has more than 16 hexadecimal digits.

Too Long Asc Field

Argument has more than 8 characters.

Too Long ID/LAM

Identifer objects cannot be over 255 characters long.

Too Long Label

All labels must be 1-15 characters long.

Too Long Name

Library command names must be 1-16 characters long.

Too Long Offset

Offset is too long for the `REL(n)` instruction.

Too Long Title

Library title must be shorter than 256 characters.

Too Many ('s

Expression doesn't have enough closing parenthesis.

Too Many)'s

Expression doesn't have enough opening parenthesis.

Too Many Labels

Lambda binding can only declare 22 labels.

Undefined Label

Label has not been defined.

Undefined Result

An invalid object was assembled.

Unresolved Expression

Expression was not resolved. (Some opcodes require it in the first pass.)

Used Before Declaration

Library command was used before it was declared, and so it was compiled wrong in the first pass (unrecoverable error).

Value Changed

Cannot change value assigned by `EQU`.

Zero Length

Data segment length field must be non-zero.

B Library source code example

The following source code example is for a simplified version of my personal 'crash'-library which sets up my favourite user-keys and flags after a warm-start. Note especially that several of the key assignments come from the HACK library, so it has to be installed in order to assemble the source code.

```
* Disable flag define
DEFINE fDISABLE 64
* Unsupported entries
ASSEMBLE
=CONTRAST       EQU #00101      * Contrast register
='EvalNoCK:     EQU #18F6A      * Strip CK<n> from next
=StoUserKeys    EQU #41E32      * Store user keys
=FIRSTPROC      EQU #7067E      * Hook to first program executed
=G_FIRSTPROC    EQU #807FC      * after warmstart.
RPL
* Library starts
xROMID 4                        ( Typical crash-library romid )
** No title - library won't show up in library menu
xCONFIG CrashCfg
** No message table

* The following declarations are not necessary, they are here merely
* to emphasise the fact that the data objects have names. This is for
* convenience only as it enables easy editing of the output when splitting
* the library with for example L->DIR from the HACK library.

EXTERNAL xOwnKeys
EXTERNAL xOwnFlags

* The configuration object does all the work
LABEL CrashCfg
::

* Abort if disable flag is set

  fDISABLE TestUserFlag ?SEMI

* Set favourite contrast

  CODE
        GOSBVL  =SAVPTR                 Set contrast to value 14
        D0=(5)  =CONTRAST
        LC(2)   14
        DAT0=C  B
        GOVLNG  =GETPTRLOOP
  ENDCODE

* FIRSTPROC is not updateable, so we cannot use it safely
* when the library is covered

  CODE
        ?C<B    A                       PC below RPL return stack?
        GOYES   +                       Yes - then library is covered
+       GOVLNG  =PushT/FLoop
  ENDCODE
  ?SEMI
```

```
* Set FIRSTPROC - the program run after configuration loop finishes

  '
  ::
    ' StartupProc        ( Default FIRSTPROC )
    >R 'R >R             ( 'emulate' first instructions.. )
    'REVAL               ( ..which initialize internal variables )

     xOwnKeys StoUserKeys               ( Set userkeys )
     %2 InitMenu%                       ( Set VAR menu )
     xOwnFlags 'EvalNoCK: xSTOF         ( Set flags )
  ;                                     ( and drop into StartupProc )

  CODE
        GOSBVL  =PopASavptr             A[A] = ->prog
        D0=(5)  =IRAM@                  RAM base address variable
        C=DAT0  S                       C[S] = 7/8 for S/G
        C=C+C   S                       CRY set if G
        D0=(5)  =FIRSTPROC              SX value
        GONC    +                       Skip if S
        D0=(5)  =G_FIRSTPROC            GX value
+       DAT0=A  A                       Store ->prog
        GOVLNG  =GETPTRLOOP
  ENDCODE
;

** Own flags **

ASSEMBLE
        CON(1)  8                       *Regular command*
RPL
xNAME OwnFlags
{ HXS 10 0FF05819C0300083               ( System flags )
  HXS 10 0000000000000010               ( User flags )
}

** Own userkeys **

DEFINE NO NULL{}                        ( Shorthand )

ASSEMBLE
        CON(1)  8                       *Regular command*
RPL
xNAME OwnKeys
{ NO NO NO NO NO NO NO
  { NO NO xEC NO NO NO }                (PRG)
  NO NO NO NO
  { NO xUP NO NO NO NO }                (')
  { xXSTO xSTOX xXRCL NO NO NO }        (STO)
  { NO xCOERCE NO NO NO NO }            (EVAL)
  NO NO NO NO NO NO NO NO NO
  { NO xVV xED NO NO NO }               (ENTER)
  { NO xASS xDIS NO NO NO }             (+/-)
  { NO xPG xDOB NO NO NO }              (EEX)
  NO NO NO NO NO NO NO NO NO NO NO
  NO NO NO NO NO NO NO NO NO NO NO
}
```