Enabling User Interaction – Integrating Menus and the Keyboard

You've probably noticed in just about every software application you've ever installed a method for handling commands. In many windows applications, you will find that commands can be instigated by events such as keyboard shortcuts (accelerators), menu items, buttons, mouse events (such as right click, drag/drop and double-click), timers, and even a USB connection. Depending on the function of the application, each of these events will be managed by the appropriate command handler and in turn, a single action or sequence of commands will be performed. With HP programming, the two most common forms of interacting with a program is the keyboard and on-screen menu.

If you were like us, the first thing you did when you removed your HP out of the box was create your own custom menu and keyboard shortcuts using the standard CST and `USER` keys methods. Just looking at the standard HP menus and keyboard no doubt made you feel cluttered by the number of functions that are available to you, and perhaps frustrated that there was so much of little use to your business needs. Persevering one afternoon with a stiff coffee and your user manual, you soon enough had direct access to many system commands and via a single key press. Your computing world was one bit simpler.

So how do you incorporate these into a System RPL program? How are the desired menu keys drawn to the screen? Where on the screen can I put them? How do you handle hierarchical menus? How can these menu commands be mapped using the keyboard? What about handling a single key press? What about a right-shift (green) or a left-shift (purple) key press? With a small amount of effort, access to your favourite program commands can be handled using some relatively simple programming techniques.

Parameterised Outer Loop – the Command Handler

Knowing exactly when a key has been pressed is not a simple process. Writing the necessary code to manually poll or watch all sources of input for the appropriate input command is a complex task and will usually involve some assembly programming. To handle the input messages correctly, you will need write a program that runs in a continuous loop to process every user input action, testing what key was pressed and then handling it with the appropriate command. Fortunately, System RPL provides an easy, high-level method for input and command handling in a very simple loop that won't compromise your processing power – the *Parameterised Outer Loop:* `ParOuterLoop`.

Parameterised outer loop functions just like a Windows message map. It is a structure that enables you to create a comprehensive program that can receive keystrokes and perform different actions based on the key that was pressed. The loop is repeated infinitely until an exit condition or a system command is sent, such as the standard key commands `CANCEL` or `DROP`. Basically, you simply determine which keyboard messages (commands) you want to trap and then define the function(s) to handle that command.

Using a parameterised outer loop to handle your program input requires you to state nine input arguments. These are described as follows:

1. **Screen Display** — This object is evaluated before each key is evaluated and should be used to refresh the screen (where the keys do not), local variable initialisation, and any other special error conditions.

2. **Key Handlers** — This object sets up the custom key mapping to override the functionality of a particular key. These objects may return values to the stack.

3. **Key Handle Flag** — `TRUE`: Non assigned keys perform their normal action.
   `FALSE`: The keys not assigned are cancelled.

4. **Standard Handle Flag** — `TRUE`: Standard key definitions are used for non-application keys. `FALSE`: The key processing defaults are used.

5. **Menu** — The menu specification. There are a few ways of handling menu input. For this issue, we will use the method of using key assignments to handle menu buttons, whereby a standard menu label is created using graphic objects (grobs).

6. **Initial Menu Row** — If a single menu level spans more than one row (ie. contains more than six items), this argument specifies the row which should appear first. Hence, this should be `ONE`.

7. **Run-time Environment Flag** — `TRUE`: If any command or object evaluation forces a suspended environment or restart, the loop instead sends an error message. `FALSE`: The command or object is evaluated as normal.

8. **Exit condition** — This object is evaluated before each update or key evaluation, and determines whether the loop should continue. If the result is `TRUE`, the loop exits.

9. **Error Function** — The definition (or code body) of the object to be evaluated when an error message is sent.

After placing these arguments on the stack, execute the command `ParOuterLoop`. The command does not return any results to the stack.

Lets look at the code to use `ParOuterLoop`. Using our program to convert polar to rectangular coordinates, we'll write a command handler for processing some keyboard commands using a simple menu. Logically, we'll call it `COMMAND_HANDLER`. Basically, we want Key A to initialise a vector list and run a loop that prompts the user for bearings and distances, Key B to display the resultant missing bearing and distance, and Key C to display the equivalent departure and latitude.

```
************************************************************
* FUNCTION NAME:  COMMAND_HANDLER
* INPUT:          Nil
* OUTPUT:         Error on Exit.
* COMMENTS:       Sets menu display and key conditions.
************************************************************
ASSEMBLE
  CON(1)  8    * Tell parser 'Non algebraic'
RPL
NULLNAME  COMMAND_HANDLER
::
  GARBAGE                                          ( clear memory )
  FALSE  { LAM exit } BIND                ( exit condition variable )
  DEFINE kpNoShift  ONE       ( single key press in un-shifted plane )
  DEFINE kpLeftShift TWO   ( Purple key press for left-shifted plane )
  DEFINE kcMenuKey1 ONE                     ( A Key, Row 1, Column 1 )
  DEFINE kcMenuKey2 TWO                     ( B Key, Row 1, Column 2 )
  DEFINE kcMenuKey3 THREE                   ( C Key, Row 1, Column 3 )
  DEFINE kcMenuKey6 SIX                     ( F Key, Row 1, Column 6 )

  ( set up the main keyboard buttons - for ease of reading )
  ' ::
    ( 1. Draw the screen display and menu buttons - using standard menu grobs. )
    ( See the following section on Menus for the definition of MENU_BUTTONS )
    MENU_BUTTONS_
  ;
  ( 2. set up the key handlers )
  ' ::
    kpNoShift #=casedrop       ( process un-shifted plane )
    ::
      kcOn  ?CaseKeyDef        ( was the On Key pressed? )
      ::
```

```
        TakeOver TRUE          ( exit ParOuterLoop )
        ' LAM exit STO         ( store exit condition )
      ;
      kcMenuKey1 ?CaseKeyDef   ( was the A Key pressed? )
      ::
        TakeOver MAIN_         ( run our main loop for data input )
      ;
      kcMenuKey2 ?CaseKeyDef   ( was the B Key pressed? )
      ::
        TakeOver MBD           ( compute missing bearing & distance )
      ;
      kcMenuKey3 ?CaseKeyDef   ( was the C Key pressed? )
      ::
        TakeOver MDL           ( display missing departure & latitude )
      ;
      kcMenuKey6 ?CaseKeyDef   ( was the F Key pressed? )
      ::
        TakeOver CLR_glob      ( initialise global g_Lat and g_Dep to zero )
      ;
      DROP 'DoBadKeyT          ( prevent suspended environment )
    ;
    kpLeftShift #=casedrop     ( process left-shifted plane )
    ::
      kcMenuKey1 ?CaseKeyDef   ( was the A Key pressed? )
      ::
        TakeOver
        $ "Left Shift A Key!"  ( place a simple message on )
        DISPROW1               ( the first row of the screen )
      ;
      DROP 'DoBadKeyT          ( prevent suspended environments )
    ;
    2DROP 'DoBadKeyT
  ;
  TrueTrue      ( 3. Key handle flag and 4. Standard handle flag )
  NULL{}        ( 5. Menu built by MENU_BUTTONS )
  ONEFALSE      ( 6. Initial menu row and 7. Run time environment flag )
  ' LAM exit    ( 8. exit condition )
  ' ERRJMP      ( 9. error function )
  ParOuterLoop  ( run the ParOuterLoop )
  ABND          ( abandon temporary variables )
  RECLAIMDISP   ( resize and clear display )
  ClrDAsOK      ( redraw display )
;
```

Lets look at the above code in a little more detail. Once we have forced a garbage collection of unwanted data in the memory heap, we create a temporary local variable `(LAM exit)` to hold the exit condition. Since the value held by this variable is used to evaluate whether the loop will halt or not, giving it a default value of `FALSE` will ensure that the loop initiates as normal.

For the keyboard assignment, any key in the six planes can be assigned to have a new function, whether right- or left-shifted. Here, we assign command handlers for the first three keys and one for a left-shifted first key. For all other keys not assigned (and the key handle flag parameter is `TRUE`), the standard key definition is executed.

The menu items can be handled in two different ways. However, since we have a function to print the menu to the screen, we can set this parameter to `NULL`. This will be discussed later.

The next parameters set the initial menu row and a switch for handling errors. If the flag is set to `TRUE`, any user command creating a suspended environment (such as `HALT`) or a cold restart will generate an error.

Each time the loop is executed, the temporary local variable `(LAM exit)` is evaluated. If the result is `TRUE`, the loop is exited. Finally, call the system command `ParOuterLoop` to handle the user input messages.

For this application to run a little smoother, we need to modify the code for our `MAIN_` function. Using the code from the last edition, the modifications are in bold below. Note that we will use global

variables to store the departure and latitude. These globals are handled by the function that calls `MAIN_` and thus, should not be deleted here.

```
*****************************************************************
* FUNCTION NAME:  MAIN_
* INPUT:          Two named local variables
* OUTPUT:         NONE
* COMMENTS:       The main Program: Polar to Rectangular.
*                 Requires global variables g_Dep and g_Lat
*****************************************************************
ASSEMBLE
    CON(1)  8
RPL
NULLNAME  MAIN_              ( use NULLNAME to hide the function from the user )
::
  ( Remove local variable declaration and null string )
  BEGIN                      ( Begin the loop )
  ::
    $ "Bearing? "            ( Put bearing prompt 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 g_Dep %+ ' LAM g_Dep STO           ( store total Departure )
          LAM g_Lat %+ ' LAM g_Lat STO           ( store total Latitude )
          FALSE                                  ( continue loop )
        ;
        ( at this point, the loop is ready to continue back to the start. )
        ( The only difference is that we are not prefixing an output string )
        ( 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 )
  ( remove local variable clean up )
;
*****************************************************************
```

Menus

There are no standards for how menus should be designed, so the look and feel is up to you. Whether you have a neatly designed hierarchical menu system that functions neatly from a single entry menu, or whether you have various data-centric menu systems that are accessible via a single custom key handle, the choice is ultimately dependent on which is most comfortable to you. Remember, the 48s and 49s don't come with large Windows style screens, hence simple, logical and intuitive menus are very much an important design criteria.

The same applies with handling commands. That is, whether to perform an action using the data on the stack or to instigate the user to enter in data for a subsequent computation. In the oncoming articles, we will assume that a menu item or a key press requires the user to enter some data on the stack before performing any functions on the data.

We noted previously that there are two ways to handle menu interaction. For this article, we have used the method of drawing graphic objects (grobs) to the screen and using keyboard handlers to respond to a key stroke. The other method, which we will discuss in the next article, uses the `ParOuterLoop` (see argument 5) to set up and handle the menu drawing and key stroke processing.

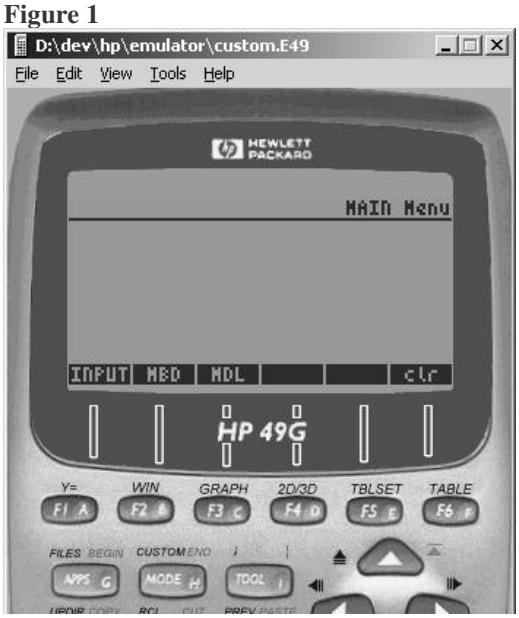
For this article's program, let's create a s imple menu that has menu items called "INPUT", "MBD", "MDL" and "clr". Here's the code that does just that.

```
*************************************************************
* FUNCTION NAME:      MENU_BUTTONS
* INPUT:              Nil
* OUTPUT:             Menu in screen-pixel format (ie. XYGROBS)
* COMMENTS:           Sets up custom menu display
*************************************************************
ASSEMBLE
    CON(1)  8
RPL
NULLNAME  MENU_BUTTONS
::
  { "INPUT" "MBD" "MDL" " " " " " "clr" }
  { LAM MenuLine } BIND
  ::
    ( draw some information to the screen )
    RECLAIMDISP
    95 BINT1 $ "MAIN Menu" $>grob XYGROBDISP
    ZERO SEVEN 131 SEVEN LINEON
    TURNMENUOFF
  ;
  ( Draw labels )
  ZERO FIFTYSIX LAM MenuLine BINT1 BINT1 SUBCOMP INCOMPDROP
  TWENTYTWO FIFTYSIX LAM MenuLine BINT2 BINT2 SUBCOMP INCOMPDROP
  44 FIFTYSIX LAM MenuLine BINT3 BINT3 SUBCOMP INCOMPDROP
  66 FIFTYSIX LAM MenuLine BINT4 BINT4 SUBCOMP INCOMPDROP
  88 FIFTYSIX LAM MenuLine BINT5 BINT5 SUBCOMP INCOMPDROP
  110 FIFTYSIX LAM MenuLine BINT6 BINT6 SUBCOMP INCOMPDROP
  SIX ZERO DO MakeStdLabel XYGROBDISP LOOP
  ABND ( abandon temporary variables )
;
```

In the above code, the local variable `MenuLine` is given an initial value of a list defining the menu button labels. The next object (i.e. the code between `::` and `;`), when evaluated, draws the text "MAIN Menu" to the screen at the position of `x=95` and `y=1`, where x and y are screen coordinates are in pixels from the top left corner - `0,0`. Then, from screen position `0,7` to `131,7` a line is drawn.

To draw the menu using grobs, we first turn the last drawn menu off using `TURNMENUOFF`. Then, using a loop, `MakeStdLabel` takes each of the six arguments from the stack (the individual elements from the list) and converts it into a menu label. For example, after the first run through the loop, the first element "INPUT" is retrieved from the list and placed on the screen at position `0,56`. The remainder of the menu at this point is blank. Figure 1 shows the menu and simple text.

Figure 1



Keyboard Shortcuts

Basically, the keyboard is divided into six columns and, depending on whether you have a HP48 or a HP49, 9 or 10 rows respectively. So defining which keys to map is a matter of counting across from the left and then down starting at Key A. For example, key NINETEEN on the HP49 is the "O" key (EQW), and TWENTYFIVE is the "T" key (COS).

As mentioned previously, the example in this article uses the following key assignments for the menu keys in row one:

```
DEFINE kcMenuKey1 ONE                    ( A Key, Row 1, Column 1 )
...
```

We could however, assign any of the keys on the keyboard to have this functionality. In this case – where we are using the method of defining keys to handle the functionality of menu grobs, it makes sense to use the keys directly below the menu displayed in the screen.

Close

To wrap all this up into a neat application, we will again use our BD2EN function (from the last article) to setup the program environment. This time however, instead of calling MAIN_ to control the user input we call COMMAND_HANDLER. In addition, all we need to do is add in our global variables to handle the departure and latitude, and some new functions to handle the display of the missing bearing and distance and departure and latitude.

To add declaration, assignment and clean up of the globals, modify the code for BD2EN as follows.

```
*****************************************************************
ASSEMBLE
    CON(1)  8
RPL
xNAME  BD2EN
::
  %0 %0
  { LAM g_Dep LAM g_Lat } BIND   ( create local variables, initialised to zero )
  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 )
  ::
    COMMAND_HANDLER            ( call our ParOuterLoop 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 )
  ABND              ( delete named local variables )
;
*****************************************************************
```

To display the total departure and latitude, simply call the global variables and place them in the first row of the screen display.

```
*****************************************************************
* FUNCTION NAME:      MDL
* INPUT:              Nil
* OUTPUT:             Displays Departure and Latitude
* COMMENTS:           Displays summation of total traverse data -rect-
*                     Requires global variables g_Dep and g_Lat
*****************************************************************
ASSEMBLE
    CON(1)  8
RPL
NULLNAME  MDL
::
  BINT4 DOFIX                          ( fix precision to 4 decimal places )
  $ "Dep: " LAM g_Dep a%>$ &$          ( append global to departure string )
  $ "\0aLat: " LAM g_Lat a%>$ &$       ( append global to latitude string )
  &$ BlankDA12 %1 xDISP                ( clear the screen, display output )
  WaitForKey 2DROP                     ( wait for next key input )
;
```

To compute the missing bearing and distance, simply convert the departure and latitude to polar coordinates first before displaying the results. Ideally, we would write a separate function to remove the duplication of code in displaying the data.

```
*****************************************************************
* FUNCTION NAME:      MBD
* INPUT:              Nil
* OUTPUT:             Displays missing Bearing and Distance
* COMMENTS:           Displays summation of total traverse data -polar-
*                     Requires global variables g_Dep and g_Lat
*****************************************************************
ASSEMBLE
    CON(1)  8
RPL
NULLNAME  MBD
::
  BINT4 DOFIX                          ( fix precision to 4 decimal places )
  LAM g_Lat LAM g_Dep RECTOPOL         ( recall globals and convert to polar )
  $ "Brg: " SWAP a%>$ &$               ( append global to bearing string )
  SWAP $ "\0aDist: " SWAP a%>$ &$      ( append global to distance string )
  &$ BlankDA12 %1 xDISP                ( clear the screen, display output )
  WaitForKey 2DROP                     ( wait for next key input )
;



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

Finally, a function to initialise the departure and latitude to zero for subsequent input of new traverse data.

```
*****************************************************************
* FUNCTION NAME:      CLR_glob
* INPUT:              Nil
* OUTPUT:             Sets global Departure and Latitude to zero
* COMMENTS:           Initialises traverse data for new input
*                     Requires global variables g_Dep and g_Lat
*****************************************************************
ASSEMBLE
    CON(1)  8
RPL
NULLNAME  CLR_glob
::
  %0 ' LAM g_Dep STO         ( store 0 in departure )
  %0 ' LAM g_Lat STO         ( store 0 in latitude )
;
```