Enabling User Interaction – Advanced Menu and Keyboard Input

Designing a simple yet comprehensive user interface to handle data entry, whilst also providing the user with the most efficient means for interacting with the program, is perhaps the most difficult task of writing a program. Once you've got this part of the program sorted out, implementing the computational functionality of your program is in most cases somewhat trivial.

In our last article, we introduced the Parameterised Outer Loop (POL). POLs are very efficient, however the standard POL provides the developer with a very general means for user interaction. In this article, we will extend the standard POL to take full advantage of the available user input options, and thereby provide you with an understanding of how you can build advanced functionality into your user interface.

Our example code will demonstrate how to set up a simple, context-sensitive interface that can be used to accept data for our ongoing survey program.

An Advanced User Interface

POLs are a very general construct and enable the developer to select from a variety of options for accepting data input, such as Prompts, Input Forms and Browsers (or Choose Boxes). For this reason, POLs require elaborate arguments. As stated in the previous article, the POL requires nine inputs, those being:

1. Screen Display
2. Key Handlers
3. Key Handle Flag
4. Standard Handle Flag
5. Menu
6. Initial Menu Row
7. Run-time Environment Flag
8. Exit Condition
9. Error Function

Despite the large number of inputs, we actually only require three inputs from the stack to set the context within which we require data entry. The remaining inputs for the POL are simply standard inputs. So to take full advantage of the available user interface options provided within the System RPL design space, we really should take a look at how POLs actually work.

The SystemRPL command `ParOuterLoop` itself decompiles to a set of calls to other SystemRPL commands as shown in Figure 1 (note the appropriate error handling). None of the commands return anything, and the only one that takes any arguments is `POLSetUI`, which requires the same nine arguments required by `ParOuterLoop`.

Figure 1 `ParOuterLoop`

```
::
  POLSaveUI  ( save current user interface in a temporary environment )
  ERRSET     ( start error trap )
  ::
    POLSetUI ( set new user interface, according to the parameters given )
    POLKeyUI ( Displays, reads and evaluates keys. Handles errors and )
             ( exits according to the user interface specified by POLSetUI )
  ;
```

```
  ERRTRAP
  POLResUI&Err ( if an error happened, restore )
               ( the saved interface and error )
  POLRestoreUI ( Restores the user interface saved by POLSaveUI and )
               ( abandons the temporary environment )
;
```

Therefore, if we plan to use the same key handlers, exit condition and error functions for our entire user interface (regardless of what data we require), then we could write our own POL that is flexible enough to be adapted to any data context. For the example program we have been working with (`BD2EN`), the two contexts for our data entry are bearings and distances. Hence, we should consider an interface (and context menus accordingly) that provides additional functionality depending on whether we are entering in bearings *or* distances.

We've called our POL function `iEditor`. The arguments required for `iEditor` are the title string to be displayed (or user prompt), the initial value to be entered, and the context menu specification. Therefore, each time we need to create an instance of a new user interface, we onlly place 3 arguments on the stack and then call `iEditor`. But, before we look at the code for `iEditor`, we'll begin with the arguments it requires.

Arg 1: The Title String

Firstly, we need a title string to prompt the user for the data we actually require. This can be done simply by placing a sting on the stack, for example:

```
$ "Enter bearing: "
```

If we would like to append the last entered bearing to the end of this string, then we could declare a global variable initially at run-time, initialise it each time a new value is entered, and then append it as the title string is displayed. Appending the value is performed as follows:

```
$ "Enter bearing: " LAM g_Brg a%>$ &$
```

Arg 2: The Context Menu

In our last article, we described how we could handle menu interaction through the use of graphic objects (grobs) and keyboard handlers. In this example, we will define the menus using a list of strings and objects, which each define a menu item caption and the functionality to be executed (upon selecting the menu item) respectively.

For bearing input, our context menu has been designed to allow cancelling, accepting and the deletion of the current input, setting the last bearing entered, reversing (180±) and converting between decimal degrees and degrees, minutes and seconds. Of course, you could provide further functionality if you require, such as data addition and subtraction options. Figure 2 defines the code for our bearing input context menu. Note that it has two rows.

Figure 1 `brgContextMenu`

```
*****************************************************************
* FUNCTION NAME:  brgContextMenu
* INPUT:          NONE
* OUTPUT:         Menu specification as a 'list'
* COMMENTS:       Requires global variable g_Brg
*****************************************************************
ASSEMBLE
    CON(1)  8
RPL
NULLNAME brgContextMenu
::
```

```
    {
      { "DEL"  :: TakeOver DODEL.L ; }
      { "LAST" :: TakeOver InitEd&Modes LAM g_Brg a%>$ CMD_PLUS ; }
      { "REV"  :: TakeOver RCL_CMD DUPLEN$
                  #0=ITE
                  :: DROP LAM g_Brg ;       ( NULL? recall last input bearing )
                  :: palparse NOTcase 3DROP ;
                  CKREAL %180 %+ 1REV %MOD BINT4 DOFIX
                  a%>$ InitEd&Modes CMD_PLUS
                ; }
      NullMenuKey
      { "DMS->" :: TakeOver RCL_CMD DUPLEN$
                   #0=ITE
                   :: DROP LAM g_Brg ;
                   :: palparse NOTcase 3DROP ;
                   CKREAL %>HMS a%>$ InitEd&Modes CMD_PLUS
                 ; }
      { "->DMS" :: TakeOver RCL_CMD DUPLEN$
                   #0=ITE
                   :: DROP LAM g_Brg ;
                   :: palparse NOTcase 3DROP ;
                   CKREAL %HMS> a%>$ InitEd&Modes CMD_PLUS
                 ; }
      {  :: TakeOver "INS" INSERT? Box/StdLabel ;
         :: TakeOver Tog.Insert SetDA12NoCh ; }
      <SkipKey
      >SkipKey
      { "INFO"  :: TakeOver DOTEXTINFO ; }
      { "CANCL" :: TakeOver iEditorCANCL ; }
      { "OK"    :: TakeOver iEditorOK ; }
    }
;
```

Note that the menu is a list that generally follows the structure:

{ MenuItem1 MenuItem2 MenuItem3 … MenuItemN }

where each menu item can be one of the following:

- NullMenuKey
- KeyObj
- { Label KeyProcNS }
- { Label KeyProcNS KeyProcLS }
- { Label KeyProcNS KeyProcLS KeyProcRS }

Label	is the object (either a string or a 21*8 GROB) to be displayed as a label
KeyProc	is the action to be taken upon a key press.
NS, LS, RS	are No-Shift, Left Shift and Right Shift when the key is pressed.

If *KeyProc*, is a sub program with *TakeOver* as the first command, the sub program will override the normal execution until the sub program is complete. It is through this functionality that we are able to change the value on the stack or in the editor without actually exiting the POL.

You will notice in the menu specification many commands that relate the internal editor of the HP49G such as CMD_PLUS. An explanation of the editor is beyond the scope of this article and the reader is encouraged to refer to the document "*Programming in SystermRPL (Second Edition), 2002* by *Eduardo de Mattos Kalinowski & Carsten Dominik"* for more information on menu specifications and the editor commands. This document can be downloaded at www.hpcalc.org

For distance input, we have created a context menu that caters for converting between links and metres. Hence, you could expand this menu to include *any* functionality you require.

Figure 2 distContextMenu

```
*****************************************************************
* FUNCTION NAME:  distContextMenu
* INPUT:          NONE
* OUTPUT:         Menu specification as a 'list'
* COMMENTS:       Requires global variable g_Dist
*****************************************************************
ASSEMBLE
    CON(1)  8
RPL
NULLNAME distContextMenu
::
    {
       { "DEL" :: TakeOver DODEL.L ; }
       { "LAST" :: TakeOver InitEd&Modes LAM g_Dist a%>$ CMD_PLUS ; }
       { "L->M" :: TakeOver RCL_CMD DUPLEN$
                   #0=ITE
                   :: DROP LAM g_Dist ;       ( NULL? recall last input distance )
                   :: palparse NOTcase 3DROP ;
                   CKREAL % 0.201168 %* BINT3 DOFIX
                   a%>$ InitEd&Modes CMD_PLUS
                 ; }
       {  :: TakeOver "INS" INSERT? Box/StdLabel ;
          :: TakeOver Tog.Insert SetDA12NoCh ; }
       { "CANCL" :: TakeOver iEditorCANCL ; }
       { "OK"    :: TakeOver iEditorOK ; }
       <SkipKey
       >SkipKey
       NullMenuKey NullMenuKey NullMenuKey
       { "INFO" ::  TakeOver DOTEXTINFO ; }
    }
;
```

Arg 3: The initial value

If we want to place an initial value in our editor, we simply put the value's string representation on the stack. Otherwise, you can use a null string.

The Editor

The code for the POL editor is shown in Figure 3.

Figure 3 iEditor

```
*****************************************************************
* FUNCTION NAME:  iEditor
* INPUT:          Title String, Menu specification, initial value
* OUTPUT:         POL environment
* COMMENTS:       an advanced POL environment for handling context
*                 sensitive menus and custom functionality
*                 within the standard editor.
*****************************************************************
ASSEMBLE
    CON(1)  8
RPL
NULLNAME   iEditor
::
  POLSaveUI
  ERRSET
  ::
    DUP
    InitEdLine                      ( Clear Editline )
    CMD_PLUS                        ( Insert String )
    BINT0 SetCursor                 ( Set Cursor Position )
    INSERT_MODE                     ( Default to insert Mode )
    SetPrgmEntry                    ( Set Program Entry Mode )
    FALSE                           ( Define Exit condition )
    4NULLLAM{} BIND                 ( Bind Editor Variables )
    ' ::
      DA2aOK?NOTIT                  ( Stack area invalid; redraw )
      ::
```

```
        BlankDA12                       ( Prepare Screen )
        ZERO SEVEN 131 SEVEN LINEON     ( draw a line on the screen )
        4GETLAM                         ( Retrieve title string )
        CODE                            ( define scope for assembly )
          GOSBVL  =PopASavptr
          C=A     A
          GOSBVL  =GetStrLenC
          ST=0    11
          GOSBVL  =D0->Row1
          GOSBVL  =MINI_DISP          * Display title string
          GOVLNG  =GETPTRLOOP
        ENDCODE
      ;
      DispCommandLine                 ( Draw CommandLine )
      DA3OK?NOTIT ?DispMenu           ( Draw Menu )
      ClrDAsOK
    ;
    ' ::
      DUP BINT1 #<> SWAP BINT4 #<> AND casedrpfls
      FORTYSEVEN ?CaseKeyDef
      :: TakeOver iEditorCANCL ;
      FIFTYONE ?CaseKeyDef
      :: TakeOver iEditorOK ;
      FIFTY ?CaseKeyDef
      :: TakeOver $ "SPC Key Not Used" FlashWarning ;
      DROPFALSE
    ;
    TrueFalse               ( 3. Key handle flag and 4. Standard handle flag )
    3GETLAM                 ( 5. Menu Row )
    ONEFALSE                ( 6. No suspended env's and 7. Run time env flag )
    ' 1GETLAM               ( 8. Exit condition )
    ' SysErrorTrap          ( 9. Error function )
    POLSetUI
    ClrDAsOK
    POLKeyUI
    ABND
  ;
  ERRTRAP
  ::
    DEL_CMD
    ?ClrAlg
    POLResUI&Err
  ;
  POLRestoreUI
  InitEd&Modes
  ClrDAsOK
;
```

Let's take a minute to see what we're doing here. Remembering that prior to calling `iEditor`, we placed 3 arguments on the stack. Then, after we define our error condition (`FALSE`) inside `ERRSET`, we bind the four stack items to four null lams (so that we have access to these objects later) by:

```
4NULLLAM{} BIND
```

Having set up the initial objects, we now begin to define the nine inputs required for `POLSetUI`.

1. Screen Display

The screen display is given by the following object, defined by:

```
` ::
    DA2aOK?NOTIT          ( Only redraw if stack area invalid)
    ...                             ...

    ClrDAsOK              ( Refresh the screen
;
```

Included in this screen display object is a small section of *assembly* (defined within `CODE` and `ENDCODE` words) to display the title string in MINI_FONT in the top left hand corner of the screen. Assembly is a programming language in itself and is well beyond the scope of these articles. It is used here to assist in the 'speeding up' of the refresh rate after each key press.

2. Key Handlers

As the POL will respond to the key strokes A-F (top six keys as defined by our bearing and distance context menu specifications), only two keys from the keyboard require their actions to be processed - those being the ENTER and CANCEL keys:

```
` ::
    DUP BINT1 #<> SWAP BINT4 AND casedrpfls
    FORTYSEVEN ?CaseKeyDef :: TakeOver iEditorCANCL ;
    FIFTYONE ? CaseKeyDef :: TakeOver iEditorOK;
    DROPFALSE
  ;
```

The remaining 7 inputs are described in sufficient detail by the comments in the code for `iEditor` above.

Note that two more functions are required to deal with the actions associated with processing the ENTER & CANCEL keys (whether through the menu or from the keyboard). Figures 4 and 5 outline the code for these functions.

Figure 4 `iEditorCANCL`

```
*****************************************************************
* FUNCTION NAME:  iEditorCANCL
* INPUT:          NONE
* OUTPUT:         Exit condition for the POL
*****************************************************************
NULLNAME iEditorCANCL
::
     DEL_CMD FalseTrue 1PUTLAM
;
```

`iEditorCANCL` clears the editline, places "false" on the stack and sets the "exit condition" to true therefore ending our POL editor. Why do we place a "false" on the stack? Well it is always a good idea after executing a function to return a flag as to whether we were successful or not. Obviously, if we are unsuccessful then we not wish to continue with the manipulation of the string. For example, if we were editing a bearing and the user pressed CANCL then there is little point in "storing" or updating the current bearing.

Figure 5 `iEditorOK`

```
*****************************************************************
* FUNCTION NAME:  iEditorOK
* INPUT:          NONE
* OUTPUT:         Parses the editor string, then leaves a
*                 real and the exit condition for the POL
*****************************************************************
ASSEMBLE
    CON(1)  8
RPL
NULLNAME iEditorOK
::
     Parse.2
     NOTcase
     ::
       5ROLLDROP
       ParseFail2
     ;
     SWAPDROPTRUE
     TRUE 1PUTLAM
;
```

This function processes the results from the editor and tries to convert the object in the editor to a string if necessary. If this is not possible, an error message is displayed and our POL editor is not exited. If it is successful, then no further action is required and the parsed value is placed on the stack.

So that we can incorporate the POL editor into our BD2EN program, we need to slightly modify our MAIN_ function. We have also made MAIN_ more responsive to keying in invalid data, such that it will continue looping around asking us for new data. Also two separate loops (one for bearing and distance) are run until valid data is entered or until the user presses CANCEL. Figure 6 shows the amended code.

Figure 6 MAIN_

```
*****************************************************************
* FUNCTION NAME:  MAIN_
*****************************************************************
ASSEMBLE
    CON(1)  8
RPL
NULLNAME  MAIN_              ( use NULLNAME to hide the function from the user )
::
  BEGIN                      ( begin data entry loop )
  ::
    BEGIN                        ( Begin the brg input loop )
    ::
      $ "Enter bearing: " LAM g_Brg a%>$ &$     ( append last brg to prompt )
      brgContextMenu                            ( prepare the context menu )
      NULL$ iEditor                             ( initial value is NULL )
      ITE
      ::
        CheckReal                ( return value holds next test condition )
        ITE
        ::
          DUP ' LAM g_Brg STO    ( store global Bearing )
          TrueTrue               ( exit this loop and continue )
        ;
        :: FALSE ;               ( invalid data? continue loop )
      ;
      :: TRUE FalseTrue ;        ( cancel? exit this loop and all input )
    ;
    UNTIL
    IT
    ::
      BEGIN                      ( Begin the dist input loop )
      ::
        $ "Enter distance: " LAM g_Dist a%>$ &$ ( append last dist to prompt )
        distContextMenu                         ( prepare the context menu )
        NULL$ iEditor                           ( initial value is NULL )
        ITE
        ::
          CheckReal              ( return value holds loop condition )
          ITE
          ::
            DUP ' LAM g_Dist STO ( store global Distance )
            TrueTrue             ( exit this loop and continue )
          ;
          :: FALSE ;             ( invalid data? continue loop )
        ;
        :: DROP TRUE FalseTrue ; ( exit this loop and all input )
      ;
      UNTIL
      IT
      ::
        SWAP POLTOREC                       ( convert to rectangular )
        LAM g_Dep %+ ' LAM g_Dep STO            ( store total Departure )
        LAM g_Lat %+ ' LAM g_Lat STO            ( store total Latitude )
        FALSE  ( continue data input )
      ;
    ;
  ;
  UNTIL
;
```

Note that to use the LAMS g_Brg and g_Dist, we need to declare them within BD2EN as follows:

```
%0 DUP
{ LAM g_Brg LAM g_Dist } BIND   ( create global variables, initialised to zero )
```

The function CheckReal is provided here in assembly for run-time efficiency (it is worth noting here that our programs may run up to 100 times faster if we were to code them entirely using Assembly!). The function attempts to convert the string returned from the editor to a real, and upon success returns TRUE or FALSE.

```
*****************************************************************
* FUNCTION NAME:  CheckReal
* INPUT:          an object on level 1
* OUTPUT:         a real and TRUE (if the object is a real),
*                 otherwise FALSE
*****************************************************************
ASSEMBLE
    CON(1)  8
RPL
NULLNAME CheckReal
::
        ( check for valid object )
    CODE
      GOSBVL  =SAVPTR
      A=DAT1  A                              * Read Address of Object on Stack
      D1=A                                   * Point to that address
      A=DAT1  A                              * Read Prolog
      LC(5)   =DOREAL                        * Load Real Prolog
      ?A#C    A                              * Do we have a Real Object?
      GOYES   +
      GOVLNG  =GPPushTLoop                   * Push TRUE if we have a real number
+     GOVLNG  =GPPushFLoop                   * Push FALSE if it fails
    ENDCODE
    ITE
    :: TRUE ;                                ( Push TRUE if it is a real )
    ::
      ERRSET
      :: CKREAL TRUE ;                       ( Can we convert object to real? )
      ERRTRAP
      :: DROP FALSE ;              ( Push FALSE if we can not convert object )
    ;
;
```

Close

The resulting BD2EN program now boasts an extremely efficient, context sensitive user interface. Our first POL (described in our last article) defines the scope for the main menu, and our new POL editor provides data-centric functionality for entering in bearings and distances. Figures 7 and 8 show the interfaces for the bearing and distance input editors respectively.

The efficiency and effectiveness of System RPL must not be underestimated. Developing the same interface functionality as described herein using User RPL is very cumbersome and of course, VERY SLOW! Using the POL (either via an advanced interface or not), your current HP survey program could be up to 12 times faster, whilst at the same time giving you the ability to perform additional functions on your input data.

Figure 7 Bearing input editor

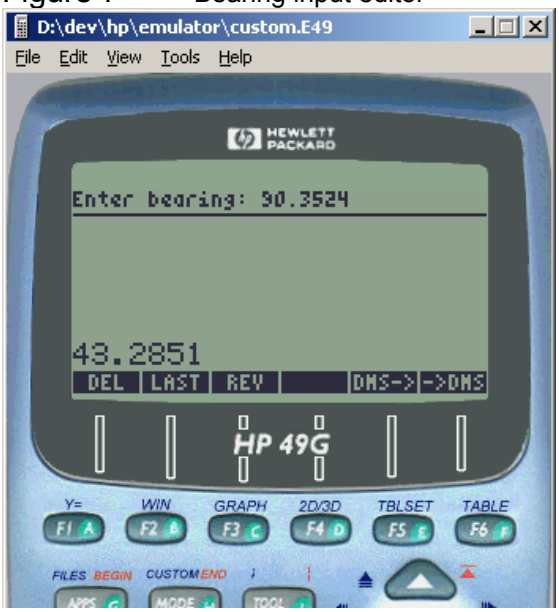

Figure 8 Distance input editor