

# Using the HP49G for System RPL Programming

**Eduardo M Kalinowski**  
**ekalin@iname.com**

**January, 2001**

## Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>The Entry Points Library</b>	<b>4</b>
<b>3</b>	<b>About Key Assignments</b>	<b>5</b>
<b>4</b>	<b>Hacking Tools</b>	<b>6</b>
4.1	Operating Tools for the HP49 . . . . .	9
<b>5</b>	<b>The Compiler</b>	<b>10</b>
<b>6</b>	<b>Disassembly</b>	<b>12</b>
6.1	Disassembly of Built-in Commands . . . . .	12

<b>7</b>	<b>The Editor, and Emacs</b>	<b>13</b>
7.1	Emacs Customization . . . . .	17
<b>8</b>	<b>Debugging</b>	<b>19</b>
<b>9</b>	<b>Conclusion</b>	<b>20</b>

# 1 Introduction

This document will describe tools for the HP49G calculator that make it a suitable programming environment for System RPL development. The HP49G calculator includes a built-in compiler, disassembler and some sort of debugger (which, to say the truth, could be improved), plus some other little tools that can be of use to the System RPL programmer. However, for big programming tasks that is not enough: some other tools are necessary to make programming easier. Because of this, some third-party tools will also be described. With a good knowledge of the built-in and third-party tools the HP49G can be used as a complete and compact programming environment. All the tools described here can be downloaded from The HP Software Archive, <http://www.hpcalc.org>.

Although several times comparisons with HP48 or PC programs are made, no experience with them is necessary.

The built-in programming tools you will need are, by default, not accesible to the user. They are in two libraries, which are not attached by default. Library 256 contains several useful commands for “hacking” with the calculator, and the disassembler. Library 257 contains MASD, the compiler. You should have these libraries always attached. If you have extable installed (and you should — see section 2), then library 256 will be automatically attached. Library 257 (MASD) does not really need to be attached, because it is possible to call MASD from library 256. Nevertheless, it is still good to have it attached.

The STARTUP variable is useful to configure the calculator. This variable (which must be in the HOME directory) contains an object to be executed after each warmstart. It can be used to set all parameters lost by a warmstart that you want to keep, or to do anything else you want. The following program will set user mode (which is lost in a warmstart); for efficient programming it is almost essencial to make some key assignments.

« -62 SF »

## 2 The Entry Points Library

For System RPL development, the extable library is virtually essential. This library contains the tables of entry points and addresses. It is with the help of this library that you can write DUP and get the correct address for this command; without it, you would need to enter PTR 3188 every time or write an equate for this command manually. In disassembly (including the System RPL stack (see section 4) — flag -85), it allows you to get the name of the commands, instead of only their addresses. This way, this library is pretty much essential

Download extable to your calculator and install it as any other library. That's all you need to do to use command names instead of addresses.

This library contains the entry point tables (not visible to the user) and a few useful commands. It appears in the library menu, and it contains five user-accessible commands.

The first command, nop, does nothing :-). I suppose there was a command in that position before, but it was removed, and another command that does nothing was put there not to change the other XLIB numbers.

The other four commands, fortunately, are sometimes useful :-). (at least for the Emacs library). The GETADR command returns the address of an entry. Just put the name of the entry (a string) in level one and run it. The inverse operation is done by GETNAME: give it an address, and it will return the name of the entry.

If you do not know exactly the name of an entry, the last two commands will help you. Put a string with the first few letters of the command in level one, run GETNAMES and, voilà, a list with the names of all commands that start with those letters is returned. The last command, GETNEAR, is even more powerful: give it a string, and all commands whose names contain that string (even if in the middle of the command) will be returned.

### 3 About Key Assignments

Even though assigning keys is not directly related to System RPL programming, I will describe here the KEYMAN library, written by Wolfgang Rautenberg. The latest version of January, 2001. This library simplifies the assignment, deletion and recalling of keys, but, most importantly, allows a key to behave differently if it is pressed longer than usual, or to behave differently if the key is double pressed.

You will find several commands inside this library. The A?D command is used to assign and delete keys. To assign something to a key, put the object in level one and press A?D shortly. Then, press the key you want to assign to (shifts, and shift-holds work, of course). The key is assigned. To delete an assignment, press A?D for a longer time, and then the key from which you want to remove the assignment. The command RclK allows one to recall the assignment of any key. It works like the previous commands: press it (briefly) and then the key. A longer press will return a list of all the keys assigned.

The commands above are just other ways to do what was already possible with the built-in commands. But the real power is in the IfE?P, IfD and IfL commands. The first serves two functions: it allows a key to have different meanings when in edit mode and when not or to have different meanings when in program mode and when not. To use it, put the object to be run in edit or program mode in level two, the object to be run in normal mode in level one, and press IfE?P. A short press will create a program that evaluates the object in level two if the calculator is in edit mode, or the object in level one if not. A longer press does the same, but the test is based on whether program entry mode is active or not.

The IfD and IfL commands are similar. To use IfD, put in level two the object to be run if the key is pressed twice like a computer mouse — double pressed — and put in level one the object to be run if the key is pressed once. Run IfD, and you will have a single program that executes one of the objects according to how the key was pressed. The command IfL is similar, but it allows different actions based on how long the key is pressed: you have seen this behaviour in the A?D command. The object to be run in a long press is in level two.

Finally, two other commands can sometimes be useful: →T0?

inserts the System RPL command `TakeOver` in the beginning of the program when the key is pressed shortly. This is necessary if you want the command to be executed while the command line is active. A longer press inserts `UnlockAlpha` in the beginning of the program, useful when it is assigned to an alpha-shifted key. Finally, SA recalls the standard assignment for any key. This is used when you want to add new functionality to a key. When this standard assignment is a command in a library (that is, a ROM Pointer, also called a XLIB name), the pointer is recalled to level two, and its contents is put in level one.

In the following sections, there will occur many examples of key assignments built with these commands.

## 4 Hacking Tools

The tools described here make the life of the programmer easier. First, the built-in tools in the HP49G will be described. Later, a third-party library will be described.

Before describing the built-in tools found in library 256, I will mention a flag that is very useful to System RPL programmers: flag -85. When this flag is set, the “System RPL Stack” is active: in the stack the objects are decompiled before being displayed. That means that, where one would see just `External` with the normal stack, the name for the entry (or PTR and the address, if no name is found) will be displayed, if you have the extable library (see section 2) installed (you should). Play with it a bit and you will see how useful it can be. Some objects (most notably real numbers and integers) keep their usual notation, but in the interactive stack all objects are decompiled. This “System RPL Stack” is like the one produced by the command `SSTK` command of the JAZZ library for the HP48 calculators.

Probably you will be switching between the two kinds of stack display all the time. It is a good idea to assign a simple program to a key to toggle this display. I have it assigned to Right-Shift `MODE`. This normally is the key that marks the end of selection in edit mode. Since this key is unused when not in edit mode, it is a good example of the use of the `KEYMAN` library. To create this assignment, first put the program to be run when in edit mode in level two. This is easy: just

use SA to recall it, running SA then pressing Right-shift MODE. Then, write a simple User RPL (or even System RPL, if you want) program to toggle flag -85 (this task is left to the reader — but read the description of OT49 in section 4.1 first). Finally, press IfE?P briefly and use →T0? on the resulting program (because you want it to be run while in edit mode), and assign it to the key, with A?D or the ASN command.

Library 256 contains some useful tools for the programmer. This library does not show up in the library menu (because it does not have a title), but you can get its menu by typing 256 MENU. If the library is attached (as it should be), you can type the commands, look up them in the catalog and an option will appear in the Apps menu, which says “Development lib”, giving access to all the commands in the library.

Here is a description of the commands present in the library:

- H “To hex”: This converts an object into a string of hexadecimal characters. A common tool since the HP48 days to ease transfer of binary objects.
- H→ “From hex”: This is the opposite transformation: creates an object from a string of hexadecimal characters.
- A “To address”: Given an object, this command returns the address of the object, which is always a five-nibble hxs. Objects whose address is less than # 80000h are in ROM, and objects whose address is greater than that are in RAM.
- A→ “From address”: This recalls the object at the specified address.
- A→H “Address to hex”: This converts # 272FEh into "EF272".
- H→A “Hex to address”: The opposite transformation: converts "EF272" into # 272FEh.
- CD “To code”: Converts a string of hex digits into a Code object.
- CD→ “From code”: Converts a Code object into a string of hex digits.
- S→H “String to hex”: Converts a string into its characters’ hexadecimal representation. For example, since 5A, 59 and 58 are the hexadecimal codes for X, Y and Z respectively, "XYZ" becomes "8595A5".

H→S “Hex to string”: the opposite transformation.

→LST “Make list”: Creates a list from a usermetaobject or another composite. A usermetaobject is any number of objects in the stack followed by a count — a real number. Be careful, because this command is not sufficiently argument-protected.

→ALG “Make algebraic”: Creates an algebraic object from a usermetaobject or another composite. This may easily result in ‘Invalid Expression’.

→PRG “Make program”: Creates a program from a usermetaobject or another composite.

COMP→ “From composite”: Explodes any composite object into a usermetaobject.

→RAM “To RAM”: Dumps any ROM object into RAM. Can extract some commands for disassembly, but see section 6.1 for more information.

SREV “Reverse string”: Reverses a string, very fast.

POKE Writes data to any address in RAM. Put in level two a hxs with the address, and in level one a string of hex digits to be written at that address. This is a very easy way of destroying any “masterpiece” you have created on the calculator :-).

PEEK Extracts raw hex digits from any address. Put the address in level two (an hxs) and the number of nibbles to get (another hxs) in level one.

APEEK “Address peek”: Like PEEK, but always gets five nibbles, returning them as a hxs.

R~SB “Real ↔ system binary”: Converts reals to bints and vice-versa.

SB~B “System binary ↔ binary”: Converts bints to hxs’s, and vice-versa.

LR~R “Long real ↔ real”: Converts long reals to reals and vice-versa.



`S~N` “String  $\leftrightarrow$  name”: Converts strings to identifiers (global names) and vice versa.

`LC~C` “Long complex  $\leftrightarrow$  complex”: Converts long complexes to complexes and vice-versa.

`ASM→` “From ASM”: Disassemble Code objects (machine-language) into source code.

`BetaTesting` Returns a string with some names (of beta testers of the calculator?).

`CRLIB` “Create library”: A library creator.

`CRC` Calculates the CRC. The argument is a string of hex digits.

`MAKESTR` “Make string”: Creates a string with the number of characters given in level one (a real number).

`SERIAL` Returns a string with the internal Serial Number of the HP49.

`ASM` Provides access to the MASD compiler. See section 5 for more information.

`ER` Used in conjunction with `ASM`. See section 5 for more information.

`→S2` Disassembles an object. See section 6 for more information.

`XLIB~` Creates a `XLIB` (rompointer) from the library number (level two) and command number (level one). Although the `~` in the name suggests this is a toggler, it can only create rompointers, not extract them.

## 4.1 Operating Tools for the HP49

Wolfgang Rautenberg (e-mail: [raut@math.fu-berlin.de](mailto:raut@math.fu-berlin.de)) is the author of a library called Operating Tools (or OT49 for short) with several commands, some of which are useful to the System RPL programmer. The latest version of this library is of February 2001.

OT49 contains a library creator and splitter. To split any library, just put its number in the stack and run `D $\leftrightarrow$ L`.

The `DType` command displays the type of the object in level one. If that object is a rompointer (XLIB) or flashpointer, its contents is recalled (unless it's pure machine-language code) and the contents' type is displayed, with an asterisk appended.

The most useful command, in my opinion is `3tog`. It toggles between three representations of composite objects: as a list, as a program and as a usermetaobject.

Another very useful command is `F1~`. It is a flag toggler. Just give the number of the system or user flag, run it, and the flag is toggled. It will also display in the header what has just been done.

The `→XU` command strips unnecessary stuff such as the `« »` delimiters from User RPL programs. The resulting program will be a little faster, but uneditable. You can, however, change it easily with the help of other OT49 commands.

## 5 The Compiler

The compiler included in the HP49G calculator is MASD. It is a newer version of the compiler found in the MetaKernel program for the HP48G calculators. If you have already used the MetaKernel, then you probably can skip most of this section. But, even if you have never used MASD, there should be no difficulties learning how to use it. There are no big differences between MASD syntax and that of other System RPL compilers such as JAZZ (for the HP48 calculators), the HP Tools or the GNU Tools.

MASD is called with the command `ASM`. It expects a string in level one, and returns the compiled object. If there are errors, the string and a list will be put in the stack. This list is used by the `ER` command, described shortly.

The first difference to be observed from those that are coming from JAZZ or one of the PC Tools is that MASD, for some unknown reason, need the source to end with a `@` character. This character must be on a line by itself, at the start of the line, and with no character after it (not even a newline). This way, it is pretty much cumbersome and useless. (To be useful, it would be the character marking the end of the

source, but there should not be all those restrictions on its placement, and text after it should be allowed — and ignored). However, if you use Emacs' RPLCPL (see section 7), the @ comes in handy.

The other thing to note concerns the current MASD mode. There are two modes, controlled by flag -92: Assembly Language mode (flag -92 cleared) and System RPL mode (flag -92 set). Probably, you will set flag -92 and thus MASD will be by default in System RPL mode. Then, nothing else needs to be changed to compile System RPL programs (just add the @ in the end). It is still possible to compile Assembly Language code in System RPL mode: just put the code between CODE and ENDCODE.

If you are in Assembly Language mode, it is possible to compile System RPL code inserting these two lines before the source:

```
!NO CODE
!RPL
```

Both are called directives. The !NO CODE directive tells MASD not to compile our source as if it were Machine Language code. (Again, you can insert assembly language code between CODE and ENDCODE). It is a good idea to always put these two lines at the start of all programs even if you use System RPL mode: this way, the source can be compiled regardless of the flag settings.

Here is a simple program source ready for MASD:

```
!NO CODE
!RPL
::
  DUPTYPEZINT?
  case
    FPTR2 ^Z>R
  DUPTYPEREAL? ?SEMI
  SETTYPEERR
;
@
```

The above is the disassembly of the CKREAL entry. As you can see, it automatically converts integers to real numbers.

While the compilation is being done, a status screen displays the compilation status (only in ROMS after version 1.19-5). This displays the time elapsed so far, the number of instructions compiled, the average number of instructions compiled by second and the status of the compilation. However, probably you will not be able to see any of the information of this status screen, because MASD is very fast: the only thing you will see is some kind of screen blink.

It is a nice idea to assign the ASM command to a key: you will need it many times.

If there was an error during compilation, the original string is put in level two, and a list is put in level one. In this case, run the ER command. It will display a list of errors for you to choose, and will jump directly to that error in the source. Correct the error, press ENTER and then choose another error, until all errors have been corrected. Then, run ASM (and ER, if necessary) again. Better yet, use the ASM2 command from library 257, which calls ASM and then, if there was any error, ER.

## 6 Disassembly

As it was briefly mentioned in the description of Library 256 (see section 4), the command  $\rightarrow S2$  is the disassembler. It will disassemble any object, in level one, into its source code suitable for reassembly with MASD. Unfortunately, there are still some bugs in MASD, which prevent some disassembled objects to be correctly re-assembled. We all hope that in a newer version this bugs will be corrected.

### 6.1 Disassembly of Built-in Commands

Often, one wants to see how one of the built-in commands in the HP's ROM is built. The JAZZ library for the HP48 calculators made that easy. Unfortunately, it is difficult to do that with only the tools in the HP49G. But, with the help of the CQIF? (*Comment Qu'Il's Font?*) library, written by Pierre Tardy (e-mail: tardyp@iname.com), currently at version 1.7.5 $\beta$ , that task is simplified. It contains several tools for the HP49G hacker. I will not describe everything from the library here,

read its documentation if you want to know what else it can do for you.

The most useful command is CQIF?. After some experimentation with this command, you will find it so useful that you will assign it to some key. In section 7, I will develop (with the help of KEYMAN and Emacs) a very useful assignment involving CQIF?.

This command is the basic way to disassemble some part of the HP's ROM. It accepts several kinds of inputs. If you give a string with the name of an entry, that entry is disassembled. You can put an address (a hxs), and run CQIF? to disassemble whatever is at that address. It will also accept the entry pointer itself, rompointerss and flashpointers. To ease the disassembly of User RPL commands, you can enter the command inside a list or program (that is, enter { DUP } or « DUP » to disassemble the User RPL command DUP).

Note that if the object you want to disassemble is written in machine language, you should tell this to CQIF? by dupping the object before running the command. With two copies of the object in the stack, a assembly language disassembly will be done.

In my experiences with CQIF?, you might need to run this command more than once to disassemble some commands. This is normal. Just remove any unnecessary junk from the stack, keeping the last result of CQIF? and run it again. Eventually you will reach the command.

Another useful command is DISPATCH. It does a virtual dispatch based on the object types. To use it, put the objects you would use as arguments to some command in the stack. Then, recall that command (probably using CQIF?) to level one. Run DISPATCH. The object that would be run for those argument types (by means of some dispatching command like CKn&Dispatch) is put in level one.

The other commands are not so useful to System RPL programs. But it is a nice idea to read the documentation and see what CQIF? can do for you.

## 7 The Editor, and Emacs

The HP49G editor is much better than the one in the HP48 calculators. However, it can be made even better. There are two variables

that are run before entering and after leaving the editor. We will see what can be done with them. I will also describe a library that enhances the editor with some very nice features.

Before starting the editor, the variable `STARTED` is run. You can put a program in this variable to be run before editing any object. And, after leaving the editor, the `EXITED` variable is run. There are many things these variables can do. A very simple (and very useful) thing is to remove the header during editing, giving a few more lines of text. After the editor is exited, the header is restored to the default setting. It is very simple to do this: `STARTED` just needs to clear the header:

```
« 0 →HEADER »
```

And `EXITED` restores the header:

```
« 2 →HEADER »
```

Change 2 to 1 if you normally use only one line of header. Note that Emacs (see below) remove the header automatically.

For even better customization of the editor there is the Emacs library, written by Carsten Dominik (e-mail: [dominik@astro.uva.nl](mailto:dominik@astro.uva.nl)) and Peter Geelhoed (e-mail: [P.F.Geelhoed@student.tn.tudelft.nl](mailto:P.F.Geelhoed@student.tn.tudelft.nl)). This library gives the editor some of the features of the famous GNU Emacs editor, such as completion, automatic indentation, incremental search, and a macro language. The latest version, at the time of this writing, is 0.5001. Again, I will not describe everything in the library — see the manual for more information.

Probably the most useful feature of the Emacs library is command completion. Just that is worth loading the library in the calculator. It is activated by the `RPLCPL` command. This is only useful in edit mode, so you will need it assigned to a key, with `TakeOver` before. If you have the `KEYMAN` library (see section 3), just put a program like this:

```
« RPLCPL »
```

and run `→T0?`. Then, assign the resulting object to a key. It is a nice idea to assign it to the same key both with and without the alpha-mode

on. Of course, you do not need a program. Just the rompointer (got with { RPLCPL } HEAD or some similar trick) is enough, but you must still run  $\rightarrow T0?$ .

To try it, enter the first few letters of any User RPL command. Press the key to which you assigned RPLCPL. If there was only one command starting with those letters, what you typed will be completed. If there were more than one, a choose box will appear from which you can select the desired command. The command line will be completed. This is something *really* useful.

Provided you have the extable library installed (as you should — see section 2), the completion also works for System RPL command names. If the last character in the string is a @ (as required by MASD), then System RPL completion is automatically used. (This is the only useful use of the @). As an added bonus, if you press the key to which RPLCPL is assigned longer, then the lookup of System RPL commands is done with GETNEAR (see section 2). You can then enter `case`, ask for completion, and get all words that have `case` in the middle — not only in the beginning. You don't need the KEYMAN library installed for this to work.

Another command that sometimes is useful is DYNCPL. It should also be assigned to some key, and also does completion. But it looks in the file you are editing for words that start with the typed letters. It is useful for the names of local variables and such. It works in a slightly different way: press the key, and the word will be completed with the first word. To accept it, press ENTER. To abort, press ON. To search for another match, press the same key that invoked DYNCPL. Any other key will accept the match and execute that key.

But wait — there is more! Another command that is useful to be assigned to a key is RPLED. This command imitates the ED command in the JAZZ library: it decompiles the object in level one, opens an editor for you to edit it, and, upon exit, recompiles the object (if you are lucky, that is. If the object cannot be compiled because of some MASD bug, use Right-Shift HALT to put the string in the stack, and exit with ON). A menu with useful operations (described below) is also displayed. If you call RPLED when in edit mode, the menu is redisplayed.

To get a description of all the commands in the menu, read the documentation that comes with Emacs. Here I will present the most

useful ones:

`CO..` calls `RPLCPL`. Left-shift `CO...` calls `DYNCPL`. See above for explanations of these commands.

With `ARG?` you can run a command several times. Press it, then enter the number of times you want the command to be repeated, followed by `ENTER`. Then, press the key. For example, `ARG?`, 4 `ENTER`, Backspace will delete four characters.

`F..ind` starts an incremental search. Press this key, then start typing the string you want to find. Type as many characters as necessary, then press `ENTER` to go to that cursor position. To cancel the search and go back to where the search started, press `ON`. Press the right arrow to find the next match.

`HALT` suspends the editor and goes back to the stack. To return to the editor, press `CONT` (Left-shift `ON`).

Pressing left-shift `Help` toggles between the minifont and the current font.

Right-shift `Help` is the menu of Emacs configuration. A choose box appears with several actions. Selecting `Options` will show a dialog, which allow you to configure two aspects of Emacs: whether the minifont is used by default, and whether the third page of the menu contains some templates for System RPL and Assembly Language development. There are also options to edit the `emacs` variable (described in section 7.1), to edit the `diagram` variable (used by the `SDiag` library, but this variable is not discussed in this document), and to make some key assignments for you.

`|→` indents the current line according to context. However, it is better to write the code already indented than to correct it later... Still, sometimes (such as when cutting and pasting), this can save some time. When left-shifted, removes `*` from the beginning of the current line (or all the selected lines), and when pressed right-shifted inserts the `*`.

`{↔}`, when pressed in a delimiter, jumps to the matching one. Works with `::` and `;`, `{` and `}` and a few others.

`( → )` shows the stack diagram for the entry point under the cursor. For this to work, the `SDiag` library, written by Denis Martinez and Carsten Dominik and distributed with Emacs, must be installed. Not all commands are documented, but this can be of great help.



DOB is a very, very useful command. If you have ever used JAZZ's ED editor, this works similarly to the Right-shift Y key. It disassembles the entry under the cursor, and its source is viewed in another editor. Exit this sub-editor with ON or ENTER to go back to the original editing section. Of course, you can call DOB again in the sub-editor. This command requires the CQIF? library (see section 6.1).

This command is also very useful, and, because of the similarity with the CQIF? command, is suitable to assignment to the same key. When it is called in edit mode, DOB is called. When not, CQIF? is called. It is very easy to create an assignment like this with KEYMAN (see section 3). First, put the list { DOB CQIF? } in the stack, and use OBJ→ or COMP→ to explode it. Drop the number of objects and run IfE?P. Use →TO? to add TakeOver to the object (since it needs to work in edit mode), and assign it to a key. If you have used the HP48 and JAZZ, Right-shift-hold +/- or Right-shift-hold 1/x will remind ED, and will not interfere with the normal operation.

Actually, the above is not really necessary. By default, DOB will call CQIF? when it is called outside edit mode.

The last page of the Emacs menu contains some templates for System RPL and Assembly Language programming. Try them, you will easily discover what they do.

If you need help with Emacs menu commands, just press Help. It will display a screen describing the two pages of the Emacs menu. Each page is represented by three rows of labels, which mean, from top to bottom, the unshifted action, the left-shifted action and the right-shifted action. Some commands are inverted, these have different actions when pressed longer.

## 7.1 Emacs Customization

Another feature the Emacs library shares with the GNU Emacs editor is the ability of customization: if present, the emacs variable in the HOME directory will be added to the Emacs menu.

This variable contains a list to define the menu, in a manner similar to the CST menus. The objects behave in a slightly different manner, though: a string is inserted at the cursor position, a program

is executed (you do not need to include TakeOver in this program), and an identifier is replaced by the contents of the variable. There can also be sub-lists of two elements: the first should be a string that will be displayed in the menu. The second element is the action to be executed, one of the object types discussed above. The second element can be another list, with actions for the unshifted, left-shifted and right-shifted presses, as in CST.

When a string is inserted in the editor, any control sequences in this string are interpreted. These control sequences form a macro language. All of them start with the | character. Here is a list of all these control sequences:

- |f Go forward one character.
- |b Go backward one character.
- |n Go to the next line.
- |p Go to the previous line.
- |a Go to the beginning of current line.
- |e Go to the end of current line.
- |< Go to the beginning of the file.
- |> Go to the end of the file.
- |F Go to the beginning of the next word.
- |B Go to the beginning of the current word, or to the beginning of the previous word if the cursor is already at the beginning of a word.
- |h Delete the previous character.
- |d Delete the character the cursor is on.
- |H Delete from the current position to the position |B would go.
- |D Delete from the current position to the position |F would go.

- |k Delete from the current position to the end of the line.
- |K Delete from the current position to the beginning of the line.
- |[ Set the start of the selection.
- |] Set the end of the selection.
- |C Clear the selection.
- |l Select the current line, excluding the final newline.
- |L Select the current line, including the final newline.
- |w Move the selection into the clipboard.
- |W Copy the selection into the clipboard.
- |z Delete the selection and append it to the clipboard contents.
- |Z Copy the selection and append it to the clipboard contents.
- |y Insert the contents of the clipboard.
- |m Insert a newline character.
- || Insert the bar character, |.
- |@ Show the current position with an arrow and makes a little pause.  
Used in debugging.
- |\* Mark a position. When the macro finishes, the cursor moves to this  
position.

## 8 Debugging

The debugging facilities for System RPL of the HP49G are the same as for User RPL: the built-in debugger (accessed by pressing **ORG**, that is, Left-shift **CAT**, **NXT** twice and **RUN**). Unfortunately, it does not work very well with some commands, which will be described later.

To start debugging, put the program or the name of the variable in which the program is stored in level one and press `DEBUG`. Then, use the other commands to examine the program. The `SST` command executes the next step in the program and displays what has just been executed. You'll need the System RPL stack (see section 4) active for this to be useful. If the command being run is a sub-routine, `SST` executes this as a single step. `SST↓` is similar, but if the command is a sub-routine, it steps into this sub-routine and executes its first command.

To see the next two actions of the program, but not execute them, press `NEXT`. To stop the program being debugged, press `KILL`. To make it resume its normal operation, press `CONT` (Left-shift `ON`).

To insert a breakpoint into your program, insert the command `HALT` (`xHALT` for System RPL programmers) in the program at the point you want the program to stop. Then use the commands above to debug the program.

The debugger does not work with commands that take arguments from the runstream, such as `'` or `IT`. Do not try stepping over one of these commands, the only thing you will get is a nice crash :-). Currently, the only way to debug these commands is by inserting `xHALT` after these commands, and using `CONT` to skip past the next `xHALT`.

## 9 Conclusion

This document has described the facilities the HP49G calculator has for programming, especially the ones useful for System RPL programmers. But, since these tools, although very good, are not enough, some tools written by other HP49G users have been described. With a good knowledge and efficient use of both the built-in and the third-party tools, the HP49G can be used as a complete programming environment for System RPL development.

I would like to thank the several people that, direct or indirectly helped me in writing the document: Joe Horn and Eric Rechlin for their description of the commands in library 256; Carsten Dominik for writing Emacs and SDiag and also for reviewing this document; Peter

Geelhoed for writing Emacs and reviewing the document; Denis Martinez for writing SDiag, Pierre Tardy for writing CQIF? and reviewing the document; and Wolfgang Rautenberg for writing OT49 and KEY-MAN, and for reviewing the document.