JAZZ 50G v1.25
System RPL and Machine Language Development Library

March 19, 2012

Contents

List of Tables

1 Copyright & Acknowledgements

The *JAZZ* library is distributed in the public domain with the hope that it will be useful. It is provided 'as is' and is subject to change without notice. No warranty of any kind is made with regard to the software or documentation. The authors shall not be liable for any error for incidental or consequential damages in connection with the software and the documentation.

1.1 Copyright

Permission to copy the whole, unmodified *JAZZ* package is granted provided that the copies are not made or distributed for resale (excepting nominal copying fees).

1.2 Extra credits & Acknowledgements

Mika Heiskanen	Original author of *JAZZ* for the HP48 series.
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 HP48 version of 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	Alphabetical sort and alphabetical 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 disassembler; the 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. (No longer used.)
Davor Jadricevic	Original small font. (No longer used.)
Al Arduengo	Improved small font. (No longer used.)
Dominique Rodriguez	First version of the LaTeX documentation.
Cary McCallister	For answering.
Andre Schoorl	For UFL (no longer used), JAZZHOOK, bug fixes, and former maintainer of *JAZZ* source on the HP48 series.

1.3 HP48 Beta Testing & 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	

1.4 Version Numbers

There are three version numbers. The internal version number is incremented from v6.8 (which as designed for the HP48 series). The current internal version number is 7.25. The version number for the HP49G+ and HP50G calculators is simply the internal version number minus six (v1.25). When the *JAZZ* library is placed onto the stack, the version number listed there reflects the compile date of that particular binary. So for example, the most recently compiled binary would show `v2012.03.18`.

2 Introduction

JAZZ was originally a library for the HP48 calculator and provided commands that enabled assembling, disassembling and debugging both system RPL and machine language on the calculator. It has since been and ported to the HP49G+ and HP50G and updated with new features. Documentation will henceforth be for the HP49G+ and HP50G. 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.

2.1 Overview

As opposed to the computer-based development tools, *JAZZ* is very tightly integrated. 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. Moreover, from within the string editor ED one can use word completion, assemble source code, study the disassembly of named and unnamed entries, call the entries catalog, and visit the stack. The following table 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
SKILL	SDB KILL command
EC	Entries catalog
ED	String editor
TED	User RPL editor
VV	String and grob viewer
EA	Entry name conversion utility
SSTK	SRPL stack display
MFED	Minifont editor

2.2 Installing and Deleting *JAZZ*

JAZZ is a regular auto-attaching library (library number 992). *JAZZ* takes approximately 71Kb of memory without entry tables. *JAZZ* currently does not work from a covered port, thus Ports 1 (ERAM), 2 (FLASH), and 3 (SD CARD) are not allowed as storage ports.

2.2.1 Installing *JAZZ*

1. Download the file `jazz50g.hp` into your calculator in binary mode.
2. Put the content of the created variable `jazz50g.hp` on the stack.
3. Delete the variable `jazz50g.hp`.
4. Store it in Port 0 (for example with 0 STO.)
5. Power-cycle the calculator (ON-C).

2.2.2 Deleting *JAZZ*

1. Exit any running *JAZZ* programs.
2. Detach the library with :0:992 DETACH.
3. Purge the library with :0:992 PURGE. If the error message 'Object in use' appears, then do ON-C and start again from 2.

2.3 Installing the Entry Tables

To provide symbolic names for ROM subroutines *JAZZ* uses preformatted entry tables. Older versions of *JAZZ* relied on the entry library (`hptab.hp`) which contained RPL.TAB and DIS.TAB and only supported symbols with a maximum of 16 characters. However, there are entries for the HP49G+ and HP50G which are longer than 16 characters. *JAZZ* 50G v1.25 and onward will instead use `extable.hp`.

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 tedious. While the tables are not obligatory, using them is highly recommended. The entries in `extable.hp` (provided in the *JAZZ* package) were built from Hewlette Packard's latest supported entries table. *Please note that the entries table provided within the JAZZ package is* **NOT** *the same as that from the* `extable` *package!*[1]

[1] The one provided within the *JAZZ* package is backward compatible, however.

2.3.1 Installing the Entry Tables

To install the tables download the file `extable.hp` into your calculator and proceed as in *JAZZ* installation process. The tables may be stored in Port 0, Port 1, or Port 2. The current version now dynamically configures memory so that storing the tables in any port will no longer result in any slowdowns, nor will the tables be copied into RAM.[2]

2.3.2 Creating Custom Entry Tables

The entries library was built from the `Suprom49.a` using a modified version of Deltef Mueller's `gentab` program. The C-source for the program is also included in the package. The syntax is

```
gentab2 < Suprom49.a > table.a
```

and then `table.a` can be assembled as instructed with SASM. The resulting table can then be compiled into `extable.hp` using the `extable` package.

2.4 Related Material Available via HTTP.

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:

`http://www.hpcalc.org/hp49/programming/entries/`

> This page contains links to documentation of supported entries, including RAM entries.

`http://www.hpcalc.org/hp49/docs/programming/`

> This page contains links to documentation on SRPL programming, assembly programming, as well as documentation for MASD, the built-in compiler.

`http://www.hpcalc.org/hp48/pc/programming/`

> This page contains links to computer programs which aid in programming the SATURN based calculators. In particular, DEBUG4X is an excellent programming environment for WINDOWS platforms.

[2] In previous versions, the tables could be stored in any port. However, storing them in covered ports required RAM to copy the tables from covered memory even if only for a single lookup.

`http://www.hpcalc.org/hp48/pc/programming/sadhp105.tgz`

SAD is a very powerful Saturn dissasembler and runs on Unix or Linux systems. The package contains extensive symbols files 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.

`comp.sys.hp48`

This is a newsgroup where calculator enthusiasts often share ideas, programming tips, and talk about HP calculators.

3 Assembler

Unlike the HP Tools, *JAZZ* provides only a single 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 The Assembler Command

COMMAND:	ASS
DESCRIPTION:	Assemble source string
STACK:	$ → ob $ → $ %erropos
KEYS:	ON key aborts
FLAGS:	1 – Report mode on when set 7 – Do not use entry tables when set

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

ASS will stop at the first error it encounters, unlike HP Tools and GNU Tools. To make the error message as informative as possible, ASS will attempt to emulate the behavior of the internal system error handler. 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`).[3] 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. Examples can be found in the stack subprogram called from within ED as well as SSTK.

All the assembler error messages are listed in Appendix C.

3.2 System RPL Assembly

The assembler always starts in SRPL mode. The tokens recognized by ASS are in Tables 2 and 4 (page 16) at the end of this section.

3.2.1 System RPL Comments

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

3.2.2 Lambda Variable Generation

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
```

The following diagram shows how a program using *JAZZ* syntax would be compiled by the assembler.

[3] The HP49G+ and HP50G evaluate the command line via a flash pointer, which is executed within a special environment that has its own error trap (`PTR 0B318`). This pointer restores the ROM view prior to execution of the command line. Therefore, running ASS via the command line will produce less informative error messages. On the other hand, calling ASS from a menu (either the library menu or a custom menu) will result in the more informative error message display.

JAZZ syntax	Assembler interpretation

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

3.2.3 Escape Sequences

To allow editing source files which contain special characters, the assembler supports the character escape sequences summarized in table 1. 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 1: Character escape sequences

3.2.4 System RPL Tokens

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 2: Miscellaneous SRPL Tokens

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.
`<int>`	A sequence integers.
`<fptrname>`	Flash pointer name (e.g. `^Qadd`)
`<rptrname>`	ROM pointer name (e.g. `~Choose`)

Table 3: RPL Token Abbreviations

Prolog	Address	Object	Tokens
DOINT	#02614h	Integer	ZINT <int>
DOLNGREAL	#0263Ah	Long Real[a]	L% <int>E<int>
DOLNGCMP	#02660h	Long Complex[b]	LC% <int>E<int> <int>E<int>
DOMATRIX	#02686h	Matrix	MATRIX ... ;
DOFLASHP	#026ACh	Flash Pointer	FPTR <hex> <hex>
			FPTR2 <fptrname>
DOAPLET	#026D5h	Aplet[c]	
DOMINIFONT	#026FEh	Minifont	MINIFONT <len> <hex>
DOBINT	#02911h	Binary Integer[d]	# <hex> or #<hex>
			<decint>
DOREAL	#02933h	Real Number	<dec>
			% <dec>
DOEREL	#02955h	Extended Real	%% <dec>
DOCMP	#02977h	Complex Number	C% <dec> <dec>
DOECMP	#0299Dh	Extended Complex	C%% <dec> <dec>
DOCHAR	#029BFh	Character	CHR <chr>
DOARRY	#029E8h	Array	ARRY <len> <hexbody>[e]
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[f]	
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>[g]
DOEXT2	#02BCCh	External 2	EXT2 <len> <hexbody>
DOEXT3	#02BEEh	External 3	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>
			ROMPTR2 ~<rptrname>

[a] Only disassembly supported in this release.
[b] Only disassembly supported in this release.
[c] No system support.
[d] Number searched in ROM first.
[e] Syntax specified in RPLCOMP.DOC is not supported.
[f] Directory assembly is supported only in MASD.
[g] DOACPTR is DOEXT1 on the HP48S/SX series.

Table 4: Object Generating Tokens

3.3 Machine Language Assembly

Since 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).

3.3.1 Machine Language Comments

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

3.3.2 Machine Language Tokens

The mnemonics recognized by the assembler are documented by `SASM.DOC` in the HP Tools package and in `newopcodes.txt`, so this manual explain them in any detail. Instead only the main changes are summarized in the tables that follow. Also, 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: Mnemonics that are not implemented.

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`	Example: `D0=D0+A` is expanded to `CD0EX C=C+A CD0EX`
`Dn=Dn-r`	Example: `D1=D1-C` is expanded to `AD1EX A=A-C AD1EX`
`Dn=Dn+P`	Example: `D0=D0+P` is expanded to `CD0EX C+P+1 CD0EX`
`Dn=Dn-P`	Example: `D0=D0-P` is expanded to `CD0EX C=-C C+P+1 C=-C CD0EX`

`r=Dn`	Example: `A=D0` is expanded to
	`AD0EX D0=A`
	Example: `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: GNU Tools mnemonics that are not implemented.

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`. Nn particular, there is no argument substitution. To define a macro, use:

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

After the definition all occurrences of `<label>` are replaced by the source lines in the macro definition. For single-line macro definition, 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
```

Since 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 advised when using symbolic names.

3.3.5 Label Generation

Sometimes assigning names for insignificant branches can be tiresome. Fortunately, the assembler recognizes local labels, which work 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.

3.3.6 Conditional Assembly

JAZZ does not use the unnecessarily complicated conditional assembly opcodes used both in GNU Tools and HP Tools. Instead *JAZZ* enables comparison operators in expressions; 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 independent of labels, so 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 might be added in future versions of *JAZZ* via the `#IF`, `#ELSE` and `#ENDIF` tokens.

3.3.7 Expressions

Expressions are evaluated using 64-bit signed integer math except for comparison operators, which 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.

3.3.8 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. In the current release of *JAZZ* the address is `#80900h` and coincides with the entry `=TopicVar31`. For more information see the DB documentation.

3.4 Library Assembly

The assembler starts library creation mode upon the declaration of the library number (ROMID). The assembler cannot produce libraries embedded in other objects, so the declaration must 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. A library's 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.

One may also declare a command to have a second (or several if used again) typeable name by using

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

This can be used to assign names to NULLNAME's too, and is a rather useful feature if one wishes to provide access to the low level subroutines. Note, however, that the secondary names are only typeable. Whether the command will have a name that is displayed by the internal decompiler is determined by the actual declaration of the command with one of `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.[4] 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.

See Appendix A For a sample source code which can be assembled with *JAZZ*.

3.5 Differences to HP Tools and GNU Tools

Aside from the obvious missing or additional mnemonics, there are several design differences between *JAZZ* and HP Tools. The most important ones are described below.

1. HP Tools MAKEROM is a separate library builder; 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".
3. *JAZZ* does not support the ARRY token as defined by HP Tools.
4. *JAZZ* does not support macro argument substitution.
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 'HPHP49-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, such as in "`WHILE DROP REPEAT`" where there is only the single object `DROP`. Thus HP Tools sometimes unnecessarily slows down the code as well as increases the code size.

[4] See Appendix E in the entries.srt document. (Section 2.4, reference ent_srt.zip.)

8. *JAZZ* does not support the special *256+ operator, while HP Tools does not support binary terms nor comparison operators in expressions.

9. *JAZZ* 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 checked whether to see if it is absolute or relative, and accordingly some mnemonics will 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 (and only on the older HP48 series). 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; no overflow
            A=A+CON S,2     Just see what happens..
            LOOP
    ENDCODE
    ```

11. As there is no specific linker in *JAZZ*, it expects 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

JAZZ has four methods of dissassembly, each differing slightly by how they dissassemble objects or blocks of memory.

4.1 The Disassembler Commands

COMMAND:	DIS
DESCRIPTION:	Disassemble object.
STACK:	ob → $
KEYS:	ON key aborts
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:	#address → $ hxs_end hxs_start → $ hxs_end "entry" → $ hxs_end
KEYS:	ON key aborts
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 end address guessing.
STACK:	hxs_start hxs_end → $
KEYS:	ON key aborts
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:	Disassemble n machine language instructions.
STACK:	#address %n → $
KEYS:	ON key aborts
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)  +
        BSS     <expr>          Data is all zeros
+       C=RSTK

        GOSUB   +
        REL(5)  +
        NIBASC  'ASCII'         Data in various formats and
        CSTRING 'ASCII'         spanning multiple lines
+       C=RSTK

        GOSUB   +
        REL(5)  +
        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 split into `CSTRING`'s or by newline characters so that the maximum length 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. There are yet other ways which the disassembler cannot reasonably be expected to guess; there are cases in which the disassembler is bound to fail.

4.3 DOB End Address Guessing

The end address guessing algorithm has changed since version 6.8a. 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 either an RPL object is found, or the "end" of machine language is reached (see below).

- 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`, `RPL2`, `ARM_LOOP`.[5]

Older versions of *JAZZ* worked under the assumption that primitive code objects ended at either the start of another primitive code object, or at the start of an RPL object (both of which we will simply call SRPL objects). This leads to erroneous end addresses when machine language ends at a new, unprologued SRPL entry.[6]

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 the HOME directory, 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.

[5] `ARM_LOOP` is not a supported token, but is the disassembly corresponding to various ROM overwrites of the opcodes `A=DAT0 A DO=DO+ 5 PC=(A)`.

[6] These entries, such as `CK&DISPATCH1`, exist even in the HP48 series.

4.5 Warnings

When using *JAZZ* to dissassemble objects (or memory), please be aware of the following discrepencies.

1. 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!

2. Composite type (RPL tokens: `{}`, `SYMBOL`, `MATRIX`) is tracked up to 64 levels. If the limit is exceeded a ";" may be output when a "}" is due.

3. 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 its data allocation. The benefits are clearly greater, though.

5 Machine Language Debugger

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.

5.1 DB Command

COMMAND:	DB
DESCRIPTION:	Debug machine language.
STACK:	id → romptr → $entry → #address → hxs_address →

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

5.2 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, or verified if ti already exists.

5.3 DB Entry Hook

Whenever DB is started it also initializes a special entry hook to enable starting DB directly from machine language. To call this hook one would then use `GOSBVL #80900` or `DEBUG`. Note that this address resides in reserved RAM and coincides with the entry `=TopicVar31` which, for now, is not used by the system. The assembler provides a *JAZZ* specific token named `DEBUG` which will automaticall be assembled to `GOSBVL #80900`. The disassembler will also output "`DEBUG`" whenever it encounters these special calls. The hook can also be initialized by the hidden command DBHOOK which will not only validate MLDLPAR but also set up the necessary hooks.

The text display grob used to display DB screens is usually 9 lines high, while DB uses the expanded 10 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 DBHOOK for the initialization. If the display size is invalid on entry, the graphics grob might be corrupted and memory could be lost. It is up to the calling program to ensure that the display is initialized (i.e. the menu area is turned off).

User flag 1 disables the entry hook so that it simply returns 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.

5.4 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`)

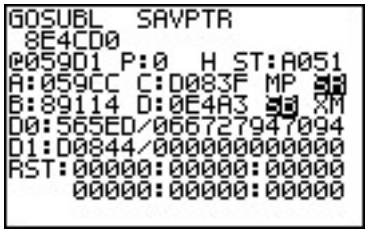
and then by pressing the key assigned for each screen.

Some people have found the small font screens too hard to read (or just too hard to get used to). Starting with *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.

5.4.1 Screen 1 – General CPU State

KEY: F1	Description
ROW1:	MNEMONIC
ROW2:	OPCODE
ROW3:	PC, C, CRY, HEX/DEC MODE, ST
ROW4:	A[A], C[A], MP, SR
ROW5:	B[A], D[A], SB, XM
ROW6:	D0 AND 6 BYTES OF @D0
ROW7:	D1 AND 6 BYTES @D1
ROW8:	TOP 3 LEVELS OF RSTK
ROW9:	NEXT 3 LEVELS OF RSTK
ROW10:	

Sample Screenshot



5.4.2 Screen 2 – General CPU State II

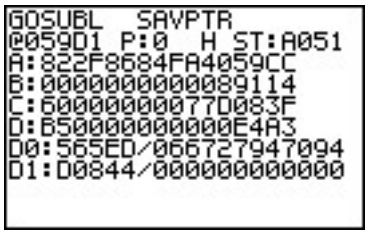
Screen 2 by default shows the CPU state using 13 rows of MINIFONT text.

KEY: F2	Description
ROW1:	PC, OPCODE
ROW2:	MNEMONIC
ROW3:	D0 AND 12 BYTES OF @D0
ROW4:	D1 AND 12 BYTES OF @D1
ROW5:	A[W], P, ST
ROW6:	B[W], ST[11-8], ST[7-4], ST[3-0]
ROW7:	C[W], CRY, HEX/DEC MODE, HST
ROW8:	D[W]
ROW9:	R0[W]
ROW10:	R1[W], RSTK 1, RSTK 5
ROW11:	R2[W], RSTK 2, RSTK 6
ROW12:	R3[W], RSTK 3, RSTK 7
ROW13:	R4[W], RSTK 4, RSTK 8

Sample Screenshot

KEY: F2	Description
ROW1:	MNEMONIC
ROW2:	PC, P, CRY, HEX/DEC MODE, ST
ROW3:	A[W]
ROW4:	B[W]
ROW5:	C[W]
ROW6:	D[W]
ROW7:	D0 AND 6 BYTES OF @D0
ROW8:	D1 AND 6 BYTES OF @D1
ROW9:	
ROW10:	

Sample Screenshot

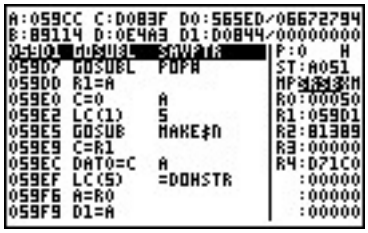


5.4.3 Screen 3 – CPU State & Instruction Stream

Screen 3 by default shows the CPU state using 13 rows of MINIFONT text.

KEY: F3	Description	
ROW1:	A[A], C[A], D0/D0[0-7]	
ROW2:	B[A], D[A], D1/D1[0-7]	
ROW3:	INSTRUCTION 1	P, CRY, MODE
ROW4:	INSTRUCTION 2	ST
ROW5:	INSTRUCTION 3	HST
ROW6:	INSTRUCTION 4	R0[A]
ROW7:	INSTRUCTION 5	R1[A]
ROW8:	INSTRUCTION 6	R2[A]
ROW9:	INSTRUCTION 7	R3[A]
ROW10:	INSTRUCTION 8	R4[A]
ROW11:	INSTRUCTION 9	RSTK 1
ROW12:	INSTRUCTION 10	RSTK 2
ROW13:	INSTRUCTION 11	RSTK 3

Sample Screenshot



KEY: F3	Description
ROW1:	MNEMONIC
ROW2:	PC, P, CRY, HEX/DEC MODE, ST
ROW3:	R0[W]
ROW4:	R1[W]
ROW5:	R2[W]
ROW6:	R3[W]
ROW7:	R4[W]
ROW8:	TOP 3 LEVELS OF RSTK
ROW9:	NEXT 3 LEVELS OF RSTK
ROW10:	

Sample Screenshot

5.4.4 Screen 4 – Data Pointers

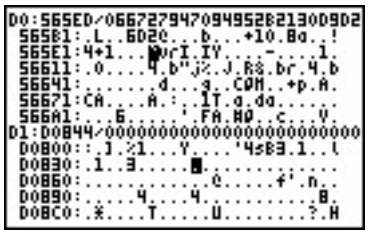
Screen 4 actually has three different views, all of which deal with the two data pointers `D0` and `D1`. The [F4] key will switch to Screen 4 if it is not already in view. Otherwise, it will switch to the next view of Screen 4.

For all three views, the [√x] key may be used to change the view to show information for both `D0` and `D1`, or only one of the two. Each screen may also have additional options. However, none have larger font equivalent.

5.4.5 Screen 4.1 – Data Stream

KEY: F4	Description
ROW1:	D0 MEMORY DUMP
ROW2:	D0 ASCII STREAM
ROW3:	.....
ROW4:	.....
ROW5:	.....
ROW6:	.....
ROW7:	.....
ROW8:	D1 MEMORY DUMP
ROW9:	D1 ASCII STREAM
ROW10:	.....
ROW11:	.....
ROW12:	.....
ROW13:	.....

Sample Screenshot



5.4.6 Screen 4.1 – RPL Stream

KEY: F4	Description
ROW1:	D0 RPL STREAM
ROW2:	.....
ROW3:	.....
ROW4:	.....
ROW5:	.....
ROW6:	.....
ROW7:	.....
ROW8:	D1 RPL STREAM
ROW9:	.....
ROW10:	.....
ROW11:	.....
ROW12:	.....
ROW13:	.....

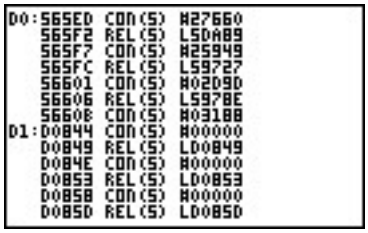
Sample Screenshot

- $\boxed{y^x}$ $\boxed{\text{0-8}}$ will show RSTK level n instead of DO.

5.4.7 Screen 4.3 – Data Table Stream

KEY: F4	Description
ROW1:	D0 TABLE STREAM
ROW2:	.....
ROW3:	.....
ROW4:	.....
ROW5:	.....
ROW6:	.....
ROW7:	.....
ROW8:	D1 TABLE STREAM
ROW9:	.....
ROW10:	.....
ROW11:	.....
ROW12:	.....
ROW13:	.....

Sample Screenshot



- $\boxed{0}$ ... $\boxed{\sqrt{x}}$ will toggle the table format.
- $\boxed{y^x}$ $\boxed{\text{0-8}}$ will show RSTK level n instead of DO.
- $\boxed{\text{TOOL}}$ $\boxed{3}$ toggles ascii/hexadecimal constants.

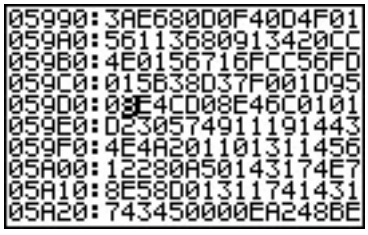
5.4.8 Screen 5 – Memory Dump

KEY: F5	Description
ROW1:	LOCATIONS 05990-0599F
ROW2:	LOCATIONS 059A0-059AF
ROW3:	LOCATIONS 059B0-059BF
ROW4:	LOCATIONS 059C0-059CF
ROW5:	LOCATIONS 059D0-059DF
ROW6:	LOCATIONS 059E0-059EF
ROW7:	LOCATIONS 059F0-05AFF
ROW8:	LOCATIONS 05A00-05A0F
ROW9:	LOCATIONS 05A10-05A1F
ROW10:	LOCATIONS 05A20-05A2F
ROW11:	LOCATIONS 05A30-05A3F
ROW12:	LOCATIONS 05A40-05A4F
ROW13:	LOCATIONS 05A50-05A5F

Sample Screenshot

KEY: F5	Description
ROW1:	LOCATIONS 05990-0599F
ROW2:	LOCATIONS 059A0-059AF
ROW3:	LOCATIONS 059B0-059BF
ROW4:	LOCATIONS 059C0-059CF
ROW5:	LOCATIONS 059D0-059DF
ROW6:	LOCATIONS 059E0-059EF
ROW7:	LOCATIONS 059F0-05AFF
ROW8:	LOCATIONS 05A00-05A0F
ROW9:	LOCATIONS 05A10-05A1F
ROW10:	LOCATIONS 05A20-05A2F

Sample Screenshot



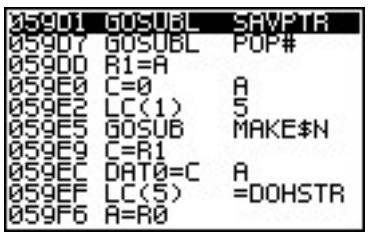
5.4.9 Screen 6 – ML Instruction Stream

KEY: F6	Description
ROW1:	INSTRUCTION 1
ROW2:	INSTRUCTION 2
ROW3:	INSTRUCTION 3
ROW4:	INSTRUCTION 4
ROW5:	INSTRUCTION 5
ROW6:	INSTRUCTION 6
ROW7:	INSTRUCTION 7
ROW8:	INSTRUCTION 8
ROW9:	INSTRUCTION 9
ROW10:	INSTRUCTION 10
ROW11:	INSTRUCTION 11
ROW12:	INSTRUCTION 12
ROW13:	INSTRUCTION 13

Sample Screenshot

KEY: F6	Description
ROW1:	INSTRUCTION 1
ROW2:	INSTRUCTION 2
ROW3:	INSTRUCTION 3
ROW4:	INSTRUCTION 4
ROW5:	INSTRUCTION 5
ROW6:	INSTRUCTION 6
ROW7:	INSTRUCTION 7
ROW8:	INSTRUCTION 8
ROW9:	INSTRUCTION 9
ROW10:	INSTRUCTION 10

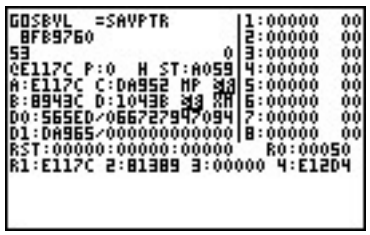
Sample Screenshot



5.4.10 Screen 7 – General CPU State & Breakpoints

KEY: APPS	Description	
ROW1:	MNEMONIC	BRK1
ROW2:	OPCODE	BRK2
ROW3:	CYCLES/TOTAL CYCLES	BRK3
ROW4:	PC, P, CRY, MODE, ST	BRK4
ROW5:	A[A], C[A], MP, SR	BRK5
ROW6:	B[A], D[A], SB, XM	BRK6
ROW7:	D0/D0[0-11]	BRK7
ROW8:	D1/D1[0-11]	BRK8
ROW9:	TOP 3 LEVELS OF RSTK, R0[A]	
ROW10:	R1[A], R2[A], R3[A], R4[A]	
ROW11:		
ROW12:		
ROW13:		

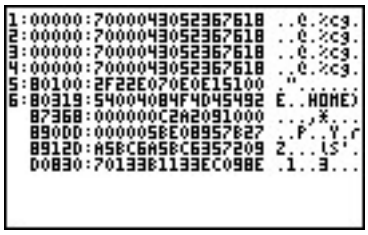
Sample Screenshot



5.4.11 Screen 8 – Watchpoints

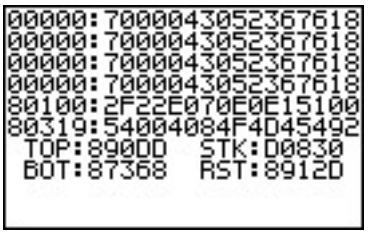
KEY: VAR	Description
ROW1:	WATCHPOINT 1
ROW2:	WATCHPOINT 2
ROW3:	WATCHPOINT 3
ROW4:	WATCHPOINT 4
ROW5:	WATCHPOINT 5
ROW6:	WATCHPOINT 6
ROW7:	TEMPOB BOTTOM WATCHPOINT
ROW8:	TEMPOB TOP WATCHPOINT
ROW9:	RPL RETURN STACK WATCHPOINT
ROW10:	RPL DATA STACK WATCHPOINT
ROW11:	
ROW12:	
ROW13:	

Sample Screenshot



KEY: VAR	Description
ROW1:	WATCHPOINT 1
ROW2:	WATCHPOINT 2
ROW3:	WATCHPOINT 3
ROW4:	WATCHPOINT 4
ROW5:	WATCHPOINT 5
ROW6:	WATCHPOINT 6
ROW7:	TEMPTOP, DSKTOP
ROW8:	TEMPBOT, RSKTOP
ROW9:	
ROW10:	

Sample Screenshot



- Watchpoints 1-6 are changeable, but are initialized by DB to 0 for watchpoints 1-4, to `(=IRAMBUFF)+11` for watchpoint 5, and to `=uart_buffer` for watchpoint 6.
- Watchpoints 7-10 are not changeable, but are updated continuously.

5.5 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 that instruction.

When stepping, the current registers are restored from MLDLPAR, then a jump to the execution buffer is made, and the jump back then causes the possibly changed registers to be saved back into MLDLPAR. The debugger will even execute the hardware bug in the `A=A+CON` and similar instructions on the HP48 series.[7] Although all new opcodes available to the HP49G+ and HP50G are also properly handled by DB, care should be taken when debugging unsupported ARM opcodes.

Obviously the method does not work for opcodes which change the PC. 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 ([EVAL])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

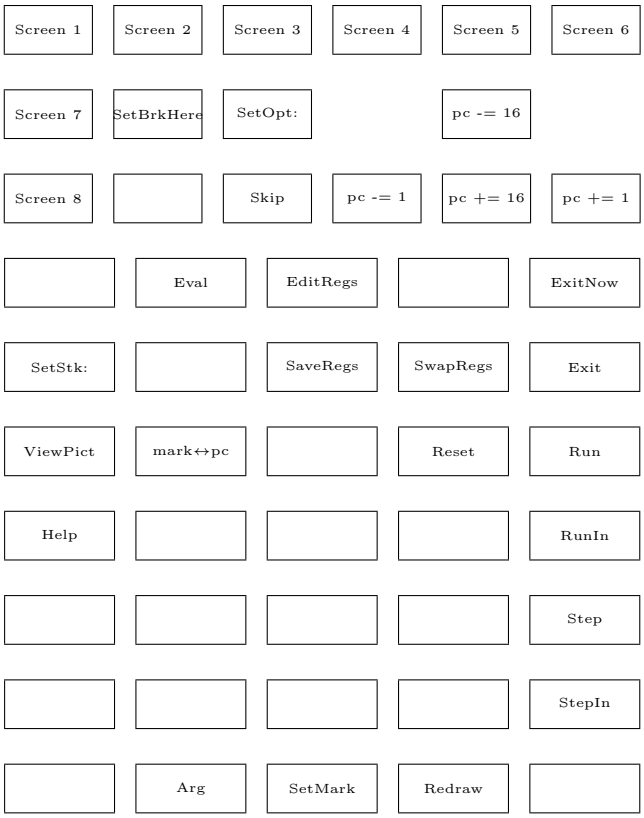
- The "ExitNow"-key ([⇐]) assumes the current CPU status is fine and executes the `LOOP` sequence to exit to RPL. Obviously this can be dangerous if the RPL status registers are not in place, or if the program hasn't completed any possible environment changes it has been doing.
- The "Exit"-key ([TAN]) 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,

[7] This bug has since been fixed in ROM 2.15 on the HP49G+ and HP50G.

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 ($\boxed{1/x}$) 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.

5.6 DB **Regular Mode Keyboard**



5.7 Movement in DB

Backward scrolling in machine language streams is a very complex task, and (at best) very prone to errors. Movement within a machine language stream is limited to the following 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 ([NXT]).
- "SetMark"-key ([.]) sets a mark at PC (or ARG), the PC and the mark can then be swapped by using the "mark↔pc"-key ([+/−]).

5.8 Register Save Buffer

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 the register usage of ROM code. One can save the registers ("SaveRegs"-key, [SIN]) before single-stepping a subroutine, and then press the "SwapRegs"-key ([COS]) 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.

5.9 Breakpoints

Up to eight breakpoint addresses can be set. 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.

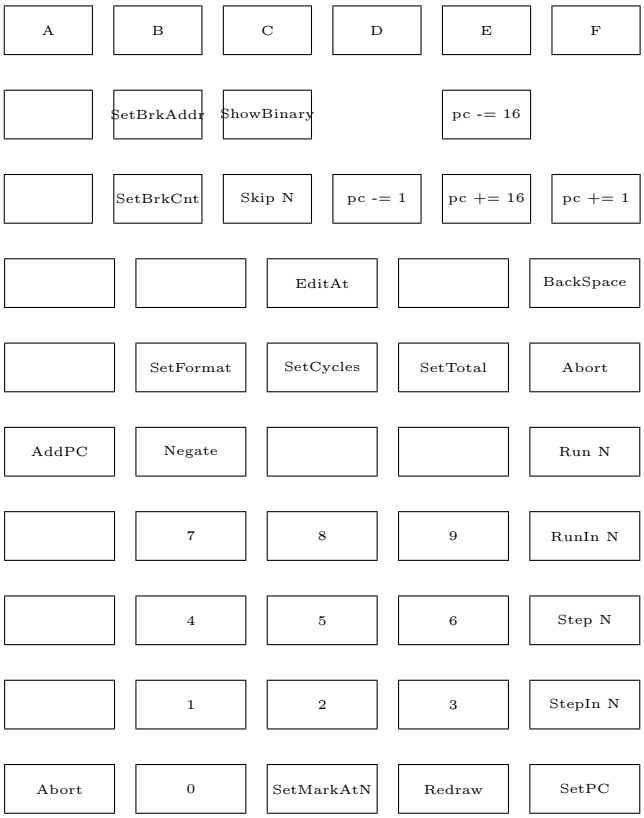
The table-comparison method does not work, though, when a breakpoint has been set into a subroutine which is stepped as a single entity with the "Step" or "Run" instructions. Thus DB stores a special breakpoint jump into DB at such breakpoint addresses. Naturally this works only when the debugged program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

Breakpoints are set as follows

- "SetBrkHere" ([MODE]) sets a breakpoint at the current PC.
- "SetBrkAddr" ([MODE]) sets a breakpoint at the address given as ARG.
- "SetBrkCntr" ([STO]) sets a breakpoint counter to zero (or ARG). The breakpoint number is given by pressing [1-8].

5.10 DB Arguments

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



5.11 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 for which they were designed.

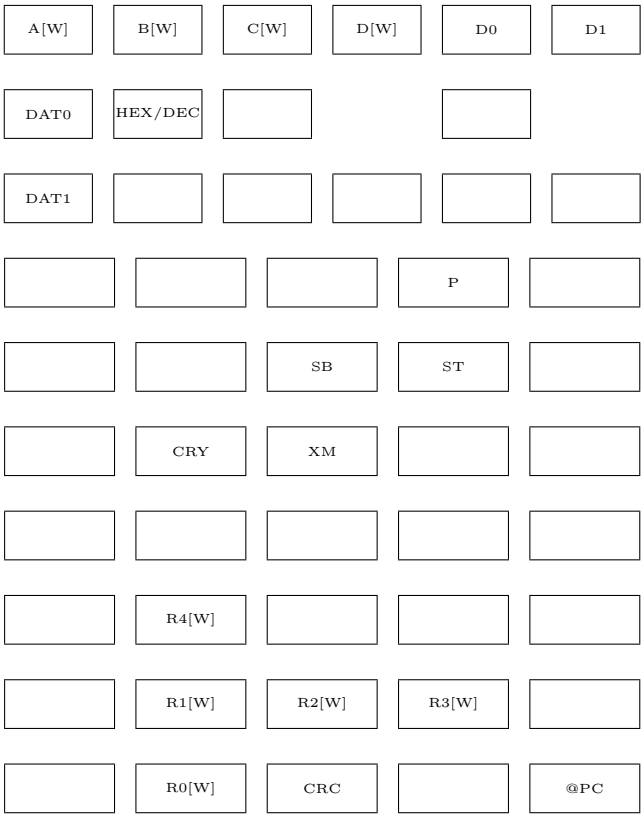
Cycles This counter is intended for measuring cycle counts for individual code segments or subroutines. "SetCycles"-key can be used to set a value to it, in particular using a zero-ARG will initialize it to zero.

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 could be determined experimentally, DB will use zero cycles.

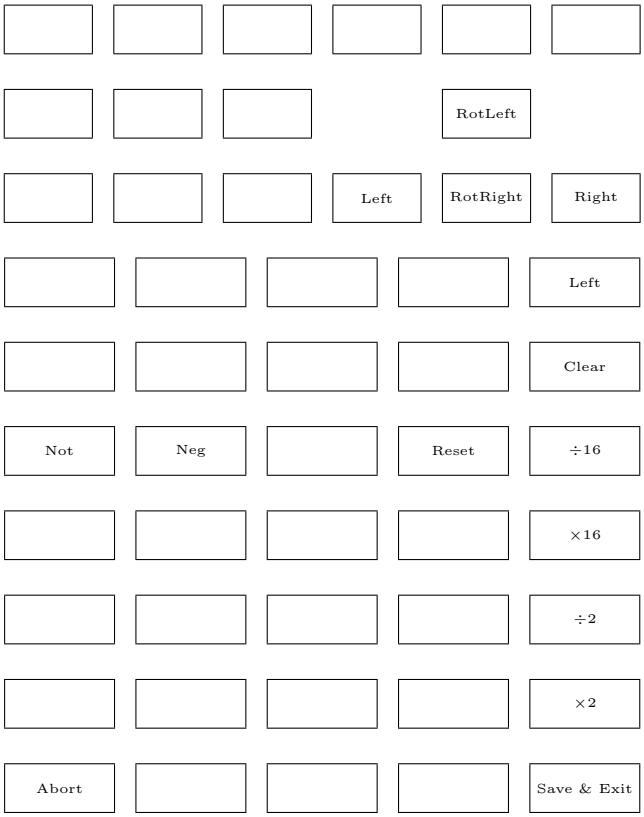
5.12 Register Editing

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



6 System RPL Debugger

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 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.

6.1 SDB Command

COMMAND:	SDB
DESCRIPTION:	Debug SRPL program. If the debugger is already running, then show the SDB menu.
STACK:	seco → id → lam → romptr →

6.2 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.

SKILL The SDB equivalent of the `KILL` command, which restores user settings prior to running SDB. For this reason, the built-int `KILL` should NOT be used

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.

7 Entries Catalog

The EC command enables browsing the entries in `extable`. As opposed to the assembler and disassembler EC does `extable` to be present.

COMMAND:	EC
DESCRIPTION:	Entries table browser.
STACK:	$\rightarrow$

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 (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 grob entry call VV else disassemble and call ED to view.
F1	Toggle alphabetical and address sorts.
F2	Toggle substring and exact match modes.
F3	View stack diagram. Requires SDIAG package.
F4	Toggle entry type.
F6	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.
'	Push entry address to the stack tagged with the name.
LS ENTER	Push entry address to the stack.
EVAL	Push entry address to the stack.
RS ENTER	Push entry name to the stack.
SYMB	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
Integer	DOINT	INT
Long Real	DOLNGREAL	L%
Long Complex	DOLNGCMP	LC%
Matrix	DOMATRIX	[M]
Flash Pointer	DOFLASHP	FPTR
Minifont	DOMINIFONT	MFNT
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
Access Pointer	DOACPTR	APTR
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
Type #1111h Pointer	#11111h	@1x5
Unprologued Pointer	(address)	SRPL
Machine Language		ML
Other		ML

Table 12: Displayed entry types in EC

8 Editor

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.

COMMAND:	ED
DESCRIPTION:	Editor.
STACK:	$ → $' $ %position → $' ob → ob'

COMMAND:	TED
DESCRIPTION:	Text editor.
STACK:	$ → $' ob → ob'

ED makes no duplicate of the string being edited if it is in temporary object area; it can edit 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.

While ED is very fast, it has to do special display adjustment calculations whenever the display is scrolled in order to support tabulation. Thus scrolling the display when very long lines are in view can appear to be slower.

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.

Sample ED Session

```
{
 # "PSTART"
 ::
  TakeOver
  MEMERR
  StackLineHeight
  PTR 2EF9A
  CODE
       AD1EX
       D1=(5)  =SysNib1
       C=DAT1  A
       D1=A
       ?CBIT=1 2
```

Status Screen

```
        Editor Status

Text size   :   1136   Insert
Clip size   :      0   Fast: OFF
Free memory : 168368   Case: OFF
Work memory : 169504   Beep: ON

Cursor pos  :     67
Cursor X    :     14
Cursor line :      8

Fstack lvl  :      1
Bank2 view  :      1
```

8.1 Editor Mode Keys

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

The status screen shows typical editing information. ED also maintains a stack of DOB calls 256 levels deep, and the current flash bank disassembled by DOB (see screenshot above).

8.2 Cursor Movement Keys

All cursor keys behave as described below regardless of alpha mode. Thus ▷ and α ▷ behave the same.

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.

8.3 Editing Keys

Unlike the HP48 series, the HP49G+ and HP50G do not have a dedicated delete key. This version of *JAZZ* implements the DEL key found on the HP48 calculators as the TAN key. Whereas the key combinations below for the ⇐ work regardless of the alpha mode, those associated with the TAN key are only valid when alpha lock is off.

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

8.4 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
RS APPS	BlockStart	Mark start of block at cursor.
RS MODE	BlockEnd	Mark end of block at cursor.
RS VAR	BlockCopy	Copy block/clip to cursor position.
or RS NXT		
RS STO	BlockCut	Copy block to clip and then delete it.
LS APPS	BlockUp	Mark end of block at cursor, start of block at start of text.
LS MODE	BlockDown	Mark start of block at cursor, end of block at end of text.
STO	BlockSwap	Swap clip and text.
F4	BlockDel	Delete block.
LS VAR	BlockKeep	Delete all but block.
LS HIST	BlockPush	Push block/clip to the stack.
LS STO	BlockPop	Pop string from stack to cursor position.
1/x	BlockRev	Reverse characters in block.
APPS	BlockToHex	Convert block to hexadecimal.
VAR	BlockToAsc	Convert block to ascii.

8.5 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. Find and replace keys normally act forward, starting at the cursor. The global replace keys, however, replace matches in the entire text and then return to the starting cursor position when done.

Key	Acronym	Description
F6	Find	Find.
NXT	FindNext	Find next match.
LS NXT	FindPrev	Find previous match.
LS F6	Repl?	Find/replace with verification.
RS F6	ReplAll	Find/replace all.
LS F5	GlobalRepl?	Global find/replace with verification.
RS F5	GlobalRepl	Global find/replace.

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.

8.6 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.

8.7 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
TOOL	MacroDo	Execute macro.
LS TOOL	MacroStart	Start macro definition.
RS TOOL	MacroEnd	End macro definition.

8.8 Editor Counter Keys

Key	Acronym	Description
F1	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.
F2	CntrOut	Inserts the current counter into text in the current format and increments the counter.

Below is an example producing line numbers using the macro editor as well as the counter.

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.

8.9 Character Catalog

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

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.

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

8.10 Editor Inputline

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

Key	Acronym	Description
[X]	Row?	Goto-row
[RS] [÷]	Arg?	Repeat next key N times.
[LS] [÷]	Pos?	Goto-cursor-position
[LS] [EVAL]	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] [EVAL]	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.

8.11 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
[LS] [']	`MATRIX`	Insert indented `MATRIX ;` into text.
[RS] [']	`SYMBOL`	Insert indented `SYMBOL ;` into text.
[RS] [−]	`ASS-RPL`	Insert `ASSEMBLE-RPL` into text.
[α] [RS] [·]	`::  ;`	Insert `::  ;` into text.
[α] [RS] [−]	`$ ""`	Insert `$ ""` into text.
[α] [LS] [+]	{}	Insert indented {} into text.
[α] [RS] [+]	`<< >>`	Insert indented ≪ ≫ into text.
[α] [LS] [SPC]	`CODE-END`	Insert `CODE-ENDCODE` into text.

Key	Acronym	Description
y^x	GOTO	Insert GOTO into text.
LS y^x	GOLONG	Insert GOLONG into text.
RS y^x	GOVLNG	Insert GOVLNG into text.
$\sqrt{x}$	GOSUB	Insert GOSUB into text.
LS $\sqrt{x}$	GOSUBL	Insert GOSUBL into text.
RS $\sqrt{x}$	GOSBVL	Insert GOSBVL into text.
SIN	GOYES	Insert GOYES into text.
LS SIN	GONC	Insert GONC into text.
RS SIN	GOC	Insert GOC into text.
COS	RTN	Insert RTN into text.
LS COS	RTNNC	Insert RTNNC into text.
RS COS	RTNC	Insert RTNC into text.

8.12 Editor Subprogram Keys

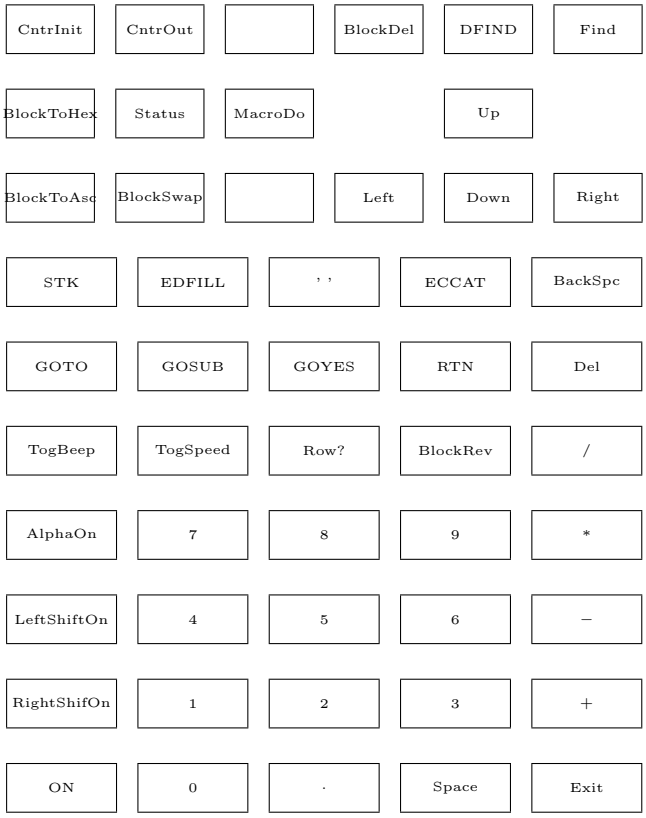
Key	Acronym	Description
LS ENTER	ASS	Assemble text. If assembly succeeds ED exits, else the error message is shown until a key press and the cursor is moved to the error location.
RS SYMB	DOB	Disassemble object under cursor and view the disassembly in a new editor. To come back simply exit the new editor.
SYMB	ECCAT	Start entries catalog in alphabetical mode. Any data pushed from the catalog is is inserted into the cursor position.
EVAL	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.
HIST	STK	Start internal system loop. The calculator is fully functional, including the possibility of starting a new editor. To come back to the editor press the CONT-key.
F3	SDIAG	Show SDIAG information for term under cursor.
F5	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

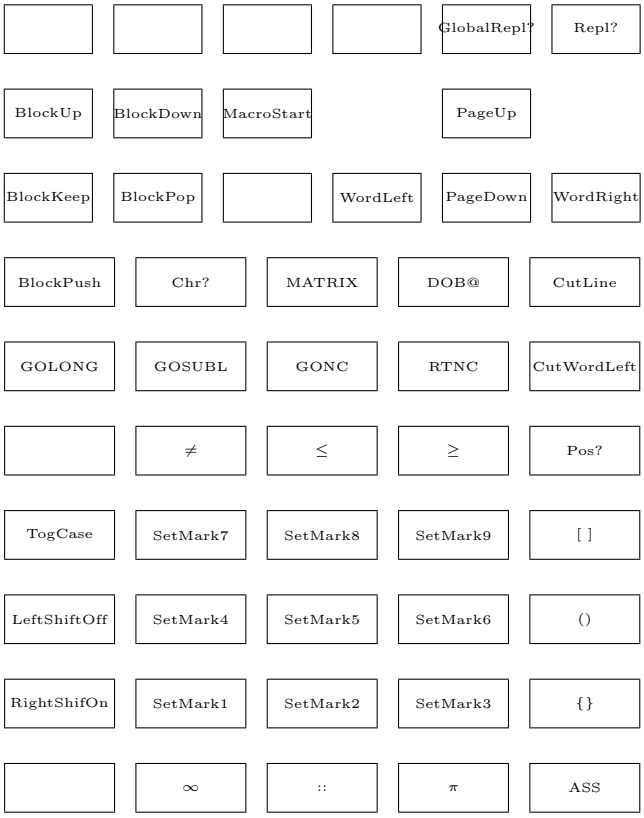
Tokens	Description
#<hex>	Generates disassembly of address.
L<hex>	Generates disassembly of address.
PTR <hex>	Generates disassembly of address.
ROMPTR <hex> <hex>	Generates disassembly of ROMPTR.
FPTR <hex> <hex>	Generates disassembly of the flash pointer.
FPTR2 <fptrname>	Generates disassembly of the flash pointer.
ID <name>	The contents are disassembled, edited, assembled and stored back.
LAM <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>	View the grob on top of the text.

8.13 Editor Keyboard Layout

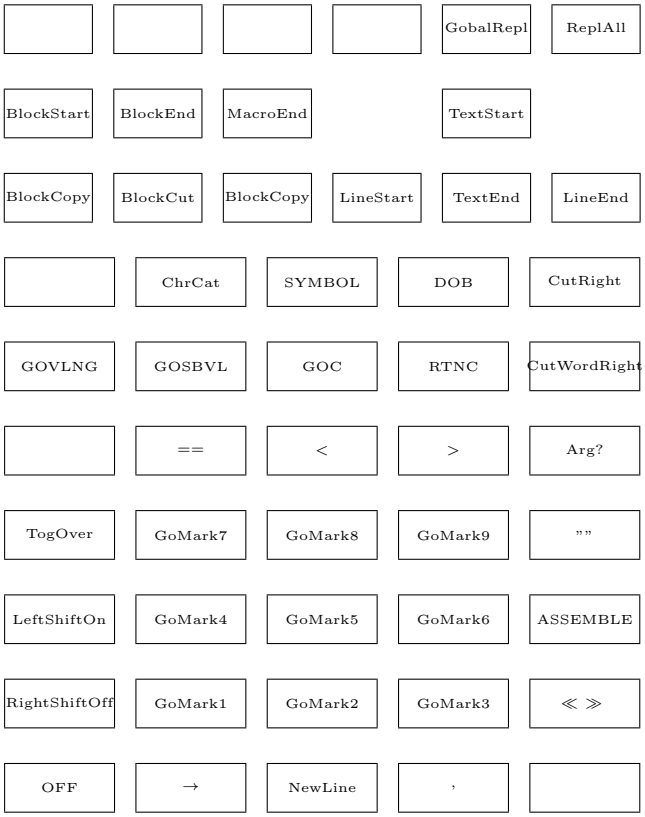
8.13.1 Non-shifted Keyboard



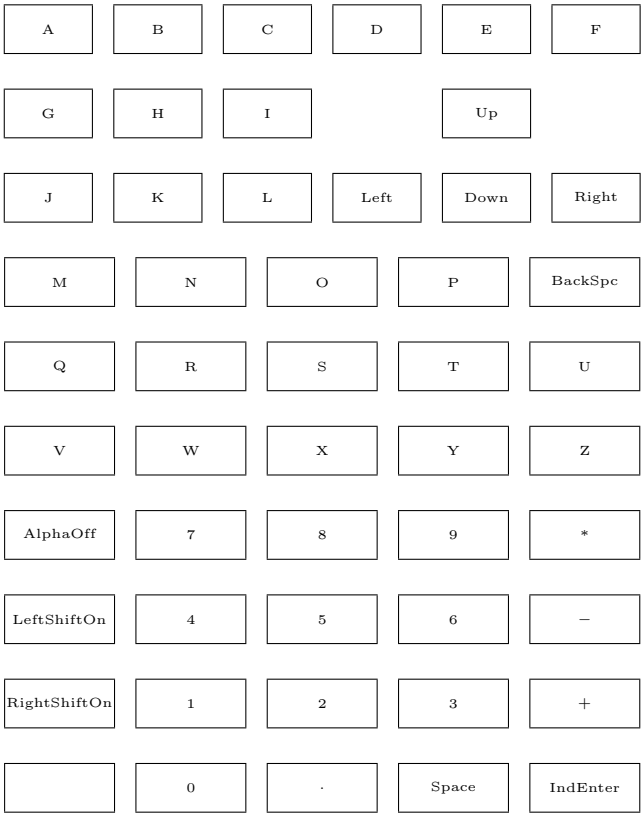
8.13.2 Left Shift Keyboard



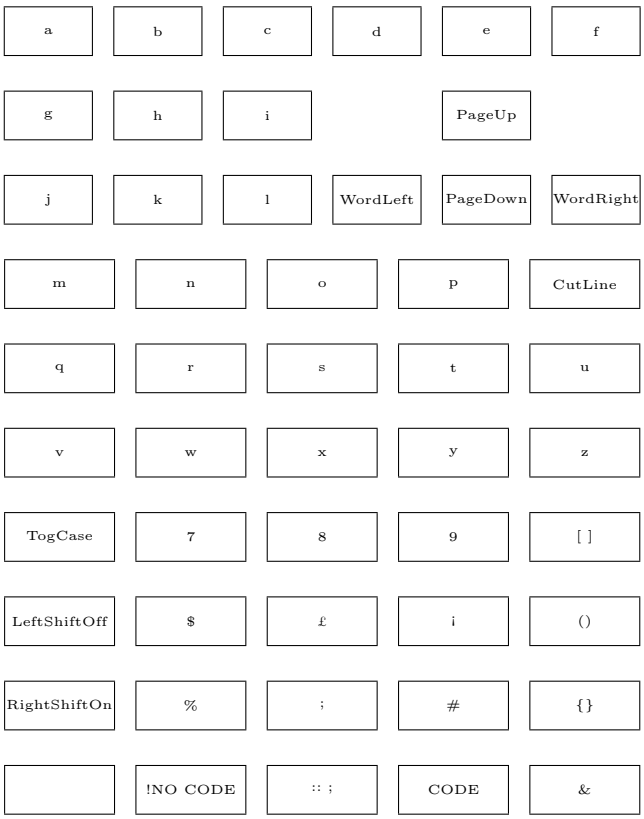
8.13.3 Right Shift Keyboard



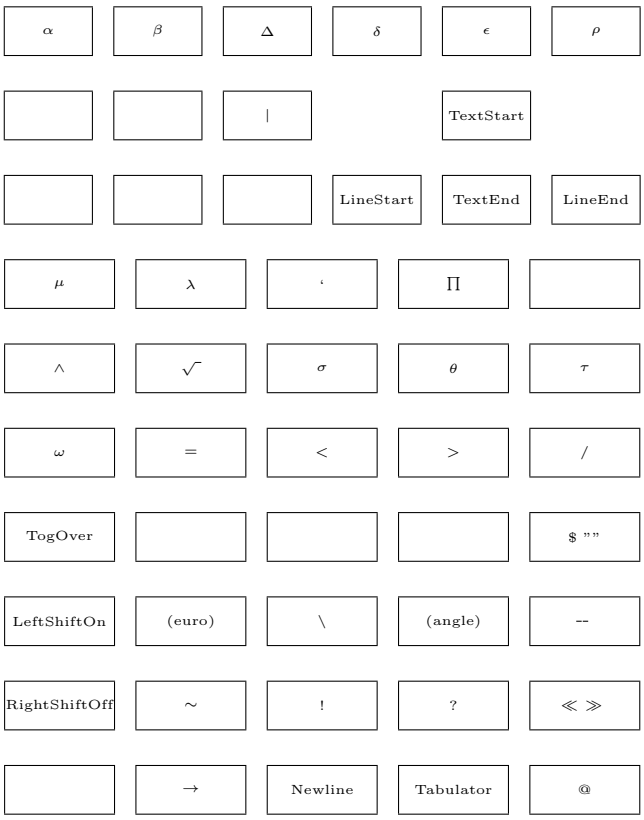
8.13.4 Alpha Shift Keyboard



8.13.5 Alpha Left Shift Keyboard



8.13.6 Alpha Right Shift Keyboard

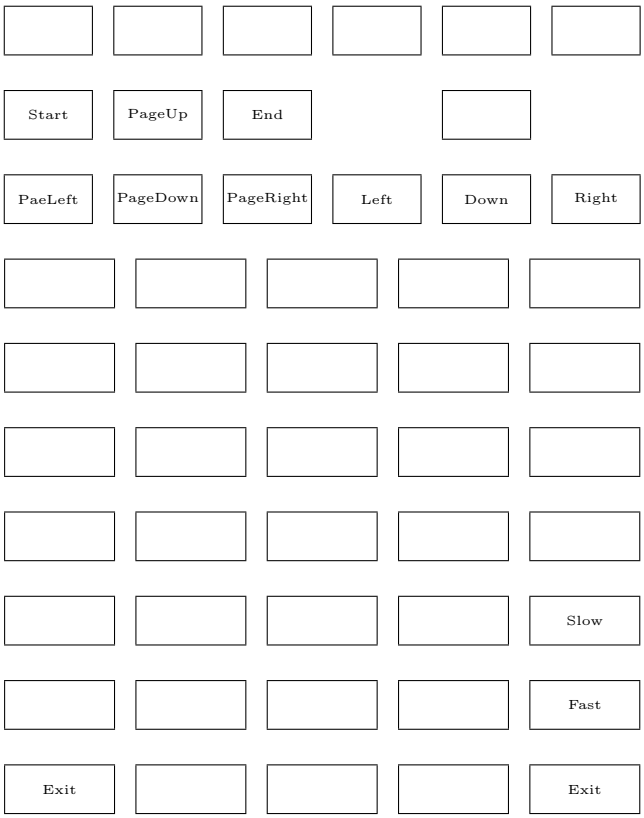


9 String and Grob Viewers

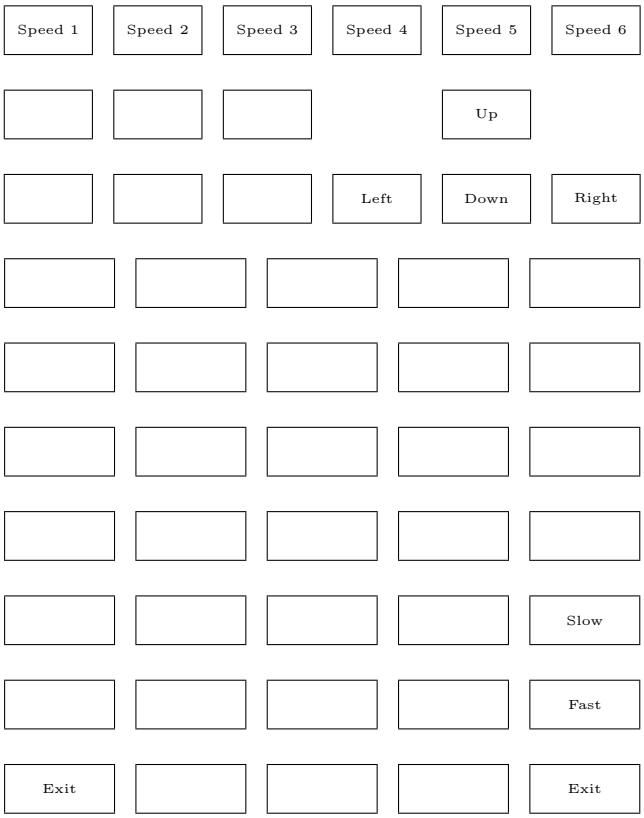
The string viewer is obsolete with ED. The grob viewer uses a "grob!" replacement with automatic cutting at the display borders. Masking grobs less than 4 bits wide is not properly implemented yet, thus grobs that narrow may appear strange or even vanish from the display. As a special feature VV recognizes the AGROB fancy print decompiler which exists for example in EQSTK and JAVA libraries. This feature is somewhat obsolete since fancy print is built-in on the HP49G+ and HP50G although there are arguably significant differences in how algebraic objects are decompiled.

COMMAND:	VV
DESCRIPTION:	String and grob viewer.
STACK:	$ → $ grob → grob ob → ob

9.1 String Viewer Keys



9.2 Grob Viewer Keys



10 Entry & Memory Utilities

The HP49G+ and HP50G port of *JAZZ* no longer supports entry table generation on the calculator itself since these tables are now so large that building them on the calculator is no longer efficient. However, *JAZZ* still maintains the entry conversion command EA.

COMMAND:	EA
DESCRIPTION:	Convert between entry addresses and entry names.
STACK:	$entry → hxs_address hxs_address → $entry ob → hxs_address

JAZZ also provides two new hidden commands for studying the ROM. The built-in command PEEK calls a flash pointer that is supposed to peek into ROM and copy the underlying nibbles into a string object. Since the system handles flash pointers by switching to the appropriate flash bank, peeking into ROM in the address range `#40000h` – `#7FFFFh` results in erroneous output (it will always return the data from the flash bank called by PEEK). *JAZZ* provides an equivalent, hidden command that avoids this problem:

COMMAND:	Peek
DESCRIPTION:	Full-ROM equivalent of PEEK.
STACK:	hxs_address hxs_length → $nibbles

COMMAND:	FPTR→
DESCRIPTION:	Disassembles a flash pointer.
STACK:	fptr → $ hxs_addressend
KEYS:	ON key aborts
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.

11 SRPL Stack Display

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 the error handler has been coded to resemble that of the HP48 series. To exit SSTK simply execute it again.

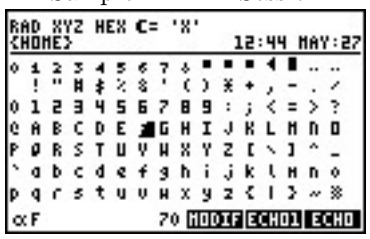
COMMAND:	SSTK
DESCRIPTION:	System RPL stack display.
STACK:	→

12 Minifont Editor

Since ED as well as several screens of DB make use of the MINIFONT, *JAZZ* now provides a utitlity to modify the system MINIFONT via MFED. The editor is essentially the same as built-in character catalog.

COMMAND:	MFED
DESCRIPTION:	Minifont editor.
STACK:	minifont → minifont'

Sample MFED Session



While editing, use the cursor keys to highlight the desired character. Press F4 to begin editing the character. Although the editor uses a 6×6 grid, all characters will be cropped down to the leftmost 6×4 pixels. The other two menu labels are disabled. Press ENTER to save the changes, or ON to abort all changes.

A 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 = library 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 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
}
```

B *JAZZ* Command Index

COMMAND:	ASS
DESCRIPTION:	Assemble source string
STACK:	$ → ob $ → $ %erropos
KEYS:	ON key aborts
FLAGS:	1 – Report mode on when set 7 – Do not use entry tables when set

COMMAND:	DIS
DESCRIPTION:	Disassemble object.
STACK:	ob → $
KEYS:	ON key aborts
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:	#address → $ hxs_end hxs_start → $ hxs_end "entry" → $ hxs_end
KEYS:	ON key aborts
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 end address guessing.
STACK:	hxs_start hxs_end → $
KEYS:	ON key aborts
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:	Disassemble n machine language instructions.
STACK:	#address %n → $
KEYS:	ON key aborts
FLAGS:	5 – Tabulator disabled when set. 6 – Put labels on own rows when set. 7 – Entry tables disabled when set.

COMMAND:	DB
DESCRIPTION:	Debug machine language.
STACK:	id → romptr → $entry → #address → hxs_address →

COMMAND:	SDB
DESCRIPTION:	Debug SRPL program. If the debugger is already running, then show the SDB menu.
STACK:	seco → id → lam → romptr →

COMMAND:	SHALT
DESCRIPTION:	SDB HALT command.
STACK:	→

COMMAND:	SKILL
DESCRIPTION:	SDB KILLcommand.
STACK:	→

COMMAND:	EC
DESCRIPTION:	Entries table browser.
STACK:	→

COMMAND:	ED
DESCRIPTION:	Editor.
STACK:	$ → $’ $ %position → $’ ob → ob’

COMMAND:	TED
DESCRIPTION:	Text editor.
STACK:	$ → $’ ob → ob’

COMMAND:	VV
DESCRIPTION:	String and grob viewer.
STACK:	$ → $ grob → grob ob → ob

COMMAND:	EA
DESCRIPTION:	Convert between entry addresses and entry names.
STACK:	$entry → hxs_address hxs_address → $entry ob → hxs_address

COMMAND:	SSTK
DESCRIPTION:	System RPL stack display.
STACK:	→

COMMAND:	MFED
DESCRIPTION:	Minifont editor.
STACK:	minifont → minifont'

COMMAND:	DBHOOK
DESCRIPTION:	Hidden command to initialize DB hooks.
STACK:	→

COMMAND:	Peek
DESCRIPTION:	Full-ROM equivalent of PEEK.
STACK:	hxs_address hxs_length → $nibbles

COMMAND:	FPTR→
DESCRIPTION:	Disassembles a flash pointer.
STACK:	fptr → $ hxs_address
KEYS:	ON key aborts
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.

C Error Messages

C.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.

C.2 Assembler Error Messages

ACPTR is a G Object

Access pointers cannot be compiled on a S/SX.

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 G/GX.

Field Selector Expected

A field selector is expected.

FPTR CMD > FFFF

FPTR CMD must be between `#0h` and `#FFFFh`.

FPTR ID > FFF

FPTR ID must be between `#0h` and `#FFFh`.

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 FPTR CMD

Second token after FPTR must be a hexadecimal value from `#0h` to `#FFFFh`.

Invalid FPTR ID

Token after FPTR must be a hexadecimal value from `#0h` to `#FFFh`.

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` to `#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.

Invalid ZINT

Integer contains non-decimal digits.

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` to `#FFFh`.

ROMPTR RomWd > FFF

Command number must be in the range `#0` to #FFFh.

Too Big #

Integers should be in the range `#0` to `#FFFFFh`.

Too Big Exponent

Expression has an exponent larger than 48.

Too Big PTR

Address should be in the range `#0` to `#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.