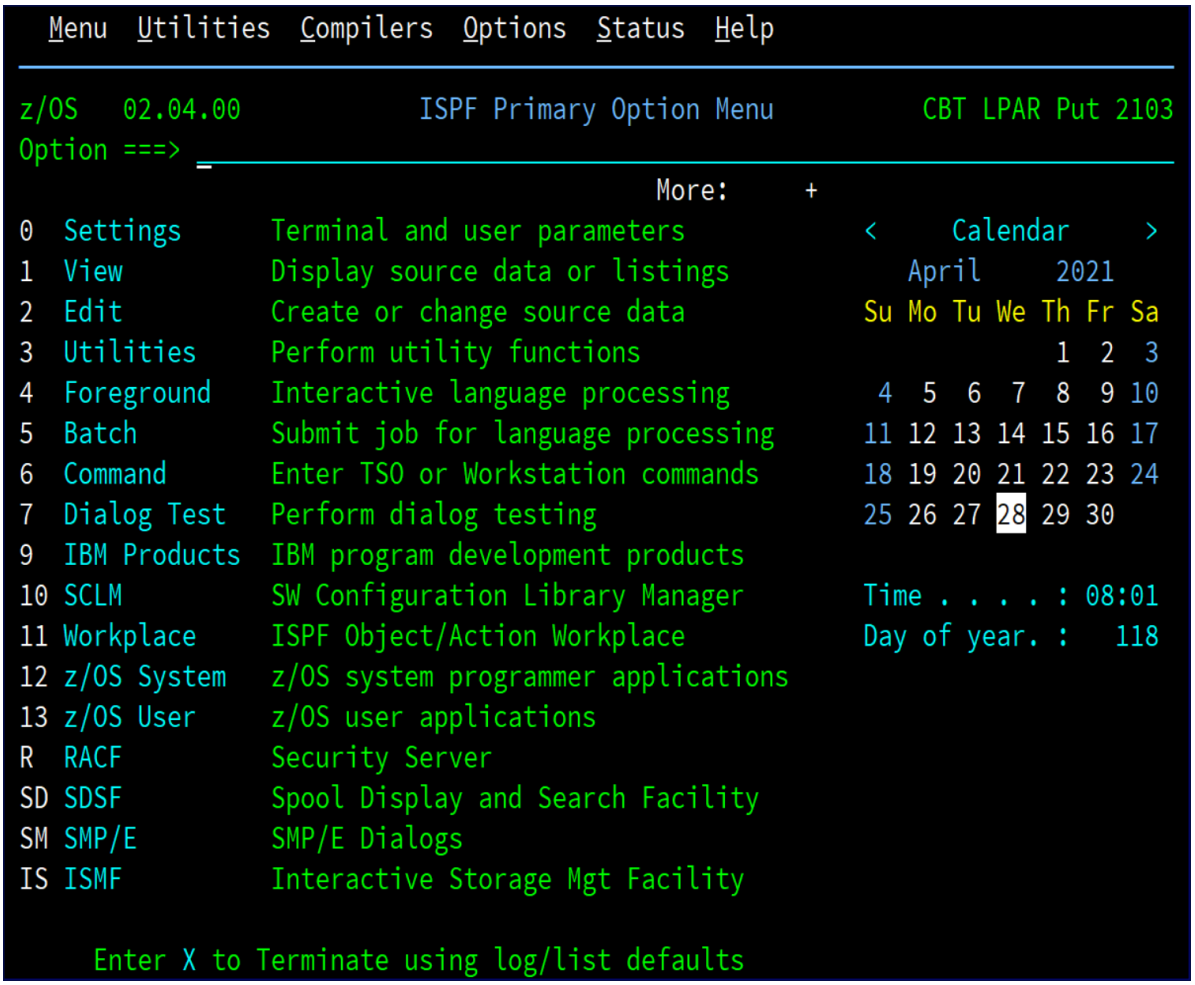
ISPF Developer Tips and Tricks



Version 1.9

Contents

Dedication

This document is dedicated, with extreme thanks, to those who came before us. To the Blue Berets who contributed and managed the MVT Mods Tapes, to those who freely shared their code and documentation via cards, tapes, Gopher, listservs, and then the Web, to those who contribute to the CBTTape, to those who speak at SHARE and other organizations sharing their experience and knowledge. And to all the IBMers who share via the various listservs, forums, and PMRs. We are all better developers because of the help of others, whether it is overt from a document or presentation, or by allowing us to learn from their code. This document is released in the spirit that you, our readers, will continue this tradition.

Caveats and Disclaimer

This document and the code samples are provided without warranty or guarantee. If you use any of the code for production purposes, you should thoroughly test and validate it before use.

There is no copyright associated with the sample code so feel free to use any/all of it within your own code (after testing and validation of course). The term, YMMV (your mileage may vary) applies.

Where to Find Updates

Updated versions will be posted to www.lbdsoftware.com, and www.cbttape.org in file 990.

How to Contribute

This document is intended to be a living document and that requires regular updates. If you would like to provide sample code, and the prose to go with it, please e-mail ispfdev.tips@gmail.com and one of the volunteer team members will acknowledge your communication.

Contribution guidelines:

1. Code should follow the sample naming conventions:
    a. PNxxx for Panels
    b. RXxxxx for REXX
    c. SKxxx for Skeletons
2. Provide it in TSO Transmit format to prevent EBCDIC/ASCII conversion issues during transport.
    a. Put the elements in a PDS
    b. Issue the TSO command: XMIT x.y DS(your.pds) OUTDS(contribution.XMIT)
        i. Note x.y can be anything
    c. Binary download and e-mail the contribution.XMIT data set
3. Prose may be plain text or in Word format.
4. Suggest where in the document to place it.
5. Examples should be generic and not depend upon products or tools that are not available on all systems (that is – don't assume every site has product xyz).

There is no guarantee that any contributions will be used – that is left to the discretion of the editors and will mostly depend on their available time.

Updates to this document and the related samples data set will be announced on the ibm-main listserv (join/view at https://listserv.ua.edu/archives/ibm-main.html) and posted at www.lbdsoftware.com.

Revision History

Version 1.9	5 June 2021	1. PNDYNPFK – dynamically set a PFK value for for an application (WJ) 2. RXSHRVAR to demonstrate how to share rexx variables between rexx exec's
Version 1.8	15 Apr 2021	3. Info on Random string 4. Section on Table Handling (Thanks Willy Jensen) 5. Sample table handling subroutine (WJ)
Version 1.7	13 May 2020	6. Document how to use an Edit macro to change text with &'s 7. Add info on how to have one REXX exec run as either an Edit Macro or TSO Command (thx to Bob Bridges) 8. Add RXNOTEPD – a sample ISPF notepad application 9. Exciting info on how to dynamically add rows to a table as needed – great for very large tables 10. Fix to PNVDSN sample panel
Version 1.6	26 January 2020	11. Update PNREXX to use 4-digit year and routine to insert commas for long numbers (thx to Doug Nadel) 12. Update PNVDSN to check for the F3/END key and bypass the check 13. Updated RXTAB with cleaner Find and RFind 14. Created RXTABLE from RXTAB with improved RFIND command table definitions
Version 1.5	27 December 2019	15. Update sample ISPF Stub (thx to Tom Conley) 16. Update info on dynamic edit color change using panel rexx
Version 1.4	26 December 2019	17. RXPOPKEY code and example updated to improve code. 18. Update LOADISPF section to reference correct sample name RXPNSL 19. Add reference to RXISPFL to the LOADISPF section as a full example 20. Elaborated on the use of LOADISPF. 21. Enhance PNTAB and RXTAB for better clarity in table handling. 22. Add PNEDITHL, RXEDITHL, and RXMEDHL to demonstrate updating colors in an ISPF Edit panel dynamically.
Version 1.3	08 October 2019	23. Add document version to sections added or updated to easily identify what changed. 24. How to pass a variable from your REXX Exec to the Skeleton REXX for use. Thanks to a tip from John Kalinich. 25. Add subroutine to change a numeric variable from all numbers to human readable (e.g. 1981 to 1,981)
Version 1.2	22 July 2019	26. Information on the ZUSERMAC variable to define the ISPF Edit Initial Macro. 27. Information on using LISTC and CSI to get a list of data sets as an alternative to LMDLIST.

		28. Update to PNVDSN to clean up the panel REXX dsname verification code. 29. Clarified the XMIT file name in the ZIP file. 30. Add to the Appendix information on the CBT Tape File 452 version of Dynamic STEPLIB 31. Miscellaneous editorial changes for clarity.
Version 1.1	31 May 2019	32. Several minor corrections from W. Jensen along with a new randomized example 33. Change date in footer 34. Sample Skeleton REXX with TSO services from W. Jensen 35. Sample using BPXWDYN for a random ddname from W. Jensen. 36. Section on using LISTDSI to return a fully qualified data set name. 37. Additional sample random routines for ddname or table names. 38. Add info on PANELID ON/OFF 39. Update the How to Contribute section 40. Added a chapter on Other Tools 41. Added section on Stem Sorting from John Kalinich.
Version 1.0	18 April 2019	Initial release

Contributors

Authors/Editors/Contributors

Lionel B. Dyck	John Kalinich	Bruce Koss
Willy Jensen	Bob Bridges	

Contributors who may not know it

Doug Nadel via his Presentation ISPF Panels Beyond the Basics which can be found at https://sites.google.com/site/schlabb/home/hints-tips/ispf	Thomas Conley	John McKown via IBM-Main
Bill Godfrey (via IBM-Main)	Albert Ferguson via IBM-Main	Robert Prins
Marvin Knight		

Introduction

This document is intended to provide the ISPF application developer, whether full time or ad hoc, with tips and tricks to exploit the capabilities of ISPF within the z/OS environment. There are a few tools that will be mentioned in the tips that the contributors feel will be useful. See the Appendix for details on those tools.

The examples are provided in REXX as it is easy to quickly develop application prototypes, many of which can be used without further efforts. In some cases, the prototype will need to be translated into a compiled language once the prototype has proven the design and function.

Become familiar with the ISPF Edit MODEL command. This great feature will provide sample code for ISPF Panels, Skeletons, REXX, and more.

Download, or bookmark, the ISPF publications from IBM for the version of z/OS that you will be working with. Given the level of ISPF development, it is unlikely that there will be new features or functions added to newer versions –one never knows, however one can hope.

To accompany this document, a PDS has been created with samples for many of the topics discussed. This PDS is distributed in TSO Transmit (XMIT) format and is included in the package ZIP file with a file suffix of .XMIT. This file must be processed by the TSO RECEIVE command after being uploaded to z/OS using a binary transfer method into a sequential data set with RECFM=FB and LRECL=80.

```
RECEIVE INDS(upload.xmit)
```

The topics with examples below will have a box with the example member names that looks like this:

```
Member-name
```

Note that the sample code may include additional comments from what is included in the examples in this document.

Sample PDS

Within the Sample PDS are members that contain ISPF Panels, Skeletons, and REXX code. All are fully functional for demonstration purposes.

Member $$README provides additional information about the samples and member $$XMIT provides information on the two members in TSO Transmit format.

Included is an ISPF dialog that demonstrates many of the ISPF table handling function using the Sample PDS.

To use the dialog:

1. Get into ISPF 3.4 and list the Sample PDS data set name
2. Then enter M to display the Member List
3. Scroll down to $DEVISPF
4. Enter EX next to it to begin the dialog which displays an ISPF table of the members and supports several commands and row selection options:

```
--------------------- ISPF Developer Tips/Tricks Examples -------- Row 1 of 92
Command ===> _                                                Scroll ===> CSR

Commands:   Find xx Locate xx Only xx  R Refresh
Selections: B Browse E Edit R Receive T Tryit X eXecute

Sel Member   Type  Description

_   $$README Text  Overview of the Samples
_   $$XMIT   Text  Doc on how to process the XMIT members
_   $DEVCPY  Rexx  Exec using LMCOPY to copy Samples into User Libraries
_   $DEVCPYP Panel ISPF Prompt Panel for $DEVCPY
_   $DEVISPF Rexx  Exec to display the Sample Members
_   $DEVPH   Panel ISPF Tutorial Panel for $DEVPP
_   $DEVPP   Panel ISPF Panel used by $DEVISPF
_   $DEVPX   Panel ISPF Popup Panel to Prompt for parms when executing Rexx
_   #RXFORM  XMIT  #RXFORM  tool in TSO Transmit format
_   #TRYIT   XMIT  #TRYIT tool in TSO Transmit format
_   CMT      Macro Rexx Edit Macro to insert Comments in Code/JCL/etc.
_   JCBAT1   JCL   Sample ISPF in Batch JCL using the TASID Utility
_   JCBAT2   JCL   Sample ISPF in Batch JCL using the RXLMD   Rexx
_   LOADISPF Rexx  REXX code to be used with other code
_   PNABC    Panel Sample ISPF Panel to demonstrate an Action Bar
_   PNAREA   Panel ISPF Panel to demonstrate a scrolling Panel
```

Hint: Execute (x) the $DEVCPY exec to copy these samples from the sample library into your own SYSEXEC and ISPPLIB libraries so that you can make changes, experiment, and learn.

General Comments

When writing code, it is important that others be able to read, understand, and if necessary, update and enhance (fix) the code. The best way to do that is to follow coding practices that have been developed over the past several decades:

1. Use meaningful variable names where possible. ISPF variables are limited to 8 characters, while REXX variables can be longer. Variables may be upper, lower, or mixed case.
    a. ISPF variable: UserName
    b. REXX variable: TSO_User_Name
2. Document the code with comments
    a. There are many styles of comments and the best one is what the developer (you) are most comfortable in using. Some like to put comments on the individual lines of code, while others like to put comments in blocks before code sections.
    b. An ISPF Edit macro, CMT, mentioned in the Appendix can make it significantly easier to enter comments. It creates block comments and supports entering comments from the command line or via a popup panel.
    c. REXX code supports comments – so do ISPF Messages, Panels, and Skeletons. That means that you can add useful comments in all those elements so that others will know what you were thinking when you developed them.
    d. Not everything requires comments. Too many comments can get in the way of the code. Be judicious and not verbose. Look at the code as if you were not the author and if you can understand it without comments then that is the ideal, otherwise add comments to make the coding understandable for whomever comes after you.
3. If there is a complex way to write code and a simple way, choose the simple way. You never know who may have to pick up your code in 5, 10, 20, or more years and fix something that you never anticipated.
4. Where possible, use standard system routines and interfaces.

Installing Your Application

Once you develop your ISPF application, the next decision is how to make it available to your users. The first thing to consider is how will the application be executed, or how will the elements be accessed. The second thing to consider is how will the user find the application so that they can use it.

Making the Application Available

Once your application is completed, the various elements (execs, panels, programs, skeletons, messages, tables, etc.) need to be made available to the users.

There are three ways to do this:

1. Copy the elements into libraries that are already allocated to the users TSO Logon (via Logon Proc or Logon Clist).
2. Write a stub Exec that performs the dynamic allocations and then invokes the application which must be installed in a library in the users SYSEXEC or SYSPROC allocations.
3. Use the Product Launch Point (PLP) discussed below.

Sample Stub

This is an example to invoke the RACF ISPF dialog and it is from Tom Conley's Dynamic ISPF Starter Set package which can be found in File 495 at www.cbttape.org where you can find samples for you to use for over 100 different software packages.

```
/* rexx */
/*********************************************************************/
/* This exec invokes IBM's RACF dialog.                              */
/*********************************************************************/
parse arg ztrail
address tso "ALTLIB ACT APPL(CLIST) DA('SYS1.HRFCLST')"
address ispexec "LIBDEF ISPPLIB DATASET ID('SYS1.HRFPANL') STACK"
address ispexec "LIBDEF ISPMLIB DATASET ID('SYS1.HRFMSG') STACK"
address ispexec "LIBDEF ISPSLIB DATASET ID('SYS1.HRFSKEL') STACK"
address ispexec "SELECT PANEL(ICHP00) NEWAPPL(RACF) OPT("ztrail")",
                "PASSLIB SCRNAME(RACF)"
address ispexec "LIBDEF ISPPLIB"
address ispexec "LIBDEF ISPMLIB"
address ispexec "LIBDEF ISPSLIB"
address tso "ALTLIB DEACT APPL(CLIST)"
```

Sample PLP Definition

This is how that would look in a PLP definition:

```
----------------------------  Product Launch Point Management -
 Command ===>

 Enter, Verify, or Change (* = required)

 *Application Name      SYNCM___ A unique application name 1-8 characters
 Application ID        SX21     1-4 characters for application id
 *Description           Syncsort ISPF Messages Dialog_______________
 Step Library          ____________________________________________
 ISPF Load Library     'SYS2.SYNC.MFX210.SYNCLOAD'_________________
 ISPF Message Dataset  'SYS2.SYNC.MFX210.SYNCMLIB'_________________
 ISPF Panel Dataset    'SYS2.SYNC.MFX210.SYNCMLIB'_________________
 ISPF Skeleton Dataset ____________________________________________
 ISPF Table Dataset    ____________________________________________
 Clist Library         ________________________________________
 REXX EXEC Library     ________________________________________
 *Application Start (select one)
    Command    ____________________________________________
    Program    SS21MSG3 Parm ______________________________
    ISPF Panel _________ Panel option ______________________
 Additional Datasets:  No_  Yes to add more
```

Adding to an Existing ISPF Menu

The application can be added to an existing ISPF Selection Panel (menu) where it can be found by those users who are aware of that Panel. In some shops, that means adding the application to ISR@PRIM as one more in a long list of ISPF applications. Others will add the application to a menu that is accessible from ISR@PRIM.

The advantage is that your application is now on a Selection Panel and thus available to your intended target users.

The disadvantage is that your application is on a Selection Panel that your intended target users, including new users, may not be aware of. If they are aware of the Panel, they may not be aware that they must scroll down to find your application.

Another challenge is making sure that all copies of the Selection Panel are updated, this is especially true for ISR@PRIM and ISP@MSTR, for which there may be multiple versions in the various libraries in the ISPPLIB allocation, and all dependent upon the order they are allocated.

Add to the ISPF Commands Table

This provides a fast path to your application from any ISPF command line. This is an excellent option, provided the application name is unique. (I once installed an ISPF application that had a command identical to one in the ISPF Command Table. That caused a lot of confusion, until it was researched).

Keep in mind that even after you add a command to the command Table, not everyone will know it's there. You may want to add it, as well, to an ISPF menu so users will see it.

Dynamic ISPF Menus

This is a plug for a tool called the Product Launch Point (PLP), which is an open source dynamic ISPF menu. This tool can be found at http://lbdsoftware.com/ispftools.html, and at http://www.cbttape.org in file 312.

PLP is table driven ISPF menu. Add the command PLP to the ISPF commands table and your user has immediate access to a searchable, scrollable, flexible ISPF menu.

PLP supports multiple menus based on how it is invoked, and it supports sub-menus.

The advantages are numerous, among them are:

1. Easily add, or update, an application.
2. When adding an application PLP allows defining libraries that will be
    a. STEPLIB'd (if a dynamic STEPLIB command is available, if you don't have a dynamic STEPLIB command see www.cbttape.org for file 452 for a free one that works very well)
    b. LIBDEF'd for ISPF libraries
    c. ALTLIB for CLIST and REXX libraries
    d. Specify an Application ID
    e. Invoke a Panel, Program, REXX/Clist – and pass parms

PLP is used at a number of large, and small, z/OS installations very successfully as they have found it greatly reduces the efforts to manage and maintain menus for access to applications, while at the same time making it very easy to find and execute the installed applications.

When PLP is initially installed the very first use must be with the ADMIN keyword to create the initial PLP table.

Disabling Keylists

Some sites have disabled application keylists via the ISPF Configuration Table, while others allow them. If you want to disable them for the period of your dialog you can use this code:

```
"VGET (ZKLUSE) PROFILE"   /* Obtain Keylist  */
SAVE_ZKLUSE = ZKLUSE      /* Save Keylist    */
ZKLUSE = "Y"              /* Keylist on      */
"VPUT (ZKLUSE) PROFILE"   /* Set Keylist     */
… code…
ZKLUSE = SAVE_ZKLUSE      /* Restore         */
"VPUT (ZKLUSE) PROFILE"   /* Set Keylist     */
```

ISPF Panels

Panel Basics

Some things to consider when developing an application using ISPF:

1. Use a consistent style for all panels in the application.
2. Keep the panel body to 23 (or less) records so that any user with a 3270 model 2 (24x80) will be able to display the panel.
3. Use hilite(uscore) for input fields to easily identify them on the panel.
4. Provide a tutorial panel for each panel where it makes sense.
5. When referencing a program function key use F# instead of PF#. Keyboards have not had the PF prefix for the function keys for decades so the newer 3270 users may not grasp what a PF key is.
6. When a small amount of information is required in a dialog consider using a popup.
7. When a dialog is processing something that takes more than a few seconds consider using a popup to display progress information.

Testing Panels

When developing ISPF Panels, it is easy to make a typographic error in the panel code. Using ISPF Test can be a challenge (but one you should take the time to learn). However, there is a tool included with the samples call TRYIT that can be used to validate your ISPF Panels easily and while you're still in ISPF Edit (you don't have to save the member to test it).

The TRYIT command syntax that you will use for ISPF Panels is:

TRYIT	For any Panel
TRYIT TUT	For Tutorial Panels
TRYIT POP	For Popup Panels

If there are any errors then there will be a short message indicating there is an issue and pressing F1 will display the longer, more detailed message with more information.

See the Appendix for more information on TRYIT.

ALWAYS test your ISPF Panels with a 3270-2 (24x80) terminal configuration. This is the lowest common terminal type available with most 3270 emulators. Many users will, by default, be using that terminal type until they become more accustomed to the capabilities of 3270 terminal and move on to a mod 3, 4, or 5 terminal configurations, or define their own custom configuration (e.g. 60x160). If the panel is too wide or has too many body records, then it will fail when displayed.

Tutorial Panels

Most, not all, ISPF panels should have a tutorial behind it. The tutorial is accessed using the HELP command (typically assigned to the F1 key).

Some things to always consider with Tutorial panels:

1. If you have multiple Tutorial panels, then connect them using the &ZUP and &ZCONT commands.
2. If you only have one Tutorial panel then set &ZCONT to the panel so that when the user presses the enter key, they will remain on that panel.
3. With multiple Tutorial panels:
    a. The 1st always has &ZCONT referencing the next panel
    b. The 2nd thru the last has &ZUP referencing the previous panel and &ZCONT referencing the next
    c. The last should have &ZCONT referencing the 1st Tutorial panel.
4. Try to keep the Tutorial panel body to 23 (or less) records. This is to allow the Tutorial to be viewed on a 3270 model 2 terminal (this configuration supports 24 rows, but you should allow for at least 1 row for an ISPF Split Screen line.
5. All the Tutorial panels should have the same general layout for consistency.
6. The user can ‘scroll’ between Tutorial panels using function keys F10 (Previous) and F11 (Next).
7. For complex dialogs consider a Tutorial Menu panel.
    a. Use the ISPF Edit MODEL command
    b. Select F0 (PANFORM)
    c. Select F5 (TUTORIAL)

Scrolling Panels

`PNAREA and RXAREA`

There are times that the data on a panel is more than the depth of the user screen and rather than creating multiple panels (the recommended technique), it is easy to define a panel area that allows scrolling.

Note that this will work for data entry panels, tutorial panels, and popup panels (table panels are already scrollable).

This sample REXX will display the following panel until the F3 (END) is pressed.

```
/* rexx */
 Address ISPExec
 do forever
  'display panel(pnarea)'
  if rc > 0 then leave
  end
```

This is part of the Panel:

```
)attr default(%+_)
} area(SCRL) Extend(ON)
)Body Expand(\\)
%Sample -\-\(~Scrolling Panel Demo%)\-\- Sample
%Command ===>_ZCMD                                                          +
+
+     Sample of a Scrolling Area - use &zpf07 and &zpf08 keys to scroll +
+     And then%F3+to Exit.
%-------------------------------------------------------------------------
}help ---------------------------------------------------------------------}
)Area Help
+With thanks to Mr. John Kalinch for allowing us to 'borrow' the FAQ for the
+PDS 8.6 command (found on the CBTTape File 182 at www.cbttape.org)
+
+ PDS Command Processor FAQ
. . .
)Init
)PROC
)END
```

Note that there is a size limit, which I suspect is byte related, but for this example at 1000 records in the area works and with a few more it fails (at 1023). If you have that many records it may be better to use ISPF Browse (or Edit or View) and place the data into a data set.

Be aware that when pressing ENTER the panel will automatically scroll back to the top.

Field Level Help

`PNFLDH, PNFLDH1, PNFLDH2, and RXFLD`

For some applications having more detailed information for specific fields on the panel is helpful. IBM has provided the ability to have Field Level Help panels that make this very easy by allowing the user to place the cursor on a specific entry field and then press F1 to display a specific Help panel.

Here is a sample data entry panel with Field Level Help:

```
)ATTR DEFAULT(%+_)
 $ TYPE(INPUT) INTENS(LOW) hilite(uscore)
)BODY Expand(\\)
+\-\%Sample Panel with Field Level Help+\-\
%Command ===>_zcmd
%
+   Place the cursor on any of the fields below and press F1 to
+   view the Field Level Help.
+
+   Field one: $z      +          Sample Input Field
+   Field two: $z      +          Sample Input Field
+
  +Press%F3+when ready to exit.
)INIT
 .zvars = '(field1 field2)'
)PROC
)Help
 Field(field1)  panel(pnfldh1)
 Field(field2)  panel(pnfldh2)
)END
```

The Field Level Help is defined in the `)HELP` section of the panel using the FIELD statements, where the field name, enclosed in parenthesis, is the entry field on the panel.

This is one of the sample Field Level Help panels. Notice this is a normal popup panel, although field level help does not require a popup, that is the typical usage.

```
)ATTR DEFAULT(%+_)
)BODY WINDOW(45,4)
+
+ This is a sample help for%Field One
+
+(Press%F3+to close)
)INIT
 &ZWINTTL = 'Field Level Help One'
)END
```

And this REXX code can be used to display and experiment with this capability:

```
Address ISPExec
 do forever
  'Display Panel(pnfldh)'
  if rc > 0 then leave
 end
```

Using Point and Shoot (PNS) with ISPF Panels

```
PNPNS, RXPNS, and RXPNSL
```

There are times where an ISPF panel needs to support point and shoot logic and this is a simple way to do it.

There are two approaches to doing this. But if the user does not have tab to Point-and-Shoot enabled then some of the usefulness is lost. Tab to Point-and-Shoot can be enabled using the ISPF command ISPFVAR PSTAB(ON).

One approach is using TYPE(PS) which allows no other attributes.

The other is TYPE(OUTPUT) with PAS(ON) which does allow most other attributes to be used such as intens, color, caps, justification, and hilite for the field.

Define some attributes

type(output) intens(high) caps(off) just(left) pas(on) hilite(uscore)

```
# type(output) intens(high) caps(off) just(asis) pas(on) hilite(uscore)
@ type(PS)
```

Use the appropriate attribute based on the justification. In some cases, you will want to use left justification and in others, you need specific placement so asis justification is what you need. These fields are enabled for overtyping but there is no effect from doing so.

If using TYPE(PS) then you are not able to define any other attributes, but the advantage is that these fields are not enabled for overtype as the TYPE(OUTPUT) fields are.

Use them in the table header

```
@Name+ Gen Abs@CRdate @MDdate +V.M @Size+
```

In this example both left and asis justification are used. Some of the headers are not defined for point and shoot (e.g. GEN Abs, V.M). The rest are defined for left justification except the MDdate which is defined using asis justification.

Or use them within a panel

Enter value:_val#xyz+

Define the output variable values in the)INIT section

```
&name   = 'Name'
&crdate = 'Created'
&size   = 'Size'
&xyz    = 'XYZ'
```

As you can see all of the variable names match the output name in the header row. The &mddate has leading and trailing blanks so that it fills out the space in the header, so the fields are aligned where they need to be.

1. Define the Point and shoot

```
)PNTS
FIELD(name)     VAR(ZCMD) VAL('SORT NAME')
FIELD(crdate)   VAR(ZCMD) VAL('SORT CREATED')
FIELD(mddate)   VAR(ZCMD) VAL('SORT CHANGED')
FIELD(size)     VAR(ZCMD) VAL('SORT SIZE')
FIELD(xyz )     VAR(ZCMD) VAL('SET XYZ')
FIELD(ZPS00001) VAR(ZCMD) VAL('Something')
```

In the `)PNTS` (point and shoot) section the syntax is:

`FIELD(variable name)`
This defines the variable of a point and shoot field in the ISPF display.

If using `TYPE(PS)` then the field name must be ZPSnnnnnn where nnnnnn is 000001 to the number of `TYPE(PS)` fields on the panel and go from left to right and top to bottom.

`VAR(variable)`
This defines the variable to be set when the point and shoot field is selected. In this case ZCMD will be updated and passed to the application.

`VAL(value)`
The value is the text that will be inserted into the variable defined in VAR(). Enclose in quotes if more than one word or if spaces are included in the text.

And assuming your dialog understands the zcmd values you're good to go.

One way to do this so that the panel gets updated but the processing code doesn't process is this technique which assumes that the point and shoot field sets ZCMD to a value of 'SET XYZ' and that causes the code to set panel variable to 'X', reset zcmd to null, and then iterate which redisplays the panel.

Here is an example ISPF Panel to demonstrate the two types of Point and Shoot fields and how they work.

Use this REXX Exec to test the following ISPF Panel:

```
/* REXX */
Address ISPExec
do forever
  'Display Panel(pnpns)'
  if rc = 8 then call done
  if rc> 8 then do
     say zerrsm
     say zerrlm
exit
     end
  if translate(zcmd) = 'EXIT' then leave
  end
exit
```

And this is the ISPF Panel with a name of PNPNS

```
)ATTR DEFAULT(%+_)
 $ type(ps)
 # type(output) caps(off) just(left) color(red) hilite(uscore)
 @ type(output) caps(off) pas(on) hilite(uscore) color(yellow)
 } type(output) caps(off) pas(on) hilite(uscore) color(blue)
)body
%Command ==>_zcmd              +
+
%Test:  $One+     $Two+
+       $Three+   $Four+
+       @Five+    @Six+
+
%Value #result                                  +
+
+Click here when ready to}Exit+
+
+Tab to any of the point and shoot fields and press%Enter+or move the
+mouse point to them and double click. See the results in the%Value+field.
+Note that the%Five+and%Six+fields are point and shoot fields but you can
+change the text that displays in them with a drawback that you can
+actually type into those fields with zero affect.
+
)Init
&five = 'Five'
&six  = 'Six'
&exit = 'Exit'
)Proc
)PNTS
 FIELD(ZPS00001) VAR(RESULT) VAL('Value of 1')
 FIELD(ZPS00002) VAR(RESULT) VAL('We have two')
 FIELD(ZPS00003) VAR(RESULT) VAL('Now three')
 FIELD(ZPS00004) VAR(RESULT) VAL('Four')
 FIELD(Five)     VAR(RESULT) VAL('Five or 5')
 FIELD(Six)      VAR(RESULT) VAL('Six or 6')
 FIELD(Exit)     VAR(zcmd)   VAL(Exit)
)end
```

PopUp Panels

`PNPOP and RXPOP`

When writing ISPF dialogs there are times when you want to prompt the user, or just provide some feedback, but you don't want to take a full window to do it. That is where a popup can be very helpful. Note that you do not have to provide a Command input (zcmd) field on a popup panel.

A Popup Panel is defined on the `)BODY` statement using the `WINDOWS(ww,hh)` field where ww is the number of columns wide, and hh is the number of rows deep.

Here is a sample panel (PNPOP):

```
)ATTR DEFAULT(%+_)
 @ type(output) caps(off) just(left)
)BODY WINDOW(45,8)
+
%This is a sample POPUP panel
   +to demonstrate the capabilities of it.
+
+and how to add a variable of the userid
+
  %Userid:@zuser
+
)INIT
&ZWINTTL = 'Sample Popup Panel'
)END
```

To keep it simple the variable &zuser, an ISPF variable, is used to demonstrate a variable.

The size of the popup panel is 45 characters wide and 8 rows deep.

To 'spiff' up the panel we have added a panel title using the &ZWINTTL variable in the `)INIT` section to define a title to be used on the popup panel.

To display the popup panel, use the following code:

```
/* REXX */
 Address ISPEXEC
 'Addpop Row(4) Column(6)'
 'Display Panel(pnpop)'
 Save_Rc = rc
 'Rempop'
```

The ADDPOP command supports optional parameters. The ROW keyword defines which row on the screen to display the popup, and the COLUMN how many characters from the left to begin the display. In this case we have the panel being displayed starting on row 4 and column 6. If ROW and COLUMN are not provided, then the popup will be at the top left of the display.

A popup can also have a `)PROC` section and can request the user to provide input to answer questions.

So be creative with popups but not to overwhelm the user.

Dynamically Turning Off PFSHOW

PNPOP and RXPOPKEY

The RXPOPKEY example demonstrates how to disable and then re-enable PFSHOW if the user has PFSHOW ON. PFSHOW ON causes the function keys to be displayed at the bottom of ISPF panels. This is not always helpful in small popup panels.

```
/* ------------------------ REXX ------------------------ *
| This code demonstrates how to dynamically turn off,    |
| and then back on, the PFSHOW setting. This is useful   |
| if the user may have PFSHOW ON to prevent the function |
| keys from overlaying the popup.                        |
* ------------------------------------------------------ */
Address ISPExec
call pfshow 'off'           /* make sure pfshow is off */
'addpop'
'display panel(pnpop)'
'rempop'
call pfshow 'reset'         /* restore pfshow setting */
exit
/* ------------------------------------------------------ *
| The pfshow routine will:                               |
| 1. check to see the passed option                      |
| 2. if Off then it will save the current pfshow setting |
|    - save the current setting                          |
|    - turn off pfshow                                   |
| 3. if the option is Reset then it will                 |
|    - test if pfshow was on and turn it back on         |
* ------------------------------------------------------ */
pfshow:
 arg pfkopt
 if pfkopt = 'RESET' then do
   if pfkeys = 'ON' then
   'select pgm(ispopf) parm(FKA,ON)'
 end
 if pfkopt = 'OFF' then do
   'vget (zpfshow)'
   pfkeys = zpfshow
    if pfkeys /= 'OFF' then
   'select pgm(ispopf) parm(FKA,OFF)'
 end
 return
```

Progress Popup Panels

`PNPROG1, PNPROG2, RXPROG1 and RXPROG2`

There are times when an ISPF dialog needs to process behind the scenes and yet you want to let the user know that something is happening. This can be accomplished using a progress popup.

Here is a very simple example. The popup simply displays a counter, from 1 to 5, while the driver exec increments the counter and then sleeps for 1 second (simulating some background action).

Sample REXX code to demonstrate the Popup Panel:

```
/* rexx */
 Address ISPExec
 do i = 1 to 5
   'control display lock'              /* Lock the Display */
   'addpop'                            /* Setup for a Pop Up Panel */
   pcount = i                          /* Update the variable for the panel */
   'display panel(pnprog1)'             /* Display the ISPF Panel */
   'rempop'                            /* Remove the Pop Up Setup */
   address syscall 'sleep 1'           /* Now sleep for 1 second to
                                          simulate work. */
 end
```

Here is the sample popup panel (named as PNPROG1):

```
)Attr
 @ Type(Output) intens(High)
)Body Window(24,3)
+
+Progress count:@z  +
+
)Init
 &zwinttl = 'Progress Counter'
 .zvars = '(pcount)'
)Proc
)End
```

The output field, pcount, is defined using zvars as the field is only 3 characters on the panel. The use of zvars is not required here as the progress field can be anything – it could be a member name, a data set name, or some other value that will be helpful for the user.

Another option is a progress meter showing the progress of something where you can provide a percentage. This ‘meter’ adds two *’s for each increment of 10%:

```
/* rexx */
/* ------------------------------------------------ *
| Provide a total number of iterations for the Demo |
* ------------------------------------------------ */
 arg total
 if total = '' then total = 100
/* ----------------------------- *
| Change to ISPExec environment |
* ----------------------------- */
 Address ISPexec
/* ------------------------------------------- *
| Define our increment so it fits in the panel |
* ------------------------------------------- */
 incr = (total % 10) + 1
/* ----------------------------- *
| Define our increment indicator |
* ----------------------------- */
 progc = '**'
 i = 0
 perc# = 0
/* --------------------------------------- *
| Now loop thru and show the progress meter |
* --------------------------------------- */
 do until i = total
   i = i + 1
   if i//incr = 0 then do
     progc = progc'**'
     perc# = perc# + 10
     perc = perc#"%"
     prog = progc '('perc')'
/* ------------------------------------- *
| Lock the display since the user will not |
| be able to do anything other than watch. |
| Then sleep for 1 second to simulate work.|
* ------------------------------------- */
     "Control Display Lock"
     'addpop'
     'display panel(pnprog2)'
      Address syscall ‘sleep 1’
     'rempop'
   end
 end
```

And here is the Panel used for the Progress meter with a single output field that is updated with the ‘meter’:

```
)Attr Default(%+_)
  @ type(output) intens(high) caps(off) color(blue)
)Body window(50,3) expand(\\)
+
+   Progress:@prog
+
)Init
 &zwinttl = 'Progress Meter'
)Proc
)End
```

Action Bars and Pull-downs

`PNABC and RXABC`

An action bar is the panel element located at the top of an application panel that contains action bar choices for the panel. Each action bar choice represents a group of related choices that appear in the pull-down associated with the action bar choice. When the user selects an action bar choice, the associated pull-down appears directly below the action bar choice. Pull-downs contain choices that, when selected by the user, perform actions that apply to the contents of the panel.

An action bar is analogous to the menu bar in Windows and macOS.

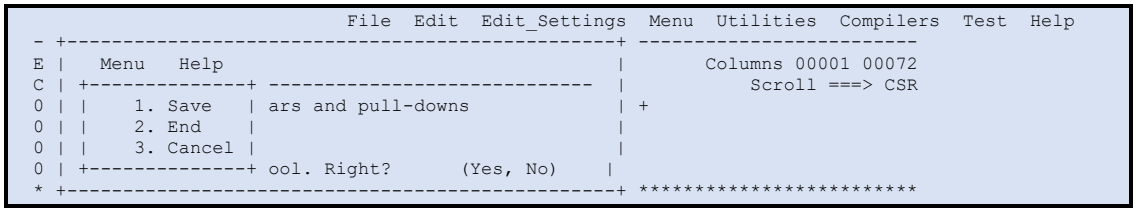
A panel can define pulldowns using the `)ABC`, `)ABCINIT`, and `)ABCPROC` panel sections.

- The `)ABC` section defines an action bar choice for a panel and its associated pull-down choices.
- The `)ABCINIT` section runs when the user selects that action bar choice.
- The `)ABCPROC` section runs when the user completes interaction with the pull-down choice and is optional.

```
)ATTR DEFAULT(%+_)
 $ TYPE(AB)           /* Action bar                */
 @ TYPE(ABSL) GE(ON)  /* Action bar separator line */
)ABC DESC(Menu) MNEM(1)
 PDC DESC('Save')   ACTION RUN(SAVE)
 PDC DESC('End')    ACTION RUN(END)
 PDC DESC('Cancel') ACTION RUN(CANCEL)
)ABCINIT
 .ZVARS = 'MENUX'
)ABC DESC(Help) MNEM(1)
 PDC DESC('Extended Help...')
 ACTION RUN(XHELP)
)ABCINIT
 .ZVARS = HELPX
)BODY WINDOW(48,6)
+$ Menu $ Help +
@-----------------------------------------------+
%         Action bars and pull-downs
%===>_ZCMD                                      +

 Action bars are cool. Right?_ans+ (Yes, No)
)END
```

And here is the sample display:



Panel Rexx

Basic Example

PNPREXX and RXPREXX

When creating ISPF panels there are times when you want to run code that is not provided by the ISPF panel language definition. The *REXX statement is used to invoke REXX code in a panel's)INIT,)REINIT, or)PROC section. The *REXX parameter specifies the names of dialog variables passed to the REXX code and optionally the member name of an external REXX program to be executed. Specifying * as the first parameter causes all the dialog variables associated with the input and output fields on the panel to be passed to the panel REXX code.

The REXX code cannot access any dialog variables except those specified on the *REXX statement or defined within the ISPF Panel. The REXX code cannot issue requests for any ISPF services. REXX coded in-line within the panel source must be terminated by with *ENDREXX.

The example below is a popup that calculates the number of days between two dates.

Save this panel as PNPREXX and then use the provided REXX code to test it.

```
)Attr
 + Type(text) Just(left) skip(on)
 @ Type(output) Just(left) intens(high)
 _ Type(input) hilite(uscore)
)Body Window(50,5) Expand(\\)
+
%Enter From Date (mm dd yyyy):_z +_z +_z   +
%Enter To Date   (mm dd yyyy):_z +_z +_z   +
+
%Days between the above dates: @z      +
)Init
 .zvars = '(fromm fromd fromy tom tod toy days)'
  &zwinttl = 'Days between Dates'
)Proc
 ver (&fromm,range,1,12)
 ver (&tom,range,1,12)
 ver (&fromd,range,1,31)
 ver (&tod,range,1,31)
 ver (&fromy,num)
 ver (&toy,num)
 vput (fromm fromd fromy tom tod toy)
 &resp = .resp
*REXX(* resp)
 if resp = 'END' then exit
 null = ''
 if fromm = null then fromm = 0
 if fromd = null then fromd = 0
 if fromy = null then fromy = 0
 if tom   = null then tom   = 0
 if tod   = null then tod   = 0
 if toy   = null then toy   = 0
 fromx = fromy''fromm''fromd
 from = date('b',fromx,'s')
 tox   = toy''tom''tod
 to   = date('b',tox,'s')
 days = to - from
 /* this translate code to insert commas stolen from
    Doug Nadel */
 days=strip(translate('0,123,456,789,abc,def', ,
   right(days,16,','), ,
   '0123456789abcdef'),'L',',')
   bytes = strip(days)
EndRexx
)End
```

This is the REXX code to display the above panel:

```
/* rexx */
 address ispexec
 'addpop row(6) column(6)'
 do forever
   'display panel(pnprexx)'
   xrc = rc
   if xrc > 0 then leave
 end
 'rempop'
```

Verify a Data Set Name within Panel REXX (1.2)

PNVDSN and RXVDSN

This example demonstrates how to use TSO services and how to set messages within a Panel.

The REXX code to display the panel is:

```
/* rexx */
 address ispexec
 do forever
   'addpop'
   'display panel(pnvdsn)'
   drc = rc
   'rempop'
   if drc > 0 then leave
 end
```

Here is the ISPF Panel code:

```
)Attr DEFAULT(%+_)
  $ type(input) intens(low) hilite(uscore) caps(on)
)Body expand(\\) window(54,5)
+Enter/Verify Data Set Name:
%===>$verdsn                                        +
+
 +Enter any data set name and press enter to verify.
                     +F3 to Exit.
)INIT
  &zwinttl = 'Verify DSN Exists via Panel Rexx'
  &resp = .resp
*REXX(* resp zedsmsg zedlmsg)
 if resp = 'END' then exit
  parse value '' with zedsmsg zedlmsg
  msgvalue = msg()
  call msg 'off'
  if sysdsn(verdsn) = 'OK' then do
     zedsmsg = 'Confirmed'
     zedlmsg = verdsn 'does exist.'
     end
  else do
     zedsmsg = 'Failure'
     zedlmsg = verdsn sysdsn(verdsn)
     end
  call msg msgvalue
*ENDREXX
 if (&zedsmsg NE &Z)
    .MSG = ISRZ001
)END
```

What this code does in the *REXX routine is to use the SYSDSN function to test the provided data set and then set the short and long message. Note the short and long messages are defined in the *REXX statement as they are not included within the panel body. The statement to save the message state in msgvalue and then turn off messages if needed to prevent TSO messages if the data set name is not valid. The message state is then reset when the Panel REXX completes.

Then outside the REXX code there is the traditional ISPF Panel VER statement for the data set name and then the statement that generates the ISPF message.

Dynamically Changing the colors of the Text in ISPF Edit

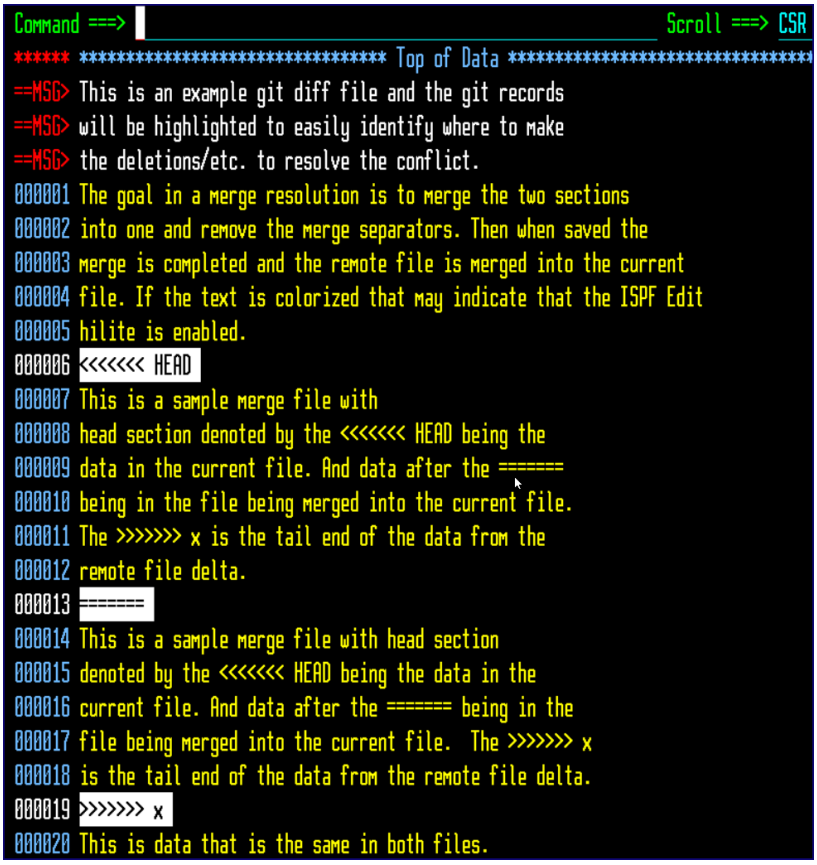
`PNEDITHL, RXEDITHL, and RXMEDHL`

There are times where your application needs to present the user with information that is very visible. Using normal ISPF Edit highlighting isn't adequate and you can't highlight using an Edit Macro.

For demonstration purposes I'm using code that I developed for the z/OS ISPF Git Interface (aka zigi) during a merge conflict resolution process. In this process the user is placed into ISPF Edit and must resolve all the conflicts. Git Diff provides that conflict delta information using three records that are inserted into the data:

<<<<<<< HEAD	Identifies that the records that follow are from the current branch file and are different from what is in the other branch file that is being merged into the current file.
=======	Is a separator between the current file and the merge file delta records and all records after it until the >>>>>>> record are in different from the current file.
>>>>>>> x	Identified the end of the delta records. The x is the name of the other branch.

This is an example, and, in this case, highlighting is on by default, so the rows are in yellow. Issue the command hilite off to turn off highlighting (suggest doing that in an edit macro):



The Panel Rexx code used is:

```
*REXX(* zdata zshadow zwidth)
  colorr = left('R',zwidth,'R')
  colorb = left('B',zwidth,'B')
  colorw = left('W',zwidth,'W')
  colorq = left('Z',zwidth,'Z')
  blank = left(' ',zwidth,' ')
  if length(zshadow) /= length(zdata) then
     zshadow = left(' ',length(zdata),' ')
  do i = 1 to length(zshadow) by zwidth
     Select
     when substr(zdata,i+8,1) = '+' then do
         len = substr(zdata,i+8,zwidth-8)
         len = translate(len,' ','00'x)
         len = length(strip(len))
         zshadow = overlay(colorb,zshadow,i+8,len+1)
     end
     when substr(zdata,i+8,1) = '-' then do
         len = substr(zdata,i+8,zwidth-8)
         len = translate(len,' ','00'x)
         len = length(strip(len))
         zshadow = overlay(colorr,zshadow,i+8,len+1)
     end
     when substr(zdata,i+8,7) = '>>>>>>>' then do
         zshadow = overlay(colorw,zshadow,i+1,6)
         len = substr(zdata,i+8,zwidth-8)
         len = translate(len,' ','00'x)
         len = length(strip(len))
         zshadow = overlay(colorq,zshadow,i+8,len+1)
         end
     when substr(zdata,i+8,7) = '<<<<<<<' then do
         zshadow = overlay(colorw,zshadow,i+1,6)
         len = substr(zdata,i+8,zwidth-8)
         len = translate(len,' ','00'x)
         len = length(strip(len))
         zshadow = overlay(colorq,zshadow,i+8,len+1)
         end
     when substr(zdata,i+8,7) = '=======' then do
         zshadow = overlay(colorw,zshadow,i+1,6)
         len = substr(zdata,i+8,zwidth-8)
         len = translate(len,' ','00'x)
         len = length(strip(len))
         zshadow = overlay(colorq,zshadow,i+8,len+1)
         end
      otherwise do
          len = substr(zdata,i+8,zwidth-8)
          len = translate(len,' ','00'x)
          len = length(strip(len))
          zshadow = overlay(blank,zshadow,i+8,len+1)
          end
      end
      end
 *ENDREXX
```

The ISPF Edit Panel, ISREDDE2, has been copied and modified for our use:

1. The menu bar was removed – no need for it in this context.
2. Panel REXX code is inserted in the)INIT section to dynamically change the data displayed by ISPF Edit.

This code isn't that complex, but it will take more than a few minutes to fully understand. Be aware that the code only has access to the records that are to be displayed and not to all of the records in the dataset.

1. The code starts with the *REXX key and includes access to:
    a. zdata – this is the main ISPF variable used by Edit (and View) that contains the data that is in the current display. Each row of data is appended to zdata until all of the data is in the display area.
    b. zshadow – this is the shadow variable that is used to add custom attributes. Each character of zshadow maps directly to a character in zdata.
    c. zwidth is the width of the ISPF panel display. This is required as the zdata is constructed one record at a time based on the screen width.
2. Next, we define some variables that are the screen width with different color attributes. This just makes coding easier and more readable later.
3. Then we create a zshadow variable that is the same width as zdata and fill it with blanks.
4. The fun stuff happens next. The rows are processed, one at a time, based on walking through zdata based on the screen width.
5. The code in this example will support both a git merge conflict file and a git diff file. The code is effectively the same:
6. Check for the substring in zdata that is at offset I plus 8, for a width of 1 or whatever we want to check for. This is a test for the data in column 1 of the actual data. The zdata character one is an attribute character, followed by the 6 digits for the row sequence number, then another attribute character and then in position 8 is the start of the real records data.
7. The next set of rexx statements:
    a. Get the actual user data for the current record into variable len
    b. Using the variable len next all hex 00's are translated to blanks
    c. Then len is update with the length of the variable len after removing all leading and trailing blanks
    d. Last the zshadow data is overlaid with the color that we want starting in zshadow position I plus 8 for the length of the variable len.
8. This is repeated for each set of text that we want to check for. In this example everything we want to look for must be in position 1 of the record.

Panel Scrolling Fields

`PNSCRL and RXSCRL`

Occasionally there are fields on a panel that are just too short for the data that is to be displayed, or entered, and the default fields will only truncate and not scroll. Fortunately, the option to enable field scrolling is easy.

Here is the sample Panel with a scrolling field:

```
)Attr Default(%+_)
  /* ISPF - Developers - Tips and Tricks */
 $ Type(input) hilite(uscore) caps(off) just(left) intens(high)
 @ type(output)
)Body window(70,10) expand(\\)
+-\-\%Sample Scrolling Field Panel+\-\-
%Command ===>_zcmd
+
+Enter data into the scrolling field and then scroll Right (F11) or
+Left (F10) to continue to enter data and to view it.
+
%==>$scrfld                                                    @z
   @lr1sc
+   (Scroll Left or Right for up to 255 characters)
+
)Init
 .cursor = scrfld
 .zvars = '(scind)'
)Proc
)Field
 Field(scrfld) ind(scind,'<>') len(255) scale(lr1sc)
)End
```

Let's look at the details of enabling a scrolling field:

1. Define the `)FIELD` section
2. Define each scrolling field using the `FIELD(field-name)` keyword and value
3. The `IND` keyword defines the scrolling indicator variable and characters. Using <> will use < to scroll left and > for scrolling right.
4. The `LEN` keyword defines the length of the scrolling field (255 in this case)
5. `SCALE` keyword defines a variable that will contain a scrolling ruler to help the user know the position within the field that is being displayed

This is what it will look like:

```
-------------------- Sample Scrolling Field Panel -------------------
Command ===>

Enter data into the scrolling field and then scroll Right (F11) or
Left (F10) to continue to enter data and to view it.

==>                                                              >
   ----+----1----+----2----+----3----+----4----+----5----+----6----+
  (Scroll Left or Right for up to 255 characters)
```

And this is the REXX code that can be used to experiment with the above panel:

```
/*   REXX   */
 Address ISPExec
 do forever
   'addpop'
   'display panel(pnscrl)'
   xrc = rc
   'rempop'
   if xrc > 0 then leave
   end
```

Dynamic Areas

PNDYN and RXDYN

This panel defines a dynamic area, but it also contains static text (field prompts), an output field (size), and an input field (somevar). All of these elements, and many more if needed, can coexist together.

The dynamic area defined in the panel is defined with `SCROLL(ON)` and `EXTEND(ON)`. This allows the application to scroll the data within the dynamic area when the user requests it, and ISPF will, at display time, extend the dynamic area depth so that the depth of the `)BODY` section equals that of the terminal that the panel is being displayed on.

It also shows how to use a shadow variable and character-level attributes to highlight individual characters within the string, overriding the field attribute for just one character, with no intervening space from a field attribute. The shadow variable is optional. Shadow variables are only needed when you want character level attributes.

```
)ATTR
 @ AREA(DYNAMIC)            SCROLL(ON) EXTEND(ON)
 01 TYPE(DATAOUT)           COLOR(RED)
 02 TYPE(DATAOUT)           COLOR(BLUE)
 03 TYPE(DATAOUT)           COLOR(GREEN)
 04 TYPE(DATAOUT)           COLOR(WHITE)
 r  TYPE(CHAR) COLOR(RED) HILITE(REVERSE)
 g  TYPE(CHAR) COLOR(GREEN) HILITE(REVERSE)
 b  TYPE(CHAR) COLOR(BLUE) HILITE(REVERSE)
 $  TYPE(TEXT)            COLOR(YELLOW)
)BODY
%------------------  EXAMPLE FOR USING A DYNAMIC AREA -------------------%
%COMMAND ===>_ZCMD                                        %SCROLL ===>_AMT +
%
+ This area is fixed.  size: &size
+
+ This is an input field%===>_somevar +
+
+This is extendable  @DYNAREA,DYNSHAD             @

      $This should be at the bottom of the screen when in full screen.
)END
```

This REXX routine shows how to set up a variable for use as a dynamic area variable, how to set up the shadow variable associated with that area, how to display the panel, and how to process UP and DOWN scrolling. RIGHT/LEFT scrolling is ignored in this example.

The data to be shown is set up to imbed the hex characters '01'x, '02'x, '03'x, and '04'x to represent colors red, blue, green, and white respectively. It will highlight each of the color names in their respective color, and show a list with colored color names, and line numbers.

The exec also sets up a shadow variable, which highlights the first character of each line with a color in reverse video. This 'shadowing' effect is only available in dynamic areas and allows a change of the attribute at the character level, with no intervening 'blank' space caused by a field attribute.

Note that the shadow variable (variables shadata and dynshad in this example) is optional. You could leave it out of the panel and the exec, and you would not have character level coloring.

The display is handled by setting a variable which contains the program data (dyndata) and then setting a second variable which will display the data (DYNAREA). The second variable is needed in this case to handle scrolling. The same manipulation with the length of the dyndata variable is done to the associated shadow variable, dynshad.

The variable dyndata starts with the first line to be displayed. When a scroll request is received, the program calculates the offset into DYNDATA of the first displayed line and creates DYNAREA from there.

To display the dynamic area panel, use the following code:

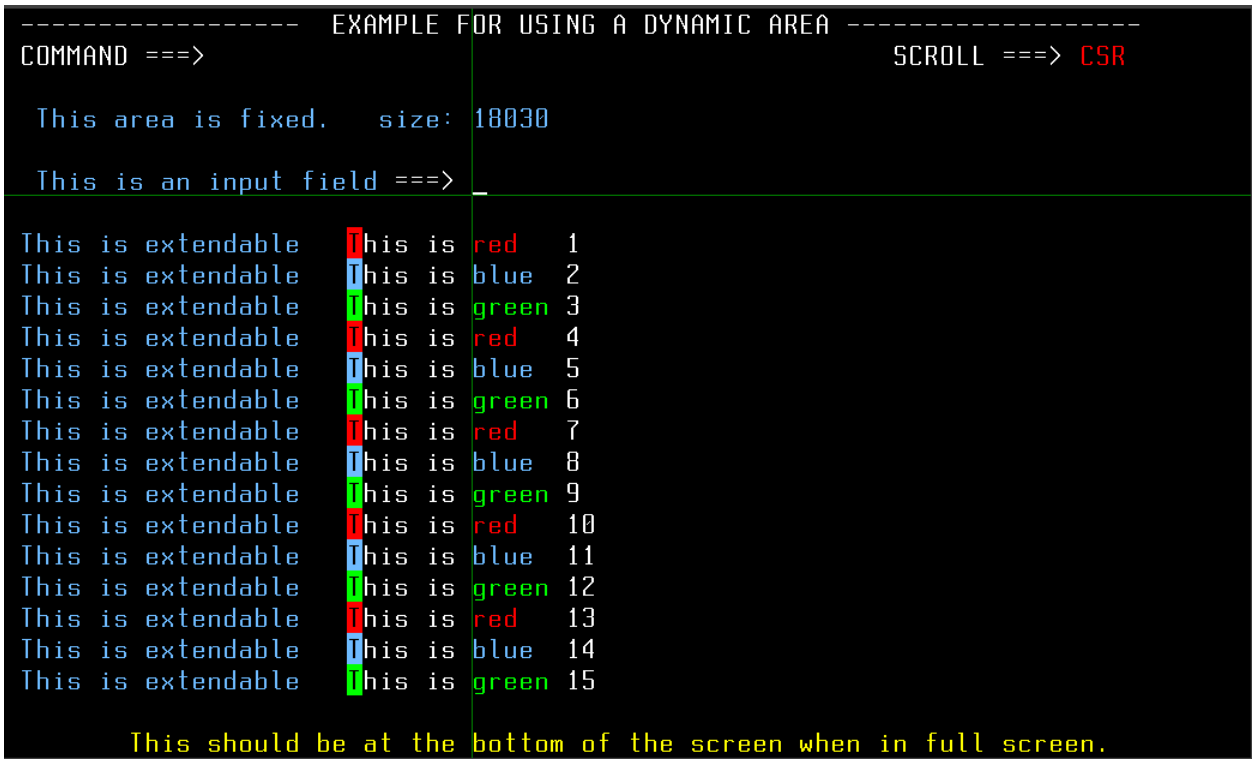
```
/* REXX - A Dynamic area example */
Address ISPEXEC                              /* Calls go to ISPF       */
red   = '01'x                                /* Assign colors to       */
blue  = '02'x                                /*     Attribute bytes    */
green = '03'x                                /*     found in the data  */
white = '04'x
maxlines = 600                               /* set max number of lines*/
dyndata  = ''                                /* initialize data        */
shadata  = ''                                /* initialize shadow var  */

Do  a = 1 to maxlines by 3                   /* Create some dummy data */
 dyndata=dyndata||white||left('This is'red  ||'red  'white||a  , 29)
 dyndata=dyndata||white||left('This is'blue ||'blue 'white||a+1, 29)
 dyndata=dyndata||white||left('This is'green||'green'white||a+2, 29)
 shadata=shadata||' r                          '
 shadata=shadata||' b                          '
 shadata=shadata||' g                          '
End

/*    Add a bottom of data maker to the end of the data      */
dyndata = dyndata||blue||centre(green||'BOTTOM'||blue,29,'*')
shadata = shadata||'                           '
curline = 1;                                 /* set current line #     */
/* ---------------------------------------------------------------------*/
/* Display loop until end or error                                      */
/* ---------------------------------------------------------------------*/

Do Until disprc > 0
 dynarea = substr(dyndata,1+(curline-1)*30) /* set dynamic variable  */
 dynshad = substr(shadata,1+(curline-1)*30) /* set shadow variable   */
 size = length(dynarea)                  /* Set a scalar variable   */
 'ISPEXEC DISPLAY PANEL(PNDYN)'          /* Display the data        */
 disprc = rc                             /* save return code        */
 'ISPEXEC VGET (ZVERB,ZSCROLLA,ZSCROLLN)' /* get scroll values       */
 Select                                  /* Process scrolling       */
   When(zverb = 'UP')   Then             /* Scroll up               */
    If zscrolla = 'MAX' Then             /*  if scroll was max      */
     curline = 1                         /*     scroll to top       */
    Else                                 /*  else a number is known */
     curline = max(1,curline-zscrolln);  /* (maximum is top)        */
   When(zverb = 'DOWN') Then             /* Scroll down             */
    If zscrolla = 'MAX' Then             /*  if scroll was max      */
     curline = maxlines                  /*     scroll to bottom    */
    Else                                 /*  else a number is known */
     curline = min(maxlines,curline+zscrolln); /* (max is bottom)    */
    Otherwise;                    /* could use left & right too     */
 End
End                                       /* End of display loop     */
```

This is what the dynamic area panel looks like when it is displayed.



Scrolling notes:

UP/DOWN scrolling

If all your data is in one variable, use (ZSCROLLN*area width) to locate the start of the displayed variable.

LEFT/RIGHT scrolling

Left/right scrolling usually involves creating a new variable to display. This is because the dynamic area uses a contiguous string.

When the user requests a scrolling action for the dynamic area, ISPF returns several variables which contain information about the requested scroll. This information is checked by the application and is then used by the application to reset the dynamic area variable.

ZVERB	Direction of scroll (UP, DOWN, LEFT, RIGHT)
ZSCROLLA	Amount to scroll (MAX, CSR, HALF, DATA, number, etc.)
ZSCROLLN	Number of lines to scroll

Additional information:

ISPF has several functions which can be used to help get additional information to assist in the definition of the dynamic area variable or scrolling requests.

PQUERY can help get information about areas contained within the panel. You specify the area name, and ISPF can return such things as the area type, and its width and depth.

The LVLINE built-in function can be called from the)INIT,)REINIT, or)PROC sections. It returns the line number of the last line within a dynamic, graphic, or scrollable area which was visible to the end-user on the currently displayed panel. This is extremely useful when the user can be in split-screen mode.

Your application can be coded to take advantage of larger-size terminal screens by using extendable areas, and by coding variable widths (on the)BODY section) and using the automatic expansion feature of ISPF. On the)BODY section, you code the EXPAND keyword, and then specify the characters you will place in the text of the panel when you wish to expand. The character in the)BODY that you place between these two characters will be used when the expansion is done.

Dynamically Set a Function Key to a value (e.g. RFIND)

PNDYNPFK

There will be occasions when you want to use a function key with 'RFIND' as the command in an application, but you find that RFIND is being intercepted by ISPF and not passed to your application. A Keylist or special ISPF application table is too much work.

This is one solution. See the sample code in RXTABLE for a different approach.

The solution shown here temporarily sets the 'RFIND' key to another value and resets the key afterwards. If there is no 'RFIND' key, then pf05 is used.

The 'RFIND' key is located in the INIT section and set to 'RRFIND'. The RRFIND command is caught in the PROC section, the ZCMD variable is set to 'RFIND' and the pfk is set back to its original value.

One caveat, if you start another panel from the one modifying the key(s), it will inherit the key settings, unless it is using keylist or is started with another application id.

ISPF Skeletons

ISPF skeleton definitions are stored in a skeleton library and accessed through the ISPF file-tailoring services. You create or change skeletons by editing directly into the skeleton library. ISPF interprets the skeletons during execution. No compilation or preprocessing step is required.

There are two types of records that can appear in the skeleton file:

- Data records - A continuous stream of intermixed text, variables, and control characters that are processed to create an output record.
- Control statements - Control the file-tailoring process.

The available control statements are:

```
)BLANK        )CM           )DEFAULT
)DO           )DOT          )ELSE
)ENDDO        )ENDDOT       )ENDREXX
)ENDSEL       )IF           )IM
)ITERATE      )LEAVE        )NOP
)REXX         )SEL          )SET
)SETF         )TB           )TBA
```

The file-tailoring services, listed in the order they are normally invoked, are:

FTOPEN Prepares the file-tailoring process and specifies whether the temporary file is to be used for output

FTINCL Specifies the skeleton to be used and starts the tailoring process

FTCLOSE Ends the file-tailoring process

FTERASE Erases an output file created by file tailoring.

File-tailoring services read skeleton files and write tailored output that can be used to drive other functions. Frequently, file tailoring is used to generate job control language statements for batch execution but can be used to generate many different outputs. The ability to code `)REXX` statements in the skeleton adds flexibility and power.

If output from file tailoring is not to be placed in a temporary file, the desired output file must be allocated to the ddname ISPFILE before invoking this service.

Simple Skeleton

`RXSKLCMD and SKCMDS`

Here is an example Skeleton used to generate a report on an ISPF Commands Table:

```
1. )TB  20
2. )SET CNT = 50
3. )DOT &TABLE
4. )SET CNT = &CNT + 3
5. )SEL &CNT > 50
6. 1 ISPF Command Table &TABLE       Date: &zdate Time: &ztime
7. )SET CNT = 1
8. )ENDSEL
9. 0CMD:  &ZCTVERB!&ZCTTRUNC
10. Action: &ZCTACT
11. Desc:   &ZCTDESC
12. )ENDDOT
```

In this skeleton the statements are:

1.)TB defines a tab which is used by the ! character to space data on the output record
2. Sets the counter to 50 to start with
3. Processed the table name in the variable &TABLE
4. Increments the counter by 3 for each table entry
5. Selects the next records to the)ENDSEL only if the counter is greater than 50
6. Inserts a record with a new page carriage control and a title
7. Resets the counter to 1
8. Ends the title record processing
9. Inserts 3 records into the output with information on the ISPF command table entry
12. Ends table processing

And here is the sample REXX code to process the skeleton:

```
/* -------------- REXX -------------------- *
| Demonstration of ISPF Skeleton Processing |
* ---------------------------------------- */
Arg Table
if table = '' then table = 'ISPCMDS'
Address ISPExec
'ftopen temp'
'ftincl skcmds'
'ftclose'
'vget (ztempf)'
"View dataset('"ztempf"')"
```

In this Code, the parameter passed is the name of the table to process, and if blank use ISPCMDS. A temporary data set is used for the location of the generated report as indicated on the FTOPEN by using the TEMP option. After the File Tailoring is closed (FTCLOSE) a VGET for the variable with the name of the temporary data set is performed and that data set is then processed using View.

Skeleton with REXX

Using the TSO Stack

RXSKLRX and SKREXX

This example demonstrates using REXX and TSO services with a Skeleton.

This is the REXX code to drive the sample:

```
/* --------------- REXX -------------------- *
| Demonstration of ISPF Skeleton Processing |
* ----------------------------------------- */
Address ISPExec
'ftopen temp'
'ftincl skrexx'
'ftclose'
'vget (ztempf)'
"View dataset('"ztempf"')"
```

And here is the sample Skeleton which places some data into the TSO Stack and then pulls the data from the stack while generating JCL.

```
)CM Demo use of TSO stack in an ISPF skeleton
)CM Make list
)REXX STACKN
 "delstack"
 queue 'Kilroy'
 queue 'was'
 queue 'here'
 stackn = queued()
)ENDREXX
)CM Make fixed front
//SKEL1    JOB (1),'BACKUP',CLASS=A,COND=(0,LT),REGION=64M
//L        EXEC PGM=IEBGENER
//SYSIN    DD DUMMY
//SYSPRINT DD SYSOUT=*
//SYSUT1   DD *
)CM Make SYSUT1 from list in stack
)DO N = 1 TO &STACKN
)REXX DATA
 parse pull data
)ENDREXX
&data
)ENDDO
)CM Make fixed back
//SYSUT2   DD SYSOUT=*
```

Sadly, there is no easy way to pass data via a stack or a stem into the Skeleton REXX. But you can put the data into a data set and then us TSO services to read that data with the Skeleton REXX (see the next example).

Using TSO Commands

RXSJKRXE and SJREXXE

Note that in Skeleton REXX you can use most TSO services and commands, including ALLOC, EXECIO, and FREE. No ISPF services may be used.

Here is an example demonstrating the use of TSO commands and the TSO Queue:

```
)CM Demo use of TSO stack in an ISPF skeleton
)REXX STACKN
"alloc f(in) ds('sys1.proclib(bpxas)') shr reuse"
'execio * diskr in (finis stem in.'
'free f(in)'
 "delstack"
 queue 'Reviewing BPXAS Proc'
 queue left('-',80,'-')
do i = 1 to in.0
  queue in.i
  end
 queue left('-',80,'-')
 queue '  '
 queue 'Including' in.0 'records.'
 stackn = queued()
)ENDREXX
)CM Get the data from the stack
)CM The variable STACKN is passed from the Skeleton REXX above
)CM with the number of records in the stack (queue)
)DO N = 1 TO &STACKN
)REXX DATA
 parse pull data
)ENDREXX
&data
)ENDDO
```

Passing a Variable to the Skeleton REXX (1.3)

RXSKLRXV and SKREXXV

Passing a variable from your code to be used within the REXX inside the Skeleton is reasonably easy. The *dsname* is passed to the REXX exec using arg and that places it into the variable pool.

```
/* --------------- REXX ------------------------------------------ *
 | Demonstration of ISPF Skeleton Processing with a passed variable |
 * ---------------------------------------------------------------- */
 arg dsname
 Address ISPExec
 'ftopen temp'
 'ftincl skrexxv'
 'ftclose'
 'vget (ztempf)'
 "View dataset('"ztempf"')"
```

Then the ISPF File Tailoring process knows to provide the variable to the Skeleton REXX as the variable name is on the)REXX statement. Any variable name on the)REXX statement is both passed to the Skeleton REXX and returned to the File Tailoring environment (see STACKN).

```
)CM Demo use of TSO stack in an ISPF skeleton with a passed variable

)REXX STACKN dsname
 "delstack"
 x = listdsi(dsname)
 queue 'DSName is:' sysdsname
 queue 'Volser is:' sysvolume
 queue 'RECFM:' sysrecfm 'LRECL:' syslrecl 'BLKSIZE:' sysblksize
 stackn = queued()
)ENDREXX

)CM Insert the information from the LISTDSI
)DO N = 1 TO &STACKN
)REXX DATA
 parse pull data
)ENDREXX
&data
)ENDDO
```

ISPF Tables

PNTAB and RXTAB and RXTABLE

Tables are one of the capabilities of ISPF that are used frequently and yet rarely fully understood. There are so many nuances with the use of tables that are left to the developer. This section will attempt to clarify ISPF Tables.

Find

Find is different, in the opinion of the authors, than Locate (see Locate below). The author prefers to use Find to search for a string in any, or a subset, of the variables within a row.

Here is a code snippet from RXTAB for doing a FIND.

```
save_floc = 0    /* Save last find location */
save_find = null /* save last find string */
Tbdispl . . .
if zcmd /= null then
if  abbrev("FIND",word(zcmd,1),1) = 1 then
   save_floc = ztdtop
if zcmd = 'RFIND' then zcmd = 'FIND' save_find
Select
When abbrev("FIND",word(zcmd,1),1) = 1 then do
 find = translate(word(zcmd,2))
 save_find = find
 wrap   = 0
 if save_floc > 0 then do
   'tbtop test'
   'tbskip test number('save_floc')'
 end
 do forever
   'tbskip test'
   if rc > 0 then do
     'tbtop test'
     'tbskip test'
     if wrap = 1 then do
       zedsmsg = 'Not Found'
       zedlmsg = find 'string not found in any member' ,
         'name. Try again.'
       'setmsg msg(isrz001)'
       leave
     end
     else wrap = 1
   end
    if pos(find,translate(vitem)) > 0 then do
      'tbquery test position(row)'
      crp = row
      save_floc = row
      if wrap = 1 then do
        zedsmsg = 'Wrapped'
        zedlmsg = 'Find restarted at the top of the table.'
        'setmsg msg(isrz001)'
      end
      else do
        zedsmsg = 'Found'
        zedlmsg = 'Found in row:' row
        'setmsg msg(isrz001)'
      end
      leave
    end
  end
 end
```

This routine will search for the provided string in a single variable (vitem), however multiple row variables can be included in the ‘`if pos(' test`.

This code also detects when the end of the table is reached and will then ‘wrap’ to the top and start the find from there again, while issuing a message that the find has ‘wrapped’.

Enabling RFIND

Repeat Find, sadly, is not enabled by default with the out of the box ISPF. There are three ways to enable RFIND that the author has identified, each with their own requirements.

Update the ISPF Command Table

This method requires that an ISPF command table be updated with the following command:

Verb:	RFIND
Truncation:	0
Action:	&USRRFIND
Description:	User Repeat Find

This adds a command that has no action as the variable &USRRFIND will start out blank. The application must then set &USRRFIND to PASSTHRU to allow the RFIND command to be processed by the application, then reset it to blank when the application no longer needs to offer this command. A suggested approach is to enable before the use of TBDISPL and then after the TBDISPL completes, and before any actions are processed, then reset it.

Note that &USRRFIND is just an example, use any variable name that makes sense.

Within the code the developer must test if the command is RFIND and then change it to be processed to FIND followed by the FIND string

Create a Temporary Command Table for the Application

This method requires more coding but does not require that the sites ISPF administrator update any of the sites ISPF command tables.

Note that using this method requires a change if you use the REXX Compiler as sysvar(‘sysicmd’) will return a null value when compiled. However, it appears that sysvar(‘syspcmd’) does return the correct value when compiled.

For this to work the application code, very early, must do the following:

```
Address ISPExec
"VGET ZAPPLID"
if zapplid <> "TABT" then do
 "TBCreate tabtcmds names(zctverb zcttrunc zctact" ,
          "zctdesc) replace share nowrite"
 zctverb = "RFIND"
 zcttrunc = 0
 zctact = "&USRRFIND"
 zctdesc = "User controlled Repeat Find (RFIND)"
 "TBAdd tabtcmds"
 "Select CMD(%"sysvar('sysicmd') options ") Newappl(tabt)" ,
        "passlib scrname(TABLE)"
 Xrc = rc
 "TBClose tabtcmds"
 Exit xrc
end
```

What this does is:

1. Enter the ISPF environment
2. VGET the ISPF system variable ZAPPLID
3. Compare the ZAPPLID to our desired application ID of TABT
4. If they match, then we do nothing and all through to processing the real code
5. If they do not match, then
    a. Create the temporary ISPF Commands Table. It must start with the application ID (TABT) and is defined with NOWRITE since there is no need to save it, with REPLACE in case it already exists and wasn't removed during a prior execution, and with SHARE to allow the table to be used across ISPF screens.
    b. Define the four command table row variables:
        i. zctverb is the command name
        ii. zcttrunc is the number of characters to abbreviate the command (0 is no abbreviation)
        iii. zctact is the action the command will initiate. For our purposes we specify an ISPF variable that we will update in the application.
        iv. zctdesc is a description (can be blank)
    c. Add that row to the command table
    d. Reinvoke the active exec with a NEWAPPL which will enable the command table to be used.
    e. Upon return from the exec the table is closed, and the exec Exits.

Alternate Method to Update the ISPF Command Table

An alternate method that does NOT require recursion is this approach:

```
Address ISPExec
zctverb = 'RFIND'
zctact = "&USRRFIND"
zctdesc = "User controlled Repeat Find (RFIND)"
zcttrunc = 0

'vget (zsctpref)'
ctab = zsctpref'cmds'
'tbtop' ctab
'tbscan' ctab 'arglist(zctdesc) condlist(EQ) Next'
if rc > 0 then 'tbadd' ctab

usrrfind = null
'vput (usrrfind)'
```

The process is:

1. Address ISPF services
2. Define the ISPF Command table variables
    a. zctverb is the command name RFIND
    b. zctact is the action – in this case a symbolic &USRRFIND
    c. zctdesc is a description
    d. zcttrunc is a truncation value – 0 means no truncation
3. Next get the ISPF variable for the site command table prefix
4. Define the command table name by adding a CMDS suffix

5. Move to the top of the command table
6. Scan for the description (zctdesc)
7. If not found (return code > 0) then add our command to the table
8. Next set the variable (usrrfind) to null (null is ‘’)
9. And put the variable into the ISPF variable pool.

Next before every TBDISPL call the usrrfind must be set to PASSTHRU:

```
usrrfind = 'PASSTHRU'
'vput (usrrfind)'
```

And then after the TBDISPL reset the variable:

```
usrrfind = ''
'vput (usrrfind)'
```

Locate a Row

Locate processing is less complex than a FIND as it is typically used to locate the 1st row with a string that matches the prefix of the selected value.

In this example the Locate always starts at the 1st row in the table. Remove the TBTOP to have the Locate process start at the currently displayed top of the table.

```
When abbrev("LOCATE",word(zcmd,1),1) = 1 then do
 vitem = translate(word(zcmd,2))
 'tbtop test'
 crp = 0
 'tbscan test arglist(vitem) position(crp) condlist(ge)'
 if rc = 0 then do
   zedsmsg = 'Found'
   zedlmsg = word(zcmd,2) 'was found in row' crp
   'Setmsg msg(isrz001)'
 end
 else do
   zedsmsg = 'Not Found'
   zedlmsg = word(zcmd,2) 'was not found'
   'Setmsg msg(isrz001)'
 end
end
```

In this example the code does the following:

1. Assuming this code is within a Select process the When tests for an abbreviation of the word LOCATE, allowing it to be abbreviated to a single character. Since the command would have multiple words, with the locate value being the second word, the abbreviation test must only work with the 1st word.
2. The row variable to be tested is then set to the translated to upper case 2nd word of the command.
3. The table is then reset to the top and the current row pointer set to 0.
4. Execute TBSCAN to scan through the table comparing the locate value to the value in each row. A successful match will place the row number in the variable crp.
5. The scan is using the CONDLIST option of GE, which means that the scan will stop when it finds a row that is greater than, or equal, to the locate value.
6. Based on the return code, a message is generated to the user that the value was found, or not found. When using a CONDLIST(GE).

Selecting Multiple Rows for Processing

Selecting multiple rows at one time requires a few additional lines of coding:

1. Set a variable (e.g. CRP) with the top displayed row of the table to 0 before the 1st TBDISPL
2. Set the ZTDSELS variable to 0 before the 1st TBDISPL
3. In the TBDISPL code section
    a. Test ZTDSELS and if 0 then
        i. TBTOP
        ii. TBSKIP to the save top display row (CRP)
        iii. TBDISPL with the PANEL parameter
    b. If ZTDSELS is > 0 then
        i. TBDISPL without the PANEL parameter
    c. Save the ZTDTOP value in a variable (e.g. CRP)

And that is it – the code will now support selecting multiple rows with one enter key.

Check out the full table processing code below to see how this all fits together.

Full Example

Using ISPF Tables can be challenging because it isn't obvious how to do some of the things that we are used to with the IBM, and other ISPF dialogs. But don't let that stop you – this chapter demonstrates some of the techniques that will make your ISPF table look polished and professional.

The first thing is to construct the ISPF Table display Panel. This panel was generated using the ISPF Edit Model command and selecting the TBDISPL Table Display Panel:

```
)Attr Default(%+_)
/* % type(text  ) intens(high) Defaults displayed for */
/* + type(text  ) intens(low )      information only  */
/* _ type( input) intens(high) caps(on ) just(left )  */
  ! type( input) intens(high) caps(on ) just(left ) hilite(uscore)
  ^ type(output) intens(low ) caps(off) just(asis )
)Body  Expand(\\)
%-\-\- Sample Table Display Panel -\-\-
%Command ===>_zcmd       \ \%Scroll ===>_amt +
%
+Commands: %REF+refresh the display%Only string+Filter on string
+Selection:%S+Browse%SS/SS+Selection Range%S##+Selection Count
+
%Sel      Status     Item
+
)Model
!z +     ^z       + ^z            +
)Init
 .ZVARS = '(vsel vstate vitem)'
 &amt = CSR
 &vset = 0
)Reinit
)Proc
 IF (&ZCMD = &Z)
    if (&ztdsels = 0000)
       &row = .csrrow
       if (&row ^= 0)
           if (&vsel = &z)
               &vsel = S
               &vset = 1
    if (&ztdsels ^= 0000)
         &row = &z
    IF (&vsel ^= &Z)
       if (&vsel = '=')
          &vsel = &osel
    &osel = &vsel
    if (&row = 0)
        &vsel = &Z
)End
```

This panel does several things:

1. Define the input field attribute of ! symbol to be used for the row selection with a hilite of uscore to show on the display with a _ to identify where to enter the selection command.
2. Defined an output field attribute using the @ symbol with caps off and justification of asis.
3. The scroll amount, which is always the 2nd input field on a table panel, is set to CSR (because the author prefers cursor scrolling instead of PAGE/HALF/##).

4. If the user has the cursor on a row and presses enter there is no selection, but this simulates a selection.
5. Note that IF statements are indentation sensitive.

Here is the REXX code, with comments, that displays the above panel.

```
/* ----------------------- REXX ----------------------- *
| Sample REXX Code to drive an ISPF Table to demonstrate |
| 1. Processing a command (Refresh)         |
| 2. Processing individual line selections  |
| 3. Processing multiple line selections    |
| 4. Processing a range selection (SS/SS)   |
| 5. Processing a range count (S###)        |
| 6. Demonstrate Only string capability     |
| 7. Uses IBM's ISRZ001 message (zedsmsg/zedlmsg)        |
| 8. Use of Find and RFIND     |
| 9. Use of Locate (TBSCAN)    |
* ------------------------------------------------------ */
```

The above is a short description of what the code is intended to do. This gives anyone reading the code a starting point to understand the process.

This next section does several things:

1. Establishes the REXX addressing environment to the ISPF (ISPExec) environment.
2. Checks for a specific ISPF Application ID of TABT, and if that isn't the active APPLID then create a temporary ISPF Commands Table and add to it the Repeat Find command using a variable that will be updated later. Without this in the command table the Repeat Find command will never be passed to our code.
3. Next reinvokes the REXX code with the ISPF NEWAPPL keyword to place the code under the desired APPLID so that the command table will be in use.
4. Also set the screen name to Table for this screen.
5. Upon return then close the table and exit.

```
/* ---------------------------------- *
| Define Addressing to ISPFExec (ISPF) |
* ---------------------------------- */
Address ISPExec
/* --------> Enable Repeat Find <------------ *
| Check for APPLID (TABT) and if not then    |
|   - create an ISPF Commands Table          |
|   - add RFIND to it with a variable        |
|   - reinvoke the current exec with NEWAPPL |
|   - on return close the commands table     |
* ------------------------------------------ */
"VGET ZAPPLID"
if zapplid <> "TABT" then do
  "TBCreate tabtcmds names(zctverb zcttrunc zctact" ,
    "zctdesc) replace share nowrite"
  zctverb = "RFIND"
  zcttrunc = 0
  zctact = "&USRRFIND"
  zctdesc = "User controlled Repeat Find (RFIND)"
  "TBAdd tabtcmds"
  "Select CMD(%"sysvar('sysicmd') options ") Newappl(tabt)" ,
    "passlib scrname(TABLE)"
  x_rc = rc
  "TBClose tabtcmds"
  Exit x_rc
End
```

Next the code's defaults are defined. In this case only the null variable is defined.

```
 /* --------------- *
 | Define defaults |
 * --------------- */
 null    = ''  /* Null variable for compare */
```

The Refresh label is used to restart processing at this location when the user invokes the Refresh command on the table.

```
 /* --------------------------- *
 | Label for Refresh Processing |
 * --------------------------- */
Refresh:
 /* ------------------------------------------------------ *
 | Define our ISPF Table with two variables and NOWRITE as |
 | it will not be saved.        |
 * ------------------------------------------------------ */
```

At this point the table is created using TBCreate. For our purposes the table name is a literal with the name of test. This could easily be a variable so that the table name can be changed at will for various reasons.

The next set of code generates test data to fill the table with. This section can be replaced by your own code to populate the table or use an existing table.

```
 "TBCreate test names(vitem vstate) nowrite"
 /* ------------------------------------ *
 | Now fill the ISPF table with test data |
 * ------------------------------------ */
 str = 'AbcDefGhiJklMnoPqrStuVwxYz0123456789'
 strc = 1
 vstate = null
 do i = 1 to 100
   vitem =  substr(str,strc,3) 'Test data number:' right(i+1000,3)
   strc = strc +3
   if strc > 34 then strc = 1
   "TBAdd test"
 End
```

With the table populated it is time to define our defaults for processing. These variables are used to control some of the table processing actions.

```
 /* ---------------------------------------------- *
 | Table defined and populated - get to the top now |
 | and define our working variables    |
 * ---------------------------------------------- */
 ztdsels = 0      /* Define # rows selected to 0 */
 crp     = 1      /* Define the starting row to 1 */
 save_floc = 0    /* Save last find location */
 save_find = null /* save last find string */
```

A do forever loop is used to process the table until the user uses F3 (END) to terminate processing:

```
/* ------------------------------------------------ *
| Process the table forever - until we say we're done |
* ------------------------------------------------ */
do forever
```

Before the Table Display (`TBDISPL`) the `usrrfind` variable is changed to PASSTHRU and VPUT is used to update the variable in the ISPF pool. This allows the Repeat Find (RFIND) command to be passed to this code for processing. See Enabling RFIND, above for more information.

```
/* ---------------------------------------- *
| Define USRRFIND for Passthru to enable RFIND |
* ---------------------------------------- */
usrrfind = 'PASSTHRU'
'vput (usrrfind)'
```

Next is the key to the table processing with the test of ztdsels (# of selected rows) so that we can process multiple rows at one time.

```
/* ------------------------------------------------- *
| Test ZTDSELS - if greater than zero than display the |
| table without the Panel name            |
| - if zero than display with the Panel name          |
|   after 1st reseting the display to the last location|
* ------------------------------------------------- */
if ztdsels > 0
then 'tbdispl test'
else do
  'tbtop test'
  'tbskip test number('crp')'
  'tbdispl test panel(pntab)'
End
```

The return code must be saved so that it can be tested but it shouldn't remain in the rc variable since the return code will be changed by the code that resets the USRRFIND variable using VPUT, which is done next to allow RFIND to be used by native ISPF routines (e.g. Browse, Edit, …).

```
trc = rc            /* Save the TBDISPL return code */
/* ---------------------- *
| Reset USRRFIND to blank |
* ---------------------- */
usrrfind = ''
'vput (usrrfind)'
```

At this point, after the USRRFIND has been reset, is the time to determine if the user used F3 to terminate processing and if so, then leave the do forever loop.

```
/* ------------------------------------------------------- *
| If the return code from the table display (TBDISPL) is     |
| greater than 4 then we are done so leave the forever loop. |
* ------------------------------------------------------- */
if trc > 4 then leave
```

This is where the real processing occurs. There are both commands (zcmd) and row selections (vsel) and both a processed separately. For zcmd there is a need to check to see if the command is RFIND, something you'll never see if you didn't update the USRRFIND variable, and if it is RFIND then the zcmd must be changed to FIND followed by the last character string used for FIND. See Find for more information on supporting FIND, and Enabling RFIND for more information on RFIND.

```
/* ------------------------------------------------------- *
| Test the zcmd for any commands that we are to           |
| process.        |
* ------------------------------------------------------- */
if zcmd /= null then
if zcmd = 'RFIND' then do
  zcmd = 'FIND' save_find
  'tbtop test'
  'tbskip test number('save_floc')'
end
Select
  /* ------------------------------- *
   | Test if there is a command      |
   * ------------------------------- */
   if zcmd /= null then
  /* ------------------------------------------------ *
  | Find and Repeat Find processed.     |
  |            |
  |   Checks for the provided string in the specified |
  |   row variables.       |
  * ------------------------------------------------ */
  When abbrev("FIND",word(zcmd,1),1) = 1 then do
    find = translate(word(zcmd,2))
    save_find = find
    wrap   = 0
    if save_floc > 0 then do
      'tbtop test'
      'tbskip test number('save_floc')'
    end
    do forever
      'tbskip test'
      if rc > 0 then do
        'tbtop test'
        'tbskip test'
        if wrap = 1 then do
          zedsmsg = 'Not Found'
          zedlmsg = find 'string not found in any member' ,
                         'name. Try again.'
         'setmsg msg(isrz001)'
        leave
        end
        else wrap = 1
      end
      if pos(find,translate(vitem)) > 0 then do
        'tbquery test position(row)'
        crp = row
```

```
            save_floc = row
            if wrap = 1 then do
                 zedsmsg = 'Wrapped'
                 zedlmsg = 'Find restarted at the top of the table.'
            'setmsg msg(isrz001)'
            end
            else do
               zedsmsg = 'Found'
               zedlmsg = 'Found in row:' row
               'setmsg msg(isrz001)'
            end
            leave
          end
        end
      end
```

This section of code demonstrates doing a Locate. See Locate a Row for more information on Locate processing.

```
      /* ------------------------------------ *
      | Demonstrate Locate Processing on the row |
      * ------------------------------------ */
      When abbrev("LOCATE",word(zcmd,1),1) = 1 then do
        vitem = translate(word(zcmd,2))
        'tbtop test'
        crp = 0
        'tbscan test arglist(vitem) position(crp) condlist(ge)'
        if rc = 0 then do
          zedsmsg = 'Found'
          zedlmsg = word(zcmd,2) 'was found in row' crp
          'Setmsg msg(isrz001)'
        end
        else do
          zedsmsg = 'Not Found'
          zedlmsg = word(zcmd,2) 'was not found'
          'Setmsg msg(isrz001)'
        end
      end
```

The Only command (abbreviated to O) is used to limit the rows displayed to only those with the specified character string. This can be done a number of ways, but this is a simple technique, which requires using the Reset command to restore all the other rows to the table display.

```
    /* ------------------------------------------------------------ *
    | Only will check for the provided string (case insensitive)  |
    | in any location within the row variables.     |
    * ------------------------------------------------------------ */
    When abbrev("ONLY",word(zcmd,1),1) = 1 then do
      str = subword(zcmd,2)
      if strip(str) = null then do
        zedsmsg = 'Only Invalid'
        zedlmsg = 'Only requires a string parameter.'
        'Setmsg msg(isrz001)'
      end
      else do
        'tbtop test'
        do forever
          'tbskip test'
          if rc > 0 then do
             'tbtop test'
             leave
          end
          if pos(str,translate(vitem vstate)) = 0 then
          'tbdelete test'
        end
      end
    end
```

Refresh closes the active ISPF Table and then restarts the code at the label Refresh which recreates the table from scratch.

```
    /* ------------------------------------------------------ *
    | Refresh will close the table and then signal (goto) the |
    | Refresh label to start over. |
    * ------------------------------------------------------ */
    When abbrev("REFRESH",zcmd,1) = 1 then do
      'tbend test'
      signal Refresh
    end
```

Because no one is perfect there needs to be a way to let the user know when they enter a command that the code can't handle. This is one way to do that.

```
    /* ------------------------------------------------------ *
    | Process unknown commands here - basically let the user |
    | know we didn't know what the command is that they      |
    | entered.       |
    * ------------------------------------------------------ */
    Otherwise do
      zedsmsg = 'Unknown'
      zedlmsg = zcmd 'is an unknown command - try again.'
      'Setmsg Msg(isrz001)'
    end
  end
```

After processing any command, it is time to check for any row selections. To keep the table from scrolling up, or down, this code saves the current top row (ztdtop) so that the table display can be repositioned when it is next displayed using TCDispl.

```
   /* ------------------------------------------------------------ *
   | Process the line selections if the selection variable (vsel) |
   | is not null.          |
   * ------------------------------------------------------------ */
   if strip(vsel) /= null then do

     /* ------------------------------------------------ *
     | This code is to support the selection by the user |
     | pressing enter on the row and not entering any    |
     | selection value (point and shoot).  |
     |            |
     | 1. save the current top row          |
     | 2. if the row selected via enter (vset = 1)       |
     |    then tbtop and then skip to that row           |
     * ------------------------------------------------ */
     crp = ztdtop
     if vset = 1 then do
       vset = 0
       "TBTop test"
       "TBSkip test NUMBER("row")"
     end
```

This is where the individual row selections are processed. This example is very simple, but it demonstrates what you can do. There are also examples of handling multiple row selections and range selections. See Selecting Multiple Rows for Processing above for more information.

```
     Select
       /* ------------------------ *
       | Process an individual row |
       * ------------------------ */
       When vsel = 'S' then do
         zedsmsg = null
         zedlmsg = 'Row data:' vitem
         'setmsg msg(isrz001)'
         say 'Row data:' vitem
         vstate = 'Selected'
         'tbput test'
       End
```

This next section of code demonstrates how to process a range of rows. See Selecting Multiple Rows for Processing for more information on this process.

```
       /* ------------------------------------------------------------------
*
       | Process a range selection where the 1st row is selected using SS |
       | and then the last row with an SS.     |
       * ------------------------------------------------------------------
*/
       When vsel = 'SS' then do
         /* ---------------------------------- *
         | When only 1 SS is specified          |
         | if block indicator on then process it |
         | if not set the block indicator and    |
```

```
        | get the 1st row location |
        * ---------------------------------- */
        if ztdsels = 1 then do
          if block = 1 then call do_ss
          else do
 block = 1
 vstate = 'SS'
 'tbput test'
 'tbquery test position(row1)'
 row2 = 0
          end
        end
        /* ------------------------------------ *
        | When Both from SS and to SS are provided |
        | - get the 1st row          |
        | - set row2 to 0|
        | - set block indicator to on |
        * ------------------------------------ */
        else do
          'tbquery test position(row1)'
          row2 = 0
          block = 1
        end
      end
      /* ------------------------------------ *
      | Process a selection with a count S###) |
      * ------------------------------------ */
      When left(vsel,1) = 'S' then do
        /* ------------------------------------------ *
        | Extract the count from the selection command |
        * ------------------------------------------ */
        vcount = substr(vsel,2)
        /* ------------------- *
        | Save current top row |
        * ------------------- */
        save_top = ztdtop
        /* -------------------------- *
        | Validate that it is numeric |
        * -------------------------- */
        if datatype(vcount) /= 'NUM' then do
          zedsmsg = null
          zedlmsg = 'Invalid selection' vsel '- expecting S###'
          'setmsg msg(isrz001)'
        end
        else do
          /* ------------------------ *
          | Get the current row number |
          | and the # of rows in table |
          * ------------------------ */
          'tbquery test position(row1) rownum(rows)'
          /* ------------------------------ *
          | Identify the last row to process |
          * ------------------------------ */
          row2 = row1 + (vcount - 1)
          /* ------------------------------ *
          | Test for going past end of table |
          * ------------------------------ */
          if row2 > rows then do
 zedsmsg = null
          zedlmsg = vsel 'is invalid as it goes beyond the end of the
table.'
 'setmsg msg(isrz001)'
          end
          else do
 /* ------------------------------------------------ *
 | Process the 1st row and update the state in the row |
```

```
 * ------------------------------------------------ */
 say 'Row data:' vitem
 vstate = 'Selected'
 'tbput test'
 /* ------------------------------ *
 | Skip 1 row and process the rest |
 * ------------------------------ */
 'tbskip test'
 do (row2-row1)
  'tbget test'
  say 'Row data:' vitem
  vstate = 'Selected'
  'tbput test'
  'tbskip test'
 end
 'tbtop test'
 'tbskip test number('save_top')'
           end
         end
       end
```

As for unknown commands, it is only polite to inform the user if they enter any row selection options that the code is unable to process.

```
       /* ------------------------------ *
       | Process unknown row selections |
       * ------------------------------ */
       Otherwise do
         zedsmsg = 'Unknown'
         zedlmsg = vsel 'is an unknown row selection - try again.'
         'setmsg msg(isrz001)'
       end
     end
     vsel = null
   end
 end
```

At this point the user has issued the F3 (END) command and processing is completed. The table must be closed, and the code can exit processing.

```
 /* ------------------------------ *
 | All done so close out the table |
 | and Exit           |
 * ------------------------------ */
 "TBEnd test"
 Exit 0
```

Here is the subroutine used for range processing.:

```
 /* ------------------------------ *
 | Process the SS/SS Range command |
 * ------------------------------ */
Do_SS:
 /* ------------------------------- *
 | Get the row number of the 2nd SS |
 * ------------------------------- */
 'tbquery test position(row2)'
 /* ---------------------------------------------------------- *
 | Test to see which SS entry is the 1st row to process and |
 | arrange row1 to the lowest row number and row2 to the    |
 | highest row number. This allows the range to be selected |
 | in either direction from any panel.        |
```

```
* ---------------------------------------------------------- */
if row2 < row1 then do
  rowx = row2
  row2 = row1
  row1 = rowx
end
/* ------------------------------------------------ *
| Position to the top row and then skip down to the |
| 1st row to process.    |
* ------------------------------------------------ */
'tbtop test'
'tbskip test number('row1')'
/* ------------------------ *
| Process the selected rows |
* ------------------------ */
do row2 - row1 +1
  /* ---------------- *
  | Get the row data |
  * ---------------- */
  'tbget test'
  say 'Row data:' vitem
  /* ------------------------------- *
  | Update the row state variable    |
  | Then update the row in the table |
  | Then skip to the next row.      |
  * ------------------------------- */
  vstate = 'Selected'
  'tbput test'
  'tbskip test'
end
/* ------------------------------------------------------ *
| When done then reset our working variables and return |
* ------------------------------------------------------ */
row1 = 0
row2 = 0
block = 0
ztdsels = 0
'tbtop test'
'tbskip test number('ztdtop')'
Return
```

Adding Rows to a Table When Needed

`PNDYNTP, PNDYNTBL and RXDYNTBL`

In the ISPF Developers Guide and Reference is a section with the title of: “Adding table rows dynamically during table display scrolling” that provides information on how to add rows to a table when needed. This is a fantastic time saver when the table may have many thousands of rows. It may just be me (Lionel), or perhaps it’s the IBM documentation, but this did not provide enough information to implement a dialog using this capability. But with some trial and terror a working example was finally arrived at and the code has been used in a production application with great success.

Note that the ISPF panel used for the TBDISPL is no different from any other table panel. The use of the TBTOP, TBSKIP, and TBDISPL is also no different.

What’s different is that when the application starts it only loads a small number of records into the table before displaying the table. Then as the user scrolls, the TBDISPL service, will return to the application with a request for more table rows to be added.

For our example we create a REXX stem with 10,000 stem records. This stem information is basic as it is for demonstration purposes.

```
/* 1st build a stem with a bunch of records */
str = 'one two three four five six seven eight nine ten'
do i = 1 to 10000
  off = i//10
  if off = 0 then off = 10
  stem.i = word(str,off) date(w) time('l')
end
stem.0 = 10000
```

Then the Table is created using TBCREATE.

```
/* now create the table and prime it with 100 records */
tbl = 'DT'time('s')   /* random table name */
Address ISPExec
'tbcreate' tbl 'names(data date time) nowrite'
ztdamtl = 100       /* set initial # of rows to add */
last = 1            /* initialize the last record added */
ztdret = 'DOWN'     /* instruct tbdispl to return to us on DOWN */

c = 0              /* set row added counter    */
x = add_rows()     /* call routine to add rows */
```

The key things here are:

1. ztdamtl is a table display variable that is set by TBDISPL to instruct the application how many rows need to be added. Since we are just starting, we are setting it to 100 before calling the add_rows subroutine.
2. last is a variable used by the application to keep track of the last stem that was added.
3. The key variable is ztdret which instructs TBDISPL to return to the application when, in this case the DOWN command, occurs if it will require more records to satisfy the scrolling request.
4. c is a counter to keep track of how many rows have been added.

The add_rows subroutine should be very familiar to anyone who has worked with ISPF tables as it just loops through a stem adding the stem values to the table using tbadd.

```
Add_Rows:
  if bottom = 1 then return 8   /* If already at the bottom then return
                                   quickly */
  msg1 = 'Adding more records.'
  msg2 = 'Starting with record' last
  call pop                      /* Inform the user of the adds in
                                   progress */
  if ztdamtl < 100              /* add at least 100 each time */
  then ztdamtl = 100
  'tbbottom' tbl                /* Get to the bottom of the table to
                                   start */
  do i = last to last+ztdamtl
    if i > stem.0 then do
      last = i
      return 8
    end
    parse value stem.i with data date time
    'tbadd' tbl
    c = c + 1
  end
  last = i
  return 0
```

In this example a popup is used to inform the user that additional records are being added to the table. This may, or may not, be something to have is a ‘real’ application but is used here so that when using the example, it is obvious additional rows are being added.

The ztdamtl variable is used to determine how many additional rows to add. In our case we test to determine if TBDISPL is requesting less than 100 rows and if so then the variable is set to 100. Adding 100 rows is transparent compared to adding 10, and by adding more it reduces the additional calls to add additional rows to the table. If the user requested ‘DOWN 999’, that is more than 100 records and the routine will honor that request (same for ‘DOWN MAX’.

It is important, especially if the table does not have keys, that the table be positioned at the bottom before adding additional rows. The TBBOTTOM does this for us.

The next section of code adds the requested rows to the table, starting with the last variable which contains the next record to be added from the last add process. When the add loop ends the last variable is reset to the value of ‘i’ from the loop, which is the next records to be added.

The return code from the add routine is either 0, for success, or 8 to indicate the end of the stem was reached and there are no more records that can be added.

Returning to the sample code from our subroutine, the next section of code displays the table:

```
dtop = 1                       /* set top of table variable */

do forever
  'tbtop' tbl                   /* Get to the top of the table */
  'tbskip' tbl 'number('dtop')'  /* Scroll down to where the user
                                 scrolled last */
  'tbdispl' tbl 'panel(pndyntbl)'  /* display the table */
  if rc > 0 then leave           /* Leave if F3/End requested */
  dtop = ztdtop                 /* save the top displayed row */
  'vget (zscrolla)'             /* get scroll amount          */
  /* query the table for information */
  'tbquery' tbl 'rownum(totrows) position(pos)'
```

In this code the key items to note are:

1. The vget of the zscrolla value. This variable contains the number of rows to scroll
2. The TBQUERY is used to get the total rows currently in the table and the cursor position. Both will be used shortly.

This next section of code processes the TBDISPL request to add additional rows, but it is primarily to make sure that the table will continue to display with the correct row as the top row in the table display.

If more rows are needed because the user has scrolled to the point that TBDISPL needs them to display, the ztdadd variable will be set with a value of YES.

This section checks to determine if there are really rows to add. In this routine the key is that if no additional rows are to be added because all rows have already been added, then the ztdadd variable is set to NO.

Then the zscrolla value is tested, and if it is CSR and the current cursor position is 0 then the top row to be displayed is adjusted for the scrolling request.

```
if ztdadd = 'YES' then      /* Add more rows ? */
if totrows = stem.0         /* if Yes have all records been added */
then ztdadd = 'NO'          /* If all added change ztdadd to NO */
if zscrolla = 'CSR' then    /* Test scroll amount for CSR */
if pos = 0 then do          /* If CSR then is cursor at 0 */
  dtop = dtop + ztdvrows    /* If at 0 then down down 1 screen of data */
  if dtop >= totrows then   /* If top >= total rows */
  dtop = totrows - ztdvrows +1  /* then reset the top row */
end
```

If the add request is real, then this next section calls the add_rows subroutine:

```
if ztdadd = 'YES' then do forever   /* Request to add more rows */
  c = 0                             /* Set added counter */
  if zscrolla = 'MAX' then ztdamtl = stem.0  /* test for max */
  x = add_rows()                    /* call add_rows subroutine */
```

The counter (c) is set to zero and we next check the zscrolla value. If it is MAX then the ztdamtl is set to the stem.0 value, which is the total number of stem records.

The add_rows subroutine is then called with a return code being placed into the variable x.

```
if c > 0 then do          /* records added */
  if ZTDSCRP > 0 then     /* update top row to display */
  dtop = ZTDSCRP
  else do
    /* ---------------------------------- *
    | Test to setup the top row to display |
    * ---------------------------------- */
    'tbquery' tbl 'rownum(totrows) position(pos)'
    if datatype(zscrolla) /= 'NUM'
    then if zscrolla /= 'MAX'
    then zscrolla = 0
    if zscrolla = 'MAX'
    then dtop = totrows - ztdvrows +1  /* adjust to see the bottom row */
    else dtop = dtop + zscrolla
  end
  leave
end
```

This code, immediately after the add_rows subroutine:

1. Checks that records were actually added (when c is greater than zero).
2. Then the TBDISPL determined new current row pointer is tested and if it is greater than zero it is used, otherwise there is more work to do. There will be times that TBDISPL is unable to determine the ZTDSCRP value and returns a zero
3. If ZTDSCRP is zero, then the following routine is used to set the top display row
    a. If zscrolla is not numeric and if it is not MAX, then it is set to zero
    b. If zscrolla is MAX, then the display top row (dtop variable) is set to the total number of rows less the number of visible display rows (ztdvrows) plus 1. The plus 1 is required from experimentation or the very last row will never be displayed.

This feature of TBDISPL is very powerful and is a great improvement for very large tables. To see the difference, modify the RXDYNTBL to add all 10,000 records initially and then compare it to how quickly the table display occurs with the initial add of 100 records. On a small, knee capped LPAR, the 100 records time was sub second with 12 service units while 10,000 records was .12 seconds and 16,794 service units. Imagine if the number of records was in excess of 300K.

More about table display

Preserve line commands for multiple selections

Normally you would use the combination of the ZTDSELS variable and additional TBDISPL commands to retrieve successive rows of a table. But the TBDISPL unfortunately ends if another display is done. This could happen if the selection action results in i.e. ISPF EDIT of a dataset.

The solution is to save the line selection commands and the row id for later processing. The following sample shows the entire process, including scrolling.

```
/* main table handler */
Do forever
   if datatype(pnltop)<>'NUM' | pnltop<1 then pnltop=1/* set top pos   */
   "tbtop" tblname                                  /* re-  */
   "tbskip" tblname "number("pnltop") NOREAD"    /*   position       */
   parse value '' with zcmd zerrlm             /* init some vars       */
   "tbdispl" tblname "panel("pnlname") rowid(pnlrowid)" /* selection */
   if rc>8 then Call Epilog 'Table display' pnlname 'rc' rc zerrlm
   if rc=8 | Abbrev('CANCEL',zcmd,3)
       then Call Epilog /* END / RETURN / CANCEL*/
   pnltop=ztdtop                                /* save 1st displayed   */
   /* save row selections */
   pnlsel.0=ztdsels                            /* init list            */
   Do n=1 to ztdsels
     if n>1 then "tbdispl" tblname "rowid(pnlrowid)"  /* next selection  */
     pnlsel.n=zsel';'pnlrowid                    /* save sel and rowid   */
   End
   if zcmd<>''   then Call PCmdHandler         /* process primary cmd  */
   if pnlsel.0>0 then Call LCmdHandler
 end

… more code …

/* handle primary commands */
PCmdHandler:
 say 'pcmd:' zcmd
 return 0

/* process row selections */
LCmdHandler:
 Do pnlseln=1 to pnlsel.0
   parse var pnlsel.pnlseln lc';'rowid
   "tbskip" tblname "row("rowid")"              /* retrieve row         */
   say 'Process row' rowid 'zsel' lc', seq='seq' ,dsn='dsn
 End
 return 0
```

Table filter using existing variable name

The following section show one method for filtering a table non-destructively. The noshow indicator becomes part of one of the existing variables, so the process does not destroy any rows. You must of course reset the indicator before saving the table, and probably also when opening the table, if it permanent. The filter makes use of the ROWS parameter of the)MODEL panel command, like shown below. The ‘PROWS’ variable is set by the program depending on if a filter is set or not.

```
)MODEL clear(zsel) rows(&prows)
```

REXX program snippet. The filter commands are processed as follows:

Include	include in addition to already included
Exclude	exclude in addition to already excluded
Only	show all with text, hide all without
Hide	hide all with text, show all without
All | Reset	show all

The table contains the following variables: DSN, TYPE, VOL and CAT. The DSN variable is used for the filter indicator. Some checks have been removed to keep the snippet small.

```
/* handle primary commands */
PCmdHandler:
 parse var zcmd cvrb cdata
 if cdata<>'' &,
  ( Abbrev('INCLUDE',cvrb,1) | Abbrev('EXCLUDE',cvrb,1),
  | Abbrev('ONLY'   ,cvrb,1) | Abbrev('HIDE'   ,cvrb,1) ) then call Filter
 if Abbrev('RESET'  ,cvrb,3) | Abbrev('ALL'    ,cvrb,1)   then call Reset
 return 0
/* Include/exclude rows with text - use var 'dsn' */
Filter:
 cnd=left(cvrb,1)
 filtn=0
 "tbtop" tblname
 "tbquery" tblname "rownum(rown)"              /* get number of rows  */
 do rown
   "tbskip" tblname                            /* get next row        */
   odsn=dsn                                    /* save dsname for test*/
   if pos(cnd,'HO')>0 then dsn=strip(dsn,'l','-')    /* reset          */
   if cnd='I' then,
     if pos(cdata,dsn type vol cat)>0 then dsn=strip(dsn,'l','-')
   if cnd='O' then,
     if pos(cdata,dsn type vol cat)=0 then dsn='-'strip(dsn,'l','-')
   if pos(cnd,'EH')>0 then,
     if pos(cdata,dsn type vol cat)>0 then dsn='-'strip(dsn,'l','-')
   if dsn<>odsn then do                        /* indicator changed?  */
     "tbput" tblname                           /* update row          */
     filtn=filtn+1                             /* for message         */
   end
 End
 if filtn=0 then return 0                      /* anything changed ?  */
 parse value '' with seq type vol cat text lcmd   /* init vars       */
 dsn='-*'                                         /* set search      */
 "tbsarg" tblname "namecond(DSN NE)"              /* set indicator   */
 parse value 0 'SCAN' with pnltop prows
 Return setmsg(filtn 'records changed')
/* Reset - show all rows */
Reset:
 "tbtop" tblname
 do forever
   parse value '-*' with dsn seq type vol cat text lcmd
   "tbscan" tblname "arglist(dsn) condlist(EQ)"
   if rc<>0 then leave
   if left(dsn,1)<>'-' then iterate
   dsn=strip(dsn,'l','-')                      /* drop indicator      */
   "tbput" tblname
 end
 parse value 0 'ALL' with pnltop prows         /* show all rows       */
 Return 0
Setmsg:
 parse arg zedlmsg
 address ispexec "setmsg msg(isrz000)"
 return 0
```

XISPTBL : Subroutine for ISPF table handling

Whenever you use a table, some handling is most often the same, like scrolling and searching. XISPTBL is an attempt to standardize table handling. It

Short description

Callable subroutine for standard ISPF table handling. The subroutine has built-in support for filtering, sorting and edit - table-row delete, edit, insert and repeat.

You can also specify primary- and line commands for which an external program is to be called.

It is recommended that the table contains variable ZSEL, otherwise a temporary table Wnnnnnnn is created as a copy. Any changes to the temporary table are reflected into the original table.

For further documentation and a short sample, see the XISPTBL program itself.

ISPF Messages

RXMSG

Every ISPF dialog needs to generate messages to inform the user of different situations. There are several ways to deal with messages, but the one trick makes it easy.

IBM provides a very useful, and generic, message in their ISRZ00 member – ISRZ002. This message is built using variables that anyone can use – so why create your own message when you can use this one.

To use this message, you need to:

1. Set variable ZERRSM to a short message
2. Set variable ZERRLM to a long message
3. Set variable ZERRALRM to YES or NO, controls an audible alarm with the message
4. Set variable ZERRHM to the name of your tutorial panel
5. Use SETMSG MSG(ISRZ002) to generate the message

You can set the required ZERRALRM and ZERRHM variables early in your code and then change ZERRSM and ZERRLM multiple times depending on the circumstances. For this you do not have to define alarm or help value. Keep in mind that the long message is limited to 512 bytes.

Notes:

1. If ZERRSM is null, then the ZERRLM message will be immediately displayed.
2. If ZERRLM is null, then pressing F1 will display the tutorial panel instead of the displaying the long message
3. You can make a small popup for the long message by adding text in 75-character increments thus:

```
/* rexx */
Address ISPExec
zerrsm = 'Error'
zerralrm = "NO"
zerrlm = left('test message',75,'*') ,
left('test message 2',75,'*')
zerrhm   = "ispftutr"
"Setmsg msg(isrz002)"
```

Which generates this long message display:

```
+-------------------------------------------------------------------------
---+
| test
message************************************************************** |
| test message
2********************************************************** |
+-------------------------------------------------------------------------
---+
```

Another option is to use the ISRZ001 message which uses variables ZEDSMSG (short message) and ZEDLMSG (Long Message).

Note – if the short message value is null then the long message will be displayed without the user having to press F1. This is true for any ISPF message and not just the ISRZ001 or ISRZ002 messages.

ISPF Edit Macros

ISPF provides the ability to create your own ISPF Edit commands, called Macros, using different programming languages. As we have done throughout this document, we will use REXX, but that can be translated into Assembler, C, or your choice of language.

Tip: Exit an Edit Macro with a return code of 1 to place the cursor in the command entry field.

Centering Text using an Edit Macro

RXCENTER

This is an example of an ISPF Edit Macro that processes selected records by centering the data on them based upon either the data width or the width provided when the macro is invoked.

```
/* rexx */
 Address ISREdit
 'Macro (dw) NOPROCESS'
 "PROCESS RANGE C"
 "(start) = linenum .zfrange"
 "(stop) = linenum .zlrange"
 if dw = '' then
 '(dw) = data_width'
 do i = start to stop
   '(data) = line' i
   data = center(strip(data),dw)
   'line' i '= (data)'
 end
```

Copy this into your own REXX library as CENTER for use.

In this example:

1. The ISPF Edit addressing environment is defined.
2. The Macro statement identifies this code as an ISPF Edit Macro, with an optional parameter (dw) and instructing ISPF Edit to NOT Process (NOPROCESS) any line commands that may be found.
    a. Note: NOPROCESS MUST be in UPPER CASE.
3. ISPF Edit is instructed to identify the RANGE of records selected, either with a single C for a single record, or CC on one record and CC on another to indicate a range.
    a. Note that the range character(s) – C in this case must be UPPER CASE, the process range may be any case.
    b. With PROCESS RANGE you can specify 1 or 2 characters to identify the range.
4. The range is then returned to the variables start and stop using the linenum service where zfrange (first) and zlrange (last) are the row numbers.
    a. If no records were tagged with a C or CC/CC, then all records will be processed
    b. C## can also be used to select a range
5. If the data width was passed as a parameter then use it, otherwise get the data width that ISPF Edit knows about.
6. Then loop from the start to the stop rows centering the data and updating the record using the LINE statement.

Invoking ISPF Edit with a Macro

`RXEMAC and RXEMACE`

It is very easy to call ISPF Edit from within code and have it open with a macro:

```
/* rexx */
 dd = 'DD'random(99999)
 'alloc f('dd') ds(temp.text) new spa(1,1) tr' ,
   'recfm(f b) lrecl(80) blksize(32720)'
 macopt = 'This is a test of the ISPF Edit Macro Parm'
 Address ISPExec
 'Control Display Save'
 'Edit Dataset(temp.text) Macro(rxemac) Parm(macopt)'
 'Control Display Restore'
 Address TSO
 'free f('dd') delete'
```

The ‘control display save/restore’ is very helpful to save the ISPF display state and restore it around the Edit.

In this example it:

1. Defines a random DDname to prevent conflicts with existing allocations
2. Allocates a data set
3. Defines the macopt variable which will be passed to the macro as a parm
4. Saves the display environment
5. Edits a data set with the data set name specifying the macro to use (rxemac) and the name of a REXX variable that contains the parm (macopt) to be passed to the Edit macro
6. Restores the display environment
7. Frees and Deletes the data set

Within the Edit Macro the parm is acquired from the macro statement:

```
/* rexx */
 Address ISREdit
 'macro (macopt)'
 'caps off'
 'number off'
 'recovery on'
 "line_after .zfirst = 'This is line number 1'"
 "line_after .zfirst = 'This is line number 2'"
 'line_before .zfirst = msgline (macopt)'
```

What this Edit Macro does is:

1. Define the ISPF Edit environment (Address ISREdit)
2. Indicate this is a macro and that a parm may be passed (macopt)
3. Turns Caps off, Numbers off, and turns Recovery on
4. Inserts 2 lines of text into the empty data set
5. Inserts a message with the passed parm text.

Define the Initial Macro Using an ISPF Variable (1.2)

There are times where there is a need/requirement to have all Edits during the application start with an Edit Macro, but the application does not invoke Edit directly (e.g. Edit is invoked via a LMDDISP or MEMLIST). In this case define the Initial Macro by setting the ISPF Variable ZUSERMAC (must be in the share or profile pool). The macro then must be able to be found in the normal search order (LPA, LINKLIST, STEPLIB, ISPLLIB, SYSEXEC, SYSPROC).

For example:

```
/* REXX */
. . .
zusermac = 'ieditmac'

Address ISPExec
'VPut (zusermac) Shared'
. . .
```

Passing Data to an Edit Macro

There are at least two easy techniques to pass data to an Edit Macro.

The 1st is demonstrated in the above section using the PARM keyword when invoking ISPF Edit. That is effectively the same as entering the data on the ISPF Edit command line when invoking the macro – see Centering Text using an Edit Macro, above.

The 2nd technique is to have the Edit Macro use ISPF VGET services to retrieve the ISPF variables from the ISPF variable pool. This technique allows for more flexibility as more than a single value can be processed by using multiple ISPF variables.

Invoking an Edit Macro on All Members of a PDS

`RXDOALL, RXPOPDO and RXPOPM`

This example demonstrates how to invoke an ISPF Edit Macro on all members of a PDS. The macro can do anything but for this example the macro will change one value to another.

```
/* --------------------  rexx procedure  ---------------------- *
 | Name:      DoAll                                             |
 | ISPF - Developers - Tips and Tricks                          |
 |                                                              |
 | Function:  This rexx exec will process the specified         |
 |            ispf edit macro against every member of the       |
 |            specified partitioned dataset.                    |
 |                                                              |
 |            Only standard system services are used. The       |
 |            LISTD TSO command with the MEMBERS keyword        |
 |            is used to extract the member names.              |
 |                                                              |
 | Syntax:    %DoAll dsname edit-macro                          |
 |                                                              |
 | Sample Edit Macro to change SYS1 to SYS2 (called chsys1t2):  |
 |  Address ISREDIT                                             |
 |  "MACRO"                                                     |
 |  "CHANGE 'DSN=SYS1.' 'DSN=SYS2.' ALL"                        |
 |  "SAVE"                                                      |
 |  "END"                                                       |
 |                                                              |
 | Sample Execution:  %Doall 'sys2.testjcl' chsys1t2            |
 |                                                              |
 * ------------------------------------------------------------ */
 /* -------------------------------- *
 | Get the target data set and macro |
 * -------------------------------- */
 arg dsn exec
 /* ---------------------------------------- *
 | Fix up data set name for use (no quotes) |
 * ---------------------------------------- */
 if left(dsn,1) <> "'" then do
   dsn = sysvar(syspref)"."dsn
 end
 else do
   dsn = substr(dsn,2,length(dsn)-2)
 end
 /* ------------------------- *
 | Setup Outtrap and do ListD |
 * ------------------------- */
 x = outtrap("lm.","*")
 "LISTD" "'"dsn"'" "MEMBERS"
 x = outtrap("off")
 /* ------------------- *
 | Process all members |
 * ------------------- */
 do i = 7 to lm.0
   parse value lm.i with mem extra
   Address ISPEXEC "EDIT DATASET('"dsn"("mem")') MACRO("exec")"
 end
```

Another example of this is the RXPOPDO exec that will process all members of the target PDS and report on all ISPF Popup Panels, those members with `)BODY` and a `WINDOW(` parameter. The report will recommend a 'centered' AddPop location (row and column).

To use this exec execute it passing it the data set name of the PDS with the ISPF Panels. Here is a sample report for the example PDS:

```
ISPF Panel PopUp Centering Suggestions

Panel    ADDPOP Command
-------- ------------------------
$DEVCPYP ADDPOP ROW(7) COLUMN(7)
$DEVPX   ADDPOP ROW(7) COLUMN(6)
PNABC    ADDPOP ROW(7) COLUMN(17)
PNFLDH1  ADDPOP ROW(7) COLUMN(18)
PNFLDH2  ADDPOP ROW(7) COLUMN(18)
PNPOP    ADDPOP ROW(7) COLUMN(18)
PNPREXX  ADDPOP ROW(7) COLUMN(16)
PNPROG1  ADDPOP ROW(7) COLUMN(29)
PNPROG2  ADDPOP ROW(7) COLUMN(16)
PNSCRL   ADDPOP ROW(7) COLUMN(6)
PNVDSN   ADDPOP ROW(7) COLUMN(14)
```

The RXPOPM example is the ISPF Edit Macro which may be used by itself on any ISPF PopUp Panel and will insert a noteline with the recommendation:

```
****** ********************************* Top of Data ******
=NOTE= Suggested ADDPOP Command: ADDPOP Row(8) Column(18)
000001 )ATTR DEFAULT(%+_)
000002   /* ISPF - Developers - Tips and Tricks */
000003   @ type(output) caps(off) just(left)
000004 )BODY WINDOW(45,8)
000005 +
```

Changing ISPF Edit Commands with Macros

`RXEM, RXEME, RXEMS and RXEMTRY`

There are times when you need to change the behavior of an ISPF Edit command. This example demonstrates how to change the way both END and SAVE work by defining an Alias of those commands to an Edit Macro.

The RXEMTRY is a sample REXX exec that will drive ISPF Edit for this demonstration.

```
/* rexx */
Address ISPExec
'vget (ztempf)'
"Edit dataset('"ztempf"') macro(rxem)"
```

RXEM is the primary ISPF Edit Macro that is called with the ISPF Edit command. It sets up the ISPF Edit environment by:

- Issuing a RESET ALL to remove a of the 'annoying' ISPF Edit information messages.
- Defining the ISPF End command to the RXEME Edit Macro.

- Defining the ISPF Save command the REXEMS Edit Macro.
- Turning off Caps and Numbers.
- Inserting 3 message lines to instruct the user how the demonstration works:

```
/*  REXX  */
Address ISREdit
'Macro'          /* First indicate we are an Edit Macro */
'Reset all'      /* Then Reset to turn off all messages */
'Define end  alias rxeme'     /* Now define END as an Alias */
'Define save alias rxems'     /* And Save as an Alias */
'caps off'       /* Now turn off Caps */
'numbers off'   /* and turn Numbers off as well */
text = 'This demonstrates changing the Save and End commands.'
'line_after .zfirst = msgline (text) '
text = 'Enter the word DONE (any case) in a record to allow Save' ,
 'or End to work.'
'line_after .zfirst = msgline (text) '
text = 'Or enter the CANCEL command to exit ISPF Edit.'
'line_after .zfirst = msgline (text) '
```

Both ISPF END and SAVE commands will not invoke the RXEME and RXEMS Rexx exec's which will both check for the word 'DONE', in upper/lower/mixed case, and only then process the real ISPF command. If the word is not found, then the user is informed using ISPF messages.

The RXEME code is below:

```
/*  Rexx  */
Address ISREdit  /* Setup Addressing to ISPF Edit */
 'Macro'          /* Indicate we are a macro */
 "Find 'done' first"           /* Do a FIND for the word 'DONE' */
 if rc = 0 then do/* if found then: */
  'define end reset'          /* Reset the alias definition for END */
  'define save reset'         /* Reset the alias definition for SAVE */
  'end'          /* Do the real Edit END command */
  exit 0         /* And edit the macro */
 end
 zedsmsg = 'Warning'
 zedlmsg = left('To exit ISPF Edit enter the word DONE somewhere',75) ,
  'in the text or enter the CANCEL command.'
 Address ISPExec
 'setmsg msg(isrz001)'
```

And the RXEMS code:

```
/*  Rexx  */
Address ISREdit  /* Setup Addressing to ISPF Edit */
'Macro'          /* Indicate we are a macro */
"Find 'done' first"           /* Do a FIND for the word 'DONE' */
if rc = 0 then do/* if found then: */
 'SAVE'         /* Do the real Edit SAVE command */
 exit 0         /* And edit the macro */
end
zedsmsg = 'Warning'
zedlmsg = left('To Save the data enter the word DONE somewhere',75) ,
 'in the text or enter the CANCEL command to exit.'
Address ISPExec
'setmsg msg(isrz001)'
```

Symbolic Handling

This section will describe how to prevent an Edit Macro from interpreting an ampersand (&) as a variable when executing an Edit command.

The editor scans edit statements within program macros to do variable substitution like the CLIST processor. Only one level of substitution is done. This is the default; use the SCAN assignment statement to prevent it.

The SCAN macro command sets scan mode, which controls the automatic replacement of variables in command lines passed to the editor.

The SCAN assignment statement either sets the value of scan mode (for variable substitution) or retrieves the value of scan mode and places it in a variable.

1) Syntax

```
ADDRESS ISREDIT
 "(varname) = SCAN"
 "SCAN OFF"
 "SCAN ON"
```

2) Return Codes

   0 Normal Completion

   24 Severe Error

3) Example – No. 1

   To set a line whose number is in variable &LNUM to:

   &SYSDATE is a CLIST built-in function

   Set scan mode off and issue the LINE command with &&SYSDATE as the CLIST function name. The CLIST processor strips off the first &, but, because scan mode is off, the editor does not remove the second &:

```
ISREDIT SCAN OFF
ISREDIT LINE &LNUM = "&&SYSDATE is a CLIST built-in function"
ISREDIT SCAN ON
```

   Because the ISPEXEC call interface for REXX EXECs allows you to specify parameters as symbolic variables, a single scan always takes place before the syntax check of a statement. Therefore, the rule of using two ampersands (&) before variable names to avoid substitution of variable names also applies to REXX EXECs.

4) Example – No. 2

To exclude all lines and find/display all occurrences of “SYS1.SDITMOD1” with a volume of &SYSRS2, no matter how many spaces are between them:

```
"ISREDIT MACRO NOPROCESS"
SPACE   = ""
VAR1    = "(SYS1.SDITMOD1)"
VAR2    = "VOL(&&SYSR2)"
LEN1    = LENGTH(VAR1)
LEN2    = LENGTH(VAR2) - 1
LEN3    = 71 - LEN1 - LEN2
ADDRESS ISREDIT
 "SCAN OFF"
 "X ALL"
 DO J = 1 to LEN3
    SPACE = SPACE" "
    TEXT = VAR1||SPACE||VAR2          "F '"TEXT"' ALL"
 END
```

Note that this may not always work per a comment from IBM. The guaranteed way to do this is to use 4 &’s thus:

```
Address ISREDIT
‘Macro’
“C ‘&&&&ABC’ ‘XYZ’ ALL’
```

This does work very nicely.

Library Services

LMDLIST – List Data Sets

RXLMD

LMDLIST will return to the application information on all the data sets under the provided high-level-qualifier. This example demonstrates that in a simple way. Note that this code can also be executed in batch as there are no display panels used.

```
/* ------------------------ rexx --------------------------- *
 | This REXX code demonstrates using the LMDLIST ISPF Service |
 | to list information about data sets under the provided     |
 | high-level-qualifier (hlq).     |
 |        |
 | If no hlq is provided then the users prefix or userid      |
 | will be used.      |
 * ------------------------------------------------------------ */
 arg hlq
/* ------------------------------- *
 | initialize our working variables |
 * ------------------------------- */
 parse value '' with null dsn
 parse value '0 0 0' with total count used
/* --------------------------- *
 | Validate and/or set the HLQ |
 * --------------------------- */
 if hlq = null then do
   if sysvar('syspref') /= null then
   hlq = sysvar('syspref')
   else hlq = sysvar('sysuid')
 end
/* ----------------------- *
 | Define ISPF Environment |
 * ----------------------- */
 Address ISPEXEC
/* ------------- *
 | Setup LMDINIT |
 * ------------- */
 "Lmdinit Listid(LMD) Level("hlq")"
/* ---------------------------------- *
 | Process ALL data sets under the HLQ |
 * ---------------------------------- */
 do forever
   "Lmdlist Listid("lmd") Stats(YES) Dataset(dsn) Option(LIST)"
/* ---------------------------- *
 | If return > 0 then finish up |
 * ---------------------------- */
   if rc > 0 then do
     "Lmdfree listid("lmd")"
     say "All done - count:" count 'used:' used
     exit 4
   end
/* ---------------------------- *
 | Update counter and report out |
 * ---------------------------- */
   count = count + 1
   say count":" "Dataset:" dsn "Vol:" zdlvol "Used:" zdlsize ,
     "Device: *"zdldev"*" ,
     "Mig:" zdlmigr c2x(zdldev)
   sysvolume = ""
   if datatype(zdlsize) = 'NUM' then
   used = used + zdlsize
 end
```

Sample Batch JCL to run this example:

```
//JOBNAME JOB (BATCH),REGION=0M,NOTIFY=&SYSUID,CLASS=A
//OUT   OUTPUT DEFAULT=YES,JESDS=ALL,OUTDISP=(HOLD,HOLD)
//* ------------------------------------------------------ *
//* Demonstration of running an ISPF application in Batch. *
//* ------------------------------------------------------ *
//TSO      EXEC  PGM=IKJEFT01
//SYSEXEC  DD  DISP=SHR,DSN=hlq.DEVTIPS.PDS   <=== Change
//SYSTSPRT DD  SYSOUT=*
//SYSTSIN  DD  *
 ISPF CMD(%TESTLMD)
//ISPPROF  DD  UNIT=VIO,
//SPACE=(TRK,(1,1,5)),
//DCB=(RECFM=FB,LRECL=80,BLKSIZE=6160)
//ISPMLIB DD   DISP=SHR,DSN=ISP.SISPMENU
//ISPPLIB DD   DISP=SHR,DSN=ISP.SISPPENU
//ISPSLIB DD   DISP=SHR,DSN=ISP.SISPSENU
//ISPTLIB DD   UNIT=VIO,DISP=(NEW,PASS),SPACE=(TRK,(1,1,5)),
//DCB=(RECFM=FB,LRECL=80,BLKSIZE=27920)
//        DD   DISP=SHR,DSN=ISP.SISPTENU
//ISPLOG  DD   DUMMY,DCB=(LRECL=120,BLKSIZE=2400,DSORG=PS,RECFM=FB)
```

Quick (Lightening Fast) List of Data Sets (1.2)

If all that is required is a list of data sets then there are two options beside the LMDLIST, and they don't require ISPF services to use them. The LISTC routine took 10 seconds on a sample hlq where LMDLIST took 15 seconds. The Catalog Search Interface (CSI) routine took less than a second.

LISTC

The first is using LISTC in combination with OUTTRAP:

```
/* REXX */
Arg HLQ

Call outtrap 'list.'
'listc level('hlq')'
Call outtrap 'off'

Do I = 1 to list.0
   Say list.i
   end
```

It isn't elegant but it does work.

Catalog Search Interface

The other option, which is even faster is to grab the IBM provided sample code found in SYS1.SAMPLIB(IGGCSIRX) and tailor it for your needs.

Miscellaneous

Browsing Data in a REXX Stem

`RXSTEM RXSTEME`

Many times, an ISPF dialog will collect information in a REXX STEM variable and need to present that to the user. This approach is very simple using a temporary data set and ISPF Library Services.

```
/* -------------------- REXX ----------------------- *
 | Sample routine to browse data from a stem variable. |
 * ---------------------------------------------------- */
 do i = 1 to 200  /* Define our test data */
   stem.i = 'Test data record number' right(i+1000,3)
 end
 stem.0 = i      /* update the stem.0 with record count */
/* ------------------------------ *
 | Randomly define a DDName to use |
 * ------------------------------ */
 dd = 'dd'random(9999)
/* ---------------------------------------- *
 | Allocate a temporary data set for our data |
 * ---------------------------------------- */
 'Alloc f('dd') new spa(5,5) tr' ,
   'recfm(v b) lrecl(80) blksize(0)'
/* ----------------------- *
 | Write out the stem data |
 * ----------------------- */
 'Execio * diskw' dd '(finis stem.'
/* ------------------------------------------------ *
 | Access the Temporary Data Set using ISPF          |
 | Library Services.       |
 | Then using ISPF Browse service to browse the data. |
 | And use Library Services to Free the Data Set.    |
 * ------------------------------------------------ */
 Address ISPExec
 'lminit dataid(ddb) ddname('dd')'
 'browse dataid('ddb')'
 'lmfree dataid('ddb')'
/* ---------------------------- *
 | Last Free the z/OS Allocation |
 * ---------------------------- */
 Address TSO
 'Free f('dd')'
```

A better and easier approach is to use the STEMEDIT command which can be found on the CBTTape in File 895. Here is the code for that:

```
/* -------------------- REXX ----------------------- *
 | Sample routine to browse data from a stem variable. |
 * ---------------------------------------------------- */
 do i = 1 to 200  /* Define our test data */
   stem.i = 'Test data record number' right(i+1000,3)
 end
 stem.0 = i      /* update the stem.0 with record count */
 call stemedit 'Browse',stem.,,,'Stem stem. Browsing'
```

Tip: Using this technique will display all the active REXX variables:

```
Call stemedit 'edit,'*'
```

Sample ISPF Notepad Application

```
RXNOTEPD
```

ISPF can be easily extended by installing your own tools with an ISPF Command Table update (see Add to the ISPF Commands Table) above. The REXX Notepad (RXNOTEPD) sample code demonstrates the beauty of this capability. Once the command table is updated then entering NOTE on any ISPF command line will open the notes partitioned dataset.

The sample code includes three options for the editor to demonstrate how easy it is to provide added functionality for the ISPF users.

Other Tricks

Edit Macro or TSO Command – same REXX Code

`RXTM and RXTSOMAC`

This sample code, when placed in a REXX exec will let the code work as either an Edit macro, or as a TSO command (inspired by an IBM-Main posting by Bob Bridges).

```
/* ---------------------------------------------------- *
 | Test if called as an Edit Macro                      |
 | RC > 0 means TSO else Edit Macro                     |
 |                                                      |
 | ISPF - Developers - Tips and Tricks                  |
 * ---------------------------------------------------- */
 Address ISREdit
 'macro (options)'
 if rc > 0 then do
   Address TSO
   tsomac = 1
   parse arg options
   say 'Lines of code via sourceline:' sourceline()
 end
 else tsomac = 0
 If tsomac = 1
 then say 'Running as a TSO Command'
 else do
   say 'Running as an ISPF Edit Macro'
   '(dataset) = dataset'
   '(member) = member'
   '(last) = linenum .zlast'
   say 'Dataset:' dataset 'Member:' member 'Lines:' last+0
   'end'
 end
 Exit
```

ISPF in Batch

JCBAT1 and JCBAT2

For ISPF in batch the batch TMP (terminal monitor program) is used. It is recommended to use IKJEFT1B which has the advantage over IKJEFT01 of returning the return code of the last executed command.

Below is an example of running the IBM TASID program in batch. This is an ISPF application written by Doug Nadel.

The Snapshot utility (Option 8) allows you to create a sequential file which contains data obtained by most of the TASID options. The name of the data set can be changed by using the Settings pulldown on the TASID main menu.

The PRINTDS command is used in this example to obtain a hardcopy of data set userid.TASID.SNAPSHOT through foreground copying to SYSOUT.

```
//JOBNAME JOB (BATCH),REGION=0M,NOTIFY=&SYSUID,CLASS=A
//OUT   OUTPUT DEFAULT=YES,JESDS=ALL,OUTDISP=(HOLD,HOLD)
//TSO      EXEC  PGM=IKJEFT1B
//STEPLIB  DD  DISP=SHR,DSN=TASID.LOAD   <=== Change
//SYSTSPRT DD  SYSOUT=*
//SYSTSIN  DD  *
 PROFILE PREFIX(userid)      <=== Change
 ISPSTART PGM(TASID) PARM(8)
 PRINTDS DSNAME(TASID.SNAPSHOT) CCHAR NOTITLE SYSOUT(R) HOLD
//ISPPROF  DD  UNIT=VIO,SPACE=(TRK,(1,1,5)),
//DCB=(RECFM=FB,LRECL=80,BLKSIZE=6160)
//ISPMLIB DD   DISP=SHR,DSN=ISP.SISPMENU
//ISPPLIB DD   DISP=SHR,DSN=ISP.SISPPENU
//ISPSLIB DD   DISP=SHR,DSN=ISP.SISPSENU
//ISPTLIB DD   UNIT=VIO,DISP=(NEW,PASS),SPACE=(TRK,(1,1,5)),
//DCB=(RECFM=FB,LRECL=80,BLKSIZE=27920)
//        DD   DISP=SHR,DSN=ISP.SISPTENU
//ISPLOG  DD   DUMMY,DCB=(LRECL=120,BLKSIZE=2400,DSORG=PS,RECFM=FB)
```

This is another example:

```
//JOBNAME JOB (BATCH),REGION=0M,NOTIFY=&SYSUID,CLASS=A
//OUT   OUTPUT DEFAULT=YES,JESDS=ALL,OUTDISP=(HOLD,HOLD)
//TSO      EXEC  PGM=IKJEFT1B
//SYSEXEC  DD  DISP=SHR,DSN=hlq.DEVTIPS.PDS   <=== Change
//SYSTSPRT DD  SYSOUT=*
//SYSTSIN  DD  *
 ISPF CMD(%TESTLMD)
//ISPMLIB DD   DISP=SHR,DSN=ISP.SISPMENU
//ISPTLIB DD  DISP=SHR,DSN=ISP.SISPTENU
//ISPPROF  DD  UNIT=VIO,SPACE=(TRK,(1,1,5)),
//DCB=(RECFM=FB,LRECL=80,BLKSIZE=6160)
//ISPPLIB DD   UNIT=VIO,DISP=(NEW,PASS),SPACE=(TRK,(1,1,5)),
//DCB=(RECFM=FB,LRECL=80,BLKSIZE=27920)
//ISPSLIB DD   UNIT=VIO,DISP=(NEW,PASS),SPACE=(TRK,(1,1,5)),
//DCB=(RECFM=FB,LRECL=80,BLKSIZE=27920)
//ISPLOG  DD   DUMMY,DCB=(LRECL=120,BLKSIZE=2400,DSORG=PS,RECFM=FB)
```

The only ‘real’ data sets required in batch is ISPMLIB and ISPTLIB, the other ISPF data sets can be temporary.

Randomize DDname and Table Names

RXRAND

There are several techniques for randomizing both DDnames and ISPF Table names with a goal to make sure that the names are unique.

Based on the REXX Exec Name

This technique uses the REXX PARSE SOURCE command to get the name of the REXX Exec that is being executed and then use that for the DDname or a temporary Table name. This information can also be used for work data set names that are allocated using a data set name. One advantage to this approach is that it is easier when debugging to determine the REXX code 'causing' the 'problem'.

To get the active REXX Exec name use this sample code:

```
/* rexx */
parse source . . exec ddname .
say 'exec:' exec
say 'ddname:' ddname
```

And what you get back is this when the code is named TSOURCE:

```
exec: TSOURCE
ddname: SYSEXEC
```

Using the REXX Random Function

This technique is more flexible while reducing the debugging information;

```
/* rexx */
dd = 'DDN'random(99999)
say 'ddname:' dd
```

With this approach the variable DD is assigned a value that starts with DDN followed by 5 random numbers between 1 and 99999. The character string DDN can easily be changed to something meaningful that will help in debugging.

Another advantage is that if this is used in an ISPF dialog that is opened in multiple ISPF screens, then you can guarantee that there will be no DD name or Table name conflicts.

Another Random String

Another approach is to use code similar to this:

```
/* ---------------------------------------------- *
 | Using the day of month and second since midnight |
 * ---------------------------------------------- */
  ddn = 'D'left(date('e'),2)''right(time('l'),5)
  say 'DDname:' ddn
```

With this approach we use the 2 digit day of the month and the 5 digits for the seconds since midnight, all preceded by a character to yield an 8 character DDname.

Using /dev/random

For a more random value use the OMVS /dev/urandom device thus:

```
x = bpxwunix('head -c 4 /dev/urandom',,so.,se.)
 random = c2x(so.1)
```

This generates a 4-character hex random string that is then converted (c2x) to human readable 8 characters. Change 4 to any number if you need a longer or shorter string.

Getting the DDname from the System

In this example, using BPXWDYN, the ddname is determined by the system and returned into a variable (dd1). Then the data in the data set is read using execio, and the allocation then freed using BPXWDYN.

This code requires that the user have OMVS access.

```
/* REXX */
ds='your.dataset.name'
cc=bpxwdyn('alloc da('ds') shr rtddn(dd1)') /* return ddname in dd1 */
if cc<>0 then say alloc' ds 'rc' cc
else do
   "execio * diskr" dd1 "(stem data. finis)"
   cc=bpxwdyn('free dd('dd1')')
 end
```

Stem Sort

Consider the following directory list application. To facilitate sorting by various fields, a directory stem named `dir.` is used.

```
PGLITE®                    TRIDJK.MBRGEN2                 Row 1       of 53
Command ===>                                                    Scroll ===> CSR

  Name       GenNum VV MM  Created     Changed          Size      Init ID
. $$$$$$$$        0 01.01 2019/04/20 2019/04/20 09:23      2         1 TRIDJK
. $$$$$$$$        1 01.06 2019/04/20 2019/05/03 12:27      5         1 TRIDJK
```

The REXX code snippet below shows how to set up a stem sort using BPXWUNIX SORT. Sort keys are defined for each field in the stem to be sorted. For the `CHANGED` and `SIZE` fields, a primary and secondary key are used. For the `CREATED` and `ID` fields, only a primary key is used.

Note that the input and output to the sort use the same stem.

```
   created = '-dr -k5,5'
   changed = '-dr -k6,6 -k7,7'
   size    = '-nr -k9,9 -k8,8'
   id      = '-dr -k11,11'

   if abbrev("CREATED",word(column,1),2) = 1 then
     call do_sort created
   if abbrev("CHANGED",word(column,1),2) = 1 then
     call do_sort changed
   if abbrev("SIZE",word(column,1),1) = 1 then
     call do_sort size
   if abbrev("ID",word(column,1),1) = 1 then
     call do_sort id

do_sort:
parse arg key
env.0=1
env.1='LC_COLLATE=S370'
stderr.0 = 0
call BPXWUNIX "/bin/sort "key, ,
              "dir.", "dir.", "stderr." "env."
 return
```

Full Stem Sort example

RXSTEMS

This example is self-contained to sort a stem.

```
/* ---------------------- REXX -------------------------- *
| From John McKown to sort a stem using unix sort in OMVS |
* ---------------------------------------------------- */
XX=SYSCALLS('ON')
If xx > 3 Then Do
  Say "SYSCALLS('ON') Failed. RC="xx
  exit
End
do i = 1 to 12
  a.i = random(999)
end
a.0=12
stdout.0=0 ; stderr.0=0
call bpxwunix "/bin/sort -n",a.,a.,stderr.
say 'a.0='a.0
do  i=1 to a.0
  say "a."i"="a.i
end
say 'stderr.0='stderr.0
do  i=1 to stderr.0
  say "stderr."i"="stderr.i
end
```

First is a verification that the user has access to OMVS. Next a stem with 12 elements is created using random 3-digit numbers.

More on Stem Sorting using BPXWUNIX

'Borrowed from an IBM-Main posting by Albert Ferguson:

```
Under zOS REXX does have a STEM Sort (via BPXWUNIX). If you have an OMVS
Segment on your USERID (and as of zOS 2.1+ you should), try this:

/*
     Do a Binary Sort (use -tn for Text)
     Using 2nd Word in each StemIn. Record as primary sort key
     Using 1st Word in each StemIn. Record as secondary sort key
     More doc on Unix Sort
at http://publibz.boulder.ibm.com/epubs/pdf/bpx1cd10.pdf
*/
x = BPXWUNIX("/bin/sort -bn -k2,2 -k1,1","StemIn.","StemOut.")

Also available via the USS side of zOS is Regex support:

/*
    Get the JESMSGLG output from a JOB, e.g. via SDSF or (E)JES and put in
the STEM JesMsgLg.
    Use the USS grep in, Extended mode, to search for all STEPS with an
RC=12, 16, or 20
    Return only those steps to the BadSteps. STEM variable ...

    Add a step using this at the very END of a JOB to determine if
something did not go as expected, and then
    possibly correct it!
*/
x = BPXWUNIX("/bin/grep -E -e'¬.{30}(.{10}  (12|16|20)'",
"JesMsgLg.","BadSteps.")
```

Converting the User Provided Data Set Name to a Full Data Set Name

Many times, the user, when prompted to enter a data set name, will enter the data set name without fully qualifying it, and without the default TSO Prefix. That isn't a problem is all references to the data set name are using standard system services that will resolve the data set name properly. If the data set name must be fully qualified within the application the easy way to do that is to use the TSO LISTDSI service, which will return the fully qualified data set name (without quotes) in the SYSDSNAME variable.

```
/* rexx */
arg dsname
x = listdsi(dsname)
Say 'Input DSN:' dsname
Say 'Full DSN:' sysdsname
```

Note that this works whether the user is running with, or without, a TSO PREFIX.

Default ISPF Terminal Type

It has been recommended that the ISPF Terminal Type 6 (3278T) model be selected from ISPF Primary Menu Option 0.

Sharing REXX Variables, including stems, with other REXX exec's

RXSHRVAR

One of the major issues with REXX on z/OS is the lack of a global pool to share REXX variables. This forces the developer to write very large REXX exec's instead of writing more modular individual REXX execs.

There are solutions on the CBT Tape – look for STEMPUSH and STEMPULL and REXXGBLV. These are assembled and linked solutions that are worth checking out if you are allowed to use open-source tools.

If you just want to roll-your-own (ryo), and if your application runs under ISPF, then you can use the ISPF Shared Variable Pool to pass REXX data from one REXX exec to another.

The RXSHRVAR sample demonstrates how to convert a REXX stem (single level only in the example but that can obviously be expanded upon) into ISPF variables and then convert the ISPF variables back to REXX stem variables.

There are two parts of the example. The 1st part converts the REXX variables to ISPF variables and the 2nd part converts the ISPF variables to REXX variables. This technique utilizes the REXX INTERPRET instruction (which can be confusing at first look so use your favorite browser search engine to learn more about it).

RXSHRVAR – Converting REXX variables to ISPF variables

```
/* ---------------------------------------------------- *
| Create unique ISPF variables from each stem and      |
| VPUT the ISPF variable into the shared variable pool. |
|                                                      |
| The samplec variable contains the count of the number |
| of variables.                                        |
* ---------------------------------------------------- */
samplec = sample.0       /* setup count variable */
'vput (samplec) shared'  /* vput the variable    */

/* ---------------------------------------------------- *
```

```
| Process each individual stem variable using the REXX      |
| INTERPRET command.                                        |
|                                                           |
| This converts the stem.# value to a valid ISPF variable   |
| which can be anything you want to call it. It just has to |
| be a valid ISPF variable name (and 8 characters or less). |
* --------------------------------------------------------- */
do i = 1 to sample.0
  interpret 'sample'i '=  sample.'i
  'vput (sample'i') shared'
end
```

RXSHRVAR – Converting ISPF variables back to REXX variables

```
/* --------------------------------------------------------- *
| Get the samplec (count) ISPF variable into a REXX variable |
| and then VERASE it to remove it from the ISPF shared       |
| variable pool.                                             |
* --------------------------------------------------------- */
'vget (samplec) shared'    /* get the count of the stem variables */
'verase (samplec) shared'  /* clean up ispf var pool    */
sample.0 = samplec         /* set sample.0 to the count */

/* --------------------------------------------------------- *
| Retrieve from the ISPF Shared Variable Pool each unique    |
| stem variable and place it back into the stem using        |
| the REXX INTERPRET command.                                |
* --------------------------------------------------------- */
do i = 1 to sample.0
  'vget (sample'i') shared'   /* get the individual var */
  'verase (sample'i') shared' /* clean up ispf variable */
  interpret 'sample.'i '= sample'i      /* set the stem           */
end
```

These examples use drop and verase to clean up the REXX and ISPF variable pools to demonstrate how to use these tools.

The comments in the code were created using the CMT ISPF Edit macro (aka ISPF Edit command), which is included with this package.

Convert a Number to Human Readable

`RXNUMC and RXNUMCE`

This piece of code was sent to the author by Doug Nadel (IBM ISPF developer and author of TASID, among other things) many years ago and it is a very useful piece of code:

```
  /* ----------------- REXX ------------------------- *
  | Take a number and display it with comma's in it. |
  |                                                  |
  | e.g.  1000 becomes 1,000                         |
  |                                                  |
  | usage:   comma_num = addcomma(number)            |
  |                                                  |
  | supports numbers up to 34 digits long            |
  |                                                  |
  | Copied from Doug Nadel                           |
  * ------------------------------------------------ */
AddComma:
  arg bignum
  cbignum =
strip(translate('0,123,456,789,abc,def,ghi,jkl,mno,pqr,stu,vwx', ,
    right(bignum,34,','), ,
    '0123456789abcdefghijklmnopqrstuvwx'),'L',',')
```

```
  return cbignum
```

This code will convert a plain numeric value, up to a maximum of 34 digits) such as 1981456 to 1,981,456.

Useful Tools

IBM ISPF includes several useful tools, among these are:

ISRDDN

Use this tool before and after testing your application to verify that all allocations have been released. To use it issue TSO ISRDDN from any ISPF Command line.

ISPLIBD

This ISPF command will display all the current ISPF LIBDEF allocations. To use it issue ISPLIBD on any ISPF Command line. Note that this will only show the allocations for the current ISPF Screen and not on other split screens.

ALTLIB Display

The ALTLIB command, with the Display parameter, will display the current ALTLIB allocations. It only displays the DDnames and not the data set names, which is a shame.

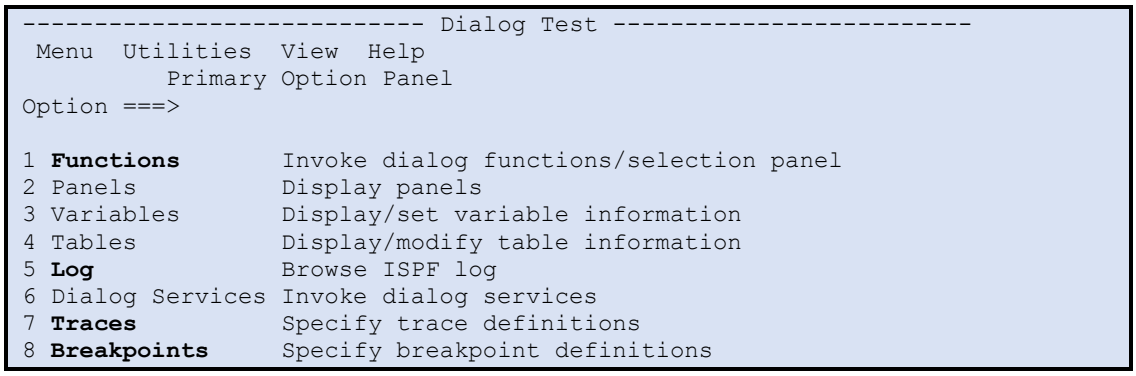
Debugging Hints/Tips

Displaying the ISPF Panel Name

Many times, it is useful to know which ISPF Panel is being displayed. To display the active Panel name, issue the **PANELID ON** command and then the reverse is **PANELID OFF**.

ISPF Dialog Test

ISPF dialog test (Primary Option 7) is a dialog that can be used to test your ISPF applications.



Whilst there are many options in Dialog Test, *Traces*, *Breakpoints* and *Functions* are the options most commonly used.

A typical debugging session begins by "activating" traces (functions and variables) and breakpoints. This is done by specifying YES in the "Active" column for these definitions.

After setting your definitions, start your dialog using *Functions* option of Dialog Test. When the dialog completes, you can browse the ISPF log (Option 5) to see the results. If your Log is allocated to SYSOUT, then you can use SDSF to browse the ISPLOG after accessing the Job Data Set Display panel.

REXX Trace

1. TRACE controls the tracing action, how much information is displayed, during processing of a REXX program. TRACE is mainly used for diagnostic purposes.

a. TRACE I (Intermediates) traces all clauses before execution and traces intermediate results during evaluation of expressions.
b. TRACE ?i will pause after each display, allowing the user to interactively issue any REXX command to assist with debugging.
c. TRACE OFF, or just TRACE, will turn off tracing.

2. Tracing can also be enabled from the ISPF Command line using the TSO commands:
   a. TSO EXECUTIL TS to initiate tracing using the ?i option
   b. TSO EXECUTIL TE to terminate tracing
3. Tracing can also be performed in both Panel Rexx and Skeleton Rexx.

Miscellaneous Tips

1. Use ISPVCALL before and after to trace general stuff
2. Use ISPFTTRC to trace and capture skeletons before and after. Great tool. Requires DD for ISPFTTRC to be allocated.
3. Example:

```
if debug
/* show all work to ISPFTTRC DD */
then ADDRESS TSO "ISPFTTRC READ(DETAIL) RECORDS(*) TBV(DETAIL)"
"FTINCL S@CPYHCD"
lrc = rc
/* RC=8 usually means member not found */
/* RC=20 usually means variable error  */
```

4. See Appendix C of ISPF Dialog Developers guide for notes on Panel and File Tailoring Traces
5. The use of)REXX section in skeletons to display variables. The calling REXX exec issues TSO “ISPFTTRC xxxxxx ”.
6. if you get RC=20 on FTINCL, display zerrlm. This assumes “ISPEXEC CONTROL ERRORS” is set.

```
ADDRESS ISPEXEC "FTINCL "myskel
ftincl_RC = rc
if ftincl_RC <>  0
then do
say ZERRSM   ZERRLM  /* optional */
   call   error_rtn "FTINCL" ftincl_rc myskel ,
   "ZERRSM("zerrsm") ZERRLM("zerrlm")"
 end
```

7. Skeleton notes
   a. Review operators, built-in functions, and control statements

```
)SEL &EML= Y
is not valid .  this is valid
)SEL &EML = Y
```

   b. Be wary of special characters like “<” “>”

```
//* look for errors > 0    - will fail
//* look for errors >>0    - will work
```

   c. May need to utilize “SCAN OFF” and “SCAN ON”
   d. Temporary datasets. Will need review for “&”

```
//  DD DSN=&&&&TEMP,    will resolve to
//  DD DSN=&&TEMP,
```

8. Watch that tailored line does not exceed the allowed record length.

Debugging ISPF Edit Macros

(contributed by Robert Prins)

Many of the edit macros presented earlier in this document are simple and if you would like to write similar ones in REXX consisting of just a few lines of code, it's relatively easy to debug them by inserting a "trace ?r" statement, and executing the code step-by-step, and typing "exit" if you see that something's wrong. Do the development in "View" and you don't even have to worry about accidentally saving the mess you made.

However, trying to use "trace ?r" to debug an edit macro that's a a bit longer, contains (long) loops, might access other macros/execs, can quickly become an endless thankless chore, as you will have to exit, look at the mess you created, swap screens, correct the code, and start all over again. Sure, you could code some direct REXX statement(s) in interactive mode to get you past the first error, make a (mental) note of what needs to be changed in the code, and run to the next problem.

Wouldn't it be nice if you could just stop the macro running in its tracks, and look at the data in the state it’s in to find out if it looks like what you expected it to look, or, for example when you're manipulating data between macro set labels, that the labels are actually on the correct lines, or, if you've realized that you made a mistake, swap to the other screen, and make the changes straight when you find the problem?

The question is rhetorical...

Well, Doug Nadel (who else?) must have thought the same in a grey past, and that's why he wrote a little gem called "ISREMSPY", a macro spy. You can embed it in your macro during development, and when it's called, just by an "ISREMSPY" statement, it will stop the macro, and present a simulated screen (sadly always using only 72 scrollable columns) that shows the last executed macro command, and the state of the data. You cannot only look at it. Press "END" and the macro will continue. Obviously, as soon as "ISREMSPY" interrupts the macro, you're back into ISPF, can swap screens, make the required corrections to your code, and continue running the macro (the old one!)

Tip: When you're developing a macro, add something like:

```
"isredit (SES) = SESSION"
if left(SES, 1) = 'E' then
   do
     zedsmsg = ''
     zedlmsg = 'Macro under development, can only be used in ''View'' mode'
     "ispexec setmsg msg(ISRZ001)"
     exit 1
   end
```

near the beginning, and you don't have to worry about accidentally messing up your data!

Appendix

Useful Tools

These tools, along with the sample code for the various chapters, are included in a TSO Transmit (XMIT) format file that is shared with this document.

CMT

```
CMT
```

CMT is an ISPF edit macro to simplify entry of comments. CMT enters all comments in the syntax of the object being edited and formats them via sent/defaulted user parameters. Defaults for execution are set/inquired via the CMT syntax #2, shown below, and some may be overridden via the main CMT syntax #1.

All comments require a target line be defined in your data.

Syntax 1 to add comment line(s) (all parms optional)

CMT #,B,text

- #=INDENT 1 - 70, C (Center) or R (Right Justify) box
- B =BOXSIZE F S (Fixed 65 chars) or V (Variable size)
- cmt text blank shows popup panel to enter multiple lines.

Syntax 2 to Set CMT defaults (also may be set on popup panel)

CMT SET <parmId> <parm value>

parmid Indent, Boxsize, Charbox or Fixlength
parm value for I: #1-70, C or R B: F,S or V
C: any character F: 1-65

Download from http://lbdsoftware.com/ispftools.html.

LOADISPF/DROPISPF

```
LOADISPF
RXLISPF
```

LOADISPF and DROPISPF are a pair of REXX routines to be copied into your REXX code that supports placing all ISPF elements (panels, messages, skeletons, CLISTs, and REXX) inline in your REXX exec. It will, when called, scan your source code and dynamically allocate temporary datasets and then ALTLIB or LIBDEF to them. Note that DROPISPF is included in the LOADISPF member since they go together and DROPISPF must be called at the close of your REXX code to release the allocations.

The syntax for defining the imbedded elements are:

Key	Description
>START	Identifies the start of the imbedded elements
>END	Identifies the end of the imbedded elements
>CLIST	The start of a CLIST
>EXEC	The start of a REXX exec
>MSG	The start of an ISPF Message member
>PANEL	The start of an ISPF Panel
>SKEL	The start of an ISPF Skeleton

Notes:

1. Any record that begins with a > will terminate the prior element. The key can be in any case (e.g. >START >start >Start).
2. The CLIST, EXEC, MSG, PANEL, and SKEL must be followed by the element name (e.g. >PANEL P1).
3. There can be multiple elements of each type.
4. Each element type results in a temporary PDS allocated with 5 directory blocks with no dataset name (thus they are temporary datasets and may be allocated to VIO).
5. CLIST and EXEC temporary datasets are allocated using RECFM=VB and LRECL=255 with the others being allocated RECFM=FB LRECL=80.
6. Some ISPF Panel special characters may result in a REXX interpreter error. The solution is to place a REXX comment (/*) before the >Start record and a closing comment (*/) after the >End record.

This is very useful, as it allows the application to be completely self-contained. There is no need to distribute, or install, multiple elements when all of the REXX routines needed, all the ISPF Messages, Panels, and Skeletons can be installed with one REXX exec.

See in the provided sample PDS member RXLISPF, which is a full example that includes imbedded elements for a REXX Exec, an ISPF Message, an ISPF Panel, and an ISPF Skeleton.

Be VERY CAREFUL if you use REXXFORM on your REXX exec that includes imbedded elements so that you don't reformat the imbedded elements.

Download both in the LOADISPF package from http://lbdsoftware.com/ispftools.html or use the copy that is included in the example PDS.

REXXFORM

```
#RXFORM
```

REXXFORM is an ISPF edit macro used to format REXX EXECs by indenting DO and SELECT groups and left-justifying lines at a selected column. REXXFORM also checks for unbalanced DO and SELECT statements while providing a consistent format to your REXX code that helps improves readability and maintainability.

REXXFORM was originally developed as a tool to format REXX code under XEDIT under z/VM. It was ported to z/OS as an ISPF Edit Macro with some enhancements to take advantage of the ISPF environment.

This command may only be used from the ISPF Edit command line.

Command Syntax is:

REXXFORM ?	Display ISPF Tutorial Panel
REXXFORM left-margin indent-column (options	Left-margin is the column all lines are justified to (default is 4) Indent-column is the column that DO and Select statements are indented to from the left-margin (default is 3) Options are:

	CI to justify comments to the left-margin CJ to justify comments to column 1

By default, all records will be processed.

To select an individual record or a range use the following syntax:

S Select a single record to format

SS/SS Select a range of records to format

S# Select current record for # -1 (S4 is current plus 3)

Download from http://lbdsoftware.com/ispftools.html

STEMEDIT

This assembler sub-routine can be invoked by a REXX EXEC to display the contents of stem variables using the ISPF BRIF, VIIF or EDIT services. STEMEDIT is a nice complement to the REXX OUTTRAP function, when it is used in the ISPF/PDF environment.

Download File 895 from www.cbttape.org.

TRYIT

```
#TRYIT
```

Overview

TRYIT is an ISPF Edit command that is designed to be used to test an Assembler program, CLIST, REXX Exec, JCL, ISPF Panel or ISPF Skeleton while it is being edited. The way this works is such that the JCL, CLIST, REXX Exec, ISPF Panel, or ISPF Skeleton does *not* must be in a library in the existing SYSPROC, SYSEXEC, ISPPLIB, or ISPSLIB allocations thus allowing the development and testing in other, less critical, data sets.

If a JCL Syntax checking product is available, then TRYIT can be used to invoke it - this is assuming the product can be invoked as an ISPF Edit Macro (e.g. CA-JCLCheck and JCLPrep).

For Assembler programs the active member will be assembled and optionally linkedited into a specified target library. After entering TRYIT the user will be prompted to enter the assembly and linkedit information if the member is determined to be an assembler program.

For CLIST and REXX Exec members the active data set in which the member resides will be allocated using the TSO ALTLIB facility and then the member executed, along with any passed optional parameters.

For ISPF Panels and ISPF Skeletons the active data set in which the member resides will be allocated using the ISPF LIBDEF facility then then the panel Displayed or Selected based upon the parameters provided to TRYIT. If there are any errors in the panel or skeleton an ISPF message will be displayed and the error may then be corrected using ISPF Edit and TRYIT used once again to verify the panel or skeleton - all without the need to split the screen and invoke ISPF Test.

Note there are limitations to the Skeleton testing as variables and imbed tables may not be available.

Because of the use of ALTLIB or LIBDEF the member being tested will be able to find subroutines or other ISPF Panels providing they reside within the data set being edited thus allowing an entire package to be developed, updated, and/or tested, in less critical libraries.

The type of member being edited is dynamically determined with a default of REXX Exec assumed if all the tests fail. The tests include:

- The data set suffix
    - Assembler: ASM ASSEM
    - CLIST: CLIST, SYSPROC, CMDPROC
    - REXX: EXEC, REXX, SYSEXEC
    - Panel: PANEL, PANELS, ISPPLIB
    - Skels: SKEL, SKELS, ISPSLIB
- CLIST: Look for PROC followed by a number on record 1
- REXX: Look for the word REXX in record 1
- Panel: Look for any of these in record 1
-)ATTR)PANEL)CCSID)PROC)BODY)INIT)REINIT ..PREP:
- JCL: First record starts with //

This provides a very easy method for iterative testing and updating of the member until the member works as desired.

Usage

Use TRYIT from any ISPF Edit command line while editing a CLIST, REXX Exec, or ISPF Panel. The syntax is:

TRYIT optional-parms

The optional-parms are:

? to display the ISPF Tutorial

For CLISTs and REXX Execs any parameters that the member being edited would need to have passed to it.

For Assembler there are no supported options currently.

For ISPF Panels one, or more, of the following:

- APPL followed by a 1 to 4-character application id to be used when SEL is specified to select the panel
- POP will cause the panel to be displayed as a popup
    - optionally POP may be followed by a row and column to be used for the popup
- SEL will Select the panel instead of just Displaying it
- TUT will display the panel as an ISPF Tutorial
- default is to just display the panel

Example:

```
Command ===>tryit opt1 opt2
```

Upon completion of the processing of the member a message is displayed in the upper right with a return code or short message with a long message, available by pressing PF1 (Help), with more information. For CLISTs and REXX Execs the return code is whatever the CLIST or REXX Exec return while for ISPF Panels the message is either a zero-return code if there are no problems or the short and long error messages generated by the ISPF Display, Selection, or Tutorial services.

For Assembler programs upon completion of the assembly the assembly listing will be displayed using ISPF browse. If a load library is specified under the link edit options, then after the linkedit the linkedit listing will also be displayed using ISPF Browse. The linkedit will only occur if the assembly completes with a return code less than 8.

Download from http://lbdsoftware.com/ispftools.html

Other Tools of Note (meaning they are worth checking out)

These are useful tools that you should consider installing. All are open source (meaning free) and each will help with your application development in some way:

PDS (the Swiss Army Knife of Utiltiies)

Get this on the CBTTape Site in File 182 – www.cbttape.org – and be sure to check the updates page as a new update may have been released after the most recent CBTTape 'Tape' was cut. This utility, which works in native TSO, includes an extensive ISPF dialog and more sub-commands that you will ever be able to remember. One of the many powerful features is the ability to subset the list of members by a wide variety of different criteria, and then to operate on that member group using any of the dozens of sub-commands.

PDSEGEN

This is an excellent ISPF dialog that fully supports working with PDSE Version 2 Member Generations. While not as extensive in the sub-command area as PDS, this tool supports over almost 2 dozen line commands and over a dozen primary commands. Support is included to copy a PDSE, with all generations, from one PDSE to another. Another capability currently not supported by IBM is the ability to unload a PDSE with all generations and then reload it. Note that PDSEGEN does not support load libraries, but then why would anyone use member generations in a load library since there is no way to access a generation other than the base (generation 0) module.

ISPFCMD

This tool can be found at www.lbdsoftware.com, or in File 312 at www.cbttape.org, and is used to add your own ISPF commands to the sites ISPF command table dynamically. This can be very helpful if you need to add a command that your ISPF sysprog is not willing to install, or to install something that is unique for you.

Here are two useful commands to add using ISPFCMD:

```
"%ispfcmds * ivar       0 select cmd(%rxivar &zparm)" ,
     "\ Display ISPF Variable"
 "%ispfcmds * rvar       0 select cmd(%rxrvar &zparm)" ,
     "\ Display REXX Variable"
```

And the code behind these are:

RXIVAR and RXRVAR

IVAR:

```
/* ------------------------ rexx------------------------- *
 * IVAR - display the contents of the ispf variable that is *
 * passed as an argument.                                   *
 * -------------------------------------------------------- */
arg var
address ispexec 'vget ('var zapplid')'
say 'Requested variable' var 'in ISPF Applid' zapplid 'value is:'
say ' '
interpret 'str ='var
say '>'str'<'
```

REXXVAR:

```
/* ---------------------- REXX ------------------------- *
 | Display any REXX variable, or evaluate and display any |
 | REXX expression.                                       |
 * ------------------------------------------------------ */
  parse arg opt
  interpret say opt
```

Then from any ISPF command line enter IVAR or RVAR thus:

```
   Menu  Utilities  Compilers  Options  Status  Help
 ------------------------------------------------------
                             ISPF Primary Option Menu
 Option ===> ivar zuser
                                                 More:
 0  Settings      Terminal and user parameters
```

And get:

```
Requested variable ZUSER in ISPF Applid ISR value is:

 >LBDYCK<
 ***
```

Or:

```
   Menu  Utilities  Compilers  Options  Status  H
 ------------------------------------------------
                             ISPF Primary Option M
 Option ===> rvar sysvar('sysuid')
                                                 More:
 0  Settings      Terminal and user parameters
```

And get:

```
LBDYCK
 ***
```

STEPLIB (1.2)

On CBTTAPE file 452 is a free, fully functional, Dynamic STEPLIB command. This command has been verified to work with z/OS 2.1, 2.2, 2.3, and 2.4. It can be very helpful when installing ISPF applications in their own libraries where LIBDEF ISPLLIB doesn't always work.

Useful Websites

CBT Tape: www.cbttape.org

> This is a repository of 100's of collections of tools contributed over decades by individuals from all over the world. The vast majority come with source and can be used 'out of the box' or as examples to learn from.

Lionel Dyck's site: www.lbdsoftware.com

> This site has many of the tools referenced in this document and many others. Primarily for use with ISPF and focused on improving the productivity of both end users and systems programmers.

The Rexx Language Association: www.rexxla.org

> The Rexx Language Association (RexxLA) is an independent, non-profit organization dedicated to promoting the use and understanding of the Rexx programming language.

Rexx Tutorial: https://www.tutorialspoint.com/rexx/

> Learn Rexx for absolute beginners.