HP 48SX MACHINE LANGUAGE

DEVELOPMENT LIBRARY

USER'S GUIDE & REFERENCE MANUAL

Jan Brittenson

HP 48SX
Machine Language Development Library
Version 1.04B

User's Guide and Reference Manual

July, 1991

This manual and any examples contained herein are provided "**as is**" and are subject to change without notice. Jan Brittenson makes no warranty **of any kind** with regard to this manual, including but not limited to the implied **warranty of merchantability** or **fitness for a particular purpose**. Jan Brittenson shall **not be liable** for any errors or for incidental or consequential damages, such as hardware failures, in connection with the furnishing, performance, or use of this manual or the examples herein.

The MLDL makes use of undocumented internal features of the HP 48 calculator. HP 48 Assembler programming in particluar is **not supported by Hewlett-Packard**.

This manual was set with LaTeX in 12pt Courier.

Contents

1 Introduction

The MLDL is a Machine Language Development Library for the HP 48, and as such is primarily intended for HP 48 Machine-Language Programmers familiar with the following topics[1].

- The SATURN instruction set
- Assembler programming
- Machine Language debugging
- RPL internals – GC, System RPL, PMC

A great source of information on the above topics, and on HP 48 internals in general, is EduCALC's "Goodies" disks. Also of value is the documentation included with Hewlett-Packard's unsupported development tools. Some of the conferences on HP's Handhelds BBS (503-750-4448, 300–2400 bps, 24hrs) are good sources of information – especially the ones linked to USENET.

The style used throughout this manual is summarized below.

- Hexadecimal numbers are typeset in **boldface** whenever they occur in the main text. Example: **0679B**. Decimal numbers are typeset according to context, but never in boldface.
- Acronyms are set in SMALL CAPS. Example: RPL
- RPL commands and ML instructions appearing in the main text are typeset in typewriter font. Example: `ATTACH`

[1] See appendix D (page 106) for a list of commonly used abbreviations.

- RPL listings are set with each line split into two parts: the leftmost part is the program code, in typewriter font, and flush right on each line is a comment in oblique type.
- The notation @#*address* is the object at *address* . Example: @#**3AC0**

This manual describes the MLDL. It is not intended to teach assembly programming in general or HP 48 internals.

The MLDL displays mnemonics according to the AG format, named after Alonzo Gariepy, who designed it explicitly to resemble the mnemonics of other commonly used processors. Hewlett-Packard also has a set of mnemonics, described in files bundled with their unsupported development tools. The HP mnemonics set and assembler syntax do not adhere to a *source-destination* model. Instructions sometimes span over several lines, it is column-oriented, and is by some people perceived as somewhat unfriendly.

STAR is an assembler implementing the AG set of mnemonics. It is a macro assembler, available as free software for a distribution charge. The standard distribution will compile and run on any UNIX system implementing ANSI C, VAX/VMS, MS-DOS, or AmigaDOS. It comes with binary executables for the latter two systems. STAR Version 1.04.4 can be ordered by mailing a check or money order for $16 (5 1/4") or $19 (3 1/2") to:

STAR Request
c/o L. Highleyman
MIT AI Lab, Rm 772
545 Technology Square
Cambridge, MA 02139

Add $1 if you would like to receive information on future STAR versions.

The MLDL is also available for free in a version that will only run from RAM. It is non-commercial and available on one of EduCALC's "Goodies" disks. Call EduCALC at (714) 582-2637 for details.

Thanks to A. Gariepy for his restructuring of the SATURN instruction set, to R. Grevelle and J. Ervin for their continuous testing and helpful advice,

and to everyone near and far who has contributed to charting the HP 48 internals. Thanks also to L. Highleyman for her professional editing of this manual.

2 ROM Card Handling

To install the MLDL ROM card, turn the calculator off, remove the card bay cover, plug in the ROM card in an unoccupied port, replace the opaque cover, and turn the calculator back on[2].

The MLDL will automatically attach to the **HOME** directory, becoming available in the LIBRARY menu.

[2]See the HP 48 *Owner's Manual, Volume II*, pages 636–638 for more details on how to install cards.

3 Library Commands

3.1 ABOUTMLDL

Paints a screen with the current version and copyright information.

3.2 MLDB

HP 48 ML Debugger. This description covers the MLDB local mode. The debugger can also operate in one of two server modes, interactive or protocol, described under MLDB Server Modes below. In local mode, the HP 48 display and keyboard, are used to control the debugger. In server mode, control is maintained via the serial port, either from a dumb ASCII terminal (interactive mode), or from a dedicated front-end (protocol mode). To ascertain that MLDB is in local mode when invoked, clear user flags 32 and 33:

32 CF 33 CF

The MLDB permits you to single-step ML programs, as well as examine registers and memory contents. Since it single-steps ML only, it is not generally useful for debugging RPL code, unless you wish to follow an RPL thread on an ML level.

3.2.1 MLDB Arguments

When invoked, MLDB expects an argument in level 1:

Global name (variable)

The value of the global name is recursively used as an argument. **Caution:** The value can be another global name, whose value is then used in turn. If the variable's value is its own name, an endless recursion will occur. This is intentional – abort with ON-C.

Code object

MLDB halts before the first instruction of the code object. Both the PC and A registers are set to the address of the first instruction.

XLIB

The XLIB must be a code object, which becomes the argument.

Binary integer

MLDB treats the binary integer as the address of a prefixed machine code routine (PMC). It halts at the first instruction of the PMC. PMCs consist of a 5-nybble pointer to the first instruction, which is usually, but not always, the address of the PMC plus 5.

@#3A81 (True)

If the token following the MLDB invocation is a code object, the MLDB halts at its first instruction. Any other type of object results in an error. When the code object program completes, RPL execution continues with the next consecutive word. When the argument is **@#3AC0** (False) the invocation of MLDB is ignored, so the invocation can be preceded by a test – for instance a user flag test, which permits the debugging of individual machine language objects embedded in RPL programs to be turned on and off by setting user flags.

Any other object type

Other objects are ignored – the debugger returns immediately. Included here is @#**3AC0** (False).

The MLDB local mode uses PICT (the graphical display) to present information. Since all information will not fit on one display, it has been divided into eight screens. Only one screen is active at any time. Switching between screens is done by means of the six menu keys – [A] through [F], the [MTH] key, and the [VAR] key.

The sample screens 1-8 in section 3.2.3 (page 11) are from the PMC at **59CC**, and can be approximately reproduced by typing the following:

#59CCh [MLDB]

3.2.2 The MLDLpar Variable

Some MLDL commands ([MLDB], [ML1], [MLPR]) automatically create an **MLDLpar** variable in the HOME directory, which contains configuration data and variables used. It is about 650 bytes. Depending on the number of directories and variables in the HP 48, a significant gain in speed can be achieved by reordering the HOME directory so the **MLDLpar** variable appears last in the [VAR] menu[3].

3.2.3 Screen 1 — General SATURN State ([A])

Mnemonic	CALL.4 #0679B
Opcode	8E4CD0
PC, P, Carry, HEX/DEC mode, ST	@:059D1 P:0 CH ST:218
A.A and C.A	A:000CC C:77794
B.A, D.A, and HST	B:729A9 D:00F96 HST:2
D0 and 6 bytes @D0	D0:409C1/9540A8240BC9
D1 and 6 bytes @D1	D1:77799/000000000000
Top 3 levels of RSTK	RST:00000:00000:00000

[3]See the **ORDER** command on page 113 in the HP 48 *Owner's Manual, Vol I.*

Mnemonic is the current instruction, pointed to by the PC. *Opcode* is the opcode of the current instruction. The third line is the current PC, P register, the status of the carry bit (a `C` if carry is set, blank if clear), HEX/DEC mode (`D` = DEC mode, `H` = HEX mode) as set by the `SETDEC` and `SETHEX` instructions, and the low three nybbles of the ST register. These three lines are common to many of the MLDB screens described below.

Lines four and five are the low five nybbles (the `.A` field) of the A, B, C, and D registers, as well as the HST register. Lines six and seven are the D0 and D1 registers, as well as the contents of the addresses they point to, as twelve-nybble integers. Thus the address pointed to by either register is displayed as the *rightmost* digit, with the next consecutive address being the next significant digit, and so on. This is because the SATURN CPU stores integers in memory with the least significant digit (nybble) at the lowest address, and the most significant digit (nybble) at the highest.

The bottom line is common to several of the screens described below. It contains the top three RSTK levels.

3.2.4 Screen 2 — Arithmetic Registers (B)

Mnemonic	`CALL.4 #0679B`
Opcode	`8E4CD0`
PC, P, Carry, Hex/Dec mode, ST	`@:059D1 P:0 CH ST:218`
Register A	`A:000000054440000CC`
Register B	`B:000000000007611E`
Register C	`C:000000000007792C`
Register D	`D:00000000000004D0`
Top 3 levels of RSTK	`RST:00000:00000:00000`

See section 3.2.3 (page 12) for an explanation of the top three and bottom lines.

Lines four to seven are registers A, B, C, and D respectively. All 16 nybbles (64 bits) are displayed.

3.2.5 Screen 3 — Data Registers (C)

Mnemonic	CALL.4 #0679B
Opcode	8E4CD0
PC, P, Carry, Hex/Dec mode, ST	0:059D1 P:0 CH ST:218
Register R0	R0:053385D439800040
Register R1	R1:00000005444059D1
Register R2	R2:0000000000075BC1
Register R3	R3:0000000544402E92
Register R4	R4:00015075A6375AA1

See section 3.2.3 (page 12) for an explanation of the top three lines.

Lines four to eight are registers R0, R1, R2, R3, and R4 respectively. All 16 nybbles (64 bits) are displayed.

3.2.6 Screen 4 — Return Stack (D)

Mnemonic	CALL.4 #0679B
Opcode	8E4CD0
PC, P, Carry, Hex/Dec mode, ST	0:059D1 P:0 CH ST:218
RSTK levels 0 and 4	RST0:00000 RST4:00000
RSTK levels 1 and 5	RST1:00000 RST5:00000
RSTK levels 2 and 6	RST2:00000 RST6:00000
RSTK levels 3 and 7	RST3:00000 RST7:00000

See section 3.2.3 (page 12) for an explanation of the top three lines.

Lines five to eight are the eight levels of RSTK, displayed as 5-nybble integers. RSTK0 is the top of the stack, and is the most recent return address or value pushed (by means of the PUSH instruction)[4].

[4]For more details on RSTK operations, see appendix E.

3.2.7 Screen 5 — Memory Dump (E)

Locations 59A0-59AF
Locations 59B0-59BF
Locations 59C0-59CF
Locations 59D0-59DF
Locations 59E0-59EF
Locations 59F0-59FF
Locations 5A00-5A0F
Locations 5A10-5A1F

```
059A0:56113680913420CC
059B0:4E0156716FCC56FD
059C0:015B38D5E0101D95
059D0:08E4CD08E46C0101
059E0:D230574911191443
059F0:4E4A201101311456
05A00:12280A50143174E7
05A10:8E58D01311741431
```

The 128 addresses surrounding the current PC are displayed, with the line corresponding to the current PC at the center. The location of the current instruction is indicated by an inverse digit (not reproduced here). Each line is 16 nybbles, with the leftmost nybble corresponding to the lowest address. The memory dump addresses can be shifted one nybble using the CST 4 option (see section 3.3.12 (page 36)).

3.2.8 Screen 6 — ML Instructions (F)

PC, P, Carry, Hex/Dec mode, ST
Next 7 instructions

```
@:059D1 P:0 CH ST:218
#CALL.4##0679B########
 CALL.4 #06641
 MOVE.W A,R1
 CLR.A C
 MOVE.P1 #5,C
 CALL.3 #05B7D
 MOVE.W R1,C
```

The top line is the current PC, P register, the status of the carry bit (C if carry is set, blank if clear), HEX/DEC mode (D = DEC mode, H = HEX mode) as set by the **SETDEC** and **SETHEX** instructions, and the low three nybbles of the ST register.

The remaining seven lines are the seven subsequent instructions, with the next instruction to be executed always appearing in inverse at the top of the screen (here somewhat crudely illustrated with # characters).

3.2.9 Screen 7 — Breakpoint Table (MTH)

Breakpoint #1
Breakpoint #2
Breakpoint #3
Breakpoints #4-#8: not used

```
1:6100 +02
2:6104 -02
3:613A 00
4:0000 00
5:0000 00
6:0000 00
7:0000 00
8:0000 00
```

Each line corresponds to a breakpoint. There are eight breakpoints, each consisting of an address and a counter. The address is displayed immediately following the breakpoint number. Trailing each line is the breakpoint counter, a signed 2-nybble (8-bit) hex integer. Any breakpoints at the current location are displayed in inverse.

3.2.10 Screen 8 — Watchpoint Table (VAR)

Watchpoint #1
Watchpoint #2
Watchpoint #3
Watchpoint #4
Watchpoint #5-#8: not used

```
00138:9DEF0C1085D1BF21
01400:8F235A05C965E186
FFFFF:12369B108DADF100
70000:F3C5A0000FB13FB3
00000:2369B108DADF1008
00000:2369B108DADF1008
00000:2369B108DADF1008
00000:2369B108DADF1008
```

Each line corresponds to a watchpoint. A watchpoint can be set to any arbitrary address. The display format is consistent with the format of the memory dump screen.

3.2.11 The HP 48 Annunciators

The display annunciators are left undisturbed by the local mode MLDB. If in server mode, only the I/O annunciator[5] is used; it will flicker with each character received over the serial line. The annunciators are left in whatever state the HP 48 ROM put them in, which means that the busy annunciator is usually lit and all other annunciators are turned off. For all practial purposes, as far as the HP 48 ROM is concerned a system RPL program is executing, even if it is temporarily halted between instructions by MLDB.

The local mode MLDB puts the SATURN microprocessor in light sleep between keystrokes[6], which means that battery consumption is generally low even if the busy annunciator is lit. Light sleep is not used when in server mode.

3.3 The Local Mode MLDB Keyboard

A number of keys will accept an argument, refered to as **ARG**. To enter **ARG**, press [0] followed by the hex integer which will make up **ARG**. It is restricted to 5 hex digits (20 bits). When [0] is pressed, the bottom display line, regardless of screen currently active, is turned into an **ARG** entry line:

```
|ARG:00000                    |
```

This remains in effect until a non-hex key is pressed, at which point that key is executed. The hex number entered becomes **ARG**. Some keys behave differently depending on whether **ARG** was entered; [DEL], for instance, does nothing, in effect cancelling **ARG**. The backarrow, [⇐], divides the argument by 16, in effect shifting it right one digit. The argument is always a 20-bit unsigned integer. The digits **A-F** are found on the menu keys. The [+/-] key negates the argument (two's complement).

---

[5]The arrow in the upper right corner

[6]Much of the HP-48 circuitry is turned off while the display is on; pressing any key causes a wake-up event which turns the power back on to allow processing of the keystroke.

3.3.1 Example: ARG Entry

This example illustrates how to enter the argument.

Begin **ARG** entry.

Keys	Display
[0]	ARG:00000

Enter the hex digit **5**.

[5]	ARG:00005

Enter the hex digit **7**.

[7]	ARG:00057

Erase the **7**. To do this, shift right one digit.

[⇐]	ARG:00005

We are now back to where we were before we entered the **7**. Enter the digit **9**.

[9]	ARG:00059

Enter two more hex digits, **D** and **1**.

[D] [1]	ARG:059D1

Negate (2's complement).

[+/-]	ARG:FA62F

Enter three **0**s.

[0] [0] [0]	ARG:2F000

Erase the last 0.

[⇐] ARG:02F00

Finally, since we are only practicing we cancel **ARG**.

[DEL]

3.3.2 Moving Around

Moving around is done with the arrow keys and the [NXT] key; notice that none of these keys actually execute the previous instruction, only the PC is affected. You can also move to a specific address with [ENTER].

Keys	Description
[△]	Decrement PC by 16 (or 16 × **ARG**)
[▽]	Increment PC by 16 (or 16 × **ARG**)
[◁]	Decrement PC by 1 (or **ARG**)
[▷]	Increment PC by 1 (or **ARG**)
[NXT]	Move to next instruction (or **ARG** instructions forward)
[ENTER]	Set PC to **ARG**.

In addition to the movement keys listed previously, there is a mark:

Keys	Description
[×]	Set mark to **ARG**, if present. Otherwise the mark is set to the current PC.
[+/-]	Swaps the PC and the mark.

3.3.3 Example: Moving Around

First, set the PC to **59D1**.

Keys

059D1 [ENTER]

Display

```
@:059D1 P:0 CH ST:218
#CALL.4##0679B########
 CALL.4 #06641
 MOVE.W A,R1
 CLR.A C
 MOVE.P1 #5,C
 CALL.3 #05B7D
 MOVE.W R1,C
```

Increment the PC by 4.

04 [▷]

```
@:059D5 P:0 CH ST:218
#CLR.A#A###############
 CALL.4 #06641
 MOVE.W A,R1
 CLR.A C
 MOVE.P1 #5,C
 CALL.3 #05B7D
 MOVE.W R1,C
```

We didn't really advance the PC by one full instruction, which means that the PC now points two nybbles into the offset of the previous CALL instruction. Set the PC back to **59D1** and switch to the memory screen.

059D1 [ENTER] [E]

```
059A0:56113680913420CC
059B0:4E0156716FCC56FD
059C0:015B38D5E0101D95
059D0:0#E4CD08E46C0101
059E0:D230574911191443
059F0:4E4A201101311456
05A00:12280A50143174E7
05A10:8E58D01311741431
```

Decrement the PC by **10**.

[△]

```
05990:3A6E80D0F40D4F01
059A0:56113680913420CC
059B0:4E0156716FCC56FD
059C0:0#5B38D5E0101D95
059D0:08E4CD08E46C0101
059E0:D230574911191443
059F0:4E4A201101311456
05A00:12280A50143174E7
```

Set the PC back to **59D1** and switch to the Instructions Screen.

059D1 [ENTER]
[F]

```
@:059D1 P:0 CH ST:218
#CALL.4##679B#########
 CALL.4 #06641
 MOVE.W A,R1
 CLR.A C
 MOVE.P1 #5,C
 CALL.3 #05B7D
 MOVE.W R1,C
```

Move the PC forward to the next instruction.

NXT

```
@:059D7 P:0 CH ST:218
#CALL.4##06641########
 MOVE.W A,R1
 CLR.A C
 MOVE.P1 #5,C
 CALL.3 #05B7D
 MOVE.W R1,C
 MOVE.A C,@D0
```

Set the mark and advance three instructions.

×
03 NXT

```
@:059D7 P:0 CH ST:218
#MOVE.P1##5,C#########
 CALL.3 #05B7D
 MOVE.W R1,C
 MOVE.A C,@D0
 MOVE.P5 #02A4E,C
 MOVE.W R0,A
 MOVE.A A,D1
```

Swap the mark and the PC.

+/-

```
@:059D7 P:0 CH ST:218
#CALL.4##06641########
 MOVE.W A,R1
 CLR.A C
 MOVE.P1 #5,C
 CALL.3 #05B7D
 MOVE.W R1,C
 MOVE.A C,@D0
```

The arrow keys are most useful for moving around in the memory dump, but can also be used to arbitrarily increment and decrement the PC to shift the instruction stream by single nybbles, as illustrated in the example above. The mark commands are useful for temporarily remembering an address.

3.3.4 Terminating the Program

There are four ways to leave the MLDB. The two first, listed below, continue RPL execution at the next token in the thread. If invoked via the LIBRARY MLDL menu, control is returned to the normal calculator operation. The third method of leaving the MLDB also listed below, resets the calculator. The fourth method is to allow the program to run to completion; see section 3.3.5 (page 22) for details.

Keys	Description
⇐	Exit.
DEL	Restore system registers to the state they were in when the debugger was invoked. This is useful if you need to exit in the middle of a program and the system registers contain random data. It does nothing if **ARG** was supplied, in effect acting as an **ARG** cancellation key.
1/x 1/x	Panic exit. Use if you know the calculator will crash when you exit, which will happen if the memory has been trashed. Reset the calculator; same as ON-C.

3.3.5 Program Stepping and Running

The program can either be single-stepped instruction by instruction, or allowed to *run free*, which means that the program will run without interference until it completes or until it encounters a breakpoint. For a discussion of breakpoints, see section 3.3.8 (page 30). Instructions can be single-stepped either *shallow* or *deep*. A shallow step is one where CALLs are stepped as a single instruction. A deep step follows CALLs into subroutines. Both shallow

and deep stepping can be given **ARG**s, in which case the **ARG** determines the number of instructions to be single-stepped. If multiple instructions are stepped, then they are susceptible to breakpoints. The table below lists the keys controlling program execution.

A single-step of more than one instruction is sensitive to the [ON] key. Pressing [ON] during a single-step of more than one instruction will halt execution. During execution, PICT is displayed unless it has been purged, a behavior which can be disabled (see section 3.3.12 (page 36)).

Keys	Description
[+]	Single-step one (or **ARG**) instruction(s), pointed to by the PC. Follow CALLs. If **ARG**, then also susceptible to breakpoints. Will display PICT until done, if more than one instruction is being single-stepped.
[−]	Same as [+], but do not follows CALLs to subroutines; instead, halt the program upon return from the subroutine. The entire subroutine is single-stepped as if it were a single instruction. Will display PICT until the subroutine returns, or until finished, if more than one instruction is being single-stepped.
[·]	Redraw the current screen but otherwise do nothing.
[EEX]	Display PICT until released.
[EVAL]	Continue execution, run until program completes, a breakpoint is reached, or a broken [−] CALL execution completes. Will display PICT while the program runs.

3.3.6 Example: Program Single-Stepping

This example illustrates some basics of single-stepping programs, and points out the difference between using [+] and [−]. It calculates tan 22.5 degrees.

The TAN PMC is at **2AC91**. It calls several ROM ML routines: **2AEA2** picks a real from the stack and returns it in registers A and B as a long real. A long real is basically a real, but it is 84 bits instead of 64, allowing higher precision. The ML routine at **2AEF6** returns the angle mode (RAD, DEG, or GRAD) in bits 0 and 1 of the ST register. **2B6F2** calculates the tangent of the long real in A and B. **2AEE5** truncates the long real to a real and puts it on the stack, which becomes the final result of the TAN PMC.

Put the angle argument on the stack and invoke MLDB on the TAN PMC (at location **2AC91**).

Keys

[STD] [DEG] [HEX]
22.5 [SPC]
#2AC91 [ENTER]
[LIBRARY] [MLDL] [MLDB]

Display

```
CALL.3 #2AEA2
 7802
@:2AC96 P:0 H ST:318
A:2AC91 C:73496
B:72A56 D:002FD HST:2
D0:7C1A5/E80007002DCC
D1:7394B/000000071BEC
RST:00000:00000:00000
```

Switch to the instruction screen.

[F]

```
@:2AC96 P:0 H ST:318
#CALL.3##2AEA2########
 CALL.3 #2AEF6
 CALL.4 #2B6F2
 JUMP.3 #2AEE5
 MOVE.A C,A
 ADD.A C,A
 MOVE.1 #7,P
```

Single-step through the routine at **2AEA2**.

```
@:2AEA2 P:0 H ST:318
#JUMP.A##29FD0########
 JUMP.A #29FF8
 JUMP.A #2A060
 JUMP.A #2A0B1
 JUMP.A #2B789
 JUMP.A #2A12A
 CALL.3 #2AEC5
```

The code at **2AEA2** is actually only a reference. It allows the use of CALLs with short offsets to addresses far away. This is a common practice when certain routines are heavily used in specific ROM areas. Single-step one more instruction.

```
@:29FD0 P:0 H ST:318
#CALL.3##29FDA########
 JUMP.4 #2BC4A
 MOVE.A @D1,C
 SWAP.A C,D1
 ADD.A #5,D1
 MOVE.W @D1,A
 MOVE.A C,D1
```

This routine first CALLs **29FDA**, and then JUMPs to **2BC4A**. The first routine pops a real off the stack, and the second converts it to a long real. Single-step again.

```
@:29FDA P:0 H ST:318
#MOVE.A#@D1,C#########
 SWAP.A C,D1
 ADD.A #5,D1
 MOVE.W @D1,A
 MOVE.A C,D1
 ADD.A #5,D1
 INC.A D
```

Single-step six more instructions.

06 [+]

```
@:29FED P:0 H ST:318
#INC.A#D##############
 SETDEC
 JUMP.A #679B
 CALL.3 #2A002
 JUMP.4 #2BCA0
 SWAP.A C,D0
 MOVE.5 #705B0,D0
```

Single-step three more instructions. The last instruction of the three will be the JUMP to **679B**.

03 [+]

```
@:0679B P:0 D ST:318
#SWAP.A#C,D0##########
 MOVE.5 #705B0,D0
 MOVE.A C,@D0
 MOVE.5 #70579,D0
 SWAP.A C,D1
 MOVE.A C,D1
 MOVE.A C@D0
```

The ML routine at **679B** saves the registers used by the system. It is fifteen instructions long. Single-step ten instructions.

0A [+]

```
@:067C4 P:0 D ST:318
#MOVE.5##7066E,D0
 MOVE.A D,C
 MOVE.A C,@D0
 RETCLRC
 MOVE.5 #7066E,D0
 MOVE.A @D0,C
 MOVE.A C,D
```

Execute four instructions. This causes us to return from the subroutine since the last instruction does a return.

04 [+]

```
@:29FD4 P:0 D ST:318
#JUMP.4##2BC4A########
 MOVE.A @D1,C
 SWAP.A C,D1
 ADD.A #5,D1
 MOVE.W @D1,A
 MOVE.A C,D1
 ADD.A #5,D1
```

Follow the jump.

[+]

```
@:2BC4A P:0 D ST:318
#SETDEC###############
 SWAP.M B,A
 MOVE.X B,A
 ADD.XS A,A
 CLR.A A
 BRCC #2BC5C
 DEC.A A
```

This is the routine that converts the real just popped from the stack into a long real. With our particular argument, 22.5, it will be 23 instructions long. Single-step 21 (**15** hex) instructions.

015 [+]

```
@:2BC9E P:0 CD ST:318
#RETCLRC##############
 MOVE.A C,D
 POP.A C
 CALL.3 #2BC4A
 PUSH.A C
 MOVE.A D,C
 SETDEC
```

We now have the long real in the A and B registers. The next instruction returns us to the main **TAN** PMC.

[+]

```
@:2AC9A P:0 D ST:318
#CALL.3##2AEF6########
 CALL.4 #2B6F2
 JUMP.3 #2AEE5
 MOVE.A C,A
 ADD.A C,A
 MOVE.1 #7,P
 NOT.A D
```

Following that single CALL took a lot of keystrokes. This time, wise from our experience, we step through the *entire* next two CALLs as if they were single instructions. The first ML routine decodes the angle mode into ST bits 0 and 1.

[−]

```
@:2A2C9A P:0 D ST:008
#CALL.3##2B6F2#########
 JUMP.3 #2AEE5
 MOVE.A C,A
 ADD.A C,A
 MOVE.1 #7,P
 NOT.A D
 SUB.A #6,D0
```

Now the angle mode is in ST bits 0 and 1. The next ML routine calculates the tangent of the long real in registers A and B. First switch to screen 2 to examine the registers.

[B]

```
CALL.4 #2B6F2
 8EE4A0
2:2AC9E P:6 CD ST:008
A:0000000000000001
B:0225000000000000
C:000000000000002FE
D:F000000000002FE
RST:00000:00000:00000
```

The long real is the argument 22.5. Calculate the actual tangent.

[−]

```
JUMP.3 #2AEE5
 6042
@:2ACA4 P:F D ST:088
A:0000000000099999
B:0414213562373092
C:0758098311271080
D:0093819582071411
RST:00000:00000:00000
```

The result is 0.414213572373092, which will be rounded to 12 digits; the ...092 tail will be truncated. Allow the TAN PMC to run to completion. Only the JUMP to the routine that rounds and pushes the result onto the stack is left for us to execute. Simply allow the program to run to completion.

[EVAL]

The stack now reads:

```
| 1:          .414213562373|
```

3.3.7 Example: Program Completion

Start with the stack display, in HEX mode, in the [LIBRARY] [MLDL] submenu. Only the relevant lines of the stack display are included in the example below. **3244** is the PMC DROP.

First we set display modes and put the two numbers 2 and 1 onto the stack, followed by the address of the DROP PMC.

Keys	Display
HEX STD	
2 SPC 1 #3244	
ENTER	

Invoke MLDB and switch to the instruction screen.

Keys	Display
MLDB F	@:03249 P:0 H ST:218 #ADD.A##5,D1########## INC.A D MOVE.A @D0,A ADD.A #5,D0 JUMP.A @A MOVE.A C,B MOVE.1 #3,P

The DROP PMC consists of only the first five instructions. Executing the JUMP @A instruction will cause the program to reach completion. Allow the program to complete without interference.

Keys
EVAL

Only the 2 is left on the stack.

3.3.8 Breakpoints

The MLDB maintains a breakpoint table of eight slots. Each slot consists of an address and a counter, both of which are displayed when the MTH key is pressed (see below). The address is 20 bits and the counter is 8 bits. Breakpoints only work in RAM, although they can be set in ROM and triggered by multiple-instruction single-steps.

When a breakpoint is triggered, its counter is incremented; if after the increment it is negative (**80–FF**), the program will continue. Setting a negative counter is therefore a way of instructing MLDB to "ignore this breakpoint n times," where n is in the range **01–7F** hex. If the counter is positive, the program is halted with a "Breakpoint Stop" or "Breakpoint Trap" message, depending on whether the breakpoint was triggered during a single-step or free run, respectively. Positive counters indicate how many times the program has halted at a specific breakpoint. The breakpoint table is accessible using the keys in the table below.

Keys	Description
[MTH]	Switch to breakpoint table screen. Any entry at the current PC will be displayed in reverse video.
[PRG]	Set breakpoint. Wait for a further key, 1–8, which specifies the breakpoint to set. **ARG** is the address the breakpoint is set to. If no **ARG** is entered, then the breakpoint is cleared (its address and counter are set to **00000** and **00**, respectively).
[STO]	Set breakpoint counter. Wait for a further key, 1–8, which specifies the breakpoint whose counter is to be set. **ARG** is the new counter value. The counter is cleared if no **ARG** is entered. Only the low eight bits of **ARG** are used.

3.3.9 Example: Breakpoints

For an illustration on how to enter **ARG**, see section 3.3.1 (page 17).

Set the PC to **59D1**.

Keys	Display
059D1 [ENTER]	CALL.4 #0679B 8E4CD0 @:059D1 P:0 CH ST:218 A:000CC C:77794 B:729A9 D:00F96 HST:2 D0:409C1/9540A8240BC9 D1:77799/000000000000 RST:00000:00000:00000

Switch to breakpoint screen.

[MTH]	1:00000 00 2:00000 00 3:00000 00 4:00000 00 5:00000 00 6:00000 00 7:00000 00 8:00000 00

Set breakpoint 1 at **59D1**.

059D1 [PRG] 1	#1:059D1##00########## 2:00000 00 3:00000 00 4:00000 00 5:00000 00 6:00000 00 7:00000 00 8:00000 00

The inverse bar in the display above indicates that the breakpoint is set at the current PC. Examine the subsequent seven instructions by switching to the instructions screen.

F

```
@:059D1 P:0 CH ST:218
#CALL.4##0679B########
 CALL.4 #06641
 MOVE.W A,R1
 CLR.A C
 MOVE.P1 #5,C
 CALL.3 #05B7D
 MOVE.W R1,C
```

Switch back to the breakpoint screen, and set breakpoint 5 at **6641**.

MTH
06641 PRG 5

```
#1:059D1##00##########
 2:00000  00
 3:00000  00
 4:00000  00
 5:06641  00
 6:00000  00
 7:00000  00
 8:00000  00
```

Single-step 64 instructions. This, like all other instances of **ARG**, is entered in hexadecimal, as **40**.

040 +

```
 1:059D1  00
 2:00000  00
 3:00000  00
 4:00000  00
#5:06641#+01##########
 6:00000  00
 7:00000  00
 8:00000  00
```

We never really got as far as 64 instructions. Instead we ran into the breakpoint at **6641** and stopped there, with a "`Breakpoint Stop`" message appearing briefly. The breakpoint table above tells us why we stopped, and that it is our first stop at this breakpoint. Setting breakpoints and single-stepping a large number of instructions is the only way to use breakpoints in ROM. Clear breakpoint 5 counter.

STO 5

```
 1:059D1  00
 2:00000  00
 3:00000  00
 4:00000  00
#5:06641##00##########
 6:00000  00
 7:00000  00
 8:00000  00
```

3.3.10 More About Breakpoints

A breakpoint is a `CALL.A` to an entry in the MLDB. When the breakpoint is reached, control is transferred to the breakpoint trap handler in the MLDB. It finds the location of the breakpoint on the return stack (RSTK), looks it up in the breakpoint table, and proceeds with the general breakpoint processing, described in section 3.3.8 (page 30). The `CALL.A` instruction occupies seven nybbles of memory. Therefore, problems can arise when the program contains a sequence like the one below:

```
point: brcc    foo     ; 3 nybbles
       inc.a   c       ; 2 nybbles
foo:   inc.a   c       ; 2 nybbles
       dec.a   a       ; 2 nybbles
```

Picture what would happen if a breakpoint is set at `point`. It occupies seven nybbles of memory, and thus overwrites the `BRCC`, subsequent `INC`, and the `INC` at `foo`. Assume another part of the program makes a jump to `foo`. This will result in a jump into the last two nybbles of the breakpoint instruction! The program is bound to behave erratically: if we're lucky this means mysterious results, if we're unlucky the calculator will crash.

During a + single-step of multiple instructions (actually, during any single-step with an **ARG**), the breakpoints are never inserted into the program. Instead, each consecutive PC is matched against the breakpoint table, and if a breakpoint is set at exactly that address, the single-stepping will stop.

This usage of breakpoints is entirely safe, and works under all conditions. It can, of course, be used for ROM as well as RAM programs. The drawback is execution speed, although the [−] stepping variant is usually faster than the [+] variant. Graphics and other CPU-heavy applications take seemingly forever to run. There is no simple solution; you must take the seven-nybble limit into consideration when writing such programs. Insert **NOP**s at places where you know you will want to insert breakpoints.

The [−] single-step does not follow **CALL**s. It executes the entire subroutine as if it were one single instruction. When it returns, control returns to MLDB, which then proceeds with the next instruction in sequence, the one following the **CALL**. Time-consuming functions like graphics routines can be placed in separately debugged and verified subroutines, and the main program calling these routines can be debugged with [−]. This will generally shorten stepping times.

During [−] single-step, when a subroutine that is allowed to run free returns or encounters a breakpoint, the program halts. The return address *into* MLDB will be on top of the return stack if it encountered a breakpoint. Pressing [EVAL] to continue at this point will cause the subroutine to continue running free, until it again reaches a breakpoint or returns, at which point it returns to MLDB and continues its [−] single-step. The subroutine can also be single-stepped when it has encountered a breakpoint, but *only* until it returns to the caller, at which point it should be allowed to return by pressing [EVAL].

3.3.11 Watchpoints

The MLDB keeps track of watchpoints in the watchpoint table, which consists of eight entries. Each entry can be set to any arbitrary address, which will appear as a memory dump line (see section 3.2.7 (page 14)) in the watchpoint table screen (screen 8, see section 3.2.10 (page 15)). By default, the addresses are set to **00000**. Watchpoints are useful for monitoring memory contents.

Keys	Description
VAR	If no **ARG**: switch to the watchpoint table screen. See section 3.2.10 (page 15) for a description of the table format.
VAR	If **ARG**: set watchpoint. Expects a further key, 1–8, which specifies the watchpoint to set. **ARG** is the address the watchpoint is set to. Unused watchpoints are by convention set to **00000**.

3.3.12 Options

There are three rather specialized "options" available, numbered 3, 4, and 5. Option 3 is used to switch to ASCII mode, which aids in debugging programs that do any kind of text processing. Since bytes take up two nybbles, the ordinary memory dump would not be useful for examining ASCII characters in memory should the ASCII characters happen to be stored at odd addresses, if the memory dump were always evenly aligned. Option 4 toggles the alignment between even which is the default, and odd. Option 5 toggles the automatic display of PICT during program execution.

Notice that since the ASCII mode affects **ARG** entry and all other integers displayed (instructions being the exception), it is only intended to be briefly toggled in and out of. The same applies to the memory dump alignment shift. All options are reset when the MLDB is initially invoked.

Keys	Description
CST 3	Toggle ASCII mode. All numerical data (except in instructions) will appear as seven-bit ASCII characters. Non-printable characters appear as dots ("."), and characters with the high bit set will appear in inverse.
CST 4	Toggle memory dump alignment. Affects only the memory dump; instead of the memory dump lines being evenly aligned, it will be oddly aligned, and vice versa (see section 3.2.7 (page 14)).
CST 5	Toggle automatic PICT display during program execution.

3.3.13 Example: Options

Set the PC to **400** and switch to the memory dump screen. The cursor is placed at the current PC.

Keys

0400 ENTER E

Display

```
003D0:5A08086300808421
003E0:5808084390800191
003F0:115228084B928008
00400:08F19C015E01E5D4
00410:015D08089115C07F
00420:A14C719D0D015A08
00430:0843D6C4CA347840
00440:0C213415231B8310
```

Switch to ASCII mode. Characters with the high bit set appear in inverse (not reproduced here). Notice that the field sizes are unchanged, which simplifies recognizing which characters correspond to which hex digits when toggling back and forth.

CST 3

```
.P    :%..6..H.
.‘    :...4....
.p    :.%..4)..
..    :..I.e.^M
..    :.U...Q.w
.     :.D.YP.%.
.O    :.4mL,C..
.@    :@.CQ218.
```

Toggle back to the “normal” screen.

CST 3

```
003D0:5A08086300808421
003E0:5808084390800191
003F0:115228084B928008
00400:08F19C015E01E5D4
00410:015D08089115C07F
00420:A14C719D0D015A08
00430:0843D6C4CA347840
00440:0C213415231B8310
```

Shift the dump one nybble.

CST 4

```
003D1:A080863008084215
003E1:8080843908001911
003F1:15228084B9280080
00401:8F19C015E01E5D40
00411:15D08089115C07FA
00421:14C719D0D015A080
00431:843D6C4CA3478400
00441:C213415231B83100
```

Switch to ASCII mode.

CST 3

```
.Q    :..h...$Q
.a    :..H.....
.q    :Q'.H....
..    :x..Q.aU.
..    :Q....Ep/
.!    :A|...Q..
.1    :HSFD:th.
.A    :,1.%....
```

Toggle out of both modes.

CST 4
CST 3

```
003D0:5A08086300808421
003E0:5808084390800191
003F0:115228084B928008
00400:08F19C015E01E5D4
00410:015D08089115C07F
00420:A14C719D0D015A08
00430:0843D6C4CA347840
00440:0C213415231B8310
```

3.4 MLDB Server Modes

User flags 32 and 33 control the MLDB mode:

		Server	
Flag	Local	Interactive	Protocol
32	Clear	Set	Set
33	Clear	Clear	Set

The normal mode of operation, described in the previous sections, is local. The other two modes are refered to as server modes, and are the interactive

mode, in which commands (see Server Mode Commands, below) are entered on a dumb ASCII terminal or emulator with full editing (see Server Command Entry, below), and protocol mode, in which the same commands are accepted as in interactive mode, except no prompts are printed and commands are not echoed when received. The protocol mode is intended for communication with software on the development system.

You must set up I/O in the HP 48 before invoking the MLDB in a server mode[7]. The MLDB server modes always communicate over the wire, effectively ignoring the state of system flag -33 (I/O device).

3.4.1 Server Mode Commands

Commands can be entered in response to the "*" prompt in interactive mode. Commands can be sent at any time in protocol mode. Generally, excessive input is ignored, as are unrecognized commands or commands with invalid arguments. Command lines of up to 80 characters can be entered. Below is a list of the 18 recognized commands. The command name is in **bold face** and optional arguments are enclosed in brackets. Don't type the brackets when entering the commands, they are used in the table for clarity.

[7]See HP 48 *Owner's Manual, Vol II*, pages 617–619 for details on how to set up the I/O.

Command	Description
=*addr*	Set the PC to *addr*. *addr* is in the range **00000-FFFFF**.
+*offs*	Add *offs* to PC. *offs* is in the range **00000-FFFFF**.
-*offs*	Subtract *offs* from PC. *offs* is in the range **00000-FFFFF**.
n [*n*]	Advance PC forward *n* instructions. If no *n* is supplied, the PC is advanced one instruction.
s [*n*]	Single-step *n* instructions. Same as the [+] key in local mode. If no *n* is entered, one instruction is stepped. A single-step of more than one instruction can be interrupted by pressing any key, which is ignored.
S [*n*]	Single-step *n* instrutions, but don't follow CALLs. Same as the [−] key in local mode. If no *n* is entered, one instruction or CALL is stepped. A single-step of more than one instruction can be interrupted by pressing any key, which is ignored.
c	Continue free-run execution until the program completes or a breakpoint is encountered. When the program completes, "Exit" is printed and MLDB exits.
t	Terminate. Exit with current registers.

(Continued from previous page)

Command	Description
T	Terminate. Exit with system registers set up exactly as they were when MLDB was invoked.
R	Reset.
i [*n*] [*addr*]	Print instructions. *n* instructions are printed, starting at *addr*. The first argument is always *n* and the second is *addr*. If no *addr* is entered, the current PC is assumed. If *n* is not entered, one (the next) instruction is printed. *n* and *addr* are both in the range **00000-FFFFF**.
x [*n*] [*addr*]	Print memory contents. *n* words of 16 nybbles are printed, starting at *addr*, each on a separate line. The first argument is always *n* and the second is *addr*. If no *addr* is entered, the current PC is assumed. If *n* isn't entered, one (the next) word is printed. *n* and *addr* are both in the range **00000-FFFFF**.
a [*n*] [*addr*]	Print memory contents in ASCII. *n* words of 32 nybbles are printed as ASCII characters, starting at *addr*, each consecutive word on a separate line. If no *addr* is entered, the current PC is assumed. If *n* is absent, one (the next) word is listed. *n* and *addr* are both in the range **00000-FFFFF**.

(Continued from previous page)

Command	Description
r	Print registers. The HEX/DEC mode is printed as `HD:0` or `HD:1`. A `0` indicates that HEX mode is active, a `1` means that DEC mode is active.
z	Print the return stack (the RSTK).
db*n* [*addr*]	Set breakpoint *n* at *addr*. If *addr* is absent, the breakpoint is cleared. *addr* is in the range **00000**– **FFFFF**. *n* must be in the range 1–8.
hb*n* [*cntr*]	Set breakpoint *n* counter to *cntr*. If *cntr* is absent, the counter is set to **00**. *cntr* is in the range **00**–**FF**. *n* must be in the range 1–8.
lb	List breakpoints.

3.4.2 MLDB Server Modes Command Entry

In both interactive and protocol mode, input can be edited, although no echo or response can be detected in protocol mode. The following table may be of help when you try to locate the editing keys on your keyboard. After 80 characters have been typed, any further entry, except the editing keys listed below, is ignored.

Key(s)	Description
Backspace, Delete, or Rubout	Erase the last character entered.
Control-W	Erase the last word entered.
Control-U or Control-X	Erase the entire line.
Control-R	Rewrite input.
Return, Enter, or Control-M	Execute command entered.

3.4.3 Example: An MLDB Interactive Mode Session

Invoke the MLDB with **3223**, the address of the **SWAP** PMC. Pass two arguments to **SWAP** – 1 and 2.

1 [SPC] 2 [SPC] #3223h [MLDB]

First, we are greeted with a header[8].

```
MLDL 1.04B
Copyright (c) 1991 Jan Brittenson
```

[8]In reality, output will rarely resemble the perfection in this example. Specifically, the number of instructions to be listed was known in advance.

Examine the **SWAP** PMC.

```
* i 9
MOVE.A @D1,C
ADD.A #5,D1
MOVE.A @D1,A
MOVE.A C,@D1
SUB.A #5,D1
MOVE.A A,@D1
MOVE.A @D0,A
ADD.A #5,D0
JUMP.A @A
```

Examine the SATURN registers.

```
* r
CY:0
P:0
PC:03228
A:0000000644403223
B:0960000000074FB3
C:0000000000075FBB
D:0000000000000335
R0:000000000007BCA5
R1:0000000644403228
R2:00000000000505C6
R3:0000000644400001
R4:00015074EE274F20
D0:7C1A5
D1:75FC0
ST:000
HST:2
HD:0
```

Stack level 1 is at **75FC0**. Examine level 1.

```
* x 1 75fc0
75FC0:ED2A29C2A2000000
```

Which is object **2A2DE**. Examine this object.

```
* x 2 2a2de
2A2DE:3392000000000000
2A2EE:0002033920000000
```

Which is a real (type prefix **2933**), the constant 2. Step a few instructions.

```
* i
MOVE.A @D1,C
* s
* i
ADD.A #5,D1
* s
* i
MOVE.A @D1,A
* s
* i
MOVE.A C,@D1
* s
* i
SUB.A #5,D1
* s
* i
MOVE.A A,@D1
* s
```

Let's see what is left of the PMC routine.

```
* i 3
MOVE.A @D0,A
ADD.A #5,D0
JUMP.A @A
```

Finally, we step a large chunk of instructions. This way we can be sure that the program completes.

```
* s 100
Exit
```

We are done. The HP 48 stack now reads:

```
2:                    2
1:                    1
```

3.5 Messages

Stopped

Appears for about a second at the top of the screen. Indicates that a single-step of multiple instructions was interrupted by pressing the ON key.

Breakpoint Stop

Appears for about a second at the top of the screen. Indicates that a single-step of multiple instructions encountered a breakpoint.

Breakpoint Trap

Appears for about a second at the top of the screen. Indicates that the program encountered a breakpoint during free run. This can occur either during a **CALL** executed with the [-] key or a free run initiated with the [EVAL] key.

Fatal Error:
Data Lost

Indicates that the **MLDLpar** variable (see section 3.2.2 (page 11)) was corrupted or purged, either directly or indirectly, by the program being debugged. No recovery is possible. Press any key to reset (same as [ON]-[C]).

Fatal Error:
ROM Card Failure

Indicates that the MLDL ROM card is either broken or a card other than the original is being used. No recovery is possible. Press any key to reset.

Fatal Error:
RAM Card Failure

Can only occur with the non-commercial RAM version of the MLDL(see section 1 (page 6)). Indicates that the RAM it is stored in is either not working properly or is write-protected. No recovery is possible. Press any key to reset.

3.6 Some MLDB System Considerations

The debugger (MLDB) has been designed specifically so that it will not alter any static system data or depend on the precise machine configuration.[9] The only system data it modifies is the keyboard buffer, since it relies on the system to respond to the keyboard interrupt and manage the buffer. Testing has shown that interfering with this will result in poor reliability. There are three instructions the debugger will refuse to single-step:

`RESET`	The effect of executing this instruction would be the same as pressing [ON]-[C] .
`CLRB #F,ST`	This instruction would lock up the calculator since it would disable all I/O interrupts, most notably the keyboard.
`INTOFF`	The effect would be similar to that of `CLRB #F,ST`.

Apart from the aspects outlined above and some system RPL code to do argument type checking and initial setup, the debugger is self-contained.

3.6.1 A Word of Caution

The [−] key lets you complete an entire `CALL`. But beware: the return stack is replaced by one that will cause the called routine to return to the debugger. Therefore, the routine called cannot rely on specific return stack contents or remove return addresses from the stack, either of which would invariably result in a system crash. One example of a ROM routine that actually does this is **CD8E**, which jumps to a location in the bank-switched ROM usually hidden behind user RAM. Most of the trig and log functions are actually located in this hidden ROM. Despite the effort put into avoiding system collisions, the HP 48 still remains a largely unprotected system.

Single-stepping a machine code program is in no way less dangerous than allowing it to run uncontrolled. It merely gives you some control over what

[9] Assuming that the automatic displaying of `PICT` has been disabled.

happens between instructions. It can even be more dangerous since the hardware may break if left in certain configurations for longer periods of time. If you single-step parts of the system ROM, you should be aware of this risk, although the author at this time has never actually heard of this occuring[10].

3.7 MLPR

Print disassembly of ML program. Accepts the same arguments as MLDB, except for @#**3A81** and @#**3AC0**, which are not recognized. The program is printed on the current print device: IR or wire.[11]. Each line printed consists of a mnemonic preceded by its address; no opcode is included, since it is usually of low interest. Use ML1 (described below) to build your own custom disassembler.

3.8 ML1

Disassemble one instruction. Allows you to build your own disassembler with its own special-purpose user interface. It takes a binary integer in level 1, and returns two values: in level 2 the mnemonic form preceded by the address, and in level 1 the address of the next instruction. Thus it is a simple task to make a number of consecutive calls to ML1. MLOPC can be used to extract the opcode as a string of hexadecimal digits. Extracting the mnemonic from the string is reasonably trivial, since it will always be of the form *xxxx*: *m* where *xxxx* is a five-digit address followed by a colon and a blank. The last part of the string, *m*, is the mnemonic.

---

[10] Under no circumstances will Jan Brittenson or the distributors of the MLDL accept any responsibility or liability for such damage, regardless of nature and extent. See page 2 for further disclaimers.

[11] See pages 602-611 in the HP 48 *Owner's Manual Volume II.* Page 611 explicitly describes the **PRTPAR** variable.

3.8.1 Example: ML1

Type:

HEX #59D1 ML1

The stack now reads:

```
2:"059D1:  CALL.4 #06...
1:                # 59D7h
```

3.9 MLOPC

Return opcode as a string of hexadecimal digits. It expects two binary integers: the starting address in level 2 and the final address plus one in level 1. It is useful for creating the opcode field in custom disassemblers. The opcode string returned by MLOPC can be up to 255 digits.

3.9.1 Example: MLOPC

Type:

HEX #59D1 #59D7 MLOPC

The stack now reads:

```
1:              "8E4CD0"
```

3.9.2 Example: MLOPC and ML1

This is a somewhat more extensive example of how to use ML1 and MLOPC. It displays a disassembly in the smallest text size by using →GROB to build screenfuls in PICT. When a key is pressed, the next screenful of ML is displayed.

The checksum for this program is **CAE4** hex.

```
≪ { # 0h # 0h } PVIEW                    display PICT
  #83h #6h BLANK → B                     create blank line GROB
  ≪ DO                                   loop screenfuls
    2 59 FOR L                           loop pixel rows
      DUP ML1                            get mnemonic
      ROT OVER MLOPC                     get opcode
      ROT DUP 1 6 SUB                    extract address from mnemonic
      ROT DUP SIZE                       opcode, #of hex digits
      "          "                       ten blanks
      10 SUB +                           pad opcode to 10 characters
      +                                  add padded opcode to address
      SWAP 7 63 SUB                      extract instruction mnemonic
      +                                  add to address-opcode
      1 →GROB                            convert to GROB
      PICT # 0h L R→B                    PICT, 0, and pixel row
      2 →LIST                            as pixel address of text line
      DUP2 B REPL                        clear text line
      ROT REPL                           fill in with line GROB
    6 STEP                               advance L to next text line
    UNTIL 0 WAIT                         main loop: read key
      55.1 ==                            loop until ⇐
    END                                  end of main loop
  ≫
≫
```

4 The SATURN Processor

4.1 Registers

The HP 48 (SATURN) CPU uses 64-bit arithmetic registers and 20-bit address registers. The unit of addressability is a four bit nybble, hence the address space of the processor is 2^{20} nybbles, or half a megabyte.

Most operations on arithmetic registers can be restricted to particlar ranges of nybbles, called fields. These fields have been chosen to optimize the decimal (BCD) arithmetic of the calculator. The format for decimal numbers is a 1 nybble sign, followed by a 12 nybble mantissa and a 3 nybble exponent, making 16 nybbles or 64 bits in all.

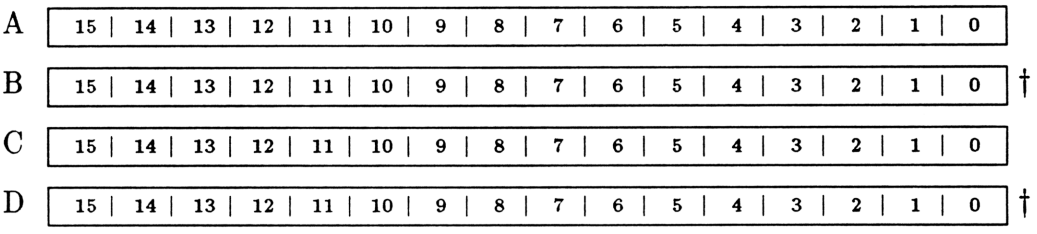
The four arithmetic registers are called A, B, C, and D. The instruction set does not allow these registers to be used interchangeably. For example, registers A and B never interact with register D. Memory operations are restricted to arithmetic registers A and C, with the bulk of the responsibility on register C.

There is a 4-bit pointer register, called P, that is used to specify the position of a one nybble field (.P) or the length of a multi-nybble field (.WP). The two 20-bit address registers are called D0 and D1, and can be used interchangeably.

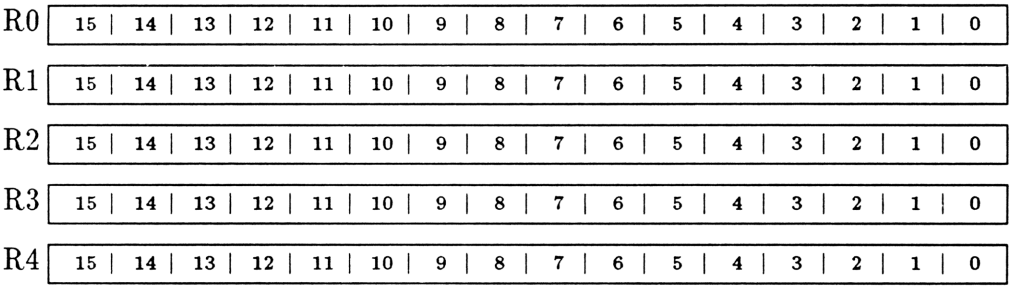
There are five 64-bit temporary registers called R0, R1, R2, R3, and R4. The operations they support are restricted to moving and swapping with registers A and C.

Below is an outline of all the registers in the CPU. Registers that are used by the system are marked with a dagger: †. The system uses only the A field (least significant 20 bits) of these registers, which must be restored after use. Since RSTK is used by interrupts, you should never `PUSH` or `CALL` more than seven addresses onto the return stack.

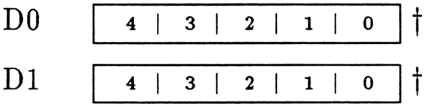
Arithmetic registers:



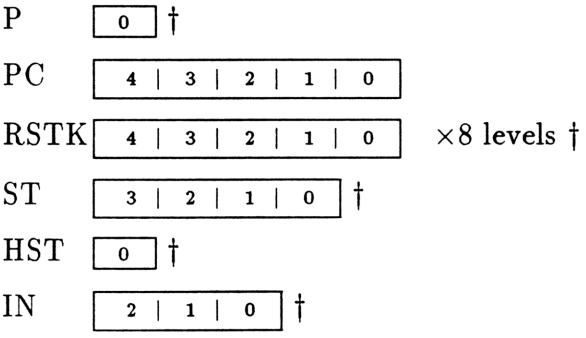
Temporary registers:



Address registers:



Control and status registers:



OUT | 2 | 1 | 0 | †

Carry | 0 |

4.2 System Register Usage

The B and D registers are used by the system. The least significant 5 nybbles of B points to the top of the RPL return stack, the least signficant 5 nybbles of D is the free word counter, the size of the free area, which is (B.A−D1.A)/5. When it reaches zero, a garbage collect is performed; if it still remains zero after the garbage collect, then the memory is full.

4.3 Instruction Fields

Each 64 bit register comprises 16 nybbles that can be grouped into fields for calculation and data movement. These nybbles are numbered from right to left starting at 0; nybble 0 is the low-order or least significant nybble and nybble 15 is the high-order or most significant nybble.

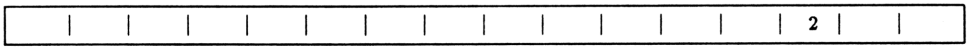
Suffix: `.P` Name: **Pointer Field**
Start: **nybble P** Size: **1 nybble**
Example: `RETZ.P B`

								P							

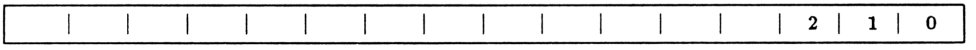
Suffix: `.WP` Name: **Word to Pointer Field**
Start: **nybble 0** Size: **P+1 nybbles**
Example: `OR.WP C,D`

									P	P-1	P-2	...	...	1	0

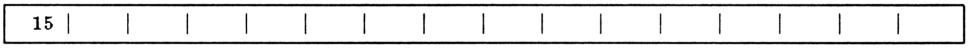
Suffix: .XS Name: **Exponent Sign Field**
Start: **nybble 2** Size: **1 nybble**
Example: `NOT.XS C`



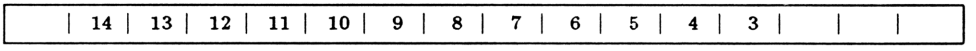
Suffix: .X Name: **Exponent Field**
Start: **nybble 0** Size: **3 nybbles**
Example: `SUB.X A,C`



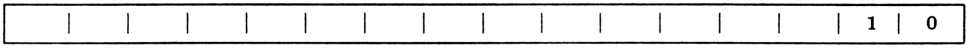
Suffix: .S Name: **Sign Field**
Start: **nybble 15** Size: **1 nybble**
Example: `CLR.S B`



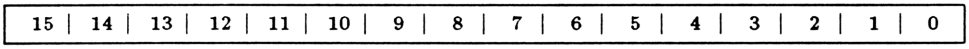
Suffix: .M Name: **Mantissa Field**
Start: **nybble 3** Size: **12 nybbles**
Example: `MOVE.M B,C`



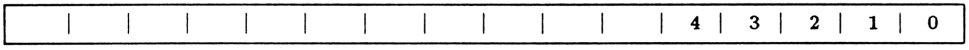
Suffix: .B Name: **Byte Field**
Start: **nybble 0** Size: **2 nybbles**
Example: `INC.B C`



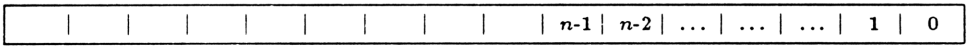
Suffix: .W Name: **Word field**
Start: **nybble 0** Size: **16 nybbles**
Example: `SWAP.W A,R2`



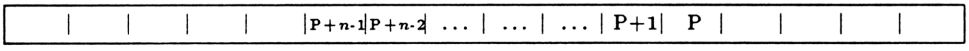
Suffix: .A Name: **Address Field**
Start: **nybble 0** Size: **5 nybbles**
Example: `ADD.A 5,D1`



Suffix: .*n* Name: *n* **Nybble Field**
Start: **nybble 0** Size: *n* **nybbles**
Example: `MOVE.8 @D0,A`



Suffix: .P*n* Name: **Pointer Load Field**
Start: **nybble P** Size: *n* **nybbles**
Example: `MOVE.P2 25,C`



4.4 Instruction Set Description

Each entry in this list consists of one or more mnemonics, a brief description, opcode, cycle, and carry information. Some entries have opcode construction tables, and most have some sort of detailed description. Some entries containing several mnemonics with different opcodes have the opcodes listed to the left of the corresponding mnemonics.

The cycle data is not always available, in which case it will be replaced by "**???**" instead. For some instructions, the number of cycles varies with

opcode or instruction field, in which case the numbers are paired with the corresponding field parenthesized. The last number may be without a field pair, in which case it applies to all field sizes not encompassed by the previous numbers. An **n** stands for the number of nybbles represented by the field used in the instruction. For field A, for example, **n**=5.

For instance, "**7(A), 3+n**" means that for field A the instruction requires 7 cycles, for any other field it requires 3+n. For an M field, this means 3+12 cycles.

The opcode construction tables are best read right-to-left. Look up the register, field, or combination of interest on the right-hand side, and note down the corresponding code to the left. Substitute it in the opcode descriptor listed for the instruction.

ADD.*f* *s*, *d*

Add register to register

Opcode: **kkx**
Cycles: **7(A), 3+n**

Carry:
Set on overflow

kk	*f*
A0	P
A1	WP
A2	XS
A3	X
A4	S
A5	M
A6	B
A7	W
C	A

x	*s*	*d*
0	B	A
1	C	B
2	A	C
3	C	D
4	A	A
5	B	B
6	C	C
7	D	D
8	A	B
9	B	C
A	C	A
B	D	C

Add contents of field *f* of register *s* to contents of field *f* of register *d*.

ADD.*f* *i*+1, *d*

Add constant to register

Opcode: **818txi**
Cycles: **???**

Carry:
Set on overflow

t	*f*
0	P
1	WP
2	XS
3	X
4	S
5	M
6	B
7	W
F	A

x	*d*
0	A
1	B
2	C
3	D

Add constant *i*+1 to field *f* of register *d*.

ADD.A *x*+1, D0
ADD.A *x*+1, D1

Add constant to address register

Opcode: **16x (D0), 17x (D1)**
Cycles: **7**

Carry:
Set on overflow

Add constant *x*+1 (range 1-16) to address register D0 or D1.

ADD.A P+1, C

Add P pointer plus one to C.

Opcode: **809**
Cycles: **8**

Carry:
Set on overflow.

Add the value of register P plus one to field A of register C.

AND.*f* *s*, *d*

And register to register

Opcode: **0Etx**
Cycles: **4+n**

Carry:
Not affected

t	*f*
0	P
1	WP
2	XS
3	X
4	S
5	M
6	B
7	W
F	A

x	*s*	*d*
0	B	A
1	C	B
2	A	C
3	C	D
4	A	B
5	B	C
6	C	A
7	D	C

Logical AND the value of field *f* of register *s*with field *f* of register *d*, placing the result in field *f* of register *d*.

83zyy	**BRBC**	***hf*, HST, PC+3+*yy***
8086xyy	**BRBC**	***x*, A, PC+5+*yy***
808Axyy	**BRBC**	***x*, C, PC+5+*yy***
86xyy	**BRBC**	***x*, ST, PC+3+*yy***
8087xyy	**BRBS**	***x*, A, PC+5+*yy***
808Bxyy	**BRBS**	***x*, C, PC+5+*yy***
87xyy	**BRBS**	***x*, ST, PC+3+*yy***

Branch or return if bit set/clear

Opcode: **see above**
Cycles: **see below**

Carry:
Set if taken, else cleared

z	*hf*
1	XM
2	SB
4	SR
8	MP

A/C: test if specified bit *x* (0–15) is cleared/set; return (RETBC and RETBS) or branch if true [no cycle data]. ST: test if specified bit *x* (0-15) is cleared/set; return (RETBC and RETBS) or branch if true [cycles: 14 taken, 7 otherwise]. HST: test if specified hardware status (HST) bit *z* mask (see bit definitions above) is clear; return (RETBC) or branch if true [cycles: 13 taken, 6 otherwise]. All these instructions branch to offset *yy* if the condition is true, or if the encoded offset is 00, a return is made.

5xx **BRCC** **PC+1+*xx***
4xx **BRCS** **PC+1+*xx***

Branch or return if carry set/clear

Opcode: **see above**
Cycles: **10 taken, 3 otherwise**

Carry:
Not affected

Test the carry status, and branch (BRCS/BRCC) to offset *xx* if condition is true, or return (RETCS/RETCC) if the encoded offset is 00.

89xyy **BREQ.1** **P, *x*, PC+3+*yy***
88xyy **BRNE.1** **P, *x*, PC+3+*yy***

Branch or return on pointer value

Opcode: **see above**
Cycles: **13 taken, 6 otherwise**

Carry:
Set if taken, else cleared

Compare register P to a constant *x*, and branch (BREQ/BRNE) or return RETEQ/RETNE if they are equal/unequal.

ttuyy	**BREQ.***f*	*s*, *d*, **PC+3+***yy*
ttuyy	**BRNE.***f*	*s*, *d*, **PC+3+***yy*
ttwyy	**BRZ.***f*	*s*, **PC+3+***yy*
ttxyy	**BRNZ.***f*	*s*, **PC+3+***yy*
zzuyy	**BRGT.***f*	*s*, *d*, **PC+3+***yy*
zzvyy	**BRLT.***f*	*s*, *d*, **PC+3+***yy*
zzwyy	**BRGE.***f*	*s*, *d*, **PC+3+***yy*
zzxyy	**BRLE.***f*	*s*, *d*, **PC+3+***yy*

Branch or return on arithmetic relation

Opcode: **see above**
Cycles: **13+n taken, 6+n otherwise**

Carry:
Set if taken, else cleared

tt	*zz*	*f*
8A	8B	A
90	98	P
91	99	WP
92	9A	XS
93	9B	X
94	9C	S
95	9D	M
96	9E	B
97	9F	W

u	*v*	*w*	*x*	*s*	*d*
0	4	8	C	A	B
1	5	9	D	B	C
2	6	A	E	C	A
3	7	B	F	D	C

Compare field *f* of register *s* to either field *f* or register *d* or a constant 0 (BRZ/BRNZ), and branch/return if the comparison is true. Conditions: EQ – equal, NE – not equal, Z – zero, NZ – nonzero, GT – greater than, LT – less than, GE – greater than or equal, LE – less than or equal.

BUSCB

Bus command "B"

Opcode: **8083**
Cycles: **???**

Carry:
Not affected

Issue bus command "B" on the system bus; not used in the HP 48.

BUSCC

Bus command "C"

Opcode: **80B**
Cycles: **6**

Carry:
Not affected

Issue bus command "C" on the system bus; not used in the HP 48.

BUSCD

Bus command "D"

Opcode: **808D**
Cycles: **???**

Carry:
Not affected

Issue bus command "D" on the system bus; not used in the HP 48.

CONFIG

Configurate device

Opcode: **805**
Cycles: **11**

Carry:
Not affected

Copy field A of the C register into the configuration register of the chip which has its daisy-in line high and its configuration flag low.

7xxx **CALL.3** **PC+4+*xxx***
8Exxxx **CALL.4** **PC+6+*xxxx***

Call subroutine relative

Opcode: **see above**
Cycles: **12(3), 15(4)**

Carry:
Not affected

Call subroutine; push the address of the next instruction onto the return stack (RSTK) and jump. Only 7 levels of stack are available, 8 if interrupts are disabled. Returning from the subroutine can be done by using any of the various return instructions; any branch instruction with a zero offset (e.g., **9FA00** for "`RETGE.W C,A`") becomes a return instruction. The relative `CALL`s are used by user programs for calling subroutines within the same program; absolute `CALL`s are used for calling system ROM subroutines.

CALL.A *xxxx*

Call subroutine absolute

Opcode: **8Fxxxxx**
Cycles: **15**

Carry:
Not affected

Call subroutine; push the address of the next instruction onto the return stack (RSTK) and jump. Only 7 levels if stack are available, 8 if interrupts are disabled. Returning from the subroutine can be done by using any of the various RET instructions; any branch instruction with a zero offset (e.g., **9FA00** for "`RETGE.W C,A`") becomes a return instruction. The absolute `CALL`s are used by user programs for calling subroutines in the system ROM; relative `CALL`s are used for calling user subroutines.

CLR.*f* *d*

Clear register

Opcode: **kkx**
Cycles: **7(A), 3+n**

Carry:
Not affected

kk	*f*
A8	P
A9	WP
AA	XS
AB	X
AC	S
AD	M
AE	B
AF	W
D	A

x	*d*
0	A
1	B
2	C
3	D

Set field *f* of register *d* to zero.

8084x **CLRB** *x*, **A**
8088x **CLRB** *x*, **C**
84x **CLRB** *x*, **ST**

Clear bit

Opcode: **see above**
Cycles: **4(ST), ???(A/C)**

Carry:
Not affected

Clear bit *x* of register C, A, or ST.

CLRB *hf*, HST

Clear hardware status bits

Opcode: **82z**
Cycles: **3**

Carry:
Not affected

z	*hf*
1	XM
2	SB
4	SR
8	MP

Clear the HST bits corresponding to the mask *hf*.

CLR.X ST

Clear ST

Opcode: **08**
Cycles: **???**

Carry:
Not affected

Clear low 12 bits of the status register (ST).

DEC.1 P

Decrement pointer register

Opcode: **0D**
Cycles: **3**

Carry:
Set on underflow, else cleared

Decrement value of the P register by 1.

DEC.*f* *d*

Decrement register

Opcode: **kkw**
Cycles: **7(A), 3+n**

Carry:
Set on underflow, else cleared

kk	*f*
A0	P
A1	WP
A2	XS
A3	X
A4	S
A5	M
A6	B
A7	W
C	A

w	*d*
C	A
D	B
E	C
F	D

Decrement field *f* of register *d* by 1.

---

802 **IN.4** **A**
803 **IN.4** **C**

IN register

Opcode: **see above**
Cycles: **7**

Carry:
Not affected

Copy contents of input register to low 4 nybbles of A or C.

INC.1 P

Increment pointer register

Opcode: **0C**
Cycles: **3**

Carry:
Not affected

Increment contents of the P register by 1.

INC.*f* *d*

Increment register

Opcode: **qqu**
Cycles: **7(A), 3+n**

Carry:
Not affected

qq	*f*
B0	P
B1	WP
B2	XS
B3	X
B4	S
B5	M
B6	B
B7	W
E	A

u	*d*
4	A
5	B
6	C
7	D

Increment contents of field *f* of register *d* by 1.

INTOFF

Disable interrupts

Opcode: **808F**
Cycles: **5**

Carry:
Not affected

Disable keyboard interrupts.

INTON

Enable interrupts

Opcode: **8080**
Cycles: **5**

Carry:
Not affected

Enable keyboard interrupts.

6xxx	**JUMP.3**	**PC+1+*xxx***
8Cxxxx	**JUMP.4**	**PC+2+*xxxx***

Jump relative

Opcode: **see above**
Cycles: **11(3), 14(4)**

Carry:
Not affected

Jump relative; set program counter (PC) to destination address. `CALL.3` and `CALL.4` are used to jump within user programs.

JUMP.A *xxxx*

Jump absolute

Opcode: **8Dxxxxx**
Cycles: **14**

Carry:
Not affected

Jump absolute; set program counter (PC) to destination address. `CALL.A` is used to jump to system ROM locations.

JUMP.A @A

Jump register A indirect

Opcode: **808C**
Cycles: **23**

Carry:
Not affected

Jump to destination whose address is held in the location pointed to by register A.

JUMP.A @C

Jump register C indirect

Opcode: **808E**
Cycles: **23**

Carry:
Not affected

Jump to destination whose address is held in the location pointed to by register C.

JUMP.A A

Jump register A direct

Opcode: **81B2**
Cycles: **???**

Carry:
Not affected

Jump to destination whose address is held in field A of register A.

JUMP.A C

Jump register C direct

Opcode: **81B3**
Cycles: **???**

Carry:
Not affected

Jump to destination whose address is held in field A of register C.

MOVE.A ID, C

Get ID of current chip

Opcode: **806**
Cycles: **11**

Carry:
Not affected

The chip which has its daisy-in line high and its configuration flag low will send its 5-nybble ID register to the system bus which will be loaded into the low-order 5 nybbles (A field) of the C register.

MOVE.*f* *s*, *d*

Move register to register

Opcode: **kkz**
Cycles: **7(A), 3+n**

Carry:
Not affected

kk	*f*
A8	P
A9	WP
AA	XS
AB	X
AC	S
AD	M
AE	B
AF	W
D	A

z	*s*	*d*
4	B	A
5	C	B
6	A	C
7	C	D
8	A	B
9	B	C
A	C	A
B	D	C

Move field *f* of register *s* to field *f* of register *d*.

14x	**MOVE.A**	*s*, *d*
14y	**MOVE.B**	*s*, *d*
15xt	**MOVE.*f***	*s*, *d*
15yi	**MOVE.*n***	*s*, *d*

Move memory to register
Move register to memory

Opcode: **see above** (*i*=*n*−**1**)
Cycles: **see below**

Carry:
Not affected

x	*y*	*s*	*d*
0	8	A	@D0
1	9	A	@D1
2	A	@D0	A
3	B	@D1	A
4	C	C	@D0
5	D	C	@D1
6	E	@D0	C
7	F	@D1	C

t	*f*
0	P
1	WP
2	XS
3	X
4	S
5	M
6	B
7	W
F	A

Move from memory to register A or C; or move from register C or A to memory. D0 or D1 holds the address where the data should be read from or written to.

Cycles for read: 18(A), 15(B), 17+n(*f*), 16+n(*n*).
Cycles for write: 17(A), 14(B), 16+n(*f*), 15+n(*n*).

13w **MOVE.A** *s*, *Dn*
13y **MOVE.4** *s*, *Dn*

Move register to address register

Opcode: **see above**
Cycles: **8(A), 7(4)**

Carry:
Not affected

w	*y*	*s*	*Dn*
0	8	A	D0
1	9	A	D1
4	C	C	D0
5	D	C	D1

Move low 5 or 4 nybbles from register A or C to register D0 or D1. The 4 nybble move leaves the fifth nybble intact.

10j **MOVE.W** **A**, *Rn*
10k **MOVE.W** **C**, *Rn*
11j **MOVE.W** *Rn*, **A**
11k **MOVE.W** *Rn*, **C**
81At0j **MOVE.***f* **A**, *Rn*
81At0k **MOVE.***f* **C**, *Rn*
81At1j **MOVE.***f* *Rn*, **A**
81At1k **MOVE.***f* *Rn*, **C**

Move register to temporary register
Move temporary register to register

Opcode: **see above**
Cycles: **19(W), ???(*f*)**

Carry:
Not affected

t	*f*
0	P
1	WP
2	XS
3	X
4	S
5	M
6	B
7	W
F	A

j	*k*	*Rn*
0	8	R0
1	9	R1
2	A	R2
3	B	R3
4	C	R4

Move from temporary register *Rn* to register A or C, or move from register C or A to temporary register *Rn*.

81B4 **MOVE.A PC, A**
81B5 **MOVE.A PC, C**

Move program counter to register

Opcode: **see above**
Cycles: **???**

Carry:
Not affected

Set field A of register A or C to the current program address. The address if that of this instruction.

3ix...x **MOVE.P*n*** ***x...x*, C**
8082ix...x **MOVE.P*n*** ***x...x*, A**

Move constant to register

Opcode: **see above** (*i*=*n*−**1**)
Cycles: **4**+*i*

Carry:
Not affected

Set n-nybble field of register C or A to constant $x...x$. The n-nybble field starts at the nybble pointed to by register P, and continues n nybbles, wrapping from the most significant digit to the least significant, if necessary.

19xx	**MOVE.2**	***xx*, D0**
1Axxxx	**MOVE.4**	***xxxx*, D0**
1Bxxxxx	**MOVE.5**	***xxxxx*, D0**
1Dxx	**MOVE.2**	***xx*, D1**
1Exxx	**MOVE.4**	***xxxx*, D1**
1Fxxxxx	**MOVE.5**	***xxxxx*, D1**

Move constant to address register

Opcode: **see above**
Cycles: **4(2), 6(4), 7(5)**

Carry:
Not affected

Set the low 2, 4, or 5 nybbles of address register D0 or D1 to a constant. The remaining nybbles, if any, are unaffected.

MOVE.1 *x*, P

Move constant to pointer register

Opcode: **2x**
Cycles: **2**

Carry:
Not affected

Set pointer register P to constant x(range 0–15).

80Cx **MOVE.1 P, C.x**
80Dx **MOVE.1 C.x, P**

Move pointer register to register nybble
Move register nybble to pointer register

Opcode: **see above**
Cycles: **6**

Carry:
Not affected

Set nybble x of register C to the value of the pointer register. Set pointer register to nybble x of register C. The least significant nybble is numbered 0; the most significant is numbered 15.

MOVE.X ST, C

Move status register to register

Opcode: **09**
Cycles: **6**

Carry:
Not affected

Set field X (low 3 nybbles) of register C to the low 3 nybbles of the status register, ST.

MOVE.X C, ST

Move register to status register

Opcode: **0A**
Cycles: **6**

Carry:
Not affected

Set low 3 nybbles of the status register, ST, to field X (low 3 nybbles) of register C.

NEG.*f* *d*

Negate register

Opcode: **kku**
Cycles: **7(A), 3+n**

Carry:
Cleared if zero, else set

kk	*f*
B8	P
B9	WP
BA	XS
BB	X
BC	S
BD	M
BE	B
BF	W
F	A

u	*d*
8	A
9	B
A	C
B	D

Negate field *f* of register *d*. Two's complement if in HEX mode; ten's complement if in DEC mode.

---

420 **NOP3**
6300 **NOP4**
64000 **NOP5**

No operation

Opcode: **see above**
Cycles: **see below**

Carry:
Not affected

No operation; do nothing. `NOP4` and `NOP5` [11 cycles] are relative jumps with offset 0. `NOP3` [10 cycles if carry set, else 3 cycles] is a `BRCS` to the next instruction.

NOT.*f* *d*

Invert register

Opcode: **kkw**
Cycles: **7(A), 3+n**

Carry:
Cleared

kk	*f*
B8	P
B9	WP
BA	XS
BB	X
BC	S
BD	M
BE	B
BF	W
F	A

w	*d*
C	A
D	B
E	C
F	D

One's complement field *f* of register *d*. Not affected by whether currently in HEX/DEC mode.

OR.*f* *s*, *d*

OR register to register

Opcode: **0Etx**
Cycles: **4+n**

Carry:
Not affected

t	*f*
0	P
1	WP
2	XS
3	X
4	S
5	M
6	B
7	W
F	A

x	*s*	*d*
8	B	A
9	C	B
A	A	C
B	C	D
C	A	B
D	B	C
E	C	A
F	D	C

Logical OR field *f* of register *s* with field *f* of register *d*.

---

800 **OUT.1** **C**
801 **OUT.3** **C**

OUT register

Opcode: **see above**
Cycles: **4(1), 6(3)**

Carry:
Not affected

Copy the least significant 1 or 3 nybbles from register C to the OUT register.

POP.A C

POP register

Opcode: **07**
Cycles: **8**

Carry:
Not affected

Pop off the top value on the 8-level return stack and place it in the least significant 5 nybbles of register C. Return instructions perform a pop into the program counter.

PUSH.A C

PUSH register

Opcode: **06**
Cycles: **8**

Carry:
Not affected

Shift the 8-level return stack down one step, and replace the top level with the contents of field A of register C. Subroutine call instructions automatically push the current program counter before jumping to the subroutine.

RESET

Hardware reset

Opcode: **80A**
Cycles: **6**

Carry:
Not affected

Cause the CPU to emit the "System Reset" bus command, resulting in all chips resetting.

---

01 **RET**
02 **RETSETC**
03 **RETCLRC**
00 **RETSETXM**
0F **RETI**

Unconditional return from subroutine or interrupt

Opcode: **see above**
Cycles: **9**

Carry:
See below

`RET` returns without affecting the carry. `RETSETC` returns with the carry set. `RETCLRC` returns with the carry clear. `RETSETXM` returns with the XM bit in the hardware status register, HST. `RETI` returns and enables interrupts; it is used by the system to return from an interrupt.

8086x00	**RETBC**	*x*, **A**
808Ax00	**RETBC**	*x*, **C**
83x00	**RETBC**	*hf*, **HST**
86x00	**RETBC**	*x*, **ST**
8087x00	**RETBS**	*x*, **A**
808Bx00	**RETBS**	*x*, **C**
87x00	**RETBS**	*x*, **ST**
500	**RETCC**	
400	**RETCS**	
89x00	**RETEQ.1**	*x*, **P**
88x00	**RETNE.1**	*x*, **P**
ttu00	**RETEQ.***f*	*s*, *d*
ttv00	**RETNE.***f*	*s*, *d*
ttw00	**RETZ.***f*	*s*
ttx00	**RETNZ.***f*	*s*
zzu00	**RETGT.***f*	*s*, *d*
zzv00	**RETLT.***f*	*s*, *d*
zzw00	**RETGE.***f*	*s*, *d*
zzx00	**RETLE.***f*	*s*, *d*

Conditional return from subroutine

Opcode: **see above and branch instruction**
Cycles: **see below and branch instruction**

Carry:
See branch instruction

For information on conditions and opcodes, refer to the corresponding branch instruction. The cycle time for return instructions is the same as for

the branch instructions, with the return-taken cycle time corresponding to that of the branch taken.

81w **RLN.W** *d*
81x **RRN.W** *d*

Rotate left one nybble
Rotate right one nybble

Opcode: **see above**
Cycles: **21**

Carry:
Not affected

w	*x*	*d*
0	4	A
1	5	B
2	6	C
3	7	D

Rotate register C left or right. **RRN** causes the sticky bit, SB, of the hardware status register, HST, to be set if the nybble rotated from position 0 to position 15 was nonzero. A zero nybble will not clear the sticky bit; this must be done with a CLRB instruction. **RLN** does not affect HST:SB.

RSI

Reset interrupt system

Opcode: **80810**
Cycles: **6**

Carry:
Not affected

Check if any interrupts are pending; if so, service them, unless already servicing interrupt, in which case wait until done as flagged by the return from interrupt (RTI) instruction.

8085x **SETB** *x*, **A**
8089x **SETB** *x*, **C**

Set register bit

Opcode: **see above**
Cycles: **???**

Carry:
Not affected

Set bit *x* (range 0-15) of register A or C.

SETB *x*, **ST**

Set status register bit

Opcode: **85x**
Cycles: **4**

Carry:
Not affected

Set bit *x* (range 0-15) of the status register, ST.

---

05 **SETDEC**
04 **SETHEX**

Set CPU arithmetic mode

Opcode: **see above**
Cycles: **3**

Carry:
Not affected

Set arithmetic mode to decimal or hexadecimal. In hexadecimal mode, all arithmetic is performed in unsigned binary. In decimal mode, the instructions listed below, when *involving registers A, B, C, or D* will operate in ten's complement. In ten's complement, each nybble has a value 0-9, corresponding to a BCD digit. The instructions affected by decimal mode are: `ADD`, `SUB`, `NEG`, `INC`, and `DEC`.

SHUTDN

Shutdown bus

Opcode: **807**
Cycles: **5**

Carry:
Not affected

Cause the CPU to emit the "Shutdown" bus command and stop the system clock. The result will be a system halt if done when the OUT register is clear (000).

---

rrw **SLN.*f*** *d*
rrx **SRN.*f*** *d*

Shift nybble register

Opcode: **see above**
Cycles: **7(A), 3+n**

Carry:
Not affected

rr	*f*
B8	P
B9	WP
BA	XS
BB	X
BC	S
BD	M
BE	B
BF	W
F	A

w	*x*	*d*
0	4	A
1	5	B
2	6	C
3	7	D

Shift field *f* of register *d* left or right one nybble. Zeros are shifted in, and the nybble shifted out is lost. The sticky bit, SB, of the hardware status register, HST, is set if the nybble shifted out by SRN was nonzero. SLN does not affect HST:SB.

819tw **SRB.*f*** *d*
81z **SRB.W** *d*

Shift bit register

Opcode: **see above**
Cycles: **20(W), 6+n**

Carry:
Not affected

t	*f*
0	P
1	WP
2	XS
3	X
4	S
5	M
6	B
7	W
F	A

w	*z*	*d*
0	C	A
1	D	B
2	E	C
3	F	D

Shift field *f* of register *d* right one bit. Zero bits are shifted in, and the bit shifted out is lost. The sticky bit, SB, of the hardware status register, HST, is set if the bit shifted out was nonzero.

SREQ

Service request check

Opcode: **80E**
Cycles: **7**

Carry:
Not affected

Emit a "Service Request?" bus command, causing the SR bit of the HST register to become set if any device responds affirmatively. The low 4 bits of the C register are set to the device, with each bit corresponding to a device. HST:SR is set to the logical OR of these bits.

---

qqy **SUB.*f*** *s*, *d*
qqw **SUBN.*f*** *s*, *d*

Subtract register

Opcode: **see above**
Cycles: **7(A), 3+n**

Carry:
Set on underflow, else cleared

qq	*f*
B0	P
B1	WP
B2	XS
B3	X
B4	S
B5	M
B6	B
B7	W
E	A

y	*w*	*s*	*d*
0	C	B	A
1	D	C	B
2	E	A	C
3	F	C	D
4		A	B
5		B	C
6		C	A
7		D	C

SUB: subtract field *f* of register *s* from field *f* of register *d*. SUBN: subtract field *f* of register *d* from field *f* of register *s*. Both place the result in field *f* of register *d*.

SUB.*f* *i*+1, *d*

Subtract constant from register

Opcode: **818tui**
Cycles: **???**

Carry:
Set on underflow, else cleared.

t	*f*
0	P
1	WP
2	XS
3	X
4	S
5	M
6	B
7	W
F	A

u	*d*
8	A
9	B
A	C
B	D

Subtract a small constant *i* (range 0-15) from field *f* of register *d*.

18i **SUB.A** *i*+**1, D0**
1Ci **SUB.A** *i*+**1, D1**

Subtract constant from address register

Opcode: **see above**
Cycles: **7**

Carry:
Set on underflow, else cleared.

Subtract a small constant *i* (range 0-15) from address register D0 or D1.

SWAP.*f* *s*, *d*

Swap register and register

Opcode: **kkw**
Cycles: **7(A), 3+n**

Carry:
Not affected

kk	*f*
A8	P
A9	WP
AA	XS
AB	X
AC	S
AD	M
AE	B
AF	W
D	A

w	*s*	*d*
C	B	A
D	C	B
E	A	C
F	C	D

Swap contents of field *f* of register *s* with contents of field *f* of register *d*.

13x **SWAP.A** *s*, *Dn*
13z **SWAP.4** *s*, *Dn*

Swap register and address register

Opcode: **see above**
Cycles: **8(A), 7(4)**

Carry:
Not affected

x	*z*	*s*	*Dn*
2	A	A	D0
3	B	A	D1
6	E	C	D0
7	F	C	D1

Swap contents of least significant 4 or 5 nybbles of register *s* with the corresponding nybbles of address register *Dn*.

12j **SWAP.W** **A**, *Rn*
12k **SWAP.W** **C**, *Rn*
81At2j **SWAP.*f*** **A**, *Rn*
81At2k **SWAP.*f*** **C**, *Rn*

Swap register with temporary register

Opcode: **see above**
Cycles: **19(W), 7???+n**

Carry:
Not affected

t	f
0	P
1	WP
2	XS
3	X
4	S
5	M
6	B
7	W
F	A

j	k	Rn
0	8	R0
1	9	R1
2	A	R2
3	B	R3
4	C	R4

Swap contents of field f of register A or C with the corresponding nybbles of temporary register Rn.

SWAP.1 P, C.n

Swap register nybble and pointer register

Opcode: **80Fn**
Cycles: **6**

Carry:
Not affected

Swap contents of nybble n of register C with contents of the pointer register, P.

SWAP.X C, ST

Swap register and status register

Opcode: **0B**
Cycles: **6**

Carry:
Not affected

Swap contents of field X of register C with the least significant 3 nybbles of the status register, ST.

SWAP.A A, PC

Swap register A and program counter

Opcode: **81B6**
Cycles: **???**

Carry:
Not affected

Swap field A of register A with the contents of the program counter (PC).

SWAP.A C, PC

Swap register C and program counter

Opcode: **81B7**
Cycles: **???**

Carry:
Not affected

Swap field A of register C with the contents of the program counter (PC).

UNCNFG

Unconfigurate device

Opcode: **804**
Cycles: **12**

Carry:
Not affected

Copy field A of the C register into each data pointer, with the device addressed by the data pointer unconfiguring.

4.5 Instruction Set Reference

On the following pages is a list of SATURN instructions, in the format in which they are listed by the MLDL commands. Square brackets refer to optional elements, and curly brackets to a set of choices. The tiny numbers to the left are the opcode sizes, in nybbles. A set of choices within square brackets is optional, with the first choice being the default. The choices are single characters unless separated by commas, in which case they are words.

The mnemonics are identical to those used by the STAR macro assembler. For example:

```
3    MOVE[.{A4}]      ac, Dn
```

This means that the instruction is `MOVE` with two arguments: the first being either register `A` or `C`, and the second either `D0` or `D1`. The suffix is optional, but if it appears it must be either `.A` or `.4`, with `.A` being the default. The opcode size is three nybbles.

```
5    BR{EQ,NE}[.1]    P, nib, dest
```

This means that the instruction is either `BREQ` or `BRNE` with three arguments: the first is always `P`, the second a small integer 0–15, and the third a jump destination. The `.1` suffix is optional. The opcode is 5 nybbles.

```
3    ADDf             s,d
6    ADD[.A]          const, ar
3    ADD[.A]          const, Dn
3    ADD[.A]          P+1, C
```

4	AND*f*	*s*,*d*	*i*, *iii*
5	BRBC	*mask*, HST, *dest*	
5	RETBC	*mask*, HST	
5	BRB{SC}	*bit*, ST, *dest*	
5	RETB{SC}	*bit*, ST	
7	BRB{SC}	*bit*, ac, *dest*	
7	RETB{SC}	*bit*, ac	
3	BRC{SC}	*dest*	
5	BR{EQ,NE}[.1]	P, *nib*, dest	
5	BR*arcondf*	*s*,*d*, *dest*	*i*
5	RET*arcondf*	*s*,*d*	*i*
5	BR{Z,NZ}*f*	ar, *dest*	
4	BUSCB		
3	BUSCC		
4	BUSCD		
3	UNCNFG		
3	CONFIG		
3	SHUTDN		
3	RESET		
3	SREQ		
3	MOVE[.A]	ID, C	
4/5/7	CALL[.{34A}]	*dest*	
2/3	CLR*f*	ar	
2	CLR[.1]	P	
2+n	CLR[.{245}]	*Dn*	
2	CLR[.X]	ST	
3	CLRB	*bit*, ST	
3	CLRB	*mask*, HST	
5	CLRB	*bit*, ac	
3	DEC[.A]	*Dn*	
2	DEC[.1]	P	
2/3	DEC*f*	ar	
3	IN[.4]	ac	
2	INC[.1]	P	
3	INC[.A]	*Dn*	
2/3	INC*f*	ar	

4	INTOFF		
4	INTON		
5	RSI		
4/6/7	JUMP[.{34A}]	*dest*	
4	JUMP[.A]	*ac*	
4	JUMP[.A]	@*ac*	
2/3	MOVE*f*	*s,d*	*i, iii*
3/4	MOVE*f*	*ac*, @*Dn*	*iv*
3/4	MOVE*f*	@*Dn*, *ac*	*iv*
4	MOVE.{1–16}	*ac*, @*Dn*	
4	MOVE.{1–16}	@*Dn*, *ac*	
3	MOVE[.{A4}]	*ac*, *Dn*	
3/6	MOVE[.W*f*]	*ac*, *Rn*	
3/6	MOVE[.W*f*]	*Rn*, *ac*	
2+n	MOVE.P{1–16}	*int*, C	
5+n	MOVE.P{1–16}	*int*, A	
2+n	MOVE.{245}	*int*, *Dn*	
2	MOVE[.1]	*nib*, P	
4	MOVE[.1]	P, C.*nib*	
4	MOVE[.1]	C.*nib*, P	
2	MOVE[.X]	ST, C	
2	MOVE[.X]	C, ST	
2/3	NEG*f*	*ar*	
2/3	NOT*f*	*ar*	
4	OR*f*	*s,d*	*i, iii*
3	OUT.{SX}	C	
2	POP[.A]	C	
2	PUSH[.A]	C	
2	RET		
2	RET{SET,CLR}	C	
5	RET{Z,NZ}*f*	*ar*	
2	RETI		
2	RETSETXM		
3	RLN[.W]	*ar*	
5	SETB	*bit*, *ac*	
3	SETB	*bit*, ST	

2	SETDEC		
2	SETHEX		
3	SLB.{W*f*}	*ar*	
2/3	SLN.{A*f*}	*ar*	
3/5	SRB.{W*f*}	*ar*	
2/3	SRN.{A*f*}	*ar*	
2/3	SUB.{A*f*}	*s*,*d*	*i*, *iii*
6	SUB*f*	*const*, *ar*	
2/3	SUBN{A*f*}	*s*,*d*	*i*, *iii*
3	SUB[.A]	*const*, *Dn*	
4	SWAP[.A]	*ac*, PC	
2/3	SWAP.{A*f*}	*s*,*d*	*i*, *iii*
3	SWAP[.{A4}]	*ac*, *Dn*	
3/6	SWAP[.{W*f*}]	*ac*, *Rn*	
4	SWAP[.1]	P, C.*nib*	
2	SWAP[.X]	C, ST	

Legend

nib	Nibble number; `0–15`
bit	Bit number; `0–15`
const	Small constant; `1–16`
int	Integer
mask	HST bit mask; see table below
ar	Arithmetic register; `A`, `B`, `C`, or `D`
ac	Register `A` or `C`
Dn	Address register; `D0` or `D1`
Rn	Temporary register; `R0`, `R1`, `R2`, `R3`, or `R4`
arcond	Arithmetic condition; {`EQ`,`NE`,`GT`,`LT`,`GE`,`LE`}
i-v	Combinatorial constraints of *s*,*d*and *f* (see below)

Arithmetic register combinations

	s,d	*s,d*	*s,d*	*s,d*
i	B,A	C,B	A,C	C,D
ii	A,A	B,B	C,C	D,D
iii	A,B	B,C	C,A	D,C

Instruction suffixes

	f	*f*	*f*	*f*	*f*	*f*	*f*	*f*
iv	P	WP	XS	X	S	M	B	W
v	A							

Hardware status bits

mask	*mask*	*mask*	*mask*
XM	SB	SR	MP

A MLDL Command Summary

ABOUTMLDL	Display logo, version, and copyright $\longrightarrow$
MLDB	SATURN Machine Language Debugger $obj \longrightarrow any_1 \ldots any_n$
MLPR	Print Machine Language Program $obj \longrightarrow obj$
ML1	Disassemble single instruction $\#address \longrightarrow$ "instruction" $\#next$
MLOPC	Return instruction opcode $\#address_1\ \#address_2 \longrightarrow$ "hexadecimal digits"

B MLDB Local Mode Keyboard Summary

Entering **ARG**:

Keys	Description
[0]	Begin **ARG** entry. The MLDB will remain in **ARG** entry mode until a non-hex key is pressed.
[0] - [9] [A] - [F]	Hex digits.
[+/-]	2's complement
[⇐]	Shift **ARG** right one digit. The most significant digit is cleared.
[DEL]	Cancel **ARG**. Return to general interactive mode.

General interaction:

Keys	No ARG	ARG
A	Screen: General CPU State	
B	Screen: Arithmetic registers	
C	Screen: Data registers	
D	Screen: Return stack	
E	Screen: Memory dump	
F	Screen: Instruction stream	
MTH	Screen: Breakpoint table	
ENTER		PC = **ARG**
◁	PC = PC - 1	PC = PC - **ARG**
▷	PC = PC + 1	PC = PC + **ARG**
△	PC = PC - 16	PC = PC - 16 × **ARG**
▽	PC = PC + 16	PC = PC + 16 × **ARG**
EEX	Display PICT	

(Continued from previous page)

Keys	No **ARG**	**ARG**
[×]	MARK = PC	MARK = **ARG**
[+/-]	MARK ↔ PC	MARK ↔ PC
[NXT]	Advance one instruction	Advance **ARG** instructions
[PRG] *b*	Clear breakpoint *b*	Set breakpoint *b* at **ARG**
[STO] *b*	Clear breakpoint *b* counter	Set breakpoint *b* counter to **ARG**
[+]	single-step one instruction	Single-step **ARG** instructions
[−]	single-step one instruction, do not follow CALLs	Single-step **ARG** instructions, do not follow CALLs
[EVAL]	Let program run free until completed, or until a breakpoint is encountered	Ignored
[CST] *t*		Toggle option *t*
[VAR]		Display watchpoint table
[VAR] *w*	Set watchpoint *w* to **ARG**	

C MLDL XLIB Numbers

The command numbers are subject to change at random and without prior notice. The library number is **444** hexadecimal, which is 1092 decimal.

Command	Number	
ABOUTMLDL	1092	0
MLDB	1092	1
MLPR	1092	2
ML1	1092	3
MLOPC	1092	4

D Common Abbreviations

GC *Garbage Collect.* Recycling of waste products.

ML *Machine Language.* Sometimes called Machine Code or Assembler.

PMC *Prefixed Machine Code.* An RPL routine implemented in ML wherein the type prefix of the PMC routine points to the first instruction.

RPL *Reverse Polish Lisp*; also *ROM-based Procedural Language*, depending on who you talk to. The language of the HP 48. System RPL refers to the internal non-keyword-based threads.