SYSTEM RPL for the HP48GX and HP49G

Authors: Timothy Ney & Roger Fraser

We have heard with much interest over the past 6 years of our experience with HP programming, many surveyors saying that they dislike the HP48/49 because of the speed problems. We have also found that many programs available to surveyors today, do not go along way to dispelling this perception. As a result, many surveyors continue to persevere with their HP42S or even a HP41.

Why does the calculator suffer from these perceived speed problems?

The answer lies in the language in which programs are written, that is UserRPL. UserRPL is the language that is documented in the calculator reference material and is readily programmed on the calculator through the use of << >> commands.

So, can the calculator be programmed to go faster?

The answer is yes, through a language known as SystemRPL or SysRPL for short. SysRPL is the built in language that is custom designed for the calculator's processor. UserRPL is actually a "subset" of SysRPL.

Why is SysRPL faster than UserRPL?

The main reason is that UserRPL commands have built in argument and error checking. In SysRPL, the programmer is responsible for all error checking and avoiding memory crashes. For example, if you know what type and how many arguments are required for a certain command or function, then it should not be necessary to perform error checking. This effectively increases execution time of the command. If you do this over several commands or even a program then execution be can somewhere around 10 times (this is a figure that we have heard a lot) fastest then UserRPL.

So, to get a good understanding of SysRPL, it is important to understand how a program is executed on the calculator. Essentially UserRPL and SysRPL are similar in that the calculator does not store the name of commands, only a series of memory addresses.

When a program runs, execution jumps to each address similar to a GOSUB statement. Some more execution may take place there or possibly jump to another address, eventually the execution will return to the original program and execution proceeds to the next address until the program is finished.

If the calculator however only stores addresses, then how can we edit UserRPL files? The calculator actually contains a table in memory that is used to cross reference the addresses to the UserRPL command *names*. When you view a UserRPL program, the information displayed is a readable form of the addresses. When you edit the program using UserRPL command *names*, the calculator searches the table for the corresponding address and constructs a program in memory containing that address.

What happens if there are addresses (or more commonly known as *entry points*) that do not have names that you can use on your calculator?

In fact there are over 3000 commands on the HP49G without names. How do we gain access to these entry points? The answer is SysRPL. It is worth noting here, that SysRPL is a

compiled language that requires special tools to develop programs. It can not be edited like UserRPL programs.

Why do we need these tools?

As we mentioned above, SysRPL commands do not have command *names*, this however is not quite true. The commands do have *names*, they are simply not stored on the calculator and therefore cannot be compiled internally like UserRPL programs. When you write a program, the SysRPL compiler searches for the names in an *entry point table* (this is a table that contains a series of command *names* and their corresponding addresses in the Calculators ROM) For example, the following table tells the compiler that when it encounters the command *name !!append$*, it should use the corresponding #623A0 address:

Example of part of a entry point table

```
=!!append$          EQU #623A0        * !!append$ equates to #623A0
=!!append$?         EQU #62312
=!!insert$          EQU #62394
=!append$           EQU #62376
=!append$SWAP       EQU #62F2F
=!insert$           EQU #622E5
```

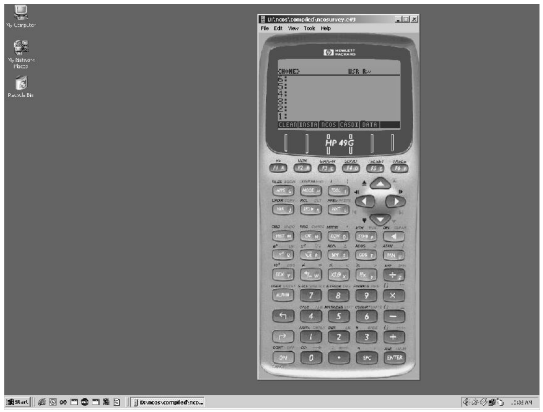
The compiler will develop a program into a series of addresses that execute one after another until the program is finished. This, as you can see, is similar to how the calculator internally compiles UserRPL programs.

What tools are available?

There are many tools available today to compile SysRPL program, ranging from PC to calculator based. A good place to start looking is at http://www.hpcalc.org. Our preferred option is the PC based HP-TOOLS (these tools are somewhat outdated now, but we are luckily enough to have access to a PC and prefer not to plug away at our HP49Gs'). However, this is purely a choice for the programmer. We have based the remainder of this article on the use of HP-TOOLS. Look for "HP Tools for WIN32 3.0.6" at http://www.hpcalc.org/hp48/pc/programming: This should be the latest version.

Before, we show you an example of a SysRPL program and how to compile it ready for the calculator; a programmer should have one more important tool. That is an *Emulator*. Exactly what is an emulator? This is a program that runs on a PC, that imitates a HP48/49 calculator. Programs can be tested by this emulator prior to uploading to your calculator to ensure that SysRPL programs are free from errors saving many crashes of your calculator. The emulator that we prefer is *EMU48.* You can download it from http://www.hpcalc.org. Installation instructions are included with the download file and are relatively easy to understand. We have sometimes written entire programs and tested them on the emulator before uploading to the calculator. Figure 1 shows EMU48 with an HP49 KML script.

Figure 1



So, what does a simple SysRPL program look like?

There are a several options available for writing and compiling a SysRPL program. One such option is to write the program source in a text file, compile the code using a HP compiler and then build the final binary file. You could simplify this procedure by using a batch file to automate the compiling and building of the final binary file. We discuss this later in the article.

The first step is to write some source code. Lets assume we want a simple HP49 program that takes a bearing and distance from the stack and converts the values into the corresponding departure and latitude (i.e. a Polar to Rectangular coordinate conversion). For this exercise, the program requires is a Distance on Stack Level two and a Bearing (in hours minutes and seconds) on Stack Level one. (Stack levels are simply the corresponding place that the objects sit on the stack) Figure 2 is a sample of source code.

Figure 2

```
ASSEMBLE
    NIBASC /HPHP49-C/
RPL
::                          ( defines the start of a source object )
  CK2&Dispatch              ( check if there are two real numbers on the stack )
  2REAL
  ::                        ( if so...)
    %HMS>                   ( convert the first value to decimal hours )
    %POL>%REC               ( convert bearing and distance to departure and latitude )
  ;
;                           ( ends the main source object )
```

So how does this program work?

Firstly, we need to look at the header information.

```
ASSEMBLE
    NIBASC /HPHP49-C/
RPL
```

All HP files contain header information (including UserRPL, but you do not see it) that lets the calculator know for which version of the HP the program was written. Since we are compiling and building source code on a PC, no header information is automatically added to the final file. Without this header information, files will not load correctly in the HP48/49. Therefore, the instruction at the top of the program, allow us to freely upload files to the HP.

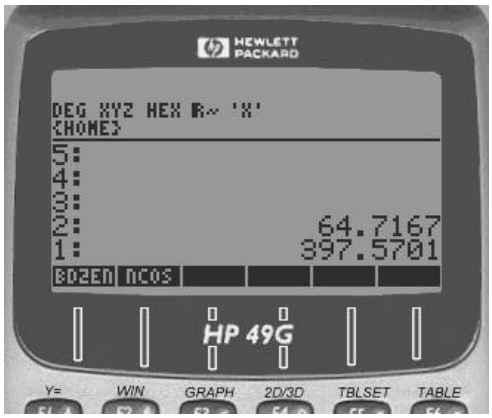
The next line contains the start of a program- shown as `::` (known as a secondary marker). The end marker is `;` this tells the calculator where the secondary begins and ends. As you can see, the program uses another secondary under **`2REAL,`** this again instructs the calculator to perform only the functions between the markers.

Where the real power of SysRPL lies is in the removal of error checking. In the example we have used a command **`CK2&Dispatch`**. This command checks for two arguments then dispatches their respective types, in this case 2 real numbers. Obviously, if we were to perform this conversion part way through program execution (and we were absolutely confident that we had two reals on the stack) we could exclude the commands `CK2&Dispatch`, `2REAL` and the secondary under `2REAL`.

The remaining commands are self-explanatory.

As an example, if we had 402.803 on stack level 2 and 80° 45' 16" on stack level 1 and we executed this program, we would end up with 64.7167 on stack level 2 and 397.5701 on stack level 1, as shown by Figure 3.

Figure 3



In up coming issues, we will show you how to create programs that perform the various tasks involved with survey calculation and data management.

So how do we create, compile and build a file suitable for the calculator?

1. The first step is to create some source code such as figure 2 and save it with an extension of .s. For example, if we have a program called *survey* then the source file would be *survey.s*. Why .s?, well this follows the general naming convention as follows:

EXTENSION	MEANING
.A	Saturn assembler source file
.EXT	External list generated by RPLCOMP
.H	Include file, used by RPLCOMP or SASM
.L	Saturn assembler list file
.LR	SLOAD output list file
.M	SLOAD control file
.MN	MAKEROM control file
.O	Saturn object code file (with Saturn header)
.OL	Loader output file
.S	RPL source file

2. The next step is to create a batch file that contains the necessary programs to compile and build the final file.

```
RPLCOMP SURVEY.S  SURVEY.A
SASM    SURVEY.A

SLOAD   -H B.M
```

   So how does the batch file work?

3. Firstly, we run **`RPLCOMP SURVEY.S  SURVEY.A.`** This will generate the file *survey.a*, which is the Saturn assembly source code file. The programs, RPLCOMP, SASM and SLOAD each have a +200 page document manual, that is beyond the scope of this article and the programmer should read this material if further information or clarification is required.

4. We now run **`SASM  SURVEY.A.`** This will generate two files *survey.l* and *survey.o*.

5. Create another file called `B.M` that contains the following

```
TITLE Binary File Compilation Program          * Title for file
REL SURVEY.O                                   * Input File
OUTPUT SURVEY.GX                               * Output File
LLIST SURVEY.LR                                * Log file
SEARCH ENTRIES49.O                             * Entry Point table
SUPRESS XREF
END
```

   With this file, we run **`SLOAD  -H B.M .`** This will produce the final binary file called *survey.gx*, that you could upload to your calculator / emulator. Ensure that you use the –H as this tells the program that an output file is required.

   You may have noticed the reference to the file *entries49.o*. This is a Saturn object code file of the entry point table discussed in the introduction to this article. Again, look at www.hpcalc.org for the latest version of the table. Many of the tables that you can download are in the form of *entries49.a* – A Saturn assembly source code file. To create a Saturn object code file (extension .o), we would use the same method as shown in 4, i.e. ***SASM entries49.a -> entries49.o***

6. Finally, you should look at the file, *SURVEY.LR*. If there are no errors, then your program should be ready. If not, you should return to the source code, correct the error and rerun the batch file.

So where do we go from here?

You can download several articles that can assist in learning System RPL. Some excellent documents include

Programming In System RPL: Eduardo Kalinowski, 1998
http://www.hpcalc.org/hp48/docs/programming/

Rplman.pdf - A guide to programming in systemRPL
http://www.hpcalc.org/hp48/docs/programming/

HP48GX / 49G Entry Reference: Carsten Dominik and Thomas Rast, 2002
http://zon.astro.uva.nl/~dominik/hpcalc/entries/

The authors are available any time to assist anyone who may want to develop SystemRPL programs or needs an application written for their calculator.

Who are the authors?

Timothy Ney

Currently a licensed surveyor employed with the Department of Natural Resources and Mines in Mackay, Queensland. Tim has 6 years experience in HP programming including UserRPL, SystemRPL and assembly (machine Code).

Timothy.Ney@nrm.qld.gov.au

Roger Fraser

Also a licensed surveyor with Natural Resources and Mines, Brisbane Queensland and is currently pursuing a PhD in Marine Spatial Data Infrastructure. Roger has over 7 years experience in writing HP calculator programs (UserRpl and SystemRPL) and has over 4 years experience in Windows software development (C, C++, VC, VB & Web), specialising in applications for the transition to GDA.

roger.fraser@nrm.qld.gov.au