Libraries

In the second instalment of programming for the HP48/49, we would like to discuss **libraries**. In particular, what are libraries, why should we use them and how do we create them?

Libraries are a collection of routines in a precompiled form that are loaded into the calculator's RAM and operate very similar to the calculators own "entry points" (commands). The precompiled form of a library consists of a library name, a hash table (contains a list of all of the routine names), a link table (contains the execution address of the routines) and an optional message table (used for generating user error messages).

What are the advantages of using a library? A good place to start is to compare a library to the implementation of a basic UserRPL program.

When we first began writing UserRPL programs, we determined reasonably quickly that a good programming technique was to break a large program into several smaller sub-programs, each stored as a separate variable on the calculator. This was for three main reasons:

1. The time taken to edit/view large files was painfully slow
2. With the lack of `GOTO` and `GOSUB` commands in UserRPL, you could reuse code by executing a variable several times.
3. As the programs became quite large, the source code was fundamentally easier and more economical to maintain.

Obviously, the main disadvantage was that the programs could not be hidden from the main menu and hence, the user could execute any of the variables, sometimes with unpredictable results. So how do we overcome this potential problem? The answer is though the use of libraries.

The main advantage of a library is that the user can only gain access to the commands the programmer wants accessed. Routines that are internal to a program and not required by the user are not shown on the menu. This effectively removes the potential to corrupt the program.

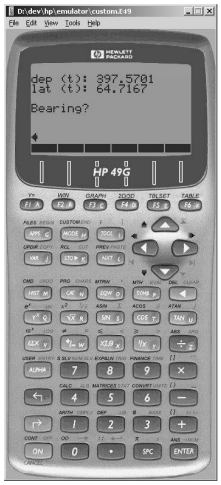
How the calculator actually performs the execution of the library using the hash and link tables is beyond the scope of this article. Instead, we will focus on how to create library routines and whether they will be accessible to the user.

How do we programmatically create library routines?

Three elements things tell the compiler (RPL compiler referred to in article 1) to turn the references (routines) into library ROM pointers (user entry points).

1. The directive `xROMID` (refer later in the article for more clarification) says to compile routines for additional user entry points and not for the calculators built in "entry points" (commands)
2. All library routines names are declared with either `xNAME`, which will show the name on the menu, or `NULLNAME` that simply creates the user entry point.
3. The compiler directive `EXTERNAL` identifies which of the `NULLNAME` routines will be compiled as an user entry points (ROM pointers)

To illustrate the concept of creating and calling library routines, lets revisit our simple polar to rectangular program: `BD2EN`. Suppose that we would like a repetitive program that asks the user for a bearing and distance. Each time the bearing and distance are entered in, the program will display the total departure and latitude. See figure below:



The input should consist of valid fractional numbers, where the bearing is in degrees, minutes and seconds, and the distance is in metres. Note that the distance, or magnitude, can be in any unit of measure. Upon displaying the output values, the program should be ready to take in new values and thus continue in a simple loop.

Firstly, we would structure the program as follows:

```
Clean the stack and initialise program environment.
Prompt user for Bearing and Distance.
While the user did not press cancel,
      Prompt for Distance.
      Check for valid data or *cancel*.
      Convert values.
      Display output and prompt user for new Bearing.
Restore stack and redraw display.
```

As an example, we shall write three transparent functions for handling the main loop, input and conversion. Ideally, we want to pass these functions some data that we *know* exists in the correct format. In this instance, they must be transparent from the user as no error checking is performed once they are called.

Before we start writing the main program, let us start with the input function, which is called `BDINPUT`. Since we may want to prompt the user for other values, we want to be able to use this function again without having to write duplicate code. Therefore, we shall pass it the string to be displayed to the user when asking for input. Note that the function does not actually have an argument calling convention. Instead, a SysRPL function takes the required value(s) from the stack.

```
************************************************************
* FUNCTION NAME:  BDINPUT
* INPUT:          User-input prompt string.
* OUTPUT:         Halts the program and waits for input. Returns
*                 TRUE if the user presses enter, FALSE if aborted
*                 by on/cancel.
************************************************************
ASSEMBLE
    CON(1)  8
RPL
NULLNAME  BDINPUT         ( use NULLNAME to hide the function from the user )
::
  NULL$         ( initial edit line )
  #ZERO#ONE     ( cursor position ie. at end, insert mode )
  BINT1         ( program/immediate entry )
  BINT2         ( alpha disabled )
  NULL{}        ( no menu )
  BINT1         ( initial menu row number )
  TRUE          ( pressing CANCEL will abort input )
  TWO           ( parse and evaluate the edit line )
  InputLine     ( prompt the user... )
```

```
;
*****************************************************************
```

Next, we want a function to convert the input values to the respective departure and latitude. The obvious name for this function is POLTOREC. We *know* that for this function to be called there must be two real numbers on the stack and thus, we do not need to perform any error checking.

```
*****************************************************************
* FUNCTION NAME:  POLTOREC
* INPUT:          Distance on level 2, bearing on level 1.
* OUTPUT:         Resultant departure and latitude.
*****************************************************************
ASSEMBLE
    CON(1)  8
RPL
NULLNAME  POLTOREC        ( use NULLNAME to hide the function from the user )
::
  %HMS>         ( convert the first value to decimal hours )
  %POL>%REC     ( convert bearing and distance to departure and latitude )
;
*****************************************************************
```

Since we are writing a program that functions around a *loop*, we can actually use the input function as an output function. To illustrate this, let us now write the main function of the program that controls the loop. So that we can display the total departure and latitude after each round, we need to use some local variables. The obvious names are Dep and Lat.

Here is the main program that controls the loop:

```
*****************************************************************
* FUNCTION NAME:  MAIN_
* INPUT:          Two named local variables
* OUTPUT:         NONE
* COMMENTS:       The main Program: Polar to Rectangular.
*****************************************************************
ASSEMBLE
    CON(1)  8
RPL
NULLNAME  MAIN_             ( use NULLNAME to hide the function from the user )
::
  %0 %0
  { LAM Dep LAM Lat } BIND   ( create local variables, initialised to zero )
  NULL$                      ( Put null string on stack so output can be displayed )
                             ( in the input prompt )
  BEGIN                      ( Begin the loop )
  ::
    $ "Bearing? " &$         ( Append bearing prompt to the string on the stack )
    BDINPUT                  ( Prompt the user for a bearing )
    ITE                      ( did the function return TRUE or FALSE? )
    ::
      CK1NOLASTWD            ( check for argument on stack )
      $ "Distance? "         ( Prompt the user for a distance )
      BDINPUT                ( create instance of InputLine )
      ITE                    ( did the function return TRUE or FALSE? )
      ::
        CK2&Dispatch         ( check that 2 reals have been entered in )
        2REAL
        ::
          SWAP POLTOREC                          ( convert to rectangular )
          LAM Dep %+ ' LAM Dep STO               ( store total Departure )
          LAM Lat %+ ' LAM Lat STO               ( store total Latitude )
          $ "dep (t): " LAM Dep a%>$ &$          ( display departure )
          $ "\0alat (t): " LAM Lat a%>$ &$ &$    ( display latitude )
          $ "\0a\0a" &$ FALSE                    ( add two carriage returns )
        ;
        ( at this point, the loop is ready to continue back to the start )
        ( the string formed above will be prefixed to the bearing input-prompt )
      ;
      :: DROP TRUE ;            ( *cancel* on distance input... exit MAIN_ )
    ;
    :: TRUE ;        ( *cancel* on bearing input... exit MAIN_ )
  ;
  UNTIL              ( repeat while valid data is entered - unless cancel is pressed )
```

```
  ABND              ( delete named local variables )
;
*****************************************************************
```

Note that before we call our function `BDINPUT`, we must leave a valid string on level 1 of the stack. This string acts like a stack (or command line) argument and must be present to correctly use the System RPL command `InputLine`.

The command `&$` concatenates two strings. On the first run, `&$` concatenates the predefined string `"Bearing? "` to a null string, given by the System RPL command `NULL$`. On the second pass the latitude and departure output string is left on the stack ready for the next concatenation of the input prompt string `"Bearing? "`.

Finally, we need to set up the program environment so we can trap any errors, initialise any system flags and, upon exit, restore the original stack display. Here is the code that does just that:

```
*****************************************************************
* FUNCTION NAME:  BD2EN
* INPUT:          NONE
* OUTPUT:         NONE
* COMMENTS:       The main Program: Polar to Rectangular.
*****************************************************************
ASSEMBLE
    CON(1)  8
RPL
xNAME  BD2EN
::
  CK0             ( Program accepts no input )
  SETDEG          ( degrees mode )
  ZERO SetHeader  ( turn off header )
  ClrDA1IsStat    ( Turn off clock )
  RECLAIMDISP     ( clear and resize the display )
  TURNMENUOFF     ( turns current menu off )
  ( set system flags. This may be written as a separate routine )
  ::
    BINT95              ( set rpn mode )
    NINETEEN            ( ->v2 yields a vector )
    SEVENTEEN           ( not in radians mode )
    EIGHTEEN            ( Degrees mode )
    BINT5 BINT1         ( start = 1, finish = 4 )
    DO ClrSysFlag LOOP  ( clear the system flags )
    105 SetSysFlag      ( approx mode on )

  ;
  ERRSET             ( set up environment to trap any errors )
  ::
    MAIN_            ( call our main function )
  ;
  ERRTRAP         ( exit on any error )
  GARBAGE         ( clear memory )
  TURNMENUON      ( turn menu back on )
  RECLAIMDISP     ( resize and clear display )
  ClrDAsOK        ( redraw display )
  BINT1 SetHeader ( redraw default header to one row )
;
*****************************************************************
```

So how do we create a library?

The process of generating a library is not simple, so the general file structures and procedures used to generate a library are illustrated below.

To create a library, we must

- Compile and assemble the source code
- Use the HP Tool **MAKEROM** to create the head, hash, end and loader control file
- Use the control file to build the library in combination with the source code
- Add a binary header to the front of the library and fill in the references to the "entry poi nt table"

We begin with the source code.

The source code **MUST** contain some configuration information that lets the compiler know that we are creating a library, what to do when we install it on our calculator, a title and a unique ID number.

Firstly, create a new source file, for example "Survey.s", and add to the start of the source the following configuration information:

```
TITLE   Survey Library v2.0      (  Title of library )
xROMID 2FC                      ( The unique ID number )
ASSEMBLE
     =SUROMID EQU #2FC
RPL

HIDDEN ROUTINES ***************************************************************

EXTERNAL  BDINPUT             ( Declare the library subroutines )
EXTERNAL  MAIN_
EXTERNAL  POLTOREC

ASSEMBLE
=Sucfg                        ( Routine that's lets the compiler know what to do after
                                          installation )
RPL
    ::
        DOBINT SUROMID
        XEQSETLIB              ( autoattach library )
    ;
```

Next, paste in the required routines as given by the aforementioned source code after this information.

Compiling the source code.

To compile this source code file, we undertake those steps as described in our first article. That is:

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

Here an additional output file has been created; "SURVEY.EXT". Upon running the program RPLCOMP, this file contains *declarations* for all routines that have been *defined* in the source code. As mentioned in the first article, all of the procedures can be simplified by using a batch file. A complete listing is shown at the end of this article.

Next we create the library with MAKEROM.

What is MAKEROM?

At this point in the construction of the final file, we must branch away from the procedure for compiling a single binary file (as described in our first article) and introduce the program required to create the library, that is **MAKEROM.** Thus the next line of your batch file would contain:

```
MAKEROM SU.MN SU.M
```

Two more files have been now introduced, "SU.MN" and "SU.M". This is why the creation of library is no trivial matter. These files are basically the input and output from MAKEROM. A sample input SU.MN is given by the following:

```
TITLE Survey Library Compiler SU.MN       ( Title of Input file )
OUTPUT SU.O                               ( Directs the Saturn output code to SU.O )
LLIST SU.LR                               ( Log File )
CONFIGURE Sucfg                           ( Location of configuration information )
NAME SULIB :Survey Program                ( Name of library )
ROMPHEAD SUHEAD.A                         ( Start )
REL SURVEY.O                              ( Location of the source code )
TABLE SUHASH.A                            ( Hash Table )
FINISH SUEND.A                            ( End )
END
```

The output from the program **MAKEROM** may look like the following (Note that the lines containing the ** are comment lines only).

```
TITLE Survey Library Compiler SU.MN       ( Title of Input File )
OUTPUT SU.O                               ( Location of the Saturn object code file)
LLIST SU.LR                               ( Log file )
**CONFIGURE SUcfg
**NAME SULIB :Survey Program
**ROMPHEAD OUTPUT/SUHEAD
REL SUHEAD.o                              ( Location of the header code)
** Parsing OUTPUT/SURVEY.EXT
REL SURVEY.O                              ( Location of the source code )
**TABLE SUHASH.A
REL SUHASH.o                              ( Location of the hash table )
**FINISH SUEND.A
REL SUEND.o                               ( Location of the end code)
END
```

A special note is made here about the **NAME**. The name "SU LIB :Survey Program" is what will appear in the menu label in the library menu. To correspond with other library names, the first field should be short enough to fit in a menu label, and can be followed by a space and whatever text you like.

The next step is to generate the Saturn object code for the three files generated by **MAKEROM** that is: SUHEAD.A, SUHASH.A and SUEND.A. This is simply done by:

```
SASM SUHEAD
SASM SUHASH
SASM SUEND
```

Two final steps are required to finalise the construction.

```
SLOAD SU.M
SLOAD -H S.M
```

Running **SLOAD** with the output from **MAKEROM** produces a final binary file that is ready to be linked to the internal routines of the HP49G, for example the 'entry point table'

The last step in the creation of a library is accomplished with yet another control file, **S.M**

```
TITLE Survey Library Compiler S.M           ( Title of this file )
OUTPUT SULIB.LIB                            ( Output file )
```

```
OPTION CODE                                 ( Options )
LLIST SULIB.LR                              ( Log file )
SUPPRESS XR
SEARCH ENTRIES49.O                          ( Entry point table )
REL BINHD49.O                               ( Include header info )
REL SU.O                                    ( Location of Saturn object code file )
CK LIB2BC SYSEND2BC                         ( Check sum )
```

The header information "BINHD49.O" can be created by simply undertaking those steps described in our first article. For example, create a file called "BINHD49.S" that contains the following code:

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

Compile with

```
RLPCOMP BINHD49.S  BINHD49.A
SASM    BINHD49.A
```

Output will be a file "BINHD49.O"

Putting it all together

To summarise the basic steps in the creation of a library, a sample batch file listing is illustrated below:

```
RPLCOMP SURVEY.S OUTPUT/SURVEY.A OUTPUT/SURVEY.EXT
SASM OUTPUT/SURVEY

MAKEROM SU.MN SU.M

SASM OUTPUT/SUHEAD
SASM OUTPUT/SUHASH
SASM OUTPUT/SUEND

SLOAD SU.M
SLOAD -H S.M
```

Output from the batch file, used in conjunction with the two control files will be a single file called "SULIB.LIB". This file can now be freely uploaded to your calculator or Emulator and installed as a library. When you cold start the calculator it will automatically be attached and ready for use.

Timothy Ney & Roger Fraser