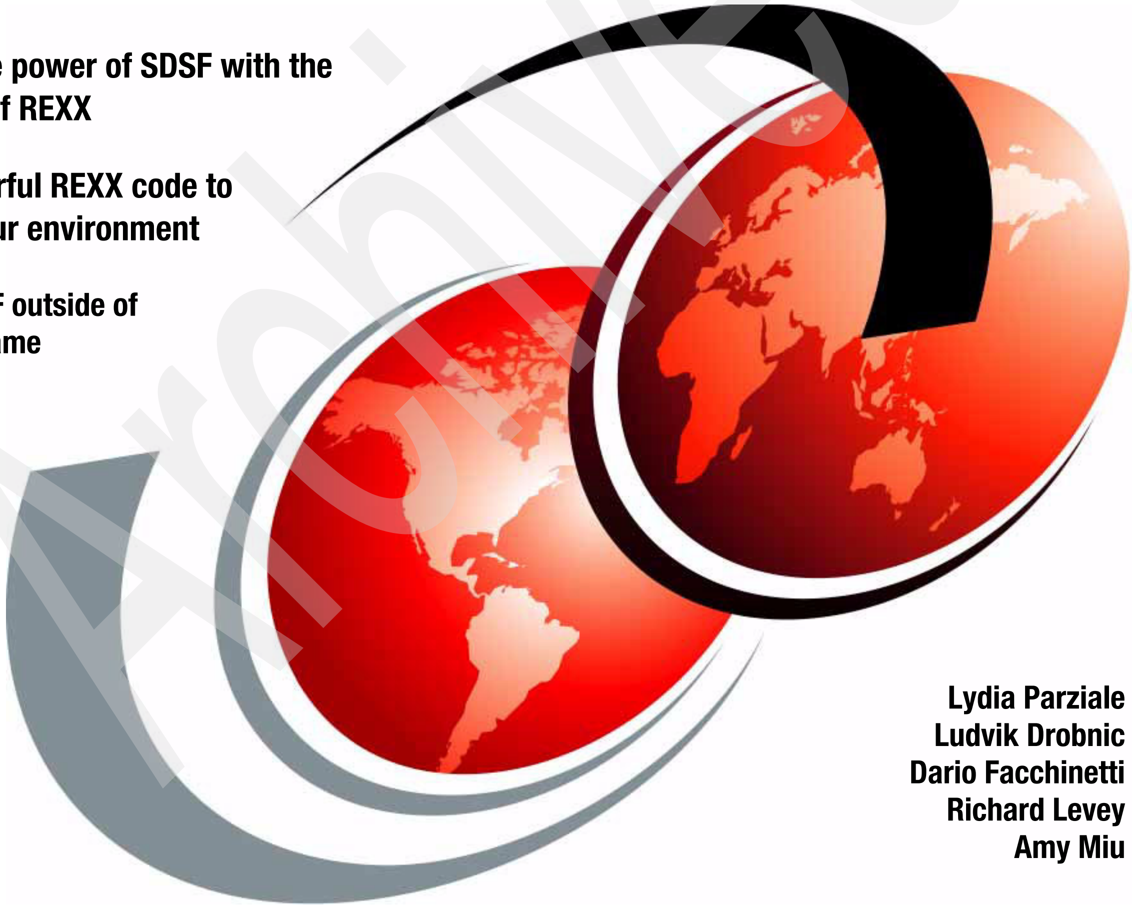
IBM

Implementing REXX Support in SDSF

- Harness the power of SDSF with the versatility of REXX
- Write powerful REXX code to manage your environment
- Access SDSF outside of your mainframe



Lydia Parziale
Ludvik Drobnic
Dario Facchinetti
Richard Levey
Amy Miu

Redbooks

ibm.com/redbooks

International Technical Support Organization

Implementing REXX Support in SDSF

June 2007

SG24-7419-00

Note: Before using this information and the product it supports, read the information in "Notices" on page xi.

First Edition (June 2007)

This edition applies to z/OS Release 1 Version 9 with APAR PK43448

Note: This book is based on a pre-GA version of a product and might not apply when the product becomes generally available. We recommend that you consult the product documentation or follow-on versions of this book for more current information.

Contents

Figures

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:
IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785 U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Trademarks

Preface

The Restructured Extended Executor (REXX™) language is a procedural language that allows you to write programs and algorithms in a clear and structural way. It is an interpreted and compiled language. It is not necessary to compile a REXX command list before executing it.

The IBM® z/OS® System Display and Search Facility (SDSF) provides a number of functions including[1]:

- Viewing the system log and searching for any literal string
- Entering system commands
- Controlling job processing (hold, release, cancel and purge jobs)
- Monitoring jobs while they are being processed
- Displaying job output before deciding to print it
- Controlling the order in which jobs are processed
- Controlling the order in which output is printed
- Controlling printers and initiators

With IBM z/OS V1.9, you can harness the versatility of REXX to interface and interact with the power of SDSF. A new function called *REXX with SDSF* z/OS V1.9 that provides access to SDSF functions through the use of the REXX programming language. This REXX support provides a simple and powerful alternative to using SDSF batch.

> **Note:** We did the work in this book using IBM z/OS V1.9 with APAR PK43448 applied.

This IBM Redbooks® publication describes the new support and provides sample REXX executables that exploit the new function and that perform real-world tasks related to operations, systems programming, system administration, and automation. This book complements the SDSF documentation, which is primarily reference information.

The audience for this book includes operations support, system programmers, automation support, and anyone with a desire to access SDSF using a REXX interface.

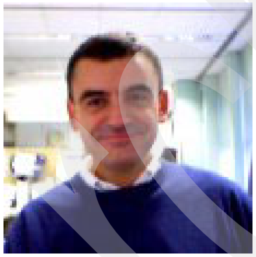
[1] See *Introduction to the New Mainframe: z/OS Basics*, SG24-6366 which is available at:
http://www.redbooks.ibm.com/redbooks/pdfs/sg246366.pdf

The team that wrote this book

This book was produced by a team of specialists from around the world working at the International Technical Support Organization (ITSO), Poughkeepsie Center.

Lydia Parziale is a Project Leader for the ITSO team in Poughkeepsie, New York, with domestic and international experience in technology management, including software development, project leadership, and strategic planning. Her areas of expertise include e-business development and database management technologies. Lydia is a Certified IT Specialist with an MBA in Technology Management and has been employed by IBM for 24 years in various technology areas.

Ludvik Drobnic Martinez is an IT Architect at *la Caixa* in Barcelona, Spain. He has 20 years of experience in application development tools in MVS™ and UNIX® systems. He is currently a member of an application architecture support team.



Dario Facchinetti is a certified Product Service Specialist working for IBM Italy in Integrated Technology Services. He has more that 25 years of experience in mainframe environment for VM, MVS, OS/390®, and z/OS. He use to work in Milan and is part of the EMEA z/OS virtual front end team.

Richard Levey is an Advisory Software Engineer for IBM Global Services in Schaumburg, IL. He works on operating system and program product exits and utility programs for z/OS systems. Rich's areas of expertise include z/OS, REXX, SCLM and assembler language and have now expanded to writing Redbooks. He has a BS in Mathematics and has been employed by IBM for 10 years.

Amy Miu is an Advisory Remote Technical Support Specialist in Canada. She has been with IBM and working with the mainframe platform for 13 years. She is part of the Canadian team that supports z/OS defect and Q&A questions from Canadian customers, as well as z/OS Q&A questions from American customers. She has been supporting SDSF since 1996. She holds a degree in Computer Science from the University of Manitoba in Canada.

Thanks to the following people for their contributions to this project:

Rich Conway, Robert Haimowitz
ITSO, Poughkeepsie Center

Ken Jonas, William Keller, John Kapernick, and Robert E Thompson
IBM Systems &Technology Group

Become a published author

Join us for a two- to six-week residency program! Help write a book dealing with specific products or solutions, while getting hands-on experience with leading-edge technologies. You will have the opportunity to team with IBM technical professionals, Business Partners, and Clients.

Your efforts will help increase product acceptance and customer satisfaction. As a bonus, you will develop a network of contacts in IBM development labs, and increase your productivity and marketability.

Find out more about the residency program, browse the residency index, and apply online at:

ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

We want our books to be as helpful as possible. Send us your comments about this book or other IBM Redbooks publications in one of the following ways:

- Use the online **Contact us** review Redbooks form found at:

  ibm.com/redbooks

- Send your comments in an e-mail to:

  redbooks@us.ibm.com

- Mail your comments to:

  IBM Corporation, International Technical Support Organization
  Dept. HYTD Mail Station P099
  2455 South Road
  Poughkeepsie, NY 12601-5400

Introduction and overview

Until now, to use the power of IBM z/OS System Display and Search Facility (SDSF), you had to invoke SDSF batch. Although SDSF batch provides the full capabilities of SDSF, the interface can be awkward to use when creating utilities. *REXX with SDSF*, new with IBM z/OS SDSF V1.9, provides a more natural programming interface that opens SDSF's strengths to a wider audience.

When using SDSF, you can access the interactive HELP panels that have some small but good examples. This IBM Redbooks publication extends those examples to help you solve typical problems found in data centers around the world.

The REXX with SDSF interface

REXX with SDSF integrates with your REXX executable (referred to in the remainder of this chapter as *REXX exec*) by executing commands and returning the results in REXX variables. To understand the new REXX with SDSF API you need to understand the commands and what they do and you need to know which variables are set and what these variables include.

Our discussion here briefly covers the REXX with SDSF API, but for more information refer to the chapter “Using SDSF with the REXX programming language” in *z/OS V1R9.0 SDSF Operation and Customization*, SA22-7670. You can also refer to the interactive tutorial panels that display when you press PF1 when using SDSF or use the REXXHELP command from any SDSF panel.

The best way to learn the interface is to run test programs while you are reading about the various features and capabilities. Nothing fixes ideas in your mind better than real experience. As you learn about commands and variables, run a small program that tries out the command or tests the variables. We have found the best approach is to display everything and to never assume that a variable will assume a specific value.

Example 1 shows the skeleton that we used to develop the examples in this book to explore all aspects of the API.

Example 1 Skeleton code used as the base for the examples in this book

```
/*  REXX  */

say;say;say            /*  Force the say text to the next page  */

/*  Load the SDSF environment and abort on failure  */

IsfRC = isfcalls( "ON" )
if IsfRC <> 0 then do
   say "RC" IsfRC "returned from isfcalls( ON )"
   exit IsfRC
   end

/*  Issue the command  */

address SDSF ""
if RC <> 0 then do
   say "RC" RC "returned from ..."
   call DisplayMessages
   end
```

```
20
21 /*  Display the user log associated with the action  */
22
23 say isfulog.0 "user log lines"
24 do i = 1 to isfulog.0
25    say "   '"isfulog.i"'"
26    end
27
28 /*  Display the responses associated with the action  */
29
30 say isfresp.0 "response lines"
31 do i = 1 to isfresp.0
32    say "   '"isfresp.i"'"
33    end
34
35 /*  Unload the SDSF environment  */
36
37 call isfcalls "OFF"
38
39 exit 0
40
41 /*  Display the messages associated with the action  */
42
43 DisplayMessages:
44    say "isfmsg: '"isfmsg"'"
45    say isfmsg2.0 "long messages in the isfmsg2 stem:"
46    do i = 1 to isfmsg2.0
47       say "   '"isfmsg2.i"'"
48       end
```

The three `say` statements on line 3 in Example 1 were put in so that the first true diagnostic output line displays on the top line of a full page rather than on the bottom of the first page where it would be separated from the really interesting information that shows on the next page. You plug your command into line 15 and follow it with whatever you need to verify your understanding of what you are testing. Whenever you have a question—whether at the beginning of your education or well into it—test, test, and test some more. Seeing is believing.

Telling SDSF to execute commands

The REXX with SDSF API mimics the interactive use of the product. You might understand many of the API's facilities better by picturing just how you would accomplish the same thing by executing SDSF at your terminal. SDSF provides support to execute commands that you enter on the command line. These commands can include:

1. Panel display commands, such as ST or DA, that result in a tabular display replacing the current panel.
2. SDSF information commands, such as WHO or QUERY, that result in a temporary information display.
3. MVS system commands, called *slash* commands, that perform system operator functions.

You use the ISFEXEC API command to request that SDSF execute these kinds of commands. Thus you enter the following command to switch the status display:

```
isfexec st
```

To execute the **who** command, you enter this command:

```
isfexec who
```

To issue the command to display outstanding replies, you enter this command:

```
isfexec /d r
```

The general syntax of the ISFEXEC command is:

```
isfexec <command> [(<options>)]
```

The result of executing a command depends on what kind of command you execute. Panel display commands create virtual tabular displays that are returned to the executable as a series of stem variables, one for each selected column. SDSF information command responses are returned in a special REXX stem variable, `isfresp`. MVS command responses are returned in a special REXX stem variable, `isfulog`.

Interactive SDSF also supports filtering commands, such as DEST, PREFIX, and INPUT that establish selection criteria for populating tabular displays. REXX with SDSF supports these commands in a different way. Rather than issue the command, you set a special REXX variable—one for each supported command—that accomplishes the same end.

For example, you implement the SDSF command SET PREFIX ADR*Q* in REXX as:

```
isfprefix = "ADR*Q*"
```

Appendix A, "REXX variables for SDSF host commands" on page 269 includes a complete list of the filtering commands and associated variables.

Panel display commands

Until now, interactive SDSF was the only ways to access SDSF data. Interactive SDSF uses the panel display commands to switch from the current panel to the named panel. For example, the DA command switches to the active job panel. The panel displays with all the fields in the primary field list as established when SDSF was installed, one column for each field. You can view fields that do not fit onto the initial panel by scrolling to the left or right or by scrolling up and down.

With REXX with SDSF, when you issue a panel display command, SDSF displays the panel by creating a virtual panel. It sets a series of stem variables to the values that you would see if you had entered the command on the interactive display. There is one stem variable for each field/column to display with one member for each row. So, if you executed the ISFEXEC ST command, the JNAME stem would be set for the job names with the job name on the first row returned in JNAME.1, the second in JNAME.2, and so on. The `0 member` includes the number of rows in the display and is the same for all the returned stem variables. All the stem variables are set when control returns from your ISFEXEC command. The API does not support the concept of scrolling.

The stem variable names are frequently the same as the column titles but not always so. SDSF uses special names, which are its internal identifiers for the values. You can use the **`colshelp`** command on any SDSF panel to open a help window that tells you the names of every variable for every panel along with the column title and an indication of whether it is a delayed column. It is important to know which stem variables are delayed (as you will see shortly). The names of each variable are listed in Appendix A, "REXX variables for SDSF host commands" on page 269.

In addition to the stem variables for the display columns, the API returns one additional stem variable. `TOKEN.i` is set to a special value that uniquely identifies the row. If you want to take some action against this row, to issue an operator command against it (**`release/hold`**, perhaps, or **`cancel`**) or to overtype the value in some column, you identify the row using the tokens.

SDSF special variables

In addition to the stem variables, SDSF also exchanges control information in special variables that begin with the letters *isf*. You can set these variables to make requests of SDSF and to interrogate the variables to get information useful to you in understanding how your call worked. Because of this naming scheme, we recommend that you avoid starting your variables with the letters *isf*. Otherwise, you might find yourself participating in a dialog that you had not intended.

For more information about the special variables, see *z/OS V1R9.0 SDSF Operation and Customization*, SA22-7670. Some of the more common variables are:

`isfcols`
: Column name list. On input to an ISF command, you set `isfcols` to the names of all the columns that you want returned. On output from the command, `isfcols` is set to the names of all the columns that are returned. The difference is when 1) `isfcols` is blank on input, 2) when SDSF adds columns (more on this later), or 3) when some of the names that you pass to SDSF are not legal columns on the display that you requested.

`isfucols`
: Updatable column name list. On output from an ISF command, this variable is set to the names of all columns in `isfcols` which can be updated by the user.

`isfrows`
: Returned row count.

`isfsort`
: Sort order. You set `isfsort` to direct SDSF to sort the rows in the virtual table before returning them to you.

`isfprefix`
: Job name pattern. Setting `isfprefix` is the same as using the PREFIX command in interactive SDSF.

`isfowner`
: Ownerid pattern. Setting `isfowner` is the same as using the OWNER command in interactive SDSF.

`isfmsg`
: ISPF short message. Capsule description of how the command completed if not blank.

`isfmsg2`
: Additional message stem. Set to additional messages, informational, warning and error. `isfmsg2.0` is the number of messages, `isfmsg2.i` is the `i-th` message.

Primary and alternate field lists

When you enter an interactive panel command at the SDSF command line, you see a group of columns that reflect the primary field definition set up when SDSF was installed. You can scroll the list left and right, but the only columns that you see are those defined as *primary fields*. Entering a question mark (**?**) primary command on the command line changes the panel to display the alternate fields as defined at installation. Again, you can scroll left and right, but the only columns you see are those defined as alternate fields.

REXX with SDSF honors the primary and alternate field lists just as interactive SDSF does. Unlike interactive SDSF, however, you can specify which set of fields you want returned to you by specifying or omitting the *alternate* option. *Omitting* the alternate option on the ISFEXEC command retrieves the *primary* fields and *specifying* it retrieves the *alternate* fields. Specifying the alternate option on the ISFEXEC command can lead to problems; however, if SDSF was installed with the primary and alternate field definitions significantly modified from the values with which the product is shipped. As shipped, the alternate definitions include all the primary fields. So, by specifying the alternate option, you can access all the columns defined for the panel. If fields have been removed from the alternate definition, it is possible that your program could require a combination of columns which, while defined for the panel you are requesting, do not exist in either the primary or alternate field lists.

Another significant difference between the interactive and program versions of SDSF are the REXX variables that are used to pass information between SDSF and REXX. When you issue a panel definition command through ISFEXEC, SDSF returns all of the stem variable names in REXX variable `isfcols`. In addition, if you set `isfcols` prior to issuing the ISFEXEC command, only those columns you specify are returned. (The TOKEN.i stem is returned regardless of its presence in the isfcols variable.) By limiting the columns that display, you can optimize your display, which saves some space and time.

Using DELAYED columns

SDSF has the concept of *delayed columns* in interactive displays that is important to understand in the REXX with SDSF environment. Simply stated, a delayed column is a column whose value cannot be immediately determined by an examination of memory. It requires a SPOOL I/O to retrieve the value and, thus, requires additional time to build and display either the interactive or virtual panel. In fact, the difference between the primary and alternate field lists in SDSF as shipped is that the alternate lists include the delayed columns and the primary lists do not.

REXX with SDSF also supports the performance enhancement of not retrieving delayed columns unless specifically requested on the ISFEXEC command. To retrieve delayed columns, you must include the **`delayed`** option with the ISFEXEC command:

```
isfexec da (alternate delayed
```

If you do not specify **`delayed`**, the delayed column values are not returned to you in stem variables, even if you explicitly request them in `isfcols`.

Primary and secondary panels

When you issue a panel display command using ISFEXEC, SDSF generates a virtual panel and creates stem variables to represent the data on the panel and on special variables (isfrows, isfcols, and so on) in order to pass additional information to your program. When you use the **?** action on one of the rows, SDSF creates a secondary panel—one subordinate to the first—and creates stem and special variables for that panel as well. Because these two panels exist simultaneously, SDSF allows you to handle their related variables in a manner that prevents them from interfering with each other. We discuss SDSF's special variables in this section and the stem variables in the next section, Overtyping data fields.

The original panel is called the *primary panel* and the `isf`*`xxx`* special variables all relate to the primary panel. These are the variables that we discussed previously in "SDSF special variables" on page 6.

The *secondary panel* has its own variables with the same function, but their name has the suffix *2*. Some of the more common variables include:

`isfcols2`	Set to the columns that you want returned before you invoke
`isfact`	SDSF sets to the columns that are returned when you get control back
`isfsort2`	Tell SDSF the desired sort order of the secondary display
`isfucols2`	SDSF informs you which columns in isfcols2 can be overtyped

Overtyping data fields

After issuing a panel display command, you might want to modify data on the virtual panel. You can overtype displayed data (if your authorization permits it) or enter line commands in the NP column such as **p** to purge data sets or **?** to display a job's data sets (JDS panel). REXX with SDSF calls overtyping data *taking an action* and provides the following ISFACT command to allow you to do it:

```
isfact <panel> token('<token>') parm(<column> <value> ...) (<options>)
```

The first thing to note is that you always code the panel name, such as `ST` or `DA`. Second, you code the *`<token>`* which identifies the row. This token is the one returned to you when you executed the ISFEXEC command to create the virtual panel. When you code the token, you must enclose it in apostrophes to ensure it is interpreted properly by SDSF.

After identifying the panel and token, you indicate which columns you want to modify and specify the new data to put into them by coding pairs of values in the parm. The first value in the pair is the column name which is the same as the stem variable names (without the tail). The second value is what you want the variable to include. If you want to set a second column at the same time, you code a second pair after the first.

There are several options you can specify, but the one we want to emphasize is the PREFIX option, which is coded as:

```
(PREFIX <string>)
```

To understand PREFIX, you need to understand the purpose for which it was intended. A typical programming problem would be to retrieve rows on a panel, such as *O*, where each row would correspond to one group of SYSOUT data sets, satisfying the conditions of the specified filters (prefix, ownerid, and so forth.), and then displaying the individual data sets by using the **`?`** line command on jobs of interest.

Now, let us say that you want to enter a line command on one of the lines of *this* display, say to purge one of the data sets. How would you do this? Here are the steps:

1. First you execute ISFEXEC **`o`** to display the virtual output panel.
2. Then execute **`isfact o token('<SYSOUT-token>') parm(np ?)`** to display the JDS virtual panel for the data sets for the row identified by **`<SYSOUT-token>`**, one of the token stem variables that is returned by ISFEXEC.
3. Execute **`isfact o token('<dataset-token>'') parm(np p)`** for the row identified by **`<dataset-token>`**, one of the token stem variables that is returned by ISFACT, that you want to purge.

Example 2 shows a skeleton of this process.

Example 2   Interference between two SDSF commands

```
/*  REXX  */

isfprefix = ...
isfowner  = ...
isfcols   = ...

address SDSF "isfexec o"

do i = 1 to isfrows
   address SDSF "isfact o token('"token.i"') parm(np ?)"
   do j = 1 to isfrows
      address SDSF "isfact o token('"token.j"') parm(np p)"
      end
   end
```

There are two serious issues with the skeleton shown in Example 2.

- The **isfact** on line 10 has overlaid the token stem variable that was returned by the **isfexec** on line 7.
- Both the **isfexec** on line 7 and **isfact** on line 10 have set variable `isfrows`, so the number of rows on the virtual output panel has been lost.

You need to address the fact that the SDSF commands have interfered with each other in the code. Having a common `isfrows` means that you need to save the value returned by ISFEXEC before issuing the ISFACT. However, sharing a common token stem array means that you now need a loop to save all the token values from the ISFEXEC. And what about common columns between the two panels? You have to save every common field of interest.

Now, you can enter the PREFIX option. You use the PREFIX string as a prefix to create new stem variable names (but not to the special variables such as `isfcols2` or `isfrows`) to eliminate the requirement to copy the values returned by the first SDSF function before issuing the subsequent function. Adding the PREFIX option provides a new version of the skeleton, as shown in Example 3.

In Example 3, PREFIX has been added to the ISFACT on line 11, and the effect is that all stem variables returned by that ISFACT now begin with `j_`. So the reference to the JDS row token on line 13 is now to `j_token` and the tokens returned by ISFEXEC are intact. In this example, we also copied `isfrows` on line 8 to remove the reuse of that variable.

Example 3 Interference solved using PREFIX

```
01 /*  REXX  */
02
03 isfprefix = ...
04 isfowner  = ...
05 isfcols   = ...
06
07 address SDSF "isfexec o"
08 Orows = isfrows
09
10 do i = 1 to Orows
11    address SDSF "isfact o token('"token.i"') parm(np ?) (prefix j_"
12    do j = 1 to isfrows
13       address SDSF "isfact o token('"j_token.j"') parm(np p)"
14       end
15    end
```

New capabilities of REXX with SDSF in z/OS V1.9

REXX with SDSF extends the capabilities on most of the SDSF functions:

- The system command in a slash command can be up to 124 characters long.
- System command responses are returned in a variable.
- A return code is set for most SDSF commands, action characters, and column field modifications.
- Each function is performed without changing the user's customized online environment.
- Users can connect to another SDSF server or JES2 node between host commands.
- The data on each tabular panel is available outside the panel.

The REXX interface itself and the new capabilities on the SDSF functions provide opportunities to implement new system management functions. Users can now:

- Issue longer system command in an offline environment.

  In batch mode, the SDSF program can only accept system command up to 42 characters long. User can now run a REXX exec in an offline environment to issue system command up to 126 characters long.

- Confirm that a system command is executed successfully.

  The SDSF short message confirms that SDSF has issued the system command successfully. User can now look in the command responses for the expected message text to confirm that the system has executed the command successfully.

  Chapter 1, "Issuing a system command" on page 37, implements a scenario that illustrates these functions.

- Better automate system workloads.

  The REXX interface sets a return code for most of the system-related SDSF functions. When using this interface in an offline environment, user can start or bypass later job steps based on the REXX exec return code.

- Better schedule operational events.

  Users can package a sequence of SDSF functions in a REXX exec and schedule it to run offline at a predetermined time without changing the user's customized interactive environment (for example, the PREFIX, OWNER, and DEST setups).

- Better control system resources in a sysplex environment.

  The REXX interface allows user to connect to a different SDSF server or JES2 node dynamically, which enables the user to better control the sysplex resources without exiting the current online SDSF session.

- Display SDSF data in a different format.

  Users can extract data from an SDSF tabular panel and present it in a different format. For example, format the data in a printable report, or even write an ISPF dialog box to display the extracted data and issue further host commands.

  Chapter 3, “Bulk job update processor” on page 97, implements a scenario that illustrates these functions.

- Display SDSF data at a remote site.

  Users can extract data from an SDSF tabular panel and send it to a remote system, which can be running on a different type of computer system. The remote system can further process the data and display it in a different presentation.

  Chapter 7, “Reviewing execution of a job” on page 173 implements a scenario that illustrates this function.

- Send up-to-date system status to a remote site.

  Users can issue system command to query the status of a job or a subsystem and pass it to a remote system. The remote system can further start, stop, or resume local workloads.

- Perform SDSF functions initiated by a remote site.

  User can set up an SDSF server to receive requests from a remote client which can run on different types of computer systems.

  Chapter 8, “Remote control from other systems” on page 187, Chapter 9, “JOB schedule and control” on page 201, and Chapter 10, “SDSF data in graphics” on page 223, implement scenarios that illustrate these functions.

- Use SDSF functions in a high-level language program.

  User can write an assembler program to accept and pass SDSF parameters from a high-level language program to a REXX exec. The REXX exec can then invoke the SDSF host commands.

  Chapter 4, “SDSF support for the COBOL language” on page 123, implements a scenario that illustrates this function.

SDSF programming practices

When using the REXX interface for SDSF functions, you need to consider the differences between the host environment and the interactive environment and make adjustments to the function invocations. Otherwise, the results and the performance can be inconsistent with that from the interactive environment.

Host environment verses interactive environment

Consider the following difference between the host environment and the interactive environment:

1. The host environment does not use any ISPF services.

   SDSF does not use the user customized setups that are saved in the ISFPROF member of the ISPF profile data set. It uses the defaults defined in the SDSF parms or those defined in the program code.

   User might get different rows from a tabular display; the command might be issued using a different console type, which can further affect the returned command responses as well as the console performance.

2. The host environment does not allow sharing of an EMCS console.

   By default, SDSF does not allow an EMCS console to share among multiple address spaces, unless the user has customized the Console.EMCS.CrossShare field in the group definition to `TRUE` or the SDSF user exit is implemented to turn on the UPRSFLG5.UPRS5CSX bit. For more details about the Console.EMCS.CrossShare field and the UPRS5CSX bit, refer to the *z/OS SDSF Customization and Operation,* SG22-7670.

   This means that when the user is current running SDSF in an interactive environment using a specific EMCS console, running a REXX exec at the same time in an online host environment gets the shared EMCS console. Running the same REXX exec in an offline host environment fails to get the shared EMCS console and uses the internal console instead.

   Regardless of the different results, both internal console and shared EMCS console do not return any command responses to SDSF. If your REXX exec requires command responses back, specify a unique EMCS console name in the `isfcons` variable.

3. The host environment treats each host command invocation as though the user:

   a. Logs on with a logon procedure called REXX.

   b. Has READ access to the JCL profile in the RACF® TSOAUTH class and no access to the ACCT or OPER profile.

c. Logs on with the terminal name which SDSF derived from SAF or TSO based on the current environment.

When RACF `SDSF` class is not activated, SDSF uses the SDSF parms (ILPROC, XLPROC, ITNAME, XTNAME, IUID, XUID, and TSOAUTH in the group definitions) to put a user into an SDSF group.

In a host environment, SDSF assumes all users have only JCL authority. If the group that the user joins when running SDSF in an interactive environment has more than JCL specified on the TSOAUTH parm, the user will fail to join this same group when running SDSF in the host environment. Thus, the user can end up in a lower authority group. When this happens, user loses the authorities on some of the SDSF functions and uses a different set of SDSF defaults.

4. The host environment uses different data and time format.

   All dates formatted will be in yyyy.ddd format. The user can use the `REXX date()` function to reformat the date to another format.

5. The host environment uses different number formats.

   In this environment, numbers are:

   – Do not include commas.
   – Are never scaled, as they are not restricted by the column width. They will not include scaling characters such as `T` or `M`. However, some values are formatted with units. For example, the `MemLimit` column on the DA panel are formatted with `MB`, `PB`, and so on.
   – Are formatted as three asterisks (***) if the data is invalid or overflowed.
   – Are formatted using a decimal point followed by one or two decimal digits when the data is a fractional number.

Recommendations

Here are the recommendations to get consistent results and performance:

1. Override the global defaults with the following REXX variables:

ISFSERVER	Specifies the server to connect to
ISFJESNAME	Specifies the JES2 system to operate on
ISFSCHARS	Affects FIND command results
ISFTIMEOUT	Affects sysplex display on a tabular panel
ISFTRMASK	Affects tracing

2. Override the user group defaults with the following REXX variables:

ISFAPPC	Filters the rows on a tabular panel
ISFDEST	Filters the rows on a tabular panel
ISFINPUT	Filters the rows on a tabular panel
ISFOWNER	Filters the rows on a tabular panel
ISFPREFIX	Filters the rows on a tabular panel

3. Override the SDSF hard-coded defaults with the following REXX variables:

ISFCONS	Affects the type of console used
ISFDELAY	Affects the command responses received
ISFSORT	Affects the row sequence of a tabular panel
ISFSYSNAME	Filters the rows on a tabular panel

4. Always list the current environment:
   a. Issue the WHO command.

      This command can confirm that the session is connected to the correct server and the correct JES2 system and that the user has joined the correct SDSF group before issuing further host commands.

      You can find a sample REXX exec to issue a WHO command in the REXXHELP command.

   b. Issue the QUERY command.

      This command can confirm that the user has authority on a specific SDSF command before issuing the ISFEXEC command.

   c. For tabular panel, set the ISFACTIONS variables to ON.

      This variable can confirm that the user has authority on a specific action before issuing the ISFACT command.

      You can find a sample REXX exec to list action characters in the REXXHELP command.

   d. For tabular panel, write out the ISFDISPLAY variable.

      This variable can confirm that the correct filter commands are currently effective.

   e. Display the host command return code.
   f. Display the SDSF short message in the `isfmsg` variable.
   g. Display all SDSF messages in the `isfmsg2` stem variable.

If the REXX exec is used frequently, you can consider writing an ISPF dialog box to save the overridden REXX variables in the dialog box's profile member and prime the REXX variables with the profile variables every time the REXX exec is run. Chapter 1, “Issuing a system command” on page 37 implements this suggestion.

For a quick reference on the REXX variables, see Table 28 in Chapter 28, “General REXX variables” on page 296. For more details about each REXX variable, refer to the *z/OS SDSF Customization and Operation,* SG22-7670.

Tune the command processing

For better performance, decide on the type of console to use. For bulk processing, an *internal console* has a shorter processing path length. For the ability to confirm successful execution of a system command, a *primary EMCS console* guarantees the return of the command responses.

For bulk update, to force a tabular action to use an internal console, the user can specify an EMCS console which the caller has no access on, which causes SDSF to use the internal console.

Areas to consider

The following are some areas you should consider when issuing the host commands:

1. System command responses are not returned when the console that is used is not a primary EMCS console (that is, a shared EMCS console or an internal console). This can happen when the REXX exec runs while the user is currently running SDSF in an interactive mode. If your REXX exec requires to get command responses back, specify a different EMCS console name in the ISFCONS variable.
2. System commands can be up to 126 characters long. SDSF requires two single quotation marks to represent a single quotation mark. These two single quotation marks are counted as two characters within the 126 characters.
3. The WAIT option is ignored when the console that is used is not a primary EMCS console.
4. Upon completion of the REXX exec, SDSF will not deactivate the primary EMCS console when the system command it issued has any unreplied WTORs.
5. ISFEXEC command returns the desired column only if the desired column is on the column-selection list of the user group. For example, if ISFEXEC is issued against the primary panel asking for a column on the alternate panel, SDSF does not return the desired column.

6. The `isfcols` variable is both an input and an output REXX variable. You might get incorrect results when two consecutive ISFEXEC commands are issued for two different tabular panel. The output in the `isfcols` variable of the first ISFEXEC command is used as the input for the second ISFEXEC command.

Debugging tools

When the ISFEXEC or the ISFACT host command fails, there are two areas that you can customize to get more details about the failure:

- VERBOSE parameter on the host command
- SDSF trace

VERBOSE parameter

The VERBOSE parameter adds diagnostic messages to the `isfmsg`2 stem variable when you use the ISFEXEC or the ISFACT host command to get tabular display type of panel entries. The messages describe each row variable that is created by SDSF.

Here are two samples to specify the VERBOSE parameter:

```
Address SDSF "ISFEXEC ST (VERBOSE)"
Address SDSF "ISFACT ST TOKEN('"TOKEN.ix"') PARM(NP ?) (VERBOSE)"
```

SDSF trace

Although SDSF TRACE is intended to be used by the IBM technical staffs, in most cases, you might find it useful in debugging security setup issues. Common security setup issues include:

- A user can perform an SDSF function for which the user does not have authority.
- A user joins the wrong SDSF group and is granted more authority than is proper.

Setup

To start an SDSF trace in the host environment, consider the following customizable areas:

- **REXX variables**

There are two variables the caller can customize to run a function trace, and these variables have no effect on initialization trace:

– ISFTRACE

  This variable sets the trace option to ON or OFF and is equivalent to entering the TRACE on or TRACE OFF command in an interactive environment. For more details about the TRACE command, access the online REXX with SDSF help tutorial (enter REXXHELP while in SDSF).

– ISFTRMASK

  This variable specifies the trace mask option, which is equivalent to entering the TRACE command with the trace mask in an interactive environment. For example, ISFTRMASK = ‘8084’ or ALL. For more details about the TRACE command, access the online REXX with SDSF help tutorial (enter REXXHELP while in SDSF).

Refer to “Types of traces” on page 19 for more details about the types of SDSF traces.

- **Trace data set allocation**

  You can preallocate a trace data set to the ISFTRACE ddname or let SDSF allocates one dynamically upon starting the trace. This allocation determines the type of SDSF trace to run. For more details about the types of SDSF traces, refer to the “Types of traces” on page 19.

  The preallocated trace data set can be a physical sequential data set with the record format of VBA and the logical record length of 137, or it can be a SYSOUT data set. If it is a sequential data set, make sure it is large enough to hold all the trace data. Each trace entry has multiple lines but only one sequence number at the end of the first line. To obtain whether the trace data is wrapped, check to make sure that the first entry has a sequence number of one.

  SDSF allocates the trace data set the same way that it does when the user enters the TRACE ON command in an online environment. Refer to *z/OS SDSF Customization and Operation,* SG22-7670 for more details about how SDSF allocates a trace data set.

- **User authority**

  If RACF SDSF class is activated, you need to have READ access on the ISFCMD.MAINT.TRACE resource. Otherwise, you need to have the TRACE command listed in the AUTH parameter of the user group definition in the SDSF parms.

Types of traces

Each ISFEXEC and ISFACT host command processing consists of two phases:

- The initialization phase
- The execution phase

So there are two corresponding types of SDSF traces:

- **The initialization trace**

  This trace records the initialization phase of a single ISFEXEC or ISFACT host command (that is, when SDSF assigns the user to an SDSF group). To start the trace, preallocate a trace data set with the ISFTRACE ddname. SDSF turns the trace option on automatically when the ISFTRACE ddname is allocated. To end the trace, unallocate the ISFTRACE ddname. SDSF does not free the ISFTRACE ddname automatically upon terminating the trace. The REXX exec should unallocate the ISFTRACE ddname before returning to the caller.

  See the REXX exec in Example 4.

Example 4 Start an initialization trace

```
/* REXX */
rcode = ISFCALLS('ON')
"alloc f(ISFTRACE) da('smith.sdsf.trace') shr"
address SDSF "ISFEXEC '/p stc1'"
"free f(ISFTRACE)"
call ISFCALLS('OFF')
return rcode
```

- **The function trace**

  This trace records the execution phase of a single ISFEXEC or ISFACT host command. To start the trace, set the ISFTRACE variable to ON before invoking the host command. To end the trace, set the ISFTRACE variable to OFF or let the REXX exec runs to its completion.

  See the REXX exec in Example 5.

Example 5 Start the function trace

```
/* REXX */
rcode = ISFCALLS('ON')
ISFTRMASK = ALL
ISFTRACE  = ON
address SDSF "ISFEXEC '/p stc1'"
ISFTRACE  = OFF
call ISFCALLS('OFF')
return rcode
```

To capture all traces for all host commands into a single data set, preallocate the ISFTRACE ddname to a SYSOUT data set. See the REXX exec in Example 6.

Example 6 Start both initialization trace and function trace for all host commands

```
/* rexx */
"alloc f(ISFTRACE) sysout(a)"
ISFTRMASK = 'ALL'
rcode = ISFCALLS('ON')
address SDSF "ISFEXEC '/p stc1'"
address SDSF "ISFEXEC '/p stc2'"
call ISFCALLS('OFF')
"free f(isftrace)"
return rcode
```

Analyzing the trace output

SDSF cuts a trace entry for each RACROUTE macro invocation. To find all the RACF resources required to perform a single host command, start both traces and look for all occurrences of the SAFRC keyword. Each SAFRC line has the following fields:

SAFRC	The Security Authorization Facility (SAF) return code For RACF 0 - access granted 4 - resource class is not activated 8 - access denied
CLASS	The RACF resource class
RESOURCE	The resource name checked by SDSF

Example 7 shows a report that is created by executing the REXX exec in Example 6 on page 20 and then using the ISPF VIEW command in the following order:

1. EXCLUDE ALL
2. FIND SAFRC ALL
3. DELETE EXCLUDE ALL

To save the trace output from a SYSOUT into a data set, you can issue the XDC or XFC action character against the SYSOUT or issue the SE action character to read the SYSOUT using the ISPF VIEW function and issue the ISPF CREATE command.

Example 7 SDSF SAFRC trace entries

```
SAFRC= 0 CLASS= SDSF     REQSTOR= ISFGROUP ATTR=02 LEN= 19 RESOURCE=GROUP.ISFSPROG.SDSF
SAFRC= 0 CLASS= SDSF     REQSTOR= ISFIPREF ATTR=02 LEN= 20 RESOURCE=ISFCMD.FILTER.PREFIX
SAFRC= 0 CLASS= SDSF     REQSTOR= ISFIOWNR ATTR=02 LEN= 19 RESOURCE=ISFCMD.FILTER.OWNER
SAFRC= 0 CLASS= SDSF     REQSTOR= ISFOPER  ATTR=02 LEN= 17 RESOURCE=ISFOPER.DEST.JES2
SAFRC= 0 CLASS= SDSF     REQSTOR= ISFADEST ATTR=02 LEN= 20 RESOURCE=ISFOPER.ANYDEST.JES2
SAFRC= 0 CLASS= SDSF     REQSTOR= ISFOPSYS ATTR=02 LEN= 14 RESOURCE=ISFOPER.SYSTEM
SAFRC= 0 CLASS= SDSF     REQSTOR= ISFCULOG ATTR=02 LEN= 21 RESOURCE=ISFCMD.ODSP.ULOG.JES2
SAFRC= 0 CLASS= OPERCMDS REQSTOR= ISFACONS ATTR=02 LEN= 15 RESOURCE=MVS.MCSOPER.SMITH
SAFRC= 0 CLASS= SDSF     REQSTOR= ISFGROUP ATTR=02 LEN= 19 RESOURCE=GROUP.ISFSPROG.SDSF
SAFRC= 0 CLASS= SDSF     REQSTOR= ISFIPREF ATTR=02 LEN= 20 RESOURCE=ISFCMD.FILTER.PREFIX
SAFRC= 0 CLASS= SDSF     REQSTOR= ISFIOWNR ATTR=02 LEN= 19 RESOURCE=ISFCMD.FILTER.OWNER
SAFRC= 0 CLASS= SDSF     REQSTOR= ISFOPER  ATTR=02 LEN= 17 RESOURCE=ISFOPER.DEST.JES2
SAFRC= 0 CLASS= SDSF     REQSTOR= ISFADEST ATTR=02 LEN= 20 RESOURCE=ISFOPER.ANYDEST.JES2
SAFRC= 0 CLASS= SDSF     REQSTOR= ISFOPSYS ATTR=02 LEN= 14 RESOURCE=ISFOPER.SYSTEM
SAFRC= 0 CLASS= SDSF     REQSTOR= ISFCULOG ATTR=02 LEN= 21 RESOURCE=ISFCMD.ODSP.ULOG.JES2
SAFRC= 0 CLASS= OPERCMDS REQSTOR= ISFACONS ATTR=02 LEN= 15 RESOURCE=MVS.MCSOPER.SMITH
```

Running REXX executables

There are some different ways that you can run a REXX exec, depending on the address space type in which it is executed: TSO/E address spaces, non TSO/E batch address spaces, or UNIX System Services address spaces.

TSO/E address spaces

TSO/E address spaces offer some extended functions to the REXX programming environment.

Interactive TSO/E address spaces

TSO/E is a base element of the z/OS operating system that allows users to interactively work with the system.You can use TSO/E in any one of the following environments:

- Line mode TSO/E

  The way programmers originally communicated interactively with the MVS operating system was with TSO/E commands typed on a terminal, one line at a time. It is a quick and direct way to use TSO/E.

- ISPF/PDF

  The Interactive System Productivity Facility (ISPF) and its Program Development Facility (ISPF/PDF) work together with TSO/E to provide panels with which users can interact. ISPF provides the underlying dialog management service that displays panels and enables a user to navigate through the panels. ISPF/PDF is a dialog of ISPF that helps maintain libraries of information in TSO/E and allows a user to manage the library through facilities such as browse, edit, and utilities.

Running REXX executables in line mode TSO/E

Line mode TSO/E is the way programmers originally communicated interactively with the MVS operating system. They issued TSO/E commands and entered text at the terminal one line at a time rather than from panels.

You can still use line mode TSO/E to communicate with the MVS system. To use line mode TSO/E, enter a command after the `READY` message displays. The `READY` message indicates that you are in line mode TSO/E.

When you are in either ISPF/PDF, you need only press the RETURN PF key (PF 4) or press the END PF key (PF 3) repeatedly until you see the `READY` message.

When you are in the READY prompt, you can specify a data set name, according to the TSO/E data set naming conventions, in several different ways. For example the data set name *USERID.REXX.EXEC(SDSFREXX)* can be specified as a fully-qualified data set that appears within single quotation marks, as shown in Example 8.

Example 8 Running a REXX exec in a fully-qualified data set

```
READY
EXEC 'userid.rexx.exec(sdsfrexx)' exec
```

A non fully-qualified data set, which has no quotation marks, can eliminate your profile prefix (usually your user ID) as well as the third qualifier exec (Example 9).

Example 9 Running a REXX exec in a non fully-qualified data set

```
READY
EXEC rexx.exec(sdsfrexx) exec           /* eliminates prefix */
READY
EXEC rexx(sdsfrexx) exec       /* eliminates prefix and exec */
```

There are also alternative ways of entering the EXEC command:

- **Explicit form**

  Enter EXEC or EX followed by the name of the data set that includes the CLIST or REXX exec. If you need prompting, you should invoke EXEC explicitly with the PROMPT option. Our examples are written using the explicit form of invocation.

- **Implicit form**

  Do not enter EXEC or EX. Enter only the name of the member to be found in a procedure library such as SYSEXEC or SYSPROC.[1] A procedure library consists of partitioned data sets allocated to the specific file (SYSPROC or SYSEXEC) either dynamically by the ALLOCATE command or as part of the LOGON procedure. TSO/E determines whether the member name is a system command before it searches the libraries. See Example 10.

Example 10 Executing a REXX implicit form

```
Menu  Options  View  Utilities  Compilers  Help
------------------------------------------------------------------------------
DSLIST - Data Sets Matching WAT                                    Row 1 of 11
Command ===> TSO @SYSCMD                                      Scroll ===> CSR

Command - Enter "/" to select action                  Message          Volume
-------------------------------------------------------------------------------
         WAT                                                           *ALIAS
         WAT.BRODCAST                                                  SBOXFF
         WAT.SC70.ISPF42.ISPPROF                                       SBOXE3
         WAT.SC70.ISPF42.ISPPROF.BATCH                                 SBOXE3
         WAT.SC70.SPFLOG1.LIST                                         SBOXB7
         WAT.SDSF.TRACE                                                SBOXE3
         WAT.SRCHFOR.LIST                                              SBOXB5
         WAT.SUPERC.LIST                                               SBOXBF
         WAT.TEST.JCL                                                  SBOX20
         WAT.TEST.OUTPUT                                               SBOXE3
         WAT.TEST.REXX                                                 SBOX20
***************************** End of Data Set list ****************************
```

[1] In the module name table, the LOADDD field contains the name of the DD from which REXX execs are fetched. The default TSO/E provides for non-TSO/E, TSO/E, and ISPF is SYSEXEC.

- **Extended implicit form**

  Enter a percent sign (%) followed by the member name. TSO/E only searches the procedure library for the specified name. This form is faster because the system does not search for commands. See Example 11.

Example 11 Executing a REXX exec extended implicit form

```
Menu  List  Mode  Functions  Utilities  Help
------------------------------------------------------------------------
                           ISPF Command Shell
 Enter TSO or Workstation commands below:

 ===> %@SYSCMD

 Place cursor on choice and press enter to Retrieve command

 => %@SYSCMD
 => ex 'JOE.TEST.REXX(@ISPCL)' 'JOE.TEST'
 => ex 'JOE.TEST.REXX(@BRLOG)' 'PATH(/tmp/JOE.syslog.sc70ts)'
 => ex 'JOE.TEST.REXX(@BRLOG)'
 => ex 'JOE.TEST.REXX(WISPCL)' 'JOE.TEST'
 => ex 'JOE.TEST.REXX(@BRLOG)' 'DSNAME(JOE.SYSLOG.SC70TS)'
 => ex 'JOE.TEST.REXX(@BRLOG)' 'DATASET(JOE.SYSLOG.SC70TS)'
 => ex 'JOE.TEST.REXX(@BRLOG)' 'JOE.SYSLOG.SC70TS'
 =>
 =>
```

Running REXX executables under ISPF/PDF

Interactive executables and executables written that involve user applications are generally run in the foreground. You can invoke an executable in the foreground in the following ways:

- From the command input field of any panel, as long as the command line is prefixed with the keyword *tso* (Example 12).

Example 12 Running a REXX exec from ISPF command input field

```
Menu  Utilities  Compilers  Options  Status  Help
------------------------------------------------------------------------------
                           ISPF Primary Option Menu
Option ===> tso exec rexx.exec(sdsfrexx) exec

0  Settings      Terminal and user parameters           User ID . : REDBOOK
1  View          Display source data or listings        Time. . . : 10:31
2  Edit          Create or change source data           Terminal. : 3278
3  Utilities     Perform utility functions              Screen. . : 2
```

```
4  Foreground    Interactive language processing        Language. : ENGLISH
5  Batch         Submit job for language processing     Appl ID . : PDF
6  Command       Enter TSO or Workstation commands      TSO logon : IKJACCT
7  Dialog Test   Perform dialog testing                 TSO prefix: REDBOOK
9  IBM Products  IBM program development products       System ID : SC70
10 SCLM          SW Configuration Library Manager       MVS acct. : ACCNT#
11 Workplace     ISPF Object/Action Workplace           Release . : ISPF 5.9
12 z/OS System   z/OS system programmer applications
13 z/OS User     z/OS user applications


     Enter X to Terminate using log/list defaults
```

- From the ISPF command processor. The ISPF command shell option allows TSO commands, CLISTs, and REXX executables to be executed under ISPF. When in Workstation mode, the command entered on Option 6 is directed to the users workstation. See Example 13.

Example 13   Running a REXX exec from the ISPF command processor

```
Menu  List  Mode  Functions  Utilities  Help
------------------------------------------------------------------------------
                             ISPF Command Shell
Enter TSO or Workstation commands below:

===> exec rexx.exec(sdsfrexx) exec

Place cursor on choice and press enter to Retrieve command

=> ex 'itso.rexx.exec(@ISPCL)' 'itso.test'
=> ex 'itso.rexx.exec(@BRLOG)' 'PATH(/tmp/ITSO.syslog.sc70ts)'
=> ex 'itso.rexx.exec(@BRLOG)'
=> ex 'itso.rexx.exec(WISPCL)' 'itso.test'
=> ex 'itso.rexx.exec(@BRLOG)' 'DSNAME(ITSO.SYSLOG.SC70TS)'
=> ex 'itso.rexx.exec(@BRLOG)' 'DATASET(ITSO.SYSLOG.SC70TS)'
=> ex 'itso.rexx.exec(@BRLOG)' 'ITSO.SYSLOG.SC70TS'
=>
=>
=>
```

- From the command prompt provided by the ISPF **cmde** command. After typing **cmde** in the input command field of any panel, a pop-up panel (ISPCMDE) with a 234-character command input field is displayed.

  You can enter up to 234 characters using the entry field provided. ISPF allows TSO commands, CLISTS, and REXX execs and parameters to be entered in the input field. This panel is processed much like the PDF Option 6 panel. Data passed to this panel is translated to uppercase characters. Data passed from this panel remains as it appears on the panel.

Example 14 Running a REXX exec from the command prompt

```
ISPF Command Entry Panel

 Enter TSO commands below:
 ===> exec rexx.exec(sdsfrexx) exec
```

- From the member list of ISPF Dataset List Utility (3.4) Enter line command **M** (Member list) beside the PDS data set that includes your REXX executables. When the list member list has been displayed, type **exec** beside the member that includes the REXX in which you are interested.

Example 15 Executing a REXX execs from ISPF Dataset List Utility

```
Menu  Functions  Confirm  Utilities  Help
------------------------------------------------------------------------------
DSLIST          WAT.TEST.REXX                                Row 00001 of 00034
Command ===>                                                   Scroll ===> CSR
           Name     Prompt        Size  Created          Changed          ID
_________ @CMDBK01                 497  2007/04/15  2007/04/26 08:45:02  WAT
_________ @PARSE                    49  2007/04/10  2007/04/10 12:34:36  WAT
_________ @SYSCMD                  530  2007/04/13  2007/04/27 14:32:05  WAT
_________ @TESTING                 186  2007/04/15  2007/04/15 22:07:48  WAT
EXEC_____ ALTLIB   *RC=0             2  2007/03/29  2007/03/29 14:32:05  WAT
_________ BOBSAMP                    1  2007/03/28  2007/03/28 16:20:09  WAT
_________ FORAMY                   119  2007/04/09  2007/04/10 10:18:20  LEVEY
_________ IEFBR14                    5  2007/04/23  2007/04/23 16:36:21  WAT
_________ IKJEFT01                  11  2007/03/31  2007/04/03 08:51:19  WAT
_________ IRXJCL                    13  2007/03/31  2007/04/10 09:51:24  WAT
_________ JIRX                      20  2007/04/03  2007/04/10 09:41:53  WAT
_________ JISPF                     16  2007/04/03  2007/04/03 14:33:24  WAT
_________ JTSO                     137  2007/04/03  2007/04/27 11:10:38  WAT
_________ MSGRTN2                   21  2007/03/27  2007/03/27 17:28:31  JOE
_________ SAMPLE1                   32  2007/03/26  2007/03/27 09:17:35  JOE
_________ SAMPLE2                   64  2007/03/26  2007/03/27 15:28:42  JOE
_________ SAMP01                    38  2007/03/27  2007/03/27 15:33:11  WAT
_________ SAMP02                    40  2007/03/27  2007/03/27 15:42:46  WAT
_________ SAMP03                    68  2007/03/27  2007/03/27 17:29:08  JOE
_________ SAMP04                    58  2007/03/27  2007/03/27 16:22:11  WAT
_________ SAMP05                    78  2007/03/27  2007/03/27 16:31:03  WAT
_________ SAMP06                    95  2007/03/27  2007/03/27 16:36:28  WAT
_________ SAMP07                    71  2007/03/27  2007/03/27 17:29:55  JOE
_________ SAMP08                    65  2007/03/27  2007/03/27 16:40:14  WAT
_________ SAMP09                    31  2007/03/27  2007/03/27 16:41:36  WAT
_________ SAMP10                    19  2007/03/27  2007/03/27 16:42:32  WAT
```

Batch TSO/E address spaces

Executables that run in the background are processed when higher priority programs are not using the system. Background processing does not interfere with a person's use of the terminal. You can run time-consuming and low priority execs in the background, or executables that do not require terminal interaction.

Running an exec in the background is the same as running a CLIST in the background. The program needed to execute TSO/E commands from the background is a terminal monitor program (TMP), which can be one of the following: IKJEFT01, IKJEFT1A, or IKJEFT1B. The EXEC (execute) statement is used to execute program IKJEFT01 or the alternate programs IKJEFT1A and IKJEFT1B. Any of these programs sets up a TSO/E environment from which you can invoke execs and CLISTs and issue TSO/E commands. For example, to run an exec named @SYSCMD contained in a partitioned data set JOE.TEST.REXX, submit the following JCL.

Example 16 Running TSO/E in batch

```
File  Edit  Edit_Settings  Menu  Utilities  Compilers  Test  Help
-------------------------------------------------------------------------------
VIEW       REDBOOK.TEST.JCL(ALLSAMP) - 01.06              Columns 00001 00072
Command ===>                                                  Scroll ===> CSR
****** ***************************** Top of Data ******************************
000001 //REDBOOKS JOB 'SG24-7419',MSGCLASS=A,CLASS=A,NOTIFY=ITSOPRG
000002 //ISPF    EXEC PGM=IKJEFT01,DYNAMNBR=40
000003 //SYSEXEC  DD  DSN=JOE.TEST.REXX,DISP=(SHR)
000004 //         DD  DSN=WAT.TEST.REXX,DISP=(SHR)
000005 //SYSTSPRT DD  SYSOUT=A,HOLD=YES
000006 //SYSTSIN  DD  *
000007  @SYSCMD CMD(D SMF)
000008 /*
****** **************************** Bottom of Data ****************************
```

The EXEC statement defines the program as IKJEFT01.PGM= specifies the module being executed. In addition to IKJEFT01, there are two other entry points available for background execution that provide additional return code and abend support. The differences among the three entry points are:

- PGM=IKJEFT01

  When a command completes with a non-zero return code, the program goes to the next command. When a command abends, the step ends with a condition code of 12 (X'C').

- PGM=IKJEFT1A

  If a command or program being processed by IKJEFT1A ends with a system abend, IKJEFT1A causes the job step to terminate with a X'04C' system completion code. IKJEFT1A also returns to the caller the completion code from the command or program in register 15.

If a command or program being processed by IKJEFT1A ends with a user abend, IKJEFT1A saves the completion code in register 15 and then terminates.

If a command, program or REXX exec being processed by IKJEFT1A returns a non-zero return code to IKJEFT1A, IKJEFT1A saves this return code in register 15 and then terminates. Non-zero return codes to IKJEFT1A from CLISTs will not affect the contents of register 15 and the TMP continues processing.

For a non-zero return code or an abend from a command or program that was not given control directly by IKJEFT1A, no return code is saved in register 15, and IKJEFT1A does not terminate.

- PGM=IKJEFT1B

  If a command or program being processed by IKJEFT1B ends with a system or user abend, IKJEFT1B causes the job step to terminate with a X'04C' system completion code. IKJEFT1B also returns to the caller the completion code from the command or program in register 15.

  If a command, program, CLIST, or REXX exec being processed by IKJEFT1B returns a non-zero return code to IKJEFT1B, IKJEFT1B saves this return code in register 15 and then terminates.

  For a non-zero return code or abend completion code from a program or command that was not given control by IKJEFT1B, no return code is saved in register 15, and IKJEFT1B does not terminate.

In a DD statement, you can assign one or more PDSs to the SYSEXEC or SYSPROC system file, in the Example 16 on page 27 we have two files concatenated: JOE.TEST.REXX and WAT.TEST.REXX. The SYSTSPRT DD allows you to print output to a specified data set or a SYSOUT class. In the input stream, after the SYSTSIN DD, you can issue TSO/E commands and invoke execs and CLISTs.

Batch non TSO/E address spaces

Because executables that run in a non-TSO/E address space cannot be invoked by the TSO/E EXEC command, you must use other means to run them. Ways to run executables outside of TSO/E are:

- From MVS batch with JCL that specifies IRXJCL in the EXEC statement.
- From a high level program using the IRXEXEC or IRXJCL processing routines.

Using IRXJCL to run a REXX exec in MVS batch

To run an exec in MVS batch, specify IRXJCL as the program name (PGM=) on the JCL EXEC statement. Specify the member name of the exec and one argument you want to pass to the exec in the PARM field on the EXEC statement. You can specify only the name of a member of a PDS. You cannot specify the name of a sequential data set. The PDS must be allocated to the DD specified in the LOADDD field of the module name table. The default is SYSEXEC.

Example 17 shows example JCL to invoke the exec @SYSCMD.

Example 17 Using IRXJCL to run a REXX exec in MVS batch

```
File  Edit  Edit_Settings  Menu  Utilities  Compilers  Test  Help
-------------------------------------------------------------------------------
VIEW       REDBOOK.TEST.JCL(JOE) - 01.01                   Columns 00001 00072
Command ===>                                                  Scroll ===> CSR
****** ***************************** Top of Data ******************************
000001 //REDBOOK@ JOB 'SG24-7419',MSGCLASS=A,CLASS=A,NOTIFY=REDBOOK
000002 //* ------------------------------------------------------------------- *
000003 //* IRXJCL TEST
000004 //* ------------------------------------------------------------------- *
000005 //BATCH   EXEC PGM=IRXJCL,
000006 //             PARM='@SYSCMD CMD(D SMF) DELAY(5)'
000007 //*                 |     | |                |
000008 //* Name of exec     <-----> |                |
000009 //* Argument                 <---------------->
000006 //SYSEXEC  DD  DSN=WAT.TEST.REXX,DISP=(SHR)
000007 //SYSTSPRT DD  SYSOUT=A
000008 //SYSPRINT DD  SYSOUT=A
****** **************************** Bottom of Data ****************************
```

As Example 17 shows, the exec @SYSCMD is loaded from DD SYSEXEC. SYSEXEC is the default setting for the name of the DD from which an exec is to be loaded. In the example, one argument is passed to the exec. The argument can consist of more than one token. In this case, the argument is:

```
@SYSCMD CMD(D SMF) DELAY(5)
```

@SYSCMD is the name of the REXX exec. All the rest of the line are parameters that are parsed by the @SYSCMD. After submitting the job and executing the REXX exec, the output in this case is as shown in Example 18.

Example 18 Output from sample Example 17

```
Display  Filter  View  Print  Options  Help
-------------------------------------------------------------------------------
 SDSF OUTPUT DISPLAY REDBOOK@ JOB29764  DSID     4 LINE 6      COLUMNS 02- 81
 COMMAND INPUT ===>                                           SCROLL ===> CSR
IEF142I REDBOOK@ BATCH - STEP WAS EXECUTED - COND CODE 0000
IEF285I   WAT.TEST.REXX                                 KEPT
IEF285I   VOL SER NOS= SBOX20.
IEF285I   REDBOOK.REDBOOK@.JOB29764.D0000101.?          SYSOUT
IEF285I   REDBOOK.REDBOOK@.JOB29764.D0000102.?          SYSOUT
IEF373I STEP/BATCH   /START 2007121.1124
IEF374I STEP/BATCH   /STOP  2007121.1124 CPU    OMIN 00.02SEC SRB    OMIN
IEF375I  JOB/REDBOOK@/START 2007121.1124
IEF376I  JOB/REDBOOK@/STOP  2007121.1124 CPU    OMIN 00.02SEC SRB    OMIN
@SYSCMD operands  : CMD(D SMF) DELAY(5)
SDSF HCE status   : established RC=00
-------------------------------------------------------------------------------
ISFEXEC options   : ( )
Original command  : /D SMF

SDSF short message: COMMAND ISSUED
SDSF long  message: ISF754I Command 'SET CONSOLE' generated from associated
SDSF long  message: ISF754I Command 'SET DELAY 5' generated from associated

SDSF ULOG messages:
SC70      2007121  11:24:34.99            ISF031I CONSOLE REDBOOK ACTIVATED
SC70      2007121  11:24:34.99           -D SMF
SC70      2007121  11:24:35.00            IEE974I 11.24.35 SMF DATA SETS 793
                                                   NAME               VOLSER
                                                 P-SYS1.SC70.MAN1      SBOXD5
                                                 S-SYS1.SC70.MAN2      SBOXD5
                                                 S-SYS1.SC70.MAN3      SBOXD5
```

IRXJCL returns a return code as the step completion code. However, the step completion code is limited to a maximum of 4095, in decimal. If the return code is greater than 4095 (decimal), the system uses the right most three digits of the hexadecimal representation of the return code and converts it to decimal for use as the step completion code.

UNIX System Services address spaces

The set of z/OS UNIX extensions to the TSO/E REXX language enable REXX programs to access z/OS UNIX callable services. The z/OS UNIX extensions, called syscall commands, have names that correspond to the names of the callable services they invoke (for example, **`access`**, **`chmod`**, and **`chown`**).

You can run an interpreted or compiled REXX program with syscall commands from TSO/E, from MVS batch, from the z/OS shells, or from a program.You can run a REXX program with syscall commands only on a system with z/OS UNIX System Services installed.

The choices to access z/OS UNIX include:

- **`rlogin`** or **`telnet`**

  The rlogin and telnet are interfaces that heritage UNIX users will find most comfortable. Access should be through an ASCII terminal. We describe these interfaces in “telnet and rlogin connections” on page 34. **`rlogin`** and **`telnet`** provide an asynchronous interface to the shell that is familiar to UNIX users.

- The TSO OMVS command

  The TSO OMVS command provides a telnet-like interface, subject to the limitations of 3270 technology.

- The ISPF shell

  The ISPF shell is an interface that heritage MVS users will find most comfortable. It exploits full-screen capabilities of ISPF.

- BPXBATCH

  BPXBATCH allows UNIX work to be executed from batch JCL. We describe this interface in “The BPXBATCH utility” on page 32.

Batch UNIX System Services address spaces

You can access z/OS UNIX services from MVS batch using the BPXBATCH utility. BPXBATCH makes it easy for you to run shell scripts and executable files that reside in z/OS UNIX files through the MVS job control language (JCL). If you do most of your work from TSO/E, using BPXBATCH saves you the trouble of going into the shell to run your scripts and executable files. REXX executables can also use BPXBATCH to run shell scripts and executable files.

The BPXBATCH utility

You can invoke BPXBATCH from a batch job or from the TSO/E environment (as a command, through a CALL command, or from a CLIST or REXX EXEC).

Example 19 Format of BPXBATCH invocation

```
EXEC PGM=BPXBATCH,PARM='SH|PGM program_name'
```

BPXBATCH accepts one parameter string as input, the combination of SH|PGM and program_name. At least one blank character must separate the parts of the parameter string. The total length of the parameter string in a JCL is up to 100 characters. If you need more than 100 characters, parameters to BPXBATCH can also be supplied through the STDPARM DD up to a limit of 65 536 characters. When the STDPARM DD is allocated BPXBATCH will use the data found in the z/OS UNIX file or MVS data set associated with this DD rather than what is found on the parameter string or in the STDIN DD. An informational message BPXM079I will be displayed indicating that this is occurring, as a warning to the user. The STDPARM DD will allow either a z/OS UNIX file, or a MVS SYSIN, PDS or PDSE member or a sequential data set.

- SH|PGM

  Specifies whether BPXBATCH is to run a shell script or command or a z/OS C executable file located in an z/OS UNIX file.

- SH

  Specifies that the shell designated in your TSO/E user ID's security product profile is to be started and is to run shell commands or scripts provided from stdin or the specified program_name.

- PGM

  Specifies that the program identified by the program_name parameter is invoked directly from BPXBATCH. This is done either through a spawn or a fork and exec. BPXBATCH creates a process for the program to run in and then calls the program. If you specify PGM, you must also specify program_name.

.BPXBATCH has logic in it to detect when it is running from a batch job. By default, BPXBATCH sets up the stdin, stdout, and stderr standard streams (files) and then calls the exec callable service to run the requested program. The exec service ends the current job step and creates a new job step to run the target program. Therefore, the target program does not run in the same job step as the BPXBATCH program; it runs in the new job step created by the exec service.

For BPXBATCH to use the exec service to run the target program, all of the following facts must be true:

- BPXBATCH is the only program running on the job step task level.
- The _BPX_BATCH_SPAWN=YES environment variable is not specified.
- The STDOUT and STDERR ddnames are not allocated as MVS data sets.

If any of these conditions is not true, then the target program runs either in the same job step as the BPXBATCH program or in a WLM initiator in the OMVS subsys category. The determination of where to run the target program depends on the environment variable settings specified in the STDENV file and on the attributes of the target program.

Running REXX execs using BPXBATCH

You can run a REXX program from the z/OS shells. The REXX program runs as a separate process. It does not run in a TSO/E address space. You cannot use TSO/E commands in the REXX program.

A REXX program that is invoked from a z/OS shell or from a program must be a text file or a compiled REXX program that resides in the hierarchical file system (HFS). It must have read and execute access permissions. Each line in the text file must be terminated by a newline character and must not exceed 2048 characters. Lines are passed to the REXX interpreter as they are. Sequence numbers are not supported; if you are using the ISPF editor to create the REXX program, be sure to set NUMBER OFF.

Example 20 Running a REXX execs with BPXBATCH utility

```
File  Edit  Edit_Settings  Menu  Utilities  Compilers  Test  Help
--------------------------------------------------------------------------------
VIEW       REDBOOK.TEST.JCL(ALLSAMP) - 01.06               Columns 00001 00072
Command ===>                                                  Scroll ===> CSR
****** ***************************** Top of Data ******************************
000001 //REDBOOK@ JOB 'SG24-7419',MSGCLASS=A,CLASS=A,NOTIFY=REDBOOK
000002 //BPXBATCH EXEC PGM=BPXBATCH,PARM='SH @SYSCMD ''CMD(D SMF) DELAY(5)'''
000003 //SYSEXEC  DD  DSN=WAT.TEST.REXX,DISP=(SHR)
000004 //SYSTSPRT DD  SYSOUT=A
000005 //STDOUT   DD  SYSOUT=A,DCB=(RECFM=VB,LRECL=1024,BLKSIZE=0)
000006 //STDERR   DD  SYSOUT=A,DCB=(RECFM=VB,LRECL=1024,BLKSIZE=0)
****** **************************** Bottom of Data ****************************
```

Note: The parameters of the REXX exec must be enclosed by two apostrophes. In the previous example, CMD(D SMF) DELAY(5) are the parameters that we send to the REXX exec @SYSCMD.

telnet and rlogin connections

To access z/OS UNIX System Services interactively, you can log in into your user account using the **rlogin** or **telnet** interface. **telnet** and **rlogin** are similar except **rlogin** supports access from trusted hosts without requiring a password (thus, security people like this option less than **telnet**).

Most platforms (including Microsoft® Windows®) include a **telnet** command or interface. On the z/OS side, **telnet** support comes with the z/OS Communications Server. It uses an inetd daemon, which must be active and set up to recognize and receive the incoming Telnet requests. The z/OS system provides asynchronous terminal support for the z/OS UNIX shell. This is different from the 3270-terminal support provided by the TSO/E OMVS command.

Example 21 Issuing @SYSCMD from a telnet connection

```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\RESIDENT>telnet wtsc70oe.itso.ibm.com

EZYTE27I login: REDBOOK
EZYTE28I REDBOOK Password:
IBM
Licensed Material - Property of IBM
5694-A01 (C) Copyright IBM Corp. 1993, 2007
(C) Copyright Mortice Kern Systems, Inc., 1985, 1996.
(C) Copyright Software Development Group, University of Waterloo, 1989.

All Rights Reserved.

U.S. Government users - RESTRICTED RIGHTS - Use, Duplication, or
Disclosure restricted by GSA-ADP schedule contract with IBM Corp.

IBM is a registered trademark of the IBM Corp.

REDBOOK @ SC70:/u/REDBOOK>@SYSCMD 'CMD(D SMF) DELAY(5)'
@SYSCMD operands  : CMD(D SMF) DELAY(5)
SDSF HCE status   : established RC=00
----------------------------------------------------------------------------------------------------
ISFEXEC options   : (VERBOSE)
Original command  : /D SMF

SDSF short message: COMMAND ISSUED
SDSF long  message: ISF754I Command 'SET CONSOLE' generated from associated variable ISFCONS.
SDSF long  message: ISF754I Command 'SET DELAY 5' generated from associated variable ISFDELAY.

SDSF ULOG messages:
SC70      2007121  14:18:49.67            ISF031I CONSOLE REDBOOK ACTIVATED
```

```
SC70      2007121  14:18:49.67           -D SMF
SC70      2007121  14:18:49.68            IEE974I 14.18.49 SMF DATA SETS 019
                                                   NAME                VOLSER SIZE(BLKS) %FULL
                                                 P-SYS1.SC70.MAN1      SBOXD5      1500    75
                                                 S-SYS1.SC70.MAN2      SBOXD5      1500    38
DUMP
                                                 S-SYS1.SC70.MAN3      SBOXD5      1500     0

Command result    : RC=0 System command issued, response received from the EMCS console.
----------------------------------------------------------------------------------------------------
SDSF HCE status   : revoked RC=00
REDBOOK @ SC70:/u/REDBOOK>
```

Note: In the parameters to the REXX exec @SYSCMD must be enclosed in apostrophes (‘).

SSH connections

OpenSSH is a suite of network connectivity tools that provide secure encrypted communications between two untrusted hosts over an insecure network. These tools provide shell functions where network traffic is encrypted and authenticated. OpenSSH is based on client and server architecture. It supports public key and private key pairs for authentication and encryption channels to ensure secure network connections and host based authentication.

1

Issuing a system command

This chapter describes a technique for issuing a system command using IBM z/OS System Display and Search Facility (SDSF) support for the REXX programming language. The technique is to enter a system command with the ISFEXEC host command.

This chapter discusses the command environment, the things to consider, the areas to customize, the command operands and the command output. It also provides a sample REXX executable (referred to in the remainder of this chapter as *REXX exec*) and includes some scenarios to exploit this interface.

1.1 Command environment

The *slash command* is an SDSF command that includes a slash character (/) followed by a system command. The system command can be an MVS command or a subsystem command (for example, /D T). When a user runs SDSF in an interactive environment, the slash command can be entered at the command line of any SDSF panels. For more details about the slash command, access the online REXX for SDSF help tutorial (enter `REXXHELP` while in SDSF).

SDSF supports the slash command with the ISFEXEC host command. This host command runs under an SDSF host environment. For more details about setting up the SDSF host environment, access the online REXX for SDSF help tutorial.

1.2 Considerations when issuing a system command in the host command environment

SDSF uses a console to issue a system command, which allows an Extended Multiple Console Support (EMCS) console to be shared only if the console was activated by SDSF in the same address space. If the EMCS console was activated by another address space or by another application, SDSF will fail to establish the EMCS console and will issue the system command with an internal console. In both cases, the command responses will not be returned by the console.

So, when issuing a system command in the host command environment, consider the items that we discuss in this section.

1.2.1 Console name

If the ISFCONS variable has a value, SDSF issues a SET CONSOLE command with it and echoes the SET command response in the ISFMSG2 stem variable. If the ISFCONS variable is not assigned, SDSF determines the EMCS console name in the same way as it does when the user enters the slash command in an interactive mode. Access the online REXX for SDSF help tutorial for more details about the SET CONSOLE command.

1.2.2 Console type

SDSF uses one of the following consoles to issue a system command:

- *Internal* console: An internal MSC console with the console ID of 0.
- *Primary* EMCS console: An EMCS console that is activated by the current user.
- *Shared* EMCS console: An EMCS console that is already activated by another user.

SDSF determines the console type in the same way that it does when a user enters the system command in an interactive mode. Access the online REXX for SDSF help tutorial for more details about how SDSF determines the console type.

1.2.3 Command authority

SDSF determines the user command authority in the same way that it does when a user issues the system command in an interactive mode. For more information on how SDSF checks system command authority, refer to the *z/OS SDSF Operation and Customization,* SG22-7670.

1.2.4 Delay time limit

If the ISFDELAY variable has a value, SDSF issues a SET DELAY command with it and echoes the SET command response in the ISFMSG2 stem variable. If the ISFDELAY variable is not assigned, SDSF determines the delay time limit in the same way that it does when the user enters the slash command in an interactive mode. Access the online REXX for SDSF help tutorial for more details about the SET DELAY command.

1.3 Customization

In an SDSF host environment, SDSF issues a system command when:

- The ISFEXEC host command is invoked with a slash command
- The ISFACT host command is invoked to issue an action character or to modify a column which generates a system command

There are two variables that control function trace:

`ISFCONS` Specifies the EMCS console name that is used to issue the system command.

`ISFDELAY` Specifies the command response delay time limit for the system command.

1.4 ISFEXEC operands

As stated earlier, the ISFEXEC API command is used to request that SDSF execute commands such as panel display commands, SDSF information commands, and MVS system commands, also known as slash (/) commands. In this section, we look a little deeper at the ISFEXEC host command and how it is used with the SDSF MVS system commands as well as the optional parameters for the slash command. For more information about the options for panel display commands or SDSF information commands, see *z/OS V1R9.0 SDSF Operation and Customization*, SA22-7670.

1.4.1 System command

The system command can be up to 126 characters. To preserve lowercase and special characters in the command text, single quotation marks are required to enclose it to make sure that the quotation marks are passed to SDSF and are not removed by REXX.

If the system command includes any single quotation mark, each single quotation mark requires two single quotation marks to represent it. For example, you enter the following command in an interactive environment:

```
/DUMP COMM='test1'
```

Then, the equivalent of the same command in the host environment is:

```
address SDSF "ISFEXEC '/DUMP COMM=''test1'''"
```

The two single quotation marks are counted as two characters of the 126 characters limit. Access the online REXX for SDSF help tutorial for the syntax of the slash command.

1.4.2 Other optional parameters

> **Note:** These optional parameters are not valid if ISFEXEC if invoked to issue other SDSF commands.

There are two optional parameters for the slash command:

`INTERNAL`
: Specifies that SDSF should issue an INTERNAL slash command (**I/**) instead of the slash command (**/**). Access the online REXX for SDSF help tutorial for more details about the **I/** command.

`WAIT`
: Specifies that SDSF should issue the WAIT slash command (**W/**) instead of the slash command (**/**). Access the online REXX for SDSF help tutorial for more details about the **W/** command.

1.5 Command output

When the slash command is issued with an internal console or a shared EMCS console, the console does not return any command responses to SDSF. The caller can find the command responses in the system log. To avoid this situation, specify a unique EMCS console name in the ISFCONS variable.

When the slash command is issued with a primary EMCS console, the console returns the command responses to SDSF. The ISFEXEC host command puts values in the following variables:

`ISFULOG`
: A stem variable that includes both the system command echo and the command responses that are returned by the console.

`ISFMSG`
: A variable that includes the SDSF short message.

`ISFMSG2`
: A stem variable that includes the SDSF messages (which include the SDSF long messages when there is an error).

1.6 REXX for SDSF system command executable samples

The @SYSCMD REXX exec issues a system command for the caller. Optionally, it can look for the expected message text in the command responses, reply to the first Reply to the Write to Operator Reply (WTOR)—if there is one, and look for the expected message text in the REPLY command responses.

This section includes some scenarios on how to use this REXX exec:

- Scenario 1 - Use the system-determined EMCS console
- Scenario 2 - Use an internal console
- Scenario 3 - Use a specific EMCS console
- Scenario 4 - Request for the initial command response
- Scenario 5 - Request for all command responses
- Scenario 6 - Confirm the execution of the system command
- Scenario 7 - Query for a started task status
- Scenario 8 - Query for a device status
- Scenario 9 - Reply to the system command generated WTOR
- Scenario 10 - Confirm the execution of the system command and reply to its WTOR
- Scenario 11 - Confirm the execution of the system command and the reply to the WTOR
- Scenario 12 - Suppress all outputs

You can find information about the program source in the compressed file, which is described in Appendix B, “Additional material” on page 305. Alternatively, you can copy all the code in the examples in this section into a single member called *@SYSCMD*.

1.6.1 Sample REXX exec - @SYSCMD

The mainline of the @SYSCMD REXX exec in Example 1-1 invokes the following subroutines:

1. The **`parse_arguments`** subroutine in Example 1-2 on page 44 gets all the keyword parameters.
2. The **`host_environment`** subroutine in Example 1-5 on page 49 sets up an SDSF host environment.
3. The **`find_emcs_console`** subroutine in Example 1-6 on page 50 finds a primary EMCS console.
4. The **`issue_command`** subroutine in Example 1-7 on page 51 issues the system command in the COMMAND keyword parameter.

5. If the console is a primary EMCS console, the **find_message** subroutine in Example 1-8 on page 52 looks for the expected message in the MESSAGE keyword parameter from the command responses.
6. The **issue_reply_command** subroutine in Example 1-9 on page 53 issues an MVS REPLY command with the information in the REPLY keyword parameter.
7. If the console is a primary EMCS console, the **find_message** subroutine in Example 1-8 on page 52 looks for the expected message in the REPLYMSG keyword parameter from the REPLY command responses.

Example 1-1 @SYSCMD mainline

```
mainline:

  parse arg arguments

  if parse_arguments(arguments)         <> 0 then call exit_routine(99)
  if host_environment('establish')      <> 0 then call exit_routine(28)
  if find_emcs_console(retry_count)     <> 0 then call exit_routine(24)
  if issue_command(system_command)      <> 0 then call exit_routine(24)
  if pos('SHARED',ISFULOG.1) > 0 | ,
     pos('FAILED',ISFULOG.1) > 0 | ,
     internal_opt <> ''                      then call exit_routine(4)
  if find_message(command_response)     <> 0 then call exit_routine(8)
  if issue_reply_command(reply_command) <> 0 then
                                             call exit_routine(rcode)
                                           /* rcode can be: 0,12,24 */
  if pos('SHARED',ISFULOG.1) > 0 | ,
     pos('FAILED',ISFULOG.1) > 0 | ,
     internal_opt <> ''                      then call exit_routine(16)
  if find_message(reply_response)        = 8 then call exit_routine(20)
  call exit_routine(0)
```

The parse_arguments subroutine

The **parse_arguments** subroutine in Example 1-2 first invokes the **parse_parms** subroutine in Example 1-3 on page 46 to extract all keyword parameter names (in the KEYWORD stem variable) and their corresponding values (in the VALUE stem variable). Then it assigns a value to every keyword parameter of the REXX exec.

For the COMMAND and the REPLY parameters, it invokes the **handle_single_quote** subroutine in Example 1-4 on page 48 to double every single quotation mark found, which meets the requirement of the ISFEXEC **host** command.

If there is an error, the subroutine returns a code of 99. Otherwise, it returns a code of 0.

Example 1-2 The parse_arguments subroutine

```
parse_arguments:

  parse arg parms

if parse_parms(parms) > 0 then
     rcode = 99
  else do

     /* successfully parsed all parameters */
     system_command   = ''
     ISFCONS          = ''
     ISFDELAY         = ''
     internal_opt     = ''
     command_response = ''
     quiet_opt        = 'N'
     reply_command    = ''
     reply_response   = ''
     retry_count      = 0
     wait_opt         = ''
     rcode            = 0   /* default return code */

     do i = 1 to operand_ix
        keyword.i = translate(keyword.i)
        select
           when (keyword.i = 'COMMAND') | (keyword.i = 'CMD') then do
              system_command = handle_single_quote(value.i)
              if datatype(system_command) = "NUM" then do
                 rcode = 99
                 leave
              end /* if */
              end /* when */
           when (keyword.i = 'CONSOLE') | (keyword.i = 'CONS') then
              ISFCONS = value.i
           when (keyword.i = 'DELAY')   | (keyword.i = 'DLY') then
              ISFDELAY = value.i
           when (keyword.i = 'INTERNAL')| (keyword.i = 'INT') then
              if left(translate(value.i),1) = 'Y' then
                 internal_opt = 'INTERNAL'
           when (keyword.i = 'MESSAGE') | (keyword.i = 'MSG') then
              command_response = translate(value.i)
```

```
            when (keyword.i = 'QUIET')   | (keyword.i = 'Q') then
               if left(translate(value.i),1) = 'Y' then
                  quiet_opt = 'Y'
            when (keyword.i = 'REPLY')   | (keyword.i = 'RPY') then do
               reply_command = handle_single_quote(value.i)
               if datatype(reply_command) = "NUM" then do
                  rcode = 99
                  leave
               end /* if */
               end /* when */
            when (keyword.i = 'REPLYMSG') | (keyword.i = 'RMSG') then
               reply_response = translate(value.i)
            when (keyword.i = 'RETRY')   | (keyword.i = 'TRY') then
               if datatype(value.i) = "NUM" then
                  retry_count = value.i
            when (keyword.i = 'WAIT')    | (keyword.i = 'W') then
               if left(translate(value.i),1) = 'Y' then
                  wait_opt = 'WAIT'
            otherwise do
               say '***** Error - Unknown parameter' keyword.i || ,
                   ', RC=99'
               rcode = 99
               leave
            end /* otherwise */
         end /* select */

      end /* do loop */

   end /* else */

   if quiet_opt = 'N' then say '@SYSCMD operands  :' parms

   return rcode
```

The parse_parm subroutine

The **parse_parm** subroutine in Example 1-3 takes the character string in the PARM1 input parameter and parses it into two stem variables: KEYWORD and VALUE. The KEYWORD stem variable contains the names of the keyword parameters. The VALUE stem variable contains the values of the corresponding keyword parameter.

If there is an error, the subroutine returns a code of 99. Otherwise, it returns a code of 0.

Example 1-3 The parse_parms subroutine

```
parse_parms:

  parse arg parm1

  /*************************************************************/
  /* This routine takes the input parameter string and parse it. */
  /* The keyword parameter is saved in stem 'keyword' and its    */
  /* content is saved in stem 'value'.                           */
  /* For example,                                                */
  /*   parm1       = DLY(3) CMD("D SMF,O")                       */
  /* variables:                                                  */
  /*   new_parm1   = DLY(3) CMD("1")                             */
  /*   save_area.1 = D SMF,O                                     */
  /*   keyword.1   = DLY                                         */
  /*   value.1     = 3                                           */
  /*   keyword.2   = CMD                                         */
  /*   value.2     = D SMF O                                     */
  /*************************************************************/

  /* Start by removing all strings enclosed in double quotes */
  new_parm1 = ""
  cursor    = 1
  save_ix   = 0

  do forever

     quote_start = pos('"',parm1,cursor)
     if quote_start = 0 then do
        /* No more quotes. Copy the rest to the new string */
        new_parm1 = new_parm1 || substr(parm1,cursor)
        leave
     end /* if */

     /* Starting quote column in quote_start. Find ending quote */
     quote_end = pos('"',parm1,quote_start+1)
     if quote_end = 0 then do
        say '***** Error - no matching double quote for quote in' ,
            'column' quote_start
        return  99
     end /* if */

     /* Move string in between quote_start and quote_end
        (omitting the quotes) to the next save area in stem
```

```
      'save_area' and replace the string with the save_area
      index number.                                                 */

   save_ix = save_ix + 1
   save_area.save_ix = substr(parm1, ,
                       quote_start+1,quote_end-quote_start-1)
   new_parm1 = new_parm1                                  || ,
               substr(parm1,cursor,quote_start-cursor+1) || ,
               save_ix                                    || ,
               '"'
   cursor = quote_end + 1

end /* do loop */

/***********************************************************/
/*   By now, for example:                                  */
/*   if parm1 has     : DLY(3) CMD("D SMF,O") CONS(ABC)    */
/*   new_parm1 has    : DLY(3) CMD("1") CONS(AC)           */
/*   save_area has    : D SMF,O                            */
/***********************************************************/

/* Parse the new string to put value in stem 'keyword' and stem
   'value'.  If 'value' has an index number, get its value from
   stem 'save_area'.                                               */

operand_ix = 0
do while new_parm1 <> ""

   parse var new_parm1 operand_name "(" operand_value ")" new_parm1
   new_parm1          = strip(new_parm1)
   operand_ix         = operand_ix + 1
   keyword.operand_ix = operand_name

   if left(operand_value,1) <> '"' then
      value.operand_ix = operand_value
   else do
      /* The value is a save_area index in double quotes. */
      parse var operand_value '"' save_ix '"'
      value.operand_ix = save_area.save_ix
   end /* else */

end /* do loop */

return 0
```

The handle_single_quote subroutine

The **handle_single_quote** subroutine in Example 1-4 scans the character string in the PARM2 input parameter and replaces every single quotation mark with two single quotation marks.

If there is a mismatched single quotation mark, the subroutine returns a code of 99. Otherwise, it returns the modified character string.

Example 1-4 The handle_single_quote subroutine

```
handle_single_quote:

  parse arg parm2

  new_parm2 = ""
  cursor    = 1
  rcode     = 0 /* default return code */

  do forever

     quote_start = pos("'",parm2,cursor)
     if quote_start = 0 then do
        /* No more quotes. Copy the rest to the new string */
        new_parm2 = new_parm2 || substr(parm2,cursor)
        leave
     end /* if */

     /* Starting quote column in quote_start. Find ending quote */
     quote_end = pos("'",parm2,quote_start+1)
     if quote_end = 0 then do
        say '***** Error - no matching single quote for quote in' ,
            'column' quote_start
        return 99
     end /* if */

     /* For every single quote found, add one more to it. */
     new_parm2 = new_parm2                                 || ,
        substr(parm2,cursor,quote_start-cursor+1)          || ,
        "'"                                                || ,
        substr(parm2,quote_start+1,quote_end-quote_start) || ,
        "'"
     cursor = quote_end + 1

  end /* do loop */

  return new_parm2
```

The host_environment subroutine

The **host_environment** subroutine in Example 1-5 sets up or revokes the SDSF host command environment (HCE) by invoking the ISFCALLS host command based on the ENVIRONMENT input parameter. It writes a message to report the status of the HCE.

If there is an error, the subroutine returns with the ISFCALLS return code. Otherwise, it returns a code of 0.

Example 1-5 The host_environment subroutine

```
host_environment:

  parse arg environment

  if translate(environment) = 'ESTABLISH' then
     rcode = ISFCALLS('ON')
  else
     rcode = ISFCALLS('OFF')

  select
     when (rcode = 0) then
        if quiet_opt = 'N' then
          if environment = 'establish' then
             say 'SDSF HCE status   : established RC=00'
          else do
             say copies('-',131)
             say 'SDSF HCE status   : revoked RC=00'
          end /* else */
     when (rcode = 1) then
        say 'SDSF HCE status   : not established RC=01'
     when (rcode = 2) then
        say 'SDSF HCE status   : not established RC=02'
     when (rcode = 3) then
        say 'SDSF HCE status   : delete failed RC=03'
     otherwise
        say 'SDSF HCE status   : failed, unrecognized RC=' rcode
  end /* select */

  return rcode
```

The find_emcs_console subroutine

When the console used is an internal console or a shared EMCS console, the console returns nothing to the REXX exec to perform further message check or reply to the WTOR.

The **find_emcs_console** subroutine in Example 1-6 first tests out the console in the ISFCONS variable with a system command (DISPLAY TIME). If the console is a not a primary EMCS console, the subroutine appends a number within the range of the **emcs_index** input parameter to form a new EMCS console name and try again, until either the **emcs_index** is reached or an primary EMCS console is found. Upon exit, the subroutine has the EMCS console name set in the ISFCONS variable.

If there is an error from the ISFEXEC host command, the subroutine returns a code of `24`. Otherwise, it returns a code of `0`.

Example 1-6 The find_emcs_console subroutine

```
find_emcs_console:

  parse arg emcs_index

  if emcs_index = 0 | internal_opt <> '' then
     rcode = 0
  else do

     /*set up customizable fields */
     test_cmd       = 'D T'
     saved_isfcons  = ''
     saved_isfdelay = ISFDELAY
     rcode          = 0         /* default return code */
     ISFDELAY       = ''

     do jx = 1 to emcs_index

        if issue_command(test_cmd) <> 0 then do
           /* ISFEXEC error */
           rcode = 24
           leave
        end  /* if */

        if (pos('SHARED',ISFULOG.1) = 0) & ,
           (pos('FAILED',ISFULOG.1) = 0) & ,
           (internal_opt = '') then
           /* primary EMCS console */
           leave
```

```
        else do
           /* shared EMCS console or internal console */
           if saved_isfcons = '' then
              saved_isfcons = word(ISFULOG.1,6)
           if length(saved_isfcons) < 8 then
              ISFCONS = saved_isfcons || jx
           else do
              say '***WARNING: original EMCS console' ,
                  saved_isfcons 'has 8 characters,' ,
                  'RETRY operand ignored'
              leave
           end /* else */
        end /* if */

     end /* do loop */

     ISFDELAY = saved_isfdelay

  end /* else */

  return rcode
```

The issue_command subroutine

The **issue_command** subroutine in Example 1-7 invokes the ISFEXEC host command to issue the system command in the **sys_cmd** input parameter. Based on the **quiet** keyword parameter, it writes out the content of the SDSF short message (in the ISFMSG variable), the SDSF messages (in the ISFMSG2 stem variable), and the messages returned by the console (in the ISFULOG stem variable, which consists of the command echo and the command responses).

It returns with the ISFEXEC command return code.

Example 1-7 The issue_command subroutine

```
issue_command:

  parse arg sys_cmd

  slash_cmd = "/" || sys_cmd
  options   = '(' || wait_opt internal_opt || ')'

  if quiet_opt = 'N' then do
     say copies('-',131)
     say 'ISFEXEC options   :' options
     if sys_cmd = test_cmd then
```

```
      say 'Test command      :' slash_cmd
   else
      say 'Original command  :' slash_cmd
end /* if */

/* issue SDSF host command */
address SDSF "ISFEXEC '"slash_cmd"' " options
rcode = rc

if quiet_opt = 'N' then do

   /* write SDSF short message */
   say ' '
   say 'SDSF short message:' ISFMSG

   /* write SDSF long messages */
   do ix = 1 to ISFMSG2.0
      say 'SDSF long  message:' ISFMSG2.ix
   end /* do loop */

   /* write command responses */
   say ' '
   say 'SDSF ULOG messages:'
   do ix = 1 to ISFULOG.0
      say ISFULOG.ix
   end /* do loop */

end /* if */

return rcode
```

The find_message subroutine

The **find_message** subroutine in Example 1-8 looks for the **response_msg** input parameter in the command responses (in the ISFULOG stem variable).

If the message ID is not found, the subroutine returns a code of 8. Otherwise, it returns a code of 0.

Example 1-8 The find_message subroutine

```
find_message:

  parse arg response_msg

  if response_msg = '' then
```

```
      rcode = 0
   else do
      rcode  = 8 /* default return code */
      do jx = 1 to ISFULOG.0
        if pos(response_msg,ISFULOG.jx) > 0 then do
           rcode  = 0
           leave
        end /* if */
      end /* do loop */
   end /* else */

   return rcode
```

The issue_reply_command subroutine

The **issue_reply_command** subroutine in Example 1-9 looks for the outstanding reply message ID in the command responses (in the ISFULOG stem variable), builds an MVS REPLY command with the REPLY_CMD input parameter and invokes the **issue_command** subroutine in Example 1-7 on page 51 to reply to the outstanding write to operator reply (WTOR).

If there is an error, the subroutine returns a code of 12. Otherwise, it returns a code of 0.

Example 1-9 The issue_reply_command subroutine

```
issue_reply_command:

   parse arg reply_cmd

   if reply_cmd = '' then
      rcode = 0
   else do
      rcode = 12 /* default return code */
      do lx = 1 to ISFULOG.0
         msg_id = word(ISFULOG.lx,4)
         if substr(msg_id,1,1) = '*' then do
            r_id = substr(msg_id,2)
            if datatype(r_id) = "NUM" then do
              rcode = issue_command('R' r_id||','||reply_cmd)
              leave
            end  /* if */
         end /* if */
      end /* do loop */
   end /* else */

   return rcode
```

The exit_routine subroutine

The **exit_routine** subroutine in Example 1-10 writes the REXX exec results based on the QUIET parameter. If the host environment was established earlier, it revokes the SDSF host environment before returning with the return code in the input parameter.

Example 1-10 The exit_routine subroutine

```
exit_routine:

  parse arg rcode

  if (quiet_opt = 'N') | (rcode > 0) then do

     say ' '
     select
        when (rcode = 0) then do
           say 'Command result    : RC=00 System command issued,' ,
               'response received from the EMCS console.'
           if command_response <> '' then
              say copies(' ',26) || 'Message' command_response ,
                  'found in the command responses.'
           if reply_command    <> '' then
              say copies(' ',26) || 'MVS REPLY command issued.'
           if reply_response   <> '' then
              say copies(' ',26) || 'Message' reply_response ,
                  'found in the REPLY response.'
           end /* when */
        when (rcode = 4) then do
           say 'Command result    : RC=04 System command issued,' ,
               'no response received from the EMCS console.'
           say copies(' ',26) || ,
               'Console used is an internal console or' ,
               'a shared EMCS console.'
           if command_response <> '' then
              say copies(' ',26) || ,
                  'Note: command responses not checked.'
           if reply_command    <> '' then
              say copies(' ',26) || 'Note: REPLY command not issued.'
           end /* when */
        when (rcode = 8) then do
           say 'Command result    : RC=08 System command issued,' ,
               'expected message' command_response 'not found.'
           if reply_command <> '' then
              say copies(' ',26) || 'Note: REPLY command not issued.'
```

```
         end /* when */
      when (rcode = 12) then
         say 'Command result    : RC=12 System command issued,' ,
             'expected REPLY prompt not found.'
      when (rcode = 16) then do
         say 'Command result    : RC=16 REPLY command issued,' ,
             'no response received from the EMCS console.'
         say copies(' ',26) || ,
             'Console used is an internal console or' ,
             'a shared EMCS console.'
         if reply_response <> '' then
            say copies(' ',26) || 'Note: REPLY response not checked.'
         end /* when */
      when (rcode = 20) then
         say 'Command result    : RC=20 REPLY command issued,' ,
             'expected REPLY message' reply_response 'not found.'
      when (rcode = 24) then
         say 'Command result    : RC=24 ISFEXEC host command failed.'
      when (rcode = 28) then
         say 'Command result    : RC=28' ,
             'ISFCALLS host environment failed.'
      when (rcode = 99) then
         say 'Command result    : RC=99 Invalid keyword parameters.'
      otherwise
         say 'Unregonized return code.'
   end /* select */

end /* if */

saved_rc = rcode

if (rcode < 28) then call host_environment('revoke')

exit saved_rc
```

Keyword parameters

Keyword parameters used by the main line of the @SYSCMD REXX exec in Example 1-1 include:

`COMMAND()` or `CMD()` — Specifies the system command issued with the slash command. Enclose the command in double quotation marks if it includes any special characters (for example, single quotation marks, parentheses, or commas). There is no need to put two single quotation marks to represent one single quotation mark here. However, each single quotation mark is counted as two characters of the 126 characters limit. This parameter is *required*, and it has no default.

`CONSOLE()` or `CONS()`
: Specifies the EMCS console with which to issue the slash command. It has the same syntax as the online SET CONSOLE command parameter. The default is `null`, which means it is a system determined value.

`DELAY()` or `DLY()`
: Specifies the command response delay time limit in seconds. It has the same syntax as the online SET DELAY command parameter. The default is `null`, which means it is a system determined value.

`MESSAGE()` or `MSG()`
: Specifies a message text to look for in the command response. Enclose the message text in double quotation marks if it includes any special characters (for example, single quotation marks, parentheses, or commas). There is no need to put two single quotation marks to represent one single quotation mark here. This parameter has no default.

`RETRY()` or `TRY()`
: Specifies the number of attempts to look for a primary EMCS console. The default is 0.

`REPLY()` or `RPY()`
: Specifies the operands for the MVS REPLY command. Enclose the entire operand string in double quotation marks if it includes any special characters (for example, single quotation marks, parentheses, or commas). There is no need to put two single quotation marks to represent one single quotation mark here. However, each single quotation mark is counted as two characters of the 126 characters limit. This parameter has no default.

`REPLYMSG()` or `RMSG()`
: Specifies the message text to look for in the REPLY command response. Enclose the message text in double quotation marks if it includes any special characters (for example, single quotation marks, parentheses, or commas). There is no need to put two single quotation marks to represent one single quotation mark. This parameter has no default.

`WAIT()` or `W()`
: Specifies to issue the WAIT slash (**`W/`**) command instead of a slash command (`/`). If the value is not

	YES or Y, the REXX exec takes it as NO or N. The default is N.
INTERNAL() or INT()	Specifies to issue the INTERNAL slash command (**I/**) instead of a slash command (/). If the value is not YES or Y, the REXX exec takes it as NO or N. The default is N.
QUIET() or Q()	Specifies whether the output messages are suppressed. If the value is not YES or Y, the REXX exec takes it as NO or N. The default is N.

Customizable REXX variable

The following variable is customizable:

TEST_CMD	Specifies the MVS command issued when the REXX exec looks for a primary EMCS console. This variable is defined in the **find_emcs_console** subroutine and the default is the MVS DISPLAY TIME command.s

Command invocation

To invoke this REXX exec, you can put the keyword parameters in any order, for examples:

```
TSO @SYSCMD COMMAND() CONSOLE() DELAY() INTERNAL() MESSAGE() QUIET()
REPLY() REPLYMSG() RETRY() WAIT()
```

or

```
TSO @SYSCMD CMD() CONS() DLY() INT() MSG() Q() RPY() RMSG() TRY() W()
```

Return codes

Return codes include:

00 Command issued, expected response found, REPLY issued
04 Command issued, not checking for expected response, REPLY not issued
08 Command issued, expected response not found
12 Command issued, no expected REPLY prompt received
16 REPLY issued, no checking for expected REPLY response
20 REPLY issued, expected REPLY message not found
24 ISFEXEC host command failed
28 ISFCALLS host environment failed.
99 invalid keyword parameters

1.6.2 Scenario 1 - Use the system-determined EMCS console

This scenario explicitly omits the CONSOLE (or CONS) keyword parameter so that SDSF determines which EMCS console to use. The REXX exec returns with a return code of 0 when the console is a primary EMCS console, a return code of 16 when the ISFEXEC host command has an error, and a return code of 20 when it fails to establish the SDSF host environment.

Invoking this scenario

To invoke the scenario, use this command line:

```
TSO @SYSCMD CMD(D IPLINFO)
```

Output for this scenario

See Example 1-11 for the REXX exec output for this scenario. The REXX exec returned with a return code of 0.

The SDSF short message shows the result of the ISFEXEC host command. The SDSF messages show how SDSF determined the EMCS console name and the response delay time.

The ISF031I message, the system command echo, and the command responses are also in the output. The ISF031I message shows the EMCS console name and its status.

Example 1-11 Scenario 1 output

```
@SYSCMD operands  : CMD(D IPLINFO)
SDSF HCE status   : established RC=00
--------------------------------------------------------------------------------
ISFEXEC options   : ( )
Original command  : /D IPLINFO

SDSF short message: COMMAND ISSUED
SDSF long  message: ISF754I Command 'SET CONSOLE' generated from associated variable
ISFCONS.
SDSF long  message: ISF754I Command 'SET DELAY' generated from associated variable
ISFDELAY.

SDSF ULOG messages:
SC70   2007123 17:26:43.48       ISF031I CONSOLE JOE ACTIVATED
SC70   2007123 17:26:43.48        -D IPLINFO
SC70   2007123 17:26:43.49      IEE254I  17.26.43 IPLINFO DISPLAY 351
                                 SYSTEM IPLED AT 07.56.45 ON 05/03/2007
                                 RELEASE z/OS 01.09.00    LICENSE = z/OS
```

```
                        USED LOADS8 IN SYS0.IPLPARM ON C730
                          ARCHLVL = 2   MTLSHARE = N
                           IEASYM LIST = XX
                          IEASYS LIST = (R3,70) (OP)
                           IODF DEVICE C730
                         IPL DEVICE D21C VOLUME Z19RB1

Command result    : RC=00 System command issued, response received from the EMCS
console.
-------------------------------------------------------------------------------------
SDSF HCE status   : revoked RC=00
```

1.6.3 Scenario 2 - Use an internal console

This scenario specifies the INTERNAL (or INT) keyword parameter so that SDSF uses an internal console. The REXX exec return a return code of 0 when the console is a primary EMCS console, a return code of 4 when the console is not a primary EMCS console, a return code of 24 when the ISFEXEC host command has an error, and a return code of 28 when it fails to establish the SDSF host environment.

Invoking this scenario

To invoke the scenario, use this command line:

```
TSO @SYSCMD CMD(D IPLINFO) INT(Y)
```

Output for this scenario

See Example 1-12 for the REXX exec output. The REXX exec returned with a return code of 4. There was no command echo or the command responses in the output.

Example 1-12 Scenario 2 output

```
@SYSCMD operands  : CMD(D IPLINFO) INT(Y)
SDSF HCE status  : established RC=00
-------------------------------------------------------------------------------------
ISFEXEC options   : ( INTERNAL)
Original command  : /D IPLINFO

SDSF short message: COMMAND ISSUED
SDSF long  message: ISF754I Command 'SET CONSOLE' generated from associated variable
ISFCONS.
SDSF long  message: ISF754I Command 'SET DELAY' generated from associated variable
ISFDELAY.
```

```
SDSF ULOG messages:

Command result    : RC=04 System command issued, no response received from the EMCS
console.
                          Console used is an internal console or a shared EMCS
console.
-----------------------------------------------------------------------------------
SDSF HCE status   : revoked RC=00
```

1.6.4 Scenario 3 - Use a specific EMCS console

This scenario specifies the CONSOLE (or CONS) keyword parameter so that SDSF will use the specified EMCS console.

The REXX exec returns with a return code of 0 when the console is a primary EMCS console, a return code of 4 when the console is not a primary EMCS console, a return code of 24 when the ISFEXEC host command has an error, and a return code of 28 when it fails to establish the SDSF host environment.

Invoking this scenario

To invoke the scenario, use this command line:

```
TSO @SYSCMD CMD(D IPLINFO) CONS(IBM)
```

Output for this scenario

See Example 1-13 for the REXX exec output. The REXX exec returned with a return code of 0.

The SDSF short message shows the result of the ISFEXEC host command. The SDSF messages show how SDSF determined the EMCS console name and the response delay time.

The ISF031I message, the system command echo and the command responses are also in the output. The ISF031I message shows the EMCS console name and its status.

Example 1-13 Scenario 3 output

```
@SYSCMD operands  : CMD(D IPLINFO) CONS(IBM)
SDSF HCE status   : established RC=00
-----------------------------------------------------------------------------------
ISFEXEC options   : ( )
Original command  : /D IPLINFO
```

```
SDSF short message: COMMAND ISSUED
SDSF long  message: ISF754I Command 'SET CONSOLE IBM' generated from associated
variable ISFCONS.
SDSF long  message: ISF754I Command 'SET DELAY' generated from associated variable
ISFDELAY.

SDSF ULOG messages:
SC70   2007123 17:26:44.77       ISF031I CONSOLE IBM ACTIVATED
SC70    2007123  17:26:44.77       -D IPLINFO
SC70   2007123 17:26:44.77      IEE254I  17.26.44 IPLINFO DISPLAY 364
                            SYSTEM IPLED AT 07.56.45 ON 05/03/2007
                            RELEASE z/OS 01.09.00    LICENSE = z/OS
                            USED LOADS8 IN SYS0.IPLPARM ON C730
                              ARCHLVL = 2   MTLSHARE = N
                               IEASYM LIST = XX
                              IEASYS LIST = (R3,70) (OP)
                               IODF DEVICE C730
                             IPL DEVICE D21C VOLUME Z19RB1

Command result    : RC=00 System command issued, response received from the EMCS
console.
------------------------------------------------------------------------------------
SDSF HCE status   : revoked RC=00
```

1.6.5 Scenario 4 - Request for the initial command response

This scenario is a modification of “Scenario 3 - Use a specific EMCS console” on page 60. It specifies the RETRY (or TRY) keyword parameter so that SDSF uses a different EMCS console when the specified EMCS console is not a primary EMCS console. The RETRY keyword parameter increases the chance of using a primary EMCS console and minimizes the chance of no command responses returned.

The REXX exec returns with a return code of 0 when the console is a primary EMCS console, a return code of 4 when the console is not a primary EMCS console, a return code of 24 when the ISFEXEC host command has an error, and a return code of 28 when it fails to establish the SDSF host environment.

Invoking this scenario

To invoke the scenario, use this command line:

```
TSO @SYSCMD CMD(D IPLINFO) CONS(IBM) TRY(2)
```

Output for this scenario

In Example 1-14, three system commands are issued. The REXX exec first issued the test system command **/D T** with the EMCS console and received a return code of 4, which means that no command response was returned. It then issued the test system command **/D T** a second time with EMCS console **IBM1** and received a return code of 0. Because a primary EMCS console was found, it issued the D IPLINFO command with the EMCS console IBM1.

Example 1-14 Scenario 4 output

```
@SYSCMD operands  : CMD(D IPLINFO) CONS(IBM) TRY(2)
SDSF HCE status   : established RC=00
---------------------------------------------------------------------------------
ISFEXEC options   : ( )
Test command      : /D T

SDSF short message: COMMAND ISSUED
SDSF long  message: ISF754I Command 'SET CONSOLE IBM' generated from associated
variable ISFCONS.
SDSF long  message: ISF754I Command 'SET DELAY' generated from associated variable
ISFDELAY.

SDSF ULOG messages:
SC70      2007123  18:31:39.39             ISF032I CONSOLE IBM ACTIVATE FAILED,
RETURN CODE 0004, REASON CODE 0000
---------------------------------------------------------------------------------
ISFEXEC options   : ( )
Test command      : /D T

SDSF short message: COMMAND ISSUED
SDSF long  message: ISF754I Command 'SET CONSOLE IBM1' generated from associated
variable ISFCONS.
SDSF long  message: ISF754I Command 'SET DELAY' generated from associated variable
ISFDELAY.

SDSF ULOG messages:
SC70   2007123 18:31:39.46       ISF031I CONSOLE IBM1 ACTIVATED
SC70    2007123 18:31:39.46       -D T
SC70     2007123  18:31:39.46  JOB02011   IEE136I LOCAL: TIME=18.31.39 DATE=2007.123
UTC: TIME=22.31.39 DATE=2007.123
---------------------------------------------------------------------------------
ISFEXEC options   : ( )
Original command  : /D IPLINFO

SDSF short message: COMMAND ISSUED
```

```
SDSF long  message: ISF754I Command 'SET CONSOLE IBM1' generated from associated
variable ISFCONS.
SDSF long  message: ISF754I Command 'SET DELAY' generated from associated variable
ISFDELAY.

SDSF ULOG messages:
SC70   2007123 18:31:40.59      ISF031I CONSOLE IBM1 ACTIVATED
SC70    2007123 18:31:40.59       -D IPLINFO
SC70   2007123 18:31:40.59      IEE254I 18.31.40 IPLINFO DISPLAY 814
                              SYSTEM IPLED AT 07.56.45 ON 05/03/2007
                              RELEASE z/OS 01.09.00   LICENSE = z/OS
                              USED LOADS8 IN SYS0.IPLPARM ON C730
                                ARCHLVL = 2  MTLSHARE = N
                                 IEASYM LIST = XX
                                IEASYS LIST = (R3,70) (OP)
                                 IODF DEVICE C730
                               IPL DEVICE D21C VOLUME Z19RB1

Command result    : RC=00 System command issued, response received from the EMCS
console.
-------------------------------------------------------------------------------------
SDSF HCE status   : revoked RC=00
```

1.6.6 Scenario 5 - Request for all command responses

Some system command takes longer to execute. This scenario specifies both the DELAY (or DLY) keyword parameter and the WAIT (or W) keyword parameter. The DELAY parameter is to set the response delay time. The WAIT parameter asks SDSF to wait the full delay interval before returning to the caller with the command responses. When you specify the two parameters together, this minimizes the chance of getting incomplete command responses.

As in "Scenario 4 - Request for the initial command response" on page 61, this scenario uses the RETRY parameter to minimize the chance of no command responses returned.

Because the system command has a comma, double quotation marks are required to enclose it.

The REXX exec returns with a return code of 0 when the console is a primary EMCS console, a return code of 4 when the console is not a primary EMCS console, a return code of 24 when the ISFEXEC host command has an error, and a return code of 28 when it fails to establish the SDSF host environment.

Invoking this scenario

To invoke the scenario, use this command line:

```
@SYSCMD CMD("F CATALOG,REPORT") CONS(IBM) TRY(2) DLY(0)
```

or

```
@SYSCMD CMD("F CATALOG,REPORT") CONS(IBM) TRY(2) DLY(5) W(Y)
```

Output for this scenario

Due to low workload volume on our test system, Example 1-15 uses a DELAY of 0 to show the effect of the DELAY parameter. When DELAY is 0, only the command echo is returned.

Example 1-15 Scenario 5 output with DELAY(0) and no WAIT

```
@SYSCMD operands : CMD(F CATALOG,REPORT) CONS(IBM) TRY(2) DLY(0)
SDSF HCE status  : established RC=00
-------------------------------------------------------------------------------------
ISFEXEC options  : ( )
Test command     : /D T

SDSF short message: COMMAND ISSUED
SDSF long  message: ISF754I Command 'SET CONSOLE IBM' generated from associated
variable ISFCONS.
SDSF long  message: ISF754I Command 'SET DELAY' generated from associated variable
ISFDELAY.

SDSF ULOG messages:
SC70   2007123 17:26:48.25       ISF031I CONSOLE IBM ACTIVATED
SC70    2007123 17:26:48.25       -D T
SC70      2007123  17:26:48.25  JOB02002   IEE136I LOCAL: TIME=17.26.48 DATE=2007.123
UTC: TIME=21.26.48 DATE=2007.123
-------------------------------------------------------------------------------------
ISFEXEC options  : ( )
Original command : /F CATALOG,REPORT

SDSF short message: COMMAND ISSUED
SDSF long  message: ISF754I Command 'SET CONSOLE IBM' generated from associated
variable ISFCONS.
SDSF long  message: ISF754I Command 'SET DELAY 0' generated from associated variable
ISFDELAY.

SDSF ULOG messages:
SC70   2007123 17:26:49.35       ISF031I CONSOLE IBM ACTIVATED
SC70    2007123 17:26:49.36       -F CATALOG,REPORT
```

```
Command result    : RC=00 System command issued, response received from the EMCS
console.
-------------------------------------------------------------------------------------
SDSF HCE status   : revoked RC=00
```

Example 1-16 for the output when the DELAY parameter has 5. It shows that the REXX exec can receive multiple messages as well as multiple-line messages.

Example 1-16 Scenario 5 output with DELAY(5) and WAIT

```
@SYSCMD operands : CMD("F CATALOG,REPORT") CONS(IBM) TRY(2) DLY(5) W(Y)
SDSF HCE status  : established RC=00
-------------------------------------------------------------------------------------
ISFEXEC options  : (WAIT )
Test command     : /D T

SDSF short message: COMMAND ISSUED
SDSF long  message: ISF754I Command 'SET CONSOLE IBM' generated from associated
variable ISFCONS.
SDSF long  message: ISF754I Command 'SET DELAY' generated from associated variable
ISFDELAY.

SDSF ULOG messages:
SC70   2007123 17:26:49.55       ISF031I CONSOLE IBM ACTIVATED
SC70    2007123 17:26:49.55       -D T
SC70      2007123  17:26:49.55  JOB02002   IEE136I LOCAL: TIME=17.26.49 DATE=2007.123
UTC: TIME=21.26.49 DATE=2007.123
-------------------------------------------------------------------------------------
ISFEXEC options  : (WAIT )
Original command : /F CATALOG,REPORT

SDSF short message: COMMAND ISSUED
SDSF long  message: ISF754I Command 'SET CONSOLE IBM' generated from associated
variable ISFCONS.
SDSF long  message: ISF754I Command 'SET DELAY 5' generated from associated variable
ISFDELAY.

SDSF ULOG messages:
SC70    2007123  17:26:51.66        ISF031I CONSOLE IBM ACTIVATED
SC70    2007123  17:26:51.66        -F CATALOG,REPORT
SC70      2007123   17:26:51.66            IEC351I CATALOG ADDRESS SPACE MODIFY
COMMAND ACTIVE
SC70    2007123  17:26:51.66        IEC359I CATALOG REPORT OUTPUT
```

```
*CAS***********************************************************
                              *  CATALOG COMPONENT LEVEL   = HDZ1190
*
                              *  CATALOG ADDRESS SPACE ASN = 0037
*
                              *  SERVICE TASK UPPER LIMIT  =  180
*
                              *  SERVICE TASK LOWER LIMIT  =   60
*
                              *  HIGHEST # SERVICE TASKS   =   19
*
                              *  CURRENT # SERVICE TASKS   =   19
*
                              *  MAXIMUM # OPEN CATALOGS   = 1,024
*
                              *  ALIAS TABLE AVAILABLE     = YES
*
                              *  ALIAS LEVELS SPECIFIED    = 1
*
                              *  SYS% TO SYS1 CONVERSION   = OFF
*
                              *  CAS MOTHER TASK          = 007FF5E8
*
                              *  CAS MODIFY TASK          = 0078EE48
*
                              *  CAS ANALYSIS TASK        = 0078E868
*
                              *  CAS ALLOCATION TASK      = 0078EC18
*
                              *  VOLCAT HI-LEVEL QUALIFIER = SYS1
*
                              *  NOTIFY EXTENT            =   80%
*
                                        *  DEFAULT VVDS SPACE        = ( 10, 10)
TRKS                  *
                                        *  ENABLED FEATURES          = DSNCHECK
DELFORCEWNG SYMREC     *
                                        *  ENABLED FEATURES          = UPDTFAIL
AUTOTUNING             *
                                        *  DISABLED FEATURES         = VVRCHECK
BCSCHECK               *
                              *  INTERCEPTS               = (NONE)
*
```

```
*CAS************************************************************
SC70      2007123  17:26:51.67            IEC352I CATALOG ADDRESS SPACE MODIFY
COMMAND COMPLETED

Command result    : RC=00 System command issued, response received from the EMCS
console.
-------------------------------------------------------------------------------------
SDSF HCE status   : revoked RC=00
```

1.6.7 Scenario 6 - Confirm the execution of the system command

This scenario specifies the MESSAGE (or MSG) keyword parameter so that the REXX exec looks for the expected system message identifier (message ID) in the command responses.

Because the system command has a comma, double quotation marks are required to enclose it.

The REXX exec returns with a return code of 0 when the console is a primary EMCS console, a return code of 4 when the console is not a primary EMCS console, a return code of 8 when the message ID is found, a return code of 24 when the ISFEXEC host command has an error, and a return code of 28 when it fails to establish the SDSF host environment.

Invoking this scenario

To invoke the scenario, use this command line:

```
TSO @SYSCMD CMD("F LLA,REFRESH") MSG(CSV210I) TRY(1)
```

Output for this scenario

In Example 1-17, message ISF031I shows that the SDSF determined console is a primary EMCS console and so the command responses are returned by the console. Message CSV210I was found in the first line of the command responses.

Example 1-17 Scenario 6 output

```
@SYSCMD operands  : CMD("F LLA,REFRESH") MSG(CSV210I) TRY(1)
SDSF HCE status   : established RC=00
-------------------------------------------------------------------------------------
ISFEXEC options   : ( )
Test command      : /D T
```

```
SDSF short message: COMMAND ISSUED
SDSF long  message: ISF754I Command 'SET CONSOLE' generated from associated variable
ISFCONS.
SDSF long  message: ISF754I Command 'SET DELAY' generated from associated variable
ISFDELAY.

SDSF ULOG messages:
SC70   2007123 17:27:01.84       ISF031I CONSOLE JOE ACTIVATED
SC70    2007123  17:27:01.84       -D T
SC70     2007123  17:27:01.84  JOB02002   IEE136I LOCAL: TIME=17.27.01 DATE=2007.123
UTC: TIME=21.27.01 DATE=2007.123
------------------------------------------------------------------------------------
ISFEXEC options  : ( )
Original command  : /F LLA,REFRESH

SDSF short message: NO RESPONSE RECEIVED
SDSF long  message: ISF754I Command 'SET CONSOLE' generated from associated variable
ISFCONS.
SDSF long  message: ISF754I Command 'SET DELAY' generated from associated variable
ISFDELAY.

SDSF ULOG messages:
SC70   2007123 17:27:02.94       ISF031I CONSOLE JOE ACTIVATED
SC70    2007123  17:27:02.94       -F LLA,REFRESH
SC70   2007123 17:27:03.99      CSV210I LIBRARY LOOKASIDE REFRESHED

Command result    : RC=00 System command issued, response received from the EMCS
console.
              Message CSV210I found in the command responses.
------------------------------------------------------------------------------------
SDSF HCE status   : revoked RC=00
```

1.6.8 Scenario 7 - Query for a started task status

This scenario is similar to "Scenario 6 - Confirm the execution of the system command" on page 67. This scenario specifies the started task status in the MESSAGE (or MSG) keyword parameter so that the REXX exec looks for the message text in the command responses.

Because both the system command and the message have parentheses, double quotation marks are required to enclose them.

The REXX exec returns with a return code of 0 when the console is a primary EMCS console, a return code of 4 when the console is not a primary EMCS console, a return code of 8 when the message text is found, a return code of 24 when the ISFEXEC host command has an error, and a return code of 28 when it fails to establish the SDSF host environment.

Invoking this scenario

To invoke the scenario, use this command line:

```
@SYSCMD CMD("$DS(CSF)") MSG("(EXECUTING/SC64)") TRY(1)
```

Output for this scenario

In Example 1-18, the message text (EXECUTING/SC64) was found in the second line of the command responses.

Example 1-18 Scenario 7 output

```
@SYSCMD operands : CMD("$DS(CSF)") MSG("(EXECUTING/SC64)") TRY(1)
SDSF HCE status  : established RC=00
--------------------------------------------------------------------------------------
ISFEXEC options  : ( )
Test command     : /D T

SDSF short message: COMMAND ISSUED
SDSF long  message: ISF754I Command 'SET CONSOLE' generated from associated variable
ISFCONS.
SDSF long  message: ISF754I Command 'SET DELAY' generated from associated variable
ISFDELAY.

SDSF ULOG messages:
SC70   2007123 17:27:05.21       ISF031I CONSOLE JOE ACTIVATED
SC70    2007123 17:27:05.21       -D T
SC70      2007123  17:27:05.21  JOB02002   IEE136I LOCAL: TIME=17.27.05 DATE=2007.123
UTC: TIME=21.27.05 DATE=2007.123
--------------------------------------------------------------------------------------
ISFEXEC options  : ( )
Original command : /$DS(CSF)

SDSF short message: COMMAND ISSUED
SDSF long  message: ISF754I Command 'SET CONSOLE' generated from associated variable
ISFCONS.
SDSF long  message: ISF754I Command 'SET DELAY' generated from associated variable
ISFDELAY.

SDSF ULOG messages:
SC70   2007123 17:27:06.33       ISF031I CONSOLE JOE ACTIVATED
SC70    2007123 17:27:06.33       -$DS(CSF)
```

```
SC70   2007123 17:27:06.33 STC29226  $HASP890 JOB(CSF)
                                            $HASP890 JOB(CSF)
STATUS=(EXECUTING/SC64),
                                            $HASP890
CLASS=STC,PRIORITY=15,
                                            $HASP890
SYSAFF=(SC64),HOLD=(NONE)
SC70   2007123 17:27:06.33 STC29523  $HASP890 JOB(CSF)
                                            $HASP890 JOB(CSF)
STATUS=(EXECUTING/SC63),
                                            $HASP890
CLASS=STC,PRIORITY=15,
                                            $HASP890
SYSAFF=(SC63),HOLD=(NONE)

Command result    : RC=00 System command issued, response received from the EMCS
console.
             Message (EXECUTING/SC64) found in the command responses.
-------------------------------------------------------------------------------------
SDSF HCE status   : revoked RC=00
```

1.6.9 Scenario 8 - Query for a device status

This scenario is similar to "Scenario 6 - Confirm the execution of the system command" on page 67 as well as "Scenario 7 - Query for a started task status" on page 68. It specifies the device number in the MESSAGE (or MSG) keyword parameter so that the REXX exec looks for that device in the command responses.

Because the system command has commas, double quotation marks are required to enclose it.

The REXX exec returns with a return code of 0 when the console is a primary EMCS console, a return code of 4 when the console is not a primary EMCS console, a return code of 8 when the device number is found, a return code of 24 when the ISFEXEC host command has an error, and a return code of 28 when it fails to establish the SDSF host environment.

Invoking this scenario

To invoke the scenario, use this command line:

```
@SYSCMD CMD("D U,,OFFLINE") MSG(0062) TRY(1)
```

Output for this scenario

In Example 1-19, device 0062 was found in the fourth line of the command responses.

Example 1-19 Scenario 8 output

```
@SYSCMD operands : CMD("D U,,OFFLINE") MSG(0062) TRY(1)
SDSF HCE status  : established RC=00
-------------------------------------------------------------------------------------
ISFEXEC options  : ( )
Test command     : /D T

SDSF short message: COMMAND ISSUED
SDSF long  message: ISF754I Command 'SET CONSOLE' generated from associated variable
ISFCONS.
SDSF long  message: ISF754I Command 'SET DELAY' generated from associated variable
ISFDELAY.

SDSF ULOG messages:
SC70   2007123 17:27:07.55       ISF031I CONSOLE JOE ACTIVATED
SC70    2007123 17:27:07.55       -D T
SC70     2007123  17:27:07.55  JOB02002   IEE136I LOCAL: TIME=17.27.07 DATE=2007.123
UTC: TIME=21.27.07 DATE=2007.123
-------------------------------------------------------------------------------------
ISFEXEC options  : ( )
Original command : /D U,,OFFLINE

SDSF short message: COMMAND ISSUED
SDSF long  message: ISF754I Command 'SET CONSOLE' generated from associated variable
ISFCONS.
SDSF long  message: ISF754I Command 'SET DELAY' generated from associated variable
ISFDELAY.

SDSF ULOG messages:
SC70   2007123 17:27:08.66       ISF031I CONSOLE JOE ACTIVATED
SC70    2007123 17:27:08.66       -D U,,OFFLINE
SC70   2007123 17:27:08.67       IEE457I 17.27.08 UNIT STATUS 426
                                            UNIT TYPE UNIT TYPE UNIT TYPE UNIT TYPE
UNIT TYPE UNIT TYPE UNIT TYPE
                                            001A SWCH 0030 3286 0031 3286 0032 3286
0033 3286 0034 3286 0035 3286
                                            0061 SWCH 0062 SWCH 0063 SWCH 0064 SWCH
0090 SWCH 0091 SWCH 0130 3286
                            0131 3286 0132 3286
```

```
Command result    : RC=00 System command issued, response received from the EMCS
console.
             Message 0062 found in the command responses.
-----------------------------------------------------------------------------------
SDSF HCE status   : revoked RC=00
```

1.6.10 Scenario 9 - Reply to the system command generated WTOR

This scenario specifies the REPLY (or RPY) keyword parameter so that the REXX exec replies to the outstanding WTOR with it.

Because the system command has a parenthesis and the reply has a comma, double quotation marks are required to enclose them.

The REXX exec returns with a return code of 0 when it successfully issues the REPLY command, a return code of 4 when the console is not a primary EMCS console, a return code of 12 when there is no outstanding reply, a return code of 16 when the reply is not issued with a primary EMCS console, a return code of 24 when the ISFEXEC host command has an error, and a return code of 28 when it fails to establish the SDSF host environment.

Invoking this scenario

To invoke the scenario, use this command line:

```
TSO @SYSCMD CMD("DUMP COMM=(MVS1)") RPY("ASID=1,END") TRY(1)
```

Output for this scenario

In Example 1-20, three system commands are issued. Look for the keywords `test` and `original` in the output. The first command was the test DISPLAY TIME command issued to find a primary EMCS console. The second command was the original DUMP command, and the third command was the original REPLY command.

Example 1-20 Scenario 9 output

```
@SYSCMD operands : CMD("DUMP COMM=(MVS1)") RPY("ASID=1,END") TRY(1)
SDSF HCE status  : established RC=00
-----------------------------------------------------------------------------------
ISFEXEC options  : ( )
Test command     : /D T

SDSF short message: COMMAND ISSUED
SDSF long  message: ISF754I Command 'SET CONSOLE' generated from associated variable
ISFCONS.
```

```
SDSF long  message: ISF754I Command 'SET DELAY' generated from associated variable
ISFDELAY.

SDSF ULOG messages:
SC70   2007123 17:27:09.87       ISF031I CONSOLE JOE ACTIVATED
SC70    2007123  17:27:09.87       -D T
SC70      2007123  17:27:09.87  JOB02002   IEE136I LOCAL: TIME=17.27.09 DATE=2007.123
UTC: TIME=21.27.09 DATE=2007.123
-------------------------------------------------------------------------------------
ISFEXEC options  : ( )
Original command : /DUMP COMM=(MVS1)

SDSF short message: COMMAND ISSUED
SDSF long  message: ISF754I Command 'SET CONSOLE' generated from associated variable
ISFCONS.
SDSF long  message: ISF754I Command 'SET DELAY' generated from associated variable
ISFDELAY.

SDSF ULOG messages:
SC70   2007123 17:27:10.98       ISF031I CONSOLE JOE ACTIVATED
SC70    2007123  17:27:10.98       -DUMP COMM=(MVS1)
SC70      2007123  17:27:10.98            *085 IEE094D SPECIFY OPERAND(S) FOR DUMP
COMMAND
-------------------------------------------------------------------------------------
ISFEXEC options  : ( )
Original command : /R 085,ASID=1,END

SDSF short message: COMMAND ISSUED
SDSF long  message: ISF754I Command 'SET CONSOLE' generated from associated variable
ISFCONS.
SDSF long  message: ISF754I Command 'SET DELAY' generated from associated variable
ISFDELAY.

SDSF ULOG messages:
SC70   2007123 17:27:12.17       ISF031I CONSOLE JOE ACTIVATED
SC70    2007123  17:27:12.17       -R 085,ASID=1,END
SC70   2007123 17:27:12.18       IEE600I REPLY TO 085 IS;ASID=1,END

Command result    : RC=00 System command issued, response received from the EMCS
console.
                MVS REPLY command issued.
-------------------------------------------------------------------------------------
SDSF HCE status   : revoked RC=00
```

1.6.11 Scenario 10 - Confirm the execution of the system command and reply to its WTOR

This scenario is a combination of "Scenario 6 - Confirm the execution of the system command" on page 67 and "Scenario 9 - Reply to the system command generated WTOR" on page 72. It specifies both the MESSAGE (or MSG) keyword parameter and the REPLY (or RPY) keyword parameter. The REXX exec first looks for the message ID in the command responses and, if there is a match, it continues to look for the outstanding WTOR ID and replies to it.

Because the system command has a parenthesis and the reply has a comma, double quotation marks are required to enclose them.

The REXX exec returns with a return code of 0 when it successfully issues the REPLY command, a return code of 4 when the console is not a primary EMCS console, a return code of 8 when the expected message is not found in the command responses, a return code of 12 when there is no outstanding reply, a return code of 16 when the reply is not issued with a primary EMCS console, a return code of 24 when the ISFEXEC host command has an error, and a return code of 28 when it fails to establish the SDSF host environment.

Invoking this scenario

To invoke the scenario, use this command line:

```
TSO @SYSCMD CMD("DUMP COMM=(MVS1)") MSG(IEE094D) RPY("ASID=1,END")
TRY(1)
```

Output

In Example 1-21, the REXX exec found the `IEE094D` message on the first line. It then found the WTOR reply ID to be `086` and issued an MVS REPLY command to reply with `ASID=1,END`.

Example 1-21 Scenario 10 output

```
@SYSCMD operands  : CMD("DUMP COMM=(MVS1)") MSG(IEE094D) RPY("ASID=1,END")
TRY(1)
SDSF HCE status  : established RC=00
--------------------------------------------------------------------------------
ISFEXEC options  : ( )
Test command     : /D T

SDSF short message: COMMAND ISSUED
SDSF long  message: ISF754I Command 'SET CONSOLE' generated from associated variable
ISFCONS.
```

```
SDSF long  message: ISF754I Command 'SET DELAY' generated from associated variable
ISFDELAY.

SDSF ULOG messages:
SC70   2007123 17:27:14.30       ISF031I CONSOLE JOE ACTIVATED
SC70    2007123  17:27:14.30       -D T
SC70      2007123  17:27:14.30  JOB02002   IEE136I LOCAL: TIME=17.27.14 DATE=2007.123
UTC: TIME=21.27.14 DATE=2007.123
--------------------------------------------------------------------------------------
ISFEXEC options  : ( )
Original command  : /DUMP COMM=(MVS1)

SDSF short message: COMMAND ISSUED
SDSF long  message: ISF754I Command 'SET CONSOLE' generated from associated variable
ISFCONS.
SDSF long  message: ISF754I Command 'SET DELAY' generated from associated variable
ISFDELAY.

SDSF ULOG messages:
SC70   2007123 17:27:15.46       ISF031I CONSOLE JOE ACTIVATED
SC70    2007123  17:27:15.46       -DUMP COMM=(MVS1)
SC70      2007123  17:27:15.46            *086 IEE094D SPECIFY OPERAND(S) FOR DUMP
COMMAND
--------------------------------------------------------------------------------------
ISFEXEC options  : ( )
Original command  : /R 086,ASID=1,END

SDSF short message: COMMAND ISSUED
SDSF long  message: ISF754I Command 'SET CONSOLE' generated from associated variable
ISFCONS.
SDSF long  message: ISF754I Command 'SET DELAY' generated from associated variable
ISFDELAY.

SDSF ULOG messages:
SC70   2007123 17:27:16.73       ISF031I CONSOLE JOE ACTIVATED
SC70    2007123  17:27:16.73       -R 086,ASID=1,END
SC70   2007123 17:27:16.73       IEE600I REPLY TO 086 IS;ASID=1,END

Command result    : RC=00 System command issued, response received from the EMCS
console.
              Message IEE094D found in the command responses.
                MVS REPLY command issued.
--------------------------------------------------------------------------------------
SDSF HCE status   : revoked RC=00
```

1.6.12 Scenario 11 - Confirm the execution of the system command and the reply to the WTOR

This scenario is a modification of "Scenario 10 - Confirm the execution of the system command and reply to its WTOR" on page 74. It specifies the MESSAGE (or MSG) keyword parameter, the REPLY (or RPY) keyword, parameter as well as the REPLYMSG keyword parameter. The REXX exec first looks for the message ID in the command responses and, if there is a match, it continues to look for the outstanding WTOR ID and replies to it. It also looks for the message ID in the REPLY command responses.

Because the system command has a parenthesis and the reply has a comma, double quotation marks are required to enclose them.

The REXX exec returns with a return code of 0 when it successfully issues the REPLY command, a return code of 4 when the console is not a primary EMCS console, a return code of 8 when the expected message is not found in the command responses, a return code of 12 when there is no outstanding reply, a return code of 16 when the reply is not issued with a primary EMCS console, a return code of 20 when the expected message is not found in the REPLY command responses, a return code of 24 when the ISFEXEC host command has an error, and a return code of 28 when it fails to establish the SDSF host environment.

Invoking this scenario

To invoke the scenario, use this command line:

```
TSO @SYSCMD CMD("DUMP COMM=(MVS1)") MSG(IEE094D) RPY("ASID=1,END")
RMSG(IEE600I) TRY(1)
```

Output for this scenario

In Example 1-22, the REXX exec found the `IEE094D` message on the first line. It then found the WTOR reply ID to be `473` and issued an MVS REPLY command to reply with `ASID=1,END`. It also found the `IEE600I` message in the REPLY command responses.

Example 1-22 Scenario 11 output

```
@SYSCMD operands  : CMD("DUMP COMM=(MVS1)") MSG(IEE094D) RPY("ASID=1,END")
RMSG(IEE600I) TRY(1)
SDSF HCE status  : established RC=00
------------------------------------------------------------------------------------
ISFEXEC options  : ( )
Test command     : /D T
```

```
SDSF short message: COMMAND ISSUED
SDSF long  message: ISF754I Command 'SET CONSOLE' generated from associated variable
ISFCONS.
SDSF long  message: ISF754I Command 'SET DELAY' generated from associated variable
ISFDELAY.

SDSF ULOG messages:
SC70   2007123 17:27:18.89       ISF031I CONSOLE JOE ACTIVATED
SC70    2007123 17:27:18.89       -D T
SC70      2007123  17:27:18.89  JOB02002   IEE136I LOCAL: TIME=17.27.18 DATE=2007.123
UTC: TIME=21.27.18 DATE=2007.123
--------------------------------------------------------------------------------------
ISFEXEC options  : ( )
Original command : /DUMP COMM=(MVS1)

SDSF short message: COMMAND ISSUED
SDSF long  message: ISF754I Command 'SET CONSOLE' generated from associated variable
ISFCONS.
SDSF long  message: ISF754I Command 'SET DELAY' generated from associated variable
ISFDELAY.

SDSF ULOG messages:
SC70   2007123 17:27:20.07       ISF031I CONSOLE JOE ACTIVATED
SC70    2007123 17:27:20.07       -DUMP COMM=(MVS1)
SC70      2007123  17:27:20.07            *087 IEE094D SPECIFY OPERAND(S) FOR DUMP
COMMAND
--------------------------------------------------------------------------------------
ISFEXEC options  : ( )
Original command : /R 087,ASID=1,END

SDSF short message: COMMAND ISSUED
SDSF long  message: ISF754I Command 'SET CONSOLE' generated from associated variable
ISFCONS.
SDSF long  message: ISF754I Command 'SET DELAY' generated from associated variable
ISFDELAY.

SDSF ULOG messages:
SC70   2007123 17:27:21.29       ISF031I CONSOLE JOE ACTIVATED
SC70    2007123 17:27:21.29       -R 087,ASID=1,END
SC70   2007123 17:27:21.30       IEE600I REPLY TO 087 IS;ASID=1,END

Command result    : RC=00 System command issued, response received from the EMCS
console.
              Message IEE094D found in the command responses.
                MVS REPLY command issued.
```

```
            Message IEE600I found in the REPLY response.
-------------------------------------------------------------------------------------
SDSF HCE status   : revoked RC=00
```

1.6.13 Scenario 12 - Suppress all outputs

This scenario is a modification of "Scenario 3 - Use a specific EMCS console" on page 60. It specifies the QUIET or Q keyword parameter so that the REXX exec suppresses all outputs, except the error messages when the return code is non-zero.

Invoking this scenario

To invoke the scenario, use this command line:

```
TSO @SYSCMD CMD(D IPLINFO) CONS(IBM) Q(Y)
```

Output for this scenario

When the console used is not a primary EMCS console, the REXX exec returns with a return code of 4. Example 1-23 shows when output when the QUIET parameter is omitted.

Example 1-23 Scenario 12 - without QUIET

```
@SYSCMD operands  : CMD(D IPLINFO) CONS(IBM)
SDSF HCE status   : established RC=00
-------------------------------------------------------------------------------------
ISFEXEC options   : ( )
Original command  : /D IPLINFO

SDSF short message: COMMAND ISSUED
SDSF long  message: ISF754I Command 'SET CONSOLE IBM' generated from associated
variable ISFCONS.
SDSF long  message: ISF754I Command 'SET DELAY' generated from associated variable
ISFDELAY.

SDSF ULOG messages:
SC70      2007123  18:31:41.77             ISF032I CONSOLE IBM ACTIVATE FAILED,
RETURN CODE 0004, REASON CODE 0000

Command result    : RC=04 System command issued, no response received from the EMCS
console.
                          Console used is an internal console or a shared EMCS
console.
-------------------------------------------------------------------------------------
SDSF HCE status   : revoked RC=00
```

Example 1-24 shows the output when the QUIET parameter has `YES` or `Y` and the return code is non-zero.

Example 1-24 Scenario 12voutput - with QUIET(Y)

```
Command result    : RC=04 System command issued, no response received from the EMCS
console.
Console used is an internal console or a shared EMCS console.
```

2

Copying SYSOUT to a PDS

This chapter describes a scenario, BUILDPDS, that copies SYSOUT data from a group of jobs to members of a partitioned data set (PDS). The scenario, BUILDPDS, accepts a number of job selection criteria and, through the REXX with SDSF API, locates and transfers SYSOUT records written to a specific DDNAME.

This scenario can be valuable to anyone interested in capturing a series of reports for further processing.

For more information about how to obtain the program source for this scenario, see Appendix B, “Additional material” on page 305.

2.1 Background and overview of this scenario

My group is responsible for operating system exits for a complex of over 30 LPARs. Each LPAR is capable of running its own unique combination of exits. We operate an RYO dynamic exit system that is controlled by system parameters located in a member of SYS1.PARMLIB. Further, we are responsible for JES exits that are controlled by JES2 initialization parameters, and we write user SVCs that are controlled by the IEASVCnn members in PARMLIB.

As it turns out, there are a large number of source data sets that include information that we need to reference to understand which exit is running where and what it is doing. In the past, we have spent a lot of time browsing data sets on each LPAR to answer questions concerning our exits. However, we have determined that it is a better use of our time to automate this process.

We decide to create several groups of batch jobs, each of which copies one member from a parameter library (typically SYS1.PARMLIB) to SYSOUT using IEBGENER. We use the /*ROUTE JCL statement to direct each job to a specific LPAR and then use SYSAFF to ensure that the job runs only where is it supposed to run.

Figure 2-1 illustrates this scenario.

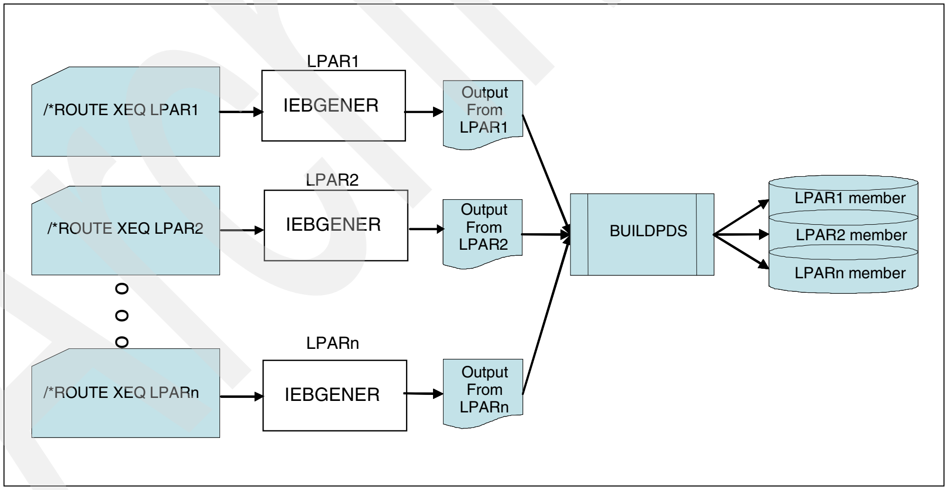


Figure 2-1 BUILDPDS flow

Figure 2-1 shows one group of jobs, with each job destined for a separate LPAR. The output returns to the JES spool of the submitting system, and we want to create a PDS on that LPAR with one member for each job. Subsequently, we run an application program to read the members and to create a composite report.

As Figure 2-1 illustrates, BUILDPDS is the program which creates the PDS members from the SYSOUT of each job.

2.2 Input to BUILDPDS

BUILDPDS accepts arguments in keyword(value) format to direct its operation. The first list includes arguments that tell BUILDPDS how to select jobs from the PRINT queue for processing. If any of these arguments is omitted, the corresponding value for the jobs does not affect whether the job is selected for SYSOUT processing.

The BUILDPDS arguments are:

JOB(<pattern>)	Specifies which job names are to be selected for SYSOUT processing. You can specify an SDSF job name filter here.
OWNER(<pattern>)	Specifies how to filter out jobs by owner ID mask. You can specify an SDSF owner filter here.
CLASS(<class>)	Specifies the execution class of the job.
DEST(<destination>)	Specifies the output destination.
COND(<cond-code>)	Specifies the maximum condition code for the job.

The remaining arguments tell BUILDPDS how to process the jobs that it selects. The DDNAME and PDS keywords must be coded but MEMBER is optional. The remaining arguments are:

DDNAME(<ddn>)	SYSOUT DDNAME copied to the PDS.
PDS(<dsname>)	The fully-qualified name of the PDS to which the members are copied. This data set is deleted and reallocated so it is important that the user ID running BUILDPDS have ALTER authority. UPDATE authority is not sufficient.
MEMBER(JOBID)	Specifies that the member names to which the SYSOUT is copied are set to the JOBID rather than the job name which is the default. You specify this argument if some of the jobs have the same name and the exact name is not important.

2.3 Program flow

BUILDPDS, like most other applications that use the REXX with SDSF API, spends most of its time massaging data and interfacing with the user and only a little time actually exercising the API. Figure 2-2 shows the main program flow. The entire process relies on the variables that are returned by the ISFEXEC ST command that is filtering the job list to remove unwanted jobs.

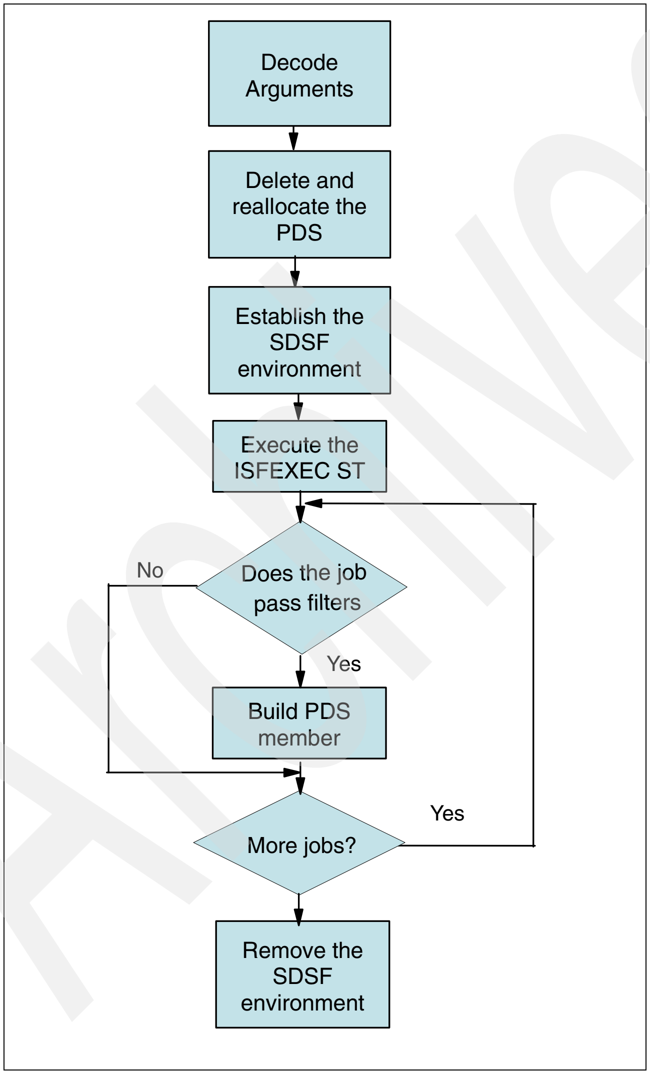


Figure 2-2 BUILDPDS main program flow

Figure 2-3 shows the logic for copying a single SYSOUT data set to a member of the PDS.

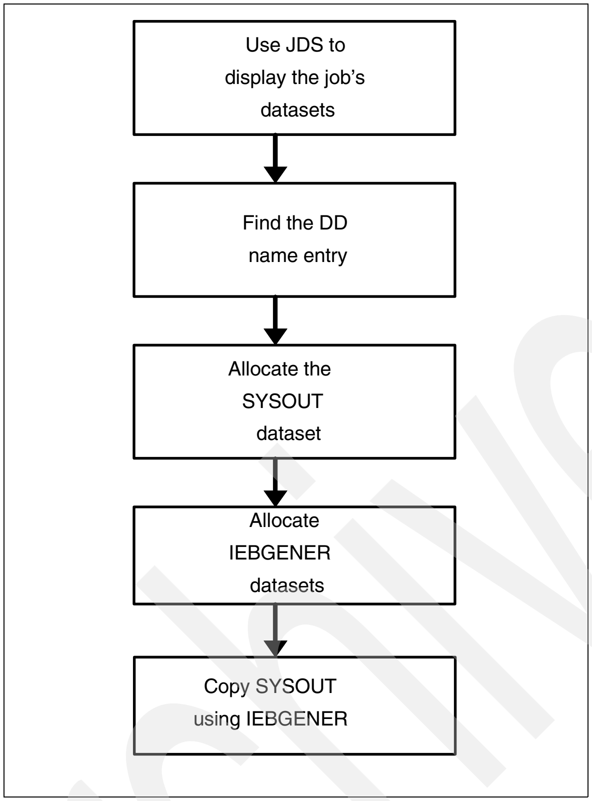


Figure 2-3 How BUILDPDS copies a single SYSOUT to the PDS

2.3.1 Decoding the arguments

Example 2-1 shows the argument decoding logic in BUILDPDS. Default values are established on lines 47 through 55 of the code, and the individual specifications are decoded in the loop on lines 57 through 113. The parse statements on lines 58 and 59 are based on the `keyword(value)` format of the operands and are broken into two statements (although they could be performed in one statement) so that the variable `ThisArg` is available for error messages.

Example 2-1 BUILDPDS Argument decoding logic

```
0047 JobPattern   = "*"
0048 OwnerPattern = "*"
0049 JobClass     = ""
0050 Destination  = ""
```

```
CondCode     = ""
DdName       = "*"
PdsDsn       = ""
Error        = "NO"
MemberRule   = "JOBNAME"

do while Arguments <> ""
   parse var Arguments ThisArg Arguments
   parse var ThisArg Keyword "(" Value ")"

   select
      when Keyword = "JOB"    then do
         JobPattern = Value
         end
      when Keyword = "OWNER"  then do
         OwnerPattern = Value
         end
      when Keyword = "CLASS"  then do
         JobClass = Value
         end
      when Keyword = "DEST"   then do
         Destination = Value
         end
      when Keyword = "COND"   then do
         if Value = "JCL" then
            CondCode = "JCL ERROR"
         else if left( Value, 1 ) = "S" then
            CondCode = "ABEND" Value
         else if left( Value, 1 ) = "U" then
            CondCode = "ABEND" Value
         else
            CondCode = CC right( Value, 4, '0' )
         end
      when Keyword = "DDNAME" then do
         DdName     = Value
         end
     when Keyword = "PDS"    then do
        PdsDsn     = Value

        call msg "off"
       Opinion = sysdsn( "'"PdsDsn"'" )
        call msg "on"
        if Opinion <> "OK" then do
          say "PDS" PdsDsn "failed validation because" Opinion
            Error = "YES"
```

```
           end
        end
     when Keyword = "MEMBER" then do
        if Value = "JOBID" then
           MemberRule = "JOBID"
        else if Value = "JOBNAME" then
           MemberRule = "JOBNAME"
        else do
            say "Bad MEMBER specification - must be JOBID or JOBNAME"
           Error = "YES"
           end
        end
     otherwise do
       say "'"ThisArg"' is not a valid argument"
        Error = "YES"
        end
     end
   end

/*  Verify that the PDS dataset name was specified  */

if PdsDsn = "" then do
  say "PDS operand (PDS dataset name) must be specified"
   Error = "YES"
   end

/*  If an error was found while processing the input arguments then we
    can go no further  */

if Error <> "NO" then do
  say "Correct arguments and rerun BUILDPDS"
   exit 1
   end
```

2.3.2 Deleting and reallocating the PDS

The next step in the BUILDPDS scenario is to delete and reallocate the PDS, as shown in Example 2-2. One downside to deleting and reallocating the PDS is that this precludes replacing only a part of the contents. However, in our case this method does not present an issue. (You can remove this code from our example if you prefer.) If you define the data set as a PDSE, then you do not have to compress the data set.

Example 2-2 Deleting and reallocating the output PDS

```
call msg "off"
ReturnCode = listdsi( "'"PdsDsn"' directory" )
call msg "on"

if ReturnCode <> 0 then do
  say "Cannot retrieve the attributes of" PdsDsn "-" SYSREASON
  exit 1
  end

if SYSDSORG <> "PO" then do
  say PdsDsn "is not a partitioned dataset"
  exit 1
  end

/*  TSO/E returns RECFM without blanks between the attribute
characters
    but requires blanks on the allocate statement  */

Hold  = SYSRECFM
Recfm = ""
do while Hold <> ""
  parse var Hold RecfmChar 2 Hold

  Recfm = Recfm RecfmChar
  end

Recfm = strip( Recfm )

/*  Build an allocation statement to recreate the PDS/library  */

if SYSADIRBLK = "NO_LIM" then
  AllocCmd = "alloc da('"SYSDSNAME"')"                 ,
                     "recfm("Recfm") lrecl("SYSLRECL")" ,
                     "blksize("SYSBLKSIZE")"            ,
```

```
0166                     "space("SYSPRIMARY SYSSECONDS")"   ,
0167                     "dir(1) dsntype(library)"          ,
0168                     SYSUNITS
0169 else
0170    AllocCmd = "alloc da('"SYSDSNAME"')"                ,
0171                     "recfm("Recfm") lrecl("SYSLRECL")" ,
0172                     "blksize("SYSBLKSIZE")"            ,
0173                     "space("SYSPRIMARY SYSSECONDS")"   ,
0174                     "dir("SYSADIRBLK")"                ,
0175                     SYSUNITS
0176
0177 /*  Delete the old dataset and allocate it anew  */
0178
0179 call msg "off"
0180 "del '"SYSDSNAME"'"
0181 DeleteRC = RC
0182 call msg "on"
0183
0184 if DeleteRC <> 0 then do
0185    say "Can't delete" SYSDSNAME":"
0186    "del '"SYSDSNAME"'"
0187    exit 1
0188    end
0189
0190 AllocCmd
0191
0192 if RC <> 0 then
0193    exit 1
```

The code in Example 2-2 requires that you run the REXX exec in a TSO environment to support the `msg`, **`listdsi`**, **`allocate`**, and **`delete`** functions. So, you can run this EXEC either interactively or under IKJEFT0x. Neither IRXJCL nor IRXEXEC are an option for this scenario.

Although the code copies RECFM, LRECL, and BLKSIZE from the currently existing output, PDSE, JES, and IEBGENER limit our choices. During testing, we discovered that JES sets the RECFM of the spool data set to Variable Block (VB) and, of course, IEBGENER demands a match to work. So, in this case, either your PDS will be VB, or you will need an alternative copy utility.

2.3.3 Interfacing with IBM z/OS System Display and Search Facility

Example 2-3 shows how BUILDPDS gets data from IBM z/OS System Display and Search Facility (SDSF). The SDSF environment is initialized on line 197, and the SDSF filters are set on lines 205 through 207. Line 209 limits the columns to only those which this executable requires. (BUILDPDS also needs the token variable, but SDSF provides this variable automatically. So, you do not have to explicitly request it.)

If line 209 had been omitted, the REXX exec would have worked identically, but the amount of storage and the amount of CPU needed by SDSF to put data into that storage would have increased. Line 210 is the call to load the status panel stem variables, and the loop through the jobs takes place on lines 219 through 232.

The assignment statement on line 218 is an example of cautious programming at its best. Using **isfrows** as the upper limit of the immediately following do loop would yield the same results. However, if some inner process were to change its value, the loop would not terminate at the same upper end. By copying the value of **isfrows** at the point in time it has been freshly set and nothing else has happened, you do not have to worry that one of the interface variables will change while you are using it.

Example 2-3 Retrieving data from SDSF

```
0195 /*  Load the SDSF environment and abort on failure  */
0196
0197 IsfRC = isfcalls( "ON" )
0198 if IsfRC <> 0 then do
0199    say "RC" IsfRC "returned from isfcalls( ON )"
0200    exit IsfRC
0201    end
0202
0203 /*  "Display" the ST panel to load the related variables  */
0204
0205 isfprefix = JobPattern
0206 isfowner  = OwnerPattern
0207 isffilter = "queue = print"
0208
0209 isfcols   = "jname jobid ownerid queue jclass prtdest retcode"
0210 address SDSF "isfexec st"
0211 if RC <> 0 then do
0212    say "RC" RC "returned from ISFEXEC ST"
0213    call DisplayMessages
0214    end
0215
0216 /*  Process every line in the ST display  */
0217
0218 StRows = isfrows
0219 do i = 1 to StRows
```

```
   /*  Apply the other filters  */

   if JobClass    <> "" & jclass.i  <> JobClass    then iterate
   if Destination <> "" & prtdest.i <> Destination then iterate
   if CondCode    <> "" & retcode.i <> CondCode    then iterate

   /*  The job is selected so go process its SYSOUT  */

   if MemberRule = "JOBID" then
      call ProcessSysout jname.i, jobid.i, token.i, DdName
   else
      call ProcessSysout jname.i, jname.i, token.i, DdName
   end

/*  Unload the SDSF environment  */

call isfcalls "OFF"

exit 0
```

The **ProcessSysout** routine, shown in Example 2-4, uses ISFACT on line 267 to get a JDS `listing` for the single ST line that is identified by the token. The option, `prefix j_` ensures that the stem variables returned by ISFACT are unique and do not conflict with those that are returned by ISFEXEC.

The rows returned by ISFACT are scanned from lines 277 to 299 to find the one that matches the user’s DDNAME, and that row is then passed to SDSF as an argument to **isfact** on line 282. Note the references to the `j_`-prefixed variables on lines 277, 278, and 282.

Example 2-4 Performing the actual copy

```
0262 ProcessSysout:
0263   parse arg JobName, MemberName, SdsfToken, DDN
0264
0265   /*  Display the job's datasets  */
0266
0267   address SDSF "isfact st token('"SdsfToken"') parm(np ?) (prefix j_"
0268   ActRC = RC
0269   if RC <> 0 then do
0270      say "JDS processing failed for job" JobName "with RC" ActRC
0271      call DisplayMessages
0272      return
0273      end
0274
0275   /*  Find the line for the specified DD name  */
```

```
0276
0277    do jX = 1 to j_ddname.0
0278       if j_ddname.jX <> DDN then iterate
0279
0280       /*  Got the correct dataset.  Now allocate the SYSOUT  */
0281
0282       address SDSF "isfact st token('"j_token.jX"') parm(np sa)"
0283       ActRC = RC
0284       if RC <> 0 then do
0285          say "SYSOUT allocation failed for" JobName "with RC" ActRC
0286          call DisplayMessages
0287          return
0288          end
0289
0290       /*  Copy the SYSOUT to the PDS member */
0291
0292       if CopySysout( PdsDsn"("MemberName")", ,
0293                      "DD:"isfddname.1 ) <> 0 then do
0294          say "Copy failed!"
0295          return 8
0296          end
0297
0298       return 0
0299       end
0300
0301    return
```

The **isfact** on line 282 has an action unique to REXX with SDSF. In the interactive SDSF, you view SYSOUT by selecting the data set using the **S** command in the NP column. You use the SA command to view SYSOUT on the virtual panel, which selects the SYSOUT and requests SDSF allocate it for you. Allocation is done and the DDNAME is passed back to you in the **isfddname** stem variable. (It is passed back in a stem because it is possible to allocate multiple data sets using SA in other circumstances.) In this case, there is only a single data set allocated, **isfddname**.1 is the variable that includes the allocated DDNAME and that DDNAME is passed to the **CopySysout** routine on lines 292 and 293.

2.3.4 Writing the data to the PDS

A shortened version of the **CopySysout** routine with error logic removed is in Example 2-5. The actual copy is done using IEBGENER; however, we invoke IEBGENER with a DDNAME substitution list.

Example 2-5 CopySysout routine (abridged)

```
CopySysout:
   parse arg CS_Output, CS_Input

   /*  Allocate the input dataset if necessary  */

   if left( CS_Input, 3 ) = "DD:" then
      Sysut1DD = substr( CS_Input, 4 )
   else do
      call bpxwdyn "ALLOC DSN('"CS_Input"') SHR RTDDN(Sysut1DD)" ,
                         "MSG(CS_Msg.)"
      . . .
      end

   /*  Allocate the output dataset if necessary  */

   if left( CS_Output, 3 ) = "DD:" then
      Sysut2DD = substr( CS_Output, 4 )
   else do

      call bpxwdyn "ALLOC DSN('"CS_Output"') SHR RTDDN(Sysut2DD)"
,
                         "MSG(CS_Msg.)"
      . . .
      end

   /* Allocate a dummy dataset for SYSIN  */

   call bpxwdyn "ALLOC DUMMY RTDDN(SysinDD) MSG(CS_Msg.)"
   . . .

   /* Allocate a temporary dataset for SYSPRINT  */

   call bpxwdyn "ALLOC UNIT(SYSALLDA) SPACE(10,10) TRACKS" ,
                      "RTDDN(SysprintDD) MSG(CS_Msg.)"
   . . .

 /*  Build the DD name substitution list  */
```

```

Parm   = ""                                /* No PARM= parm    */
DDlist = copies( '00'x, 8 )    ||,   /* DD  1: SYSLIN    */
         copies( '00'x, 8 )    ||,   /* DD  2:  n/a      */
         copies( '00'x, 8 )    ||,   /* DD  3: SYSLMOD   */
         copies( '00'x, 8 )    ||,   /* DD  4: SYSLIB    */
         left( SysinDD, 8 )    ||,   /* DD  5: SYSIN     */
         left( SysprintDD, 8 ) ||,   /* DD  6: SYSPRINT  */
         copies( '00'x, 8 )    ||,   /* DD  7: SYSPUNCH  */
         left( Sysut1DD, 8 )   ||,   /* DD  8: SYSUT1    */
         left( Sysut2DD, 8 )   ||,   /* DD  9: SYSUT2    */
         copies( '00'x, 8 )    ||,   /* DD 10: SYSUT3    */
         copies( '00'x, 8 )    ||,   /* DD 11: SYSUT4    */
         copies( '00'x, 8 )    ||,   /* DD 12: SYSTERM   */
         copies( '00'x, 8 )    ||,   /* DD 13:  n/a      */
         copies( '00'x, 8 )          /* DD 14: SYSCIN    */

/*  Call IEBGENER with two pointers: the first to a null PARM
    string and the second to the DD name substitution list */

address LINKMVS "IEBGENER Parm DDlist"
```

Every **DFSMSdfp**™ utility program is capable of being invoked with a DDNAME substitution list when being called from within a program as documented in the **DFSMSdfp** Utilities manual.

As shown in Figure 2-4, the list includes the DD names that you want the utility to use in place of the names that are defined in the utility description.

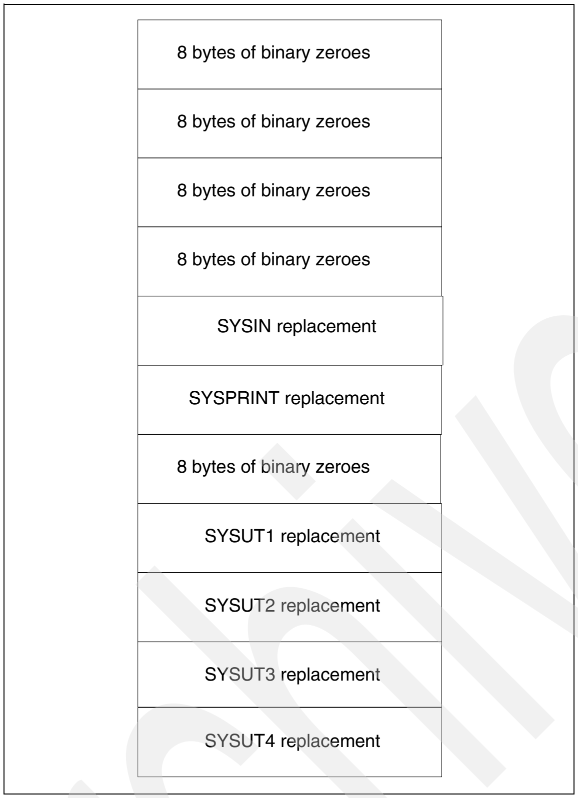


Figure 2-4 DDNAME substitution list format

The “holes” in the list which are labeled *8 bytes of binary zeroes* are to make the list compatible with similar lists that are used with different utilities. You code the list as a solid block of storage, placing the actual DD names that you want to use for the various files in the spots indicated. The DDNAME substitution list is created on lines 380 through 393 of Example 2-5. The excess entries at the end are used by other utility programs and are ignored by IEBGENER.

The **CopySysout** routine takes two arguments that are either a data set name or a DD name and distinguishes between by requiring DD names to be prefixed by *DD:*. The first argument is for the output of the copy and the second argument defines the input. This unusual arrangement is because the routine was taken from another program that had a similar requirement. In our case, the input will always be a DD name, as returned by ISFACT from the **parm(np sa)** invocation, and the output will always be a data set name (of the member in our PDS). We use the **bpxwdyn** routine to do the allocations because we want the DD name to be returned by the allocation rather than supplying it ourselves. We also allocate a dummy SYSIN file and a temporary data set SYSPRINT. The comments indicate where the various files are allocated in the logic.

We call IEBGENER on line 398 using the LINKMVS environment so that we can pass the address of the DDNAME list as the second parameter in the call. The first parameter, the PARM used to direct IEBGENER execution, is allocated as a blank string on line 379 and used as a placeholder in the call.

2.4 Suggestions for continued development

BUILDPDS has proven useful in its present form, but we considered two areas for update. The first update enhances the program to extract multiple SYSOUTs from a singe job into multiple members of the PDS. To accomplish this, you have to come up with a naming scheme that allows the different members to have different names.This enhancement exists in two forms:

- Two or more DDNAMEs in a single step
- One DDNAME in two or more steps

The second variation allows gathering all data from a single LPAR in one job.

The second update makes BUILDPDS aware of the contents of the report and uses that content to determine part or all of the member name. A variant of this capability makes BUILDPDS aware of some sort of notation in the JCL, perhaps a special-formatted comment, to identify the member name or DDNAME.

3

Bulk job update processor

This chapter describes a scenario, LISTPROC, that includes techniques that you can use to process multiple jobs with a single command. Using these techniques, you can cancel job output, modify several different job output fields, or execute a CLIST for each job. The processor is implemented as a single REXX executable (referred to in the remainder of this chapter as *REXX exec*) that extends the functionality of IBM z/OS System Display and Search Facility (SDSF) in a natural way.

The help panels for SDSF include examples of canceling a job's SYSOUT and several examples of modifying overtypable fields. However, the focus of those examples can be a bit too narrow to be useful in the typical installation. The program that we describe in this chapter expands on those examples to provide a more robust solution.

Canceling jobs and automated updating of their fields are potentially serious operations, especially when amplified by the power that REXX provides to quickly process large numbers of jobs through the REXX with SDSF interface. So, we discuss testing considerations as we examine the code.

This program is of interest to operations support and applications support personnel as well as anyone who might benefit from bulk update operations.

3.1 Scenario description

Canceling job output is a serious process that you must approach with caution. In our environment, we find a slow but persistent buildup of jobs over time that must be purged to keep the number of queued jobs manageable. We also find frequent cases where our user community falls behind our rapid update cycle and fails to update printer destinations and forms in their JCL, resulting in jobs that remain on the queue, requiring a manual update.

A facility to update jobs in bulk, for example a facility that changes all output for a specific printer destination to a different destination, would be very valuable. However, without care, unfortunately, we might find the cure far worse than the issue. It is important that we ensure that the only the jobs that we want to modify are the ones that are modified by the facility.

For information about how to obtain the program source for this scenario, see Appendix B, “Additional material” on page 305.

3.1.1 Tasks that this scenario accomplishes

The program, LISTPROC (or SYSOUT list processor), is an ISPF application that provides update, cancel, and CLIST execution functions. LISTPROC provides two different panels. As shown in Figure 3-1, the first panel allows you to specify several filters.

```
Enter job selection criteria -------------------------------------------------
COMMAND ===>                                                    SCROLL ===> CSR

 General
   Jobname       ==> ITSO*      (Generic pattern, including % and * characters)
                 ==> LE*
                 ==>
   Ownerid       ==> L%V%Y      (Generic pattern, including % and * characters)
                 ==>
                 ==>
   Execute Node  ==>            (Node on which execution took place)
   Destination   ==> BETEL*             Destination ==> JVP*
                 ==> *                              ==>
                 ==>                                ==>
                 ==>                                ==>
   Job class     ==>
   Sysout class  ==>
   Forms         ==>
   Job age       ==>            (Minimum number of days since the job ended.
                                 Enter a value between 0 and 360)
 Condition Code
   Type          ==> ABEND      (JCL, RC, ABEND, SYSTEM, USER, CANCELED)
   Numeric code  ==>            (nn for RC, ignored for JCL and CANCELED,
                                 abend code for ABEND, USER or SYSTEM)
```

Figure 3-1 The LISTPROC job selection panel

All jobs queued for output in the system are passed through the nonblank filter, s and only jobs that pass all the tests are selected for display. In this example, we want to see all jobs with the following characteristics:

- With a `jobname` that begins with letters *ITSO* and ends with the number *2* or which begins with the letters *LE*
- With an `ownerid` whose first, third, and fifth character are the letters *L*, *V*, and *Y*, and are exactly five characters long
- That ran in `jobclass U`
- That had a destination that begins with *BETEL* or *JUP*
- That abended

After you enter all the selection criteria, press Enter to display the jobs that satisfy all the criteria simultaneously. At the same time, LISTPROC saves your selection criteria so that you do not have to enter it again to select the same jobs. Figure 3-2 shows the second panel where you can see every selected job.

```
-------------------------- Selected Job Display Panel ------------- Row 1 of 22
COMMAND ===> _                                                SCROLL ===> CSR

                                          Job Out
X Job-name Job-Id    Owner-id Exec Cls Cls Forms    Dest     Max-Cond   End-Date
- -------- --------  -------- ---- -   -   -------- -------- ---------- --------
  ITSOBD04 JOB28319 LEVEY    SC70  A   A   STD      WTSCPLX2 ABEND SB37 2007.116
  ITSODBM1 JOB28314 LEVEY    SC70  A   A   STD      WTSCPLX2 ABEND SE37 2007.116
  ITSODBM1 JOB28315 LEVEY    SC70  A   A   STD      WTSCPLX2 ABEND SB37 2007.116
  ITSO01B2 JOB27016 LEVEY    SC70  A   C   STD      WTSCPLX2 ABEND S806 2007.110
  ITSO01D1 JOB25157 LEVEY    SC70  A   C   STD      WTSCPLX2 ABEND S806 2007.099
  ITSO01D1 JOB27018 LEVEY    SC70  A   C   STD      WTSCPLX2 ABEND S806 2007.110
  ITSO01F3 JOB25159 LEVEY    SC70  A   C   STD      WTSCPLX2 ABEND S806 2007.099
  ITSO01H2 JOB25161 LEVEY    SC70  A   C   STD      WTSCPLX2 ABEND S806 2007.099
  ITSO01H2 JOB27022 LEVEY    SC70  A   C   STD      WTSCPLX2 ABEND S806 2007.110
  ITSO04A1 JOB25184 LEVEY    SC70  A   C   STD      WTSCPLX2 ABENDU0123 2007.099
  ITSO04A1 JOB27045 LEVEY    SC70  A   C   STD      WTSCPLX2 ABENDU0123 2007.110
  ITSO04E2 JOB25188 LEVEY    SC70  A   C   STD      WTSCPLX2 ABENDU0123 2007.099
  ITSO04G1 JOB25190 LEVEY    SC70  A   C   STD      WTSCPLX2 ABENDU0123 2007.099
  ITSO04I3 JOB25192 LEVEY    SC70  A   C   STD      WTSCPLX2 ABENDU0123 2007.099
  ITSO11B2 JOB25205 LEVEY    SC70  C   C   STD      WTSCPLX2 ABEND S806 2007.099
  ITSO11B2 JOB27066 LEVEY    SC70  C   C   STD      WTSCPLX2 ABEND S806 2007.110
  ITSO11C3 JOB25206 LEVEY    SC70  B   C   STD      WTSCPLX2 ABEND S806 2007.099
  ITSO11C3 JOB27067 LEVEY    SC70  B   C   STD      WTSCPLX2 ABEND S806 2007.110
  ITSO11E2 JOB25208 LEVEY    SC70  B   C   STD      WTSCPLX2 ABEND S806 2007.099
  ITSO11F3 JOB27070 LEVEY    SC70  C   C   STD      WTSCPLX2 ABEND S806 2007.110
  ITSO14B2 JOB25235 LEVEY    SC70  E   C   STD      WTSCPLX2 ABENDU0123 2007.099
  ITSO14J1 JOB25243 LEVEY    SC70  E   C   STD      WTSCPLX2 ABENDU0123 2007.099
****************************** Bottom of data *******************************
```

Figure 3-2 LISTPROC Selected Job Display Panel

By examining the jobs listed in the panel, you can see whether your filter criteria selected the jobs that you want to process. If it did not, you can press PF3, modify your filter criteria, and press Enter to redisplay the selected jobs. By repeatedly updating the filters, you can eventually get the jobs in the Job Display panel close to what you want. Then, you can use the `X` line command in the X column to get to the panel shown in Figure 3-3.

```
-------------------------- Selected Job Display Panel ------------ Row 1 of 22
COMMAND ===> _                                                    SCROLL ===> CSR

                                         Job Out
X Job-name Job-Id   Owner-id Exec Cls Cls Forms    Dest     Max-Cond   End-Dat
- -------- -------- -------- ---- -   -   -------- -------- ---------- -------
  Excluded
  ITSODBM1 JOB28314 LEVEY    SC70 A   A   STD      WTSCPLX2 ABEND SE37 2007.11
  ITSODBM1 JOB28315 LEVEY    SC70 A   A   STD      WTSCPLX2 ABEND SB37 2007.11
  ITSO01B2 JOB27016 LEVEY    SC70 A   C   STD      WTSCPLX2 ABEND S806 2007.11
  ITSO01D1 JOB25157 LEVEY    SC70 A   C   STD      WTSCPLX2 ABEND S806 2007.09
  ITSO01D1 JOB27018 LEVEY    SC70 A   C   STD      WTSCPLX2 ABEND S806 2007.11
  ITSO01F3 JOB25159 LEVEY    SC70 A   C   STD      WTSCPLX2 ABEND S806 2007.09
  ITSO01H2 JOB25161 LEVEY    SC70 A   C   STD      WTSCPLX2 ABEND S806 2007.09
  Excluded
  Excluded
  ITSO04A1 JOB27045 LEVEY    SC70 A   C   STD      WTSCPLX2 ABENDU0123 2007.11
  Excluded
  Excluded
  Excluded
  Excluded
  ITSO11B2 JOB27066 LEVEY    SC70 C   C   STD      WTSCPLX2 ABEND S806 2007.11
  ITSO11C3 JOB25206 LEVEY    SC70 B   C   STD      WTSCPLX2 ABEND S806 2007.09
  ITSO11C3 JOB27067 LEVEY    SC70 B   C   STD      WTSCPLX2 ABEND S806 2007.11
  Excluded
  Excluded
  Excluded
  ITSO14J1 JOB25243 LEVEY    SC70 E   C   STD      WTSCPLX2 ABENDU0123 2007.09
******************************* Bottom of data ******************************
```

Figure 3-3 LISTPROC Selected Job Display Panel with excluded jobs

When jobs are excluded in a list, they do not participate in a cancel, update, or CLIST action. As the user, your goal is to create a list with only the SYSOUT to be modified. To help locate all jobs that you might want to exclude, you can use the SORT command to reorder the job list by the contents of any set of columns. When you finally have the jobs that you want to modify or cancel in the display, you can enter one of two commands on the command line.

- By specifying CANCEL, you can cancel the displayed jobs with purge.
- By specifying the OVERTYPE, you can change the SYSOUT class, forms specification, or destination.

Figure 3-4 shows the display after you enter the CANCEL command. The REXX with SDSF API can take a while to complete all the updates, so the progress bar shows the status of the command. The bar is updated every time an action completes.

```
------------------------- Selected Job Display Panel ------------ Row 1 of 21
COMMAND ===> CANCEL                                          SCROLL ===> CSR

                                         Job Out
X Job-name Job-Id   Owner-id Exec Cls Cls Forms    Dest     Max-Cond   End-Date
- -------- -------- -------- ---- -   -   -------- -------- ---------- --------
  ITSO01C3 JOB24955 LEVEY    SC64 A   C   BALLET   WTSCPLX2 ABEND S806 2007.096
  ITSO01F3 JOB24958 LEVEY    SC70 A   C   BALLET   WTSCPLX2 ABEND S806 2007.096
  ITSO01I3 JOB24961 LEVEY    SC70 A   C   BALLET   WTSCPLX2 ABEND S806 2007.096
  ITSO02C3 JOB24965 LEVEY    SC63 A   C   BALLET   WTSCPLX2 CC 0012    2007.096
  ITSO02F3 JOB24968 LEVEY    SC64 A   C   BALLET   WTSCPLX2 CC 0012    2007.096
  Excluded
  ITSO03C3 JOB24975 LEVEY         A   C   STD      WTSCPLX2 JCL ERROR
  ITSO03F3 JOB24978 LEVEY         A   C   STD      WTSCPLX2 JCL ERROR
  ITSO03I3 JOB24981 LEVEY         A   C   STD      WTSCPLX2 JCL ERROR
  ITSO04C3 JOB24985 LEVEY    SC70 A   C   BALLET   WTSCPLX2 ABEND U012 2007.096
  Excluded
  Excluded
  ITSO05C3 JOB24995 LEVEY    SC70 A   C   BALLET   WTSCPLX2 CC 0000    2007.096
  ITSO05F3 JOB24998 LEVEY    SC64 A   C   BALLET   WTSCPLX2 CC 0000    2007.096
  ITSO05I3 JOB25001 LEVEY    SC64 A   C   BALLET   WTSCPLX2 CC 0000    2007.096
  ITSO05C3 JOB24995 LEVEY    SC70 A   M   BALLET   BETELGUE CC 0000    2007.096
  ITSO05F3 JOB24998 LEVEY    SC64 A   M   BALLET   BETELGUE CC 0000    2007.096
  ITSO05I3 JOB25001 LEVEY    SC64 A   M   BALLET   BETELGUE CC 0000    2007.096
  Excluded
  ITSO05F3 JOB24998 LEVEY    SC64 A   P   BALLET   JUPITER  CC 0000    2007.096
  ITSO05I3 JOB25001 LEVEY    SC64 A   P   BALLET   JUPITER  CC 0000    2007.096
****************************** Bottom of data ********************************



               _-------- Progress of your CANCEL    Command -------

                    3      commands issued out of 17      commands

                 @@@@@@@@@
                 @@@@@@@@@
                 @@@@@@@@@
```

Figure 3-4 The job selection list with a progress bar

In addition to the CANCEL and OVERTYPE commands, you can use the EXECUTE command against all that were not excluded SYSOUT in your list. EXECUTE passes control to a CLIST or REXX EXEC to allow each entry in the list to be processed outside the control of LISTPROC. There are two modes of operation possible with EXECUTE, distinguished by an optional argument:

- If you omit the argument, EXECUTE invokes the CLIST once for each SYSOUT in the list, passing it all the values on the line.
- If you use the argument BATCH, EXECUTE creates a temporary data set with one record for each SYSOUT in your list that was not excluded and invoke your CLIST passing the data set's DDNAME as an argument. Your CLIST can then process all the jobs at one time.

If you find that you have excluded too many jobs, you can press PF3 and press Enter again to redisplay the original Job Selection Panel. Alternately, you can specify the RESET command to remove the exclusion from all the excluded jobs.

The Job Display panel remains until you press PF3 to either enter a different set of filters or to get out by pressing PF3 again.

3.1.2 Testing the scenario

With a utility program this powerful, we recommend three overarching goals in testing:

- Demonstrate that the program does what it is supposed to do.
- Demonstrate that the program does not do anything that it is not supposed to do.
- Make sure that the program does not take itself, and everything near it, down the sinkhole before it is fully debugged.

The first thing we did was to code the program to not take any action at all before displaying the actions it was *going* to take when it decided that action was necessary. This action has the dual benefit of letting us make sure that we were not trying to send paychecks to the public printer as well as helping us when we ran into the inevitable misunderstandings about the details of the API. The volume of debugging messages was not too bad, but if we had more actions, we would have written the debugging messages to a log data set. Log data sets have the additional advantage of not disappearing whenever you press the Enter key. Further, if you control logging through a command-line argument, you can turn it back on without much effort on your part.

The next thing we did was create a series of batch jobs to provide a predictable test bed for the display. That way, we could guarantee that every combination of factors we wanted to test would be present on the system. Along with this, we came up with a series of testing scenarios that directed our test sessions. We made sure that not only was every positive function tested (the program would do what it was supposed to) but that every negative function was tested (garbage in, error messages out). In particular, we made sure that there were an abundance of error messages and that there was at least one test case to generate each and every one of them.

3.2 Programming the interface

The REXX with SDSF interface provides a very powerful way to retrieve information from the JES job queue with a minimum of effort. In most applications programs, LISTPROC being no exception, the actual extraction of data is a minor part of the programming effort. Most of your time is spent in connecting the user to the data.

In this section, we examine how LISTPROC works. As you read through this analysis, keep in mind that LISTPROC was never intended to be a fully functioning application and would certainly benefit from extension. Its primary purpose is to illustrate the use of the REXX with SDSF API and to show how you can use the API to create a powerful and useful utility.

3.2.1 Program flow

Figure 3-5 presents the basic logic of the LISTPROC program. Initialization includes setting up the ISPF environment, which allows you to use LISTPROC interactively, thus improving the user's experience. The traditional way to set up the ISPF environment is to build panel and message libraries as part of a product's installation and to add them to the panel and message concatenation. LISTPROC was not written in this way because to do so would require the user to keep track of several different pieces, increasing the odds that the program would never run at all. Instead, LISTPROC allocates two temporary data sets, writes the panels and messages to them, and uses the LIBDEF service to add the libraries to the ISPPLIB and ISPMLIB concatenations. This method keeps all parts of the program together and makes it easier for the programmer to keep track of how the ISPF interface is constructed. It also increases the size of LISTPROC significantly.

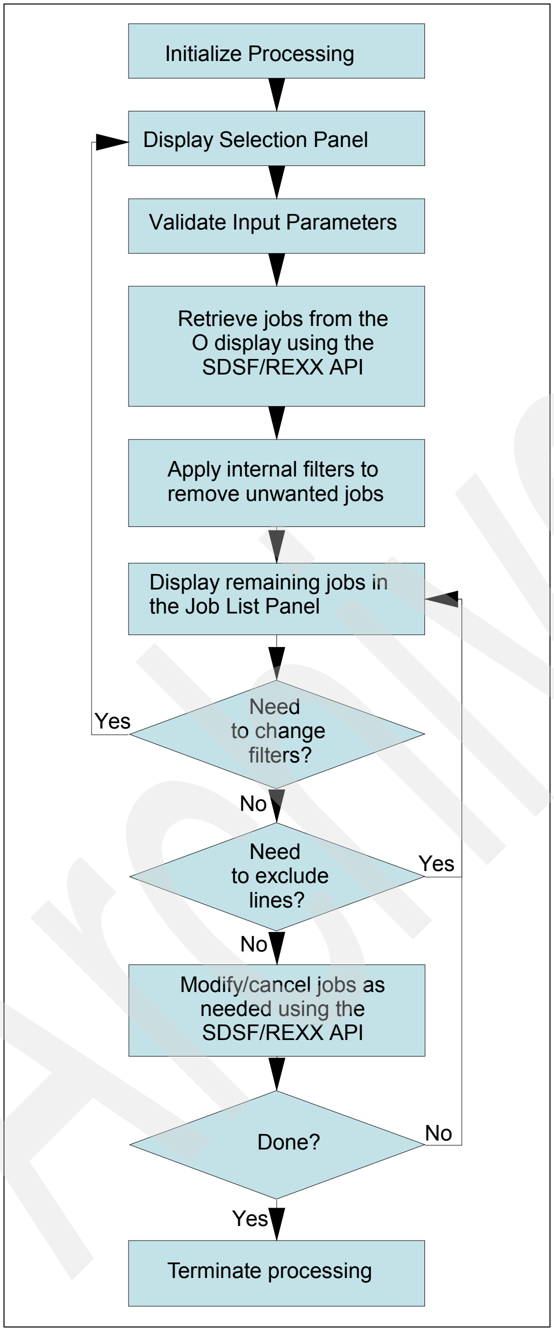


Figure 3-5 LISTPROC program logic

3.2.2 Retrieving SYSOUT information

As Figure 3-5 shows, the REXX with SDSF API only comes into play to retrieve a job's SYSOUT entries from the O panel and to perform the overtype or cancel functions. As discussed in Chapter 1, "Issuing a system command" on page 37 the REXX with SDSF interface mimics a user's interactive session. You establish the SDSF environment, issue commands to retrieve and modify data, and deactivate the SDSF environment.

Example 3-1 shows how to make the API available by creating a command processing environment that is accessible through the **`address SDSF`** command. The TSO or ISPEXEC environments are present when your EXEC begins execution (assuming that you are executing under TSO/E and ISPF). However, you need to establish the SDSF environment. The **`isfcalls`** subroutine is provided to create and to destroy this environment. If you forget to do this step, your calls to SDSF all fail with RC -3. The **`address SDSF`** instruction is not flagged by the REXX interpreter.

Example 3-1 Initializing the SDSF environment

```
IsfRC = isfcalls( "ON" )
if IsfRC <> 0 then do
   say "RC" IsfRC "returned from isfcalls( ON )"
   exit IsfRC
   end
```

SDSF distinguishes requests that you make by where you would have entered them if were you executing SDSF interactively. Requests that you would make on the command line, such as requests for specific panels (such as ST or O), SDSF commands (such as WHO), or MVS commands (such as D R) are passed to SDSF using the ISFEXEC API command. Requests that you would make by entering an action in the NP column or by modifying an overtypable field in the display area are passed to SDSF using the ISFACT API command.

Some commands, however, are implemented by setting REXX variables prior to invoking the interface. The job filtering commands—**`PREFIX`**, **`OWNER`**, **`DEST`** and **`FILTER`**—fall into this category. Variables **`isfprefix`**, **`isfowner`**, **`isfdest`**, and **`isffilter`** are set to the filtering values that their respective commands would specify.

After ISFEXEC completes, the rows of data are placed into stem variables, REXX variable **`isfcols`** is set to the names of all retrieved columns, and variable **`isfrows`** is set to the number of rows that are retrieved. In addition, the `0 member` of each stem variable also includes the number of rows.

Example 3-2 shows how LISTPROC retrieves and processes output data set information from the output panel. SDSF provides several filters to restrict the interactive display, which works well at restricting the rows that are returned by ISFEXEC. Our program optionally uses two of these—the prefix and owner—to implement the `jobname` and `ownerid` parameters. However, there are three `jobname` and three `ownerid` masks on the selection panel, and SDSF only supports one of each. The strategy of LISTPROC is to process the `jobname` and `ownerid` parameters in similar fashion:

- If zero or one `jobname` or `ownerid` patterns is specified, they are assigned to **`isfprefix`** or **`isfowner`**, respectively.
- If two or three `jobname` or `ownerid` patterns are specified, they are handled completely by LISTPROC and **`isfprefix`** or **`isfowner`** is set to asterisk (*) to force SDSF to pass all names into LISTPROC for filtering.

So **`isfprefix`** and **`isfowner`** are set on lines 370 through 378, but LISTPROC supports several more filters than SDSF. The other filters are implemented separately, which we will discussed shortly. LISTPROC limits the columns that are returned by setting the **`isfcols`** variable to the column names which will actually be processed on lines 381 through 382. SDSF is given control on line 389 to create the output display variables. We specify `alternate` because the execution system (stem variable `esys`) and end date (stem variable `daten`) are not defined as one of the primary columns, which raises an interesting issue.

This program, as is every program that uses the API, depends on the definition of which columns are on the primary panel and which are on the alternate panel. LISTPROC was written assuming the columns were assigned according to the default definitions with which SDSF is shipped. However, if your environment tailors SDSF, you might have to change the code. In the worst case, you could find that neither the primary nor alternate definitions provide all the columns that you need, and there is no way to retrieve all the data in a single call to SDSF.

Example 3-2   Initializing the SDSF environment

```
0370    if PnlJobn2 = "" then
0371       isfprefix = PnlJobn1
0372    else
0373       isfprefix = "*"
0374
0375    if PnlOwnr2 = "" then
0376       isfowner  = PnlOwnr1
0377    else
0378       isfowner  = "*"
0379
0380    isffilter = ""
0381    isfcols   = "jname   jobid   ownerid  oclass  forms   queue" ,
```

```
0382                "jclass  destn   retcode  daten   esysid"
0383    P_Count   = 0
0384
0385    /*  The isfexec retrieval of the output panel stem variables will
0386        only work if all the variables are defined on the alternate
0387        screen definition.  This is true for SDSF as distributed  */
0388
0389    address SDSF "isfexec o (alternate delayed"
0390    if RC <> 0 then do
0391       say "RC" RC "returned from ISFEXEC O"
0392       call DisplaySdsfMessages
0393       return 1
0394       end
0395
0396    /*  Delete and recreate the ISPF table to ensure we start with a
0397        clean slate  */
0398
0399    if DeleteAndRecreateIspfTable( ) = "ERROR" then
0400       return 1
0401
0402    /*  Loop through all the returned rows applying the remaining filters
0403        and adding selected jobs to the ISPF table  */
0404
0405    UnexcludedRowCount = 0
0406
0407    do RJL_i = 1 to isfrows
0408       Opinion = DoesJobPassFilters( RJL_i )
0409       if Opinion = "NO" then iterate
0410
0411       /*  The job is selected.  Add it to the dispay table  */
0412
0413       call AddJobToTheDisplay RJL_i
0414       UnexcludedRowCount = UnexcludedRowCount + 1
0415       end
```

The call to SDSF on line 389 also specifies `delayed` because the same two columns, `esys` and `daten`, are defined as delayed columns. If you fail to do this step, your call results in no data returned for the delayed columns, even if they are in your **`isfcols`** list when you invoke SDSF and **`isfcols`** on return does not include the columns either. There will, however, be `ISF742I` messages in the **`isfmsg2`** stem documenting that columns named in **`isfcols`** were not found. SDSF returns RC 0 so you have to scan **`isfmsg2`** to find the messages.

As mentioned previously, only the `jobname` and `ownerid` filters were optionally implemented through SDSF.

LISTPROC also provides filters for execution node, job class, SYSOUT class, forms, Max-RC, End Date (called job age in the panel and program) as shown in Example 3-3. `PnlJobn2`, the second `jobname` filter pattern, is only blank if there were zero or one `jobname` patterns. As mentioned earlier, when there are no more than one pattern, isfprefix is set to it and no filtering here is required; it has already been done by SDSF. The `jobname` filtering on lines 460 through 481 is only done when PnlJobn2 is non-blank. Similarly, `ownerid` filtering on lines 487 to 508 is only done when PnlOwnr2 is non-blank. Destination processing on lines 521 through 540 is done whenever there is any destination specified. Job aging is handled on lines 544 to 570 to complete filter processing.

Example 3-3 Internal job filtering logic

```
0453 DoesJobPassFilters:
0454   arg DJPF_i
0455
0456   /*  Apply the jobname filter if there are two or three of them.
0457       Otherwise filtering was done by SDSF through the isfprefix
0458       variable  */
0459
0460   if PnlJobn2 <> "" then do
0461      PatternFound = "NO"
0462
0463      do DJPF_j = 1 to 3
0464         /*  G_Pattern.JOBNAME<i> contains "YES" if jobname filter <i>
0465             is enabled */
0466
0467         JobnName = "JOBNAME"DJPF_j
0468         if G_Pattern.JobnName <> "YES" then iterate
0469         PatternFound = "YES"
0470
0471         /*  Match the jobname against the "DJPF_j-th" pattern and exit
0472             the loop if there is a match  */
0473
0474         if Generic_Match( JobnName, JName.DJPF_i ) then leave
0475         end
0476
0477      /*  If the jobname loop was terminated by running out of patterns
0478          to check then the job doesn't match any pattern  */
0479
0480      if PatternFound = "YES" & DJPF_j > 3 then return "NO"
0481      end
0482
```

```
/*  Apply the ownerid filter if there are two or three of them.
    Otherwise filtering was done by SDSF through the isfowner
    variable  */

if PnlOwnr2 <> "" then do
   PatternFound = "NO"

   do DJPF_j = 1 to 3
      /*  G_Pattern.OWNERID<i> contains "YES" if ownerid filter <i>
          is enabled */

      OwnrName = "OWNERID"DJPF_j
      if G_Pattern.OwnrName <> "YES" then iterate
      PatternFound = "YES"

      /*  Match the ownerid against the "DJPF_j-th" pattern and exit
          the loop if there is a match  */

      if Generic_Match( OwnrName, Ownerid.DJPF_i ) then leave
      end

   /*  If the ownerid loop was terminated by running out of patterns
       to check then the job doesn't match any pattern  */

   if PatternFound = "YES" & DJPF_j > 3 then return "NO"
   end

/*  Apply the queue name, job class and condition code filters  */

if JobClass    <> "" & jclass.DJPF_i <> JobClass    then return "NO"
if SysoutClass <> "" & oclass.DJPF_i <> SysoutClass then return "NO"
if ExecuteNode <> "" & esysid.DJPF_i <> ExecuteNode then return "NO"
if FormName    <> "" & forms.DJPF_i  <> FormName    then return "NO"
if CondCode    <> "" & ,
   left( retcode.DJPF_i, CondLeng )  <> CondCode    then return "NO"

/*  Apply the destination name filters  */

PatternFound = "NO"

do DJPF_j = 1 to 8
   /*  G_Pattern.DESTINATION<i> contains "YES" if destination
       filter <i> is enabled */

   DestName = "DESTINATION"DJPF_j
```

```
   if G_Pattern.DestName <> "YES" then iterate
   PatternFound = "YES"

   /*  Match the print destination against the "DJPF_j-th" pattern
       and exit the loop if there is a match  */

   if Generic_Match( DestName, Destn.DJPF_i ) then leave
   end

/*  If the destination loop was terminated by running out of
    patterns to check then the job doesn't match any pattern  */

if PatternFound = "YES" & DJPF_j > 8 then return "NO"

/*  Apply the aging filter  */

if AgeInDays <> "" & daten.DJPF_i <> "" then do
   parse var daten.DJPF_i EndYear "." EndDay

   /*  Set ElapsedDays to the number of days since the job ended.
       If the job ended in this year, it is the difference
       between the ending date and today's Julian day.  If the
       job ended last year, it is the difference in days plus the
       number of days in the year the job ended.  Otherwise the
       job ended more than one year ago and we set ElapsedDays to
       365 to force it greater than the Age (which is constrained
       to not exceed 360 by the validation logic).  */

   ElapsedDays = JulianDays - EndDay

   select
      when EndYear = JulianYear     then NOP
      when EndYear = JulianYear - 1 then
         if EndYear // 4 = 0 then
            ElapsedDays = ElapsedDays + 366
         else
            ElapsedDays = ElapsedDays + 365
      otherwise
         ElapsedDays = 365
      end

   if ElapsedDays < AgeInDays then return "NO"
   end

return "YES"
```

3.2.3 Generic filter processing

LISTPROC provides *generic filter processing* for destination and multiple job names and `ownerids` which is not supported by SDSF. The filters are implemented in a straightforward manner but generic filter processing deserves another word or two.

Generic filters in SDSF are those which allow accepting a *group* of values by specifying a non-precise pattern rather than an explicit match. Every character in the pattern stands for itself except for the percent sign, which represents any single character and the asterisk which represents any number of characters, including no characters at all. Adding generic pattern matching to your EXECs can make them more powerful and when you know how to do it you see that it is a straightforward matter.

Every generic pattern can be broken into up to three kinds of subpatterns. The subpatterns are used to test a string which matches the pattern if it matches every subpattern inside the pattern. There can be zero or one *initial* subpatterns, zero or one *final sub*patterns and zero or more *inner* subpatterns.

An initial subpattern is the beginning of the pattern up to the first asterisk. If the string begins with an asterisk, there is no initial subpattern. And if there is no asterisk at all, then the entire pattern is the initial subpattern and the initial subpattern is the only kind of subpattern. A final subpattern is the end of the pattern after the last asterisk. If the pattern ends with an asterisk then there is no final subpattern. And if there is only one asterisk in the pattern the everything following the asterisk is the final subpattern and there are no inner subpatterns at all. If the pattern contains two or more asterisks then the inner subpatterns are the characters lying between the asterisks. Example 3-4 gives an example of how a pattern is broken down into subpatterns.

Example 3-4 Sample generic pattern breakdown

```
DEST(NYC%1*RR*MM*03)

Initial subpattern NYC%1
Inner subpattern 1 RR
Inner subpattern 2 MM
Final subpattern   03
```

When a pattern is broken down, the three kinds of subpatterns are applied in turn to each string you want to match. The initial subpatterns are matched to the start of the string and the final subpatterns are matched to the end. The remainder of the string is then scanned looking for matches with each inner subpattern, one after another. Only if all subpatterns match is the string considered to have passed the filter.

LISTPROC implements generic pattern matching with three subroutines. **`Generic_Pattern`** takes a pattern and breaks it into subpatterns. **`Generic_Match`** takes a string and matches it against a specific pattern. **`Generic_Matches`** is a subroutine to **`Generic_Match`** which sees if a subpattern matches a specific piece of the string.

When SDSF has returned all rows satisfying the **`isfprefix`** and **`isfowner`** masks and LISTPROC has applied its filters to the rows, all remaining rows are displayed. The user can use the **`X`** line command to exclude individual rows. The row's appearance on the panel is changed by blanking out all fields except the job name which is replaced by `Excluded`. The user can also request that the output for all rows be canceled by using the CANCEL command or can request that one of three columns be changed for all those that were not excluded rows by using the OVERTYPE command.

3.2.4 Processing the CANCEL and OVERTYPE commands

Example 3-5 shows how LISTPROC handles the CANCEL command. When SDSF returns rows of data in response to the ISFEXEC command, it creates, in addition to the columns that you request, one additional stem variable, TOKEN.*i*, which includes a value uniquely identifying the row. When you want SDSF to perform an action against a row (that you would have requested interactively by overtyping some column), you identify the row by naming the initial command (O, ST, DA, and so forth) along with the token for the row. You then tell SDSF how to overtype the row's data by issuing an ISFACT command specifying a parameter with one or more pairs of values enclosed in parentheses. The first value is the name of the column and the second is the new value you want SDSF to put in the column. In our case, the column to overtype is the NP column and the value is `P`, the command for cancel with purge. Support for the OVERTYPE command in routine **`OvertypeColumns`** is similar.

Example 3-5 Processing the CANCEL command

```
CancelJobsInTheList:
   address ISPEXEC "tbtop lpjobtbl"

   JobCnt = 0
   call InitializeProgressBar

   do forever
      address ISPEXEC "tbskip lpjobtbl"
      if RC > 0 then leave
```

```
   if jtjname <> "Excluded" then do
      address SDSF "isfact o token('"jttoken"') parm(np p)"
      if RC <> 0 then do
         say "RC" RC "returned from ISFACT O"
         call DisplaySdsfMessages
         end

      /*  Mark all canceled jobs as feedback to the user  */

      jtowner  = "Canceled"
      jtexsys  = ""
      jtjclass = ""
      jtsclass = ""
      jtforms  = ""
      jtdest   = ""
      jtmaxrc  = ""
      jtendate = ""
      jttoken  = ""

      address ISPEXEC "tbput lpjobtbl"

      JobCnt = JobCnt + 1
      call DisplayProgressBar "Cancel", JobCnt, UnexcludedRowCount
      end
   end

call TerminateProgressBar
ListMsg = "List009"

return 0
```

A progress bar displays while the EXEC runs. Each ISFACT request is a separate SDSF operation, and SDSF has to completely initialize itself, including acquiring a console to process the generated commands, perform the action, and terminate itself. This process can take longer than the user expects, especially if there is a large number of rows.

Example 3-6 shows how the window that holds the progress bar is placed on the job list panel. The progress bar panel is member progress in the panel library and includes eight lines with a maximum width of 52 columns. The routine calculates the placement of the window so that it is centered at the bottom of the display regardless of the model of 3270 that you are emulating. The `addpop` service ensures that all future display requests are placed in the window area which is located by specifying row and column on the service invocation.

Example 3-6 Initializing the progress bar

```
InitializeProgressBar:
   /*  Add the popup window centered at the bottom with the bottom
       line unused.  &zscreend is the number of lines on the screen
       and zscreenw is the number of columns  */

   address ISPEXEC "vget (zscreenw zscreend)"

   IPB_RowsToAdd = zscreend - 12
   IPB_ColsToAdd = ( zscreenw - 52 ) / 2 - 15
   if IPB_ColsToAdd < 0 then
      IPB_ColsToAdd = 0

   zwinttl = ""

   address ISPEXEC "addpop poploc(zcmd)"         ,
                          "row("IPB_RowsToAdd")" ,
                          "column("IPB_ColsToAdd")"
   if RC <> 0 then do
      call xsay "RC" RC "adding the pop-up window"
      return 1
      end

   return
```

Displaying the progress bar consists of simply calculating the number of pips in the actual bar, suppressing user input, and displaying the progress panel as shown in Example 3-7. The **`control display lock`** setting causes the display service to return immediately as though the user had pressed the Enter key. From the user's perspective, the bar is updated continuously as the operation progresses without requiring any action.

Example 3-7 Displaying the progress bar

```
DisplayProgressBar:
   arg PBCommnd, PBCount, PBTotal

   /*  Prevent user input during progress bar display  */

   address ISPEXEC "control display lock"
   if RC <> 0 then do
```

```
   call xsay "RC" RC "locking the display"
   return 1
   end

/*  One "zit" for every 2% increase in completion  */

DPB_NumberOfZits = trunc( ( 100 * PBCount / PBTotal + .5 ) / 2 )

Progbar1 = copies( "@", DPB_NumberOfZits )
Progbar2 = Progbar1
Progbar3 = Progbar1

/*  Display the progress bar.  Because the display is locked, control
    will return immediately  */

address ISPEXEC "display panel(progress)"
if RC <> 0 then do
   call xsay "RC" RC "displaying the progress bar"
   return 1
   end

return
```

3.3 Processing the EXECUTE command

Example 3-8 shows how the EXECUTE command is processed. Command syntax is enforced on lines 782 through 790, and a temporary data set is allocated for BATCH mode processing on lines 794 to 804.

Example 3-8 EXECUTE command processing

```
0778 ExecuteExec:
0779    arg ExecuteVerb CmdName BatchMode Extraneous
0780    /*  Verify the command was EXECUTE <command> [BATCH]  */
0781
0782    if Extraneous <> "" then do
0783       ListMsg = "List000I"
0784       return 1
0785       end
0786
0787    if BatchMode <> "" & BatchMode <> "BATCH" then do
0788       ListMsg = "List000I"
0789       return 1
```

```
   end

/*  Batch mode requires a temporary dataset  */

if BatchMode = "BATCH" then do
   parse value time( "N" ) with Hour ":" Minute ":" Second
   DDname = "##" || Hour || Minute || Second

   address TSO "alloc f("DDname") space(10 10) track" ,
                                 "lrecl(255) recfm(v b)"
   if RC <> 0 then do
      say "Brother, are you hosed!"
      exit 4
      end
   end

/*  Now scan the table to find all the rows to process  */

address ISPEXEC "tbtop lpjobtbl"

JobCnt = 0

do forever
   address ISPEXEC "tbskip lpjobtbl"
   if RC > 0 then leave

   if jtjname <> "Excluded" then do
      /*  This is a row to process.  Build the argument string  */

      ArgString = "JOBNAME("ArgValue( jtjname )")" ,
                  "JOBID("ArgValue( jtjobid )")"   ,
                  "OWNERID("ArgValue( jtowner )")" ,
                  "EXEC("ArgValue( jtexsys )")"    ,
                  "JOBCLS("ArgValue( jtjclass )")" ,
                  "OUTCLS("ArgValue( jtsclass )")" ,
                  "FORMS("ArgValue( jtforms )")"   ,
                  "DEST("ArgValue( jtdest )")"     ,
                  "MAXCOND("ArgValue( jtmaxrc )")" ,
                  "ENDDATE("ArgValue( jtendate )")"

      /*  Batch mode writes the argument to the batch file; otherwise
          we execute the CLIST/EXEC for the argument now  */

      if BatchMode = "BATCH" then do
         queue ArgString
```

```
0835            address TSO "execio 1 diskw" DDname
0836            end
0837         else
0838            CmdName ArgString
0839
0840         JobCnt = JobCnt + 1
0841         end
0842      end
0843
0844   /*  Batch mode closes the batch file and executes the CLIST/EXEC with
0845       the DDNAME as the argument  */
0846
0847   if BatchMode = "BATCH" then do
0848      address TSO "execio 0 diskw" DDname "(finis"
0849
0850      address TSO CmdName DDname
0851
0852      address TSO "free f("DDname")"
0853      end
0854
0855   ListMsg = "List000J"
0856
0857   return 0
```

The ISPF table that includes the selected rows is scanned on lines 812 through 842. The argument is built on lines 819 to 828 using an internal routine. Every column value is enclosed in apostrophes to ensure it can be parsed in a simple fashion. The argument is built using a subroutine, **ArgValue**, which makes sure that any apostrophe (‘) inside the value is doubled (‘ ‘) so that REXX parses it correctly. When the value is built, the CLIST is executed on line 838 if not BATCH mode or the argument is written to the temporary data set for BATCH mode on lines 834 and 835. Finally, for BATCH mode, the CLIST is executed on lines 847 through 853.

3.3.1 A sample CLIST

Example 3-9 presents a modest example of a BATCH mode REXX exec. The EXEC reads all the argument strings and looks for restart JCL for the job in data set <HLQ>.MASTER.JCL. If found, the JCL is submitted to restart the job. The EXEC remembers all jobs that have restarted and will not restart a second job by the same name. Whether found or not, a small report is written to SYSOUT to document the actions taken by the EXEC.

Now, in fact, the EXEC has no idea whether the JCL restarts the job or, more fundamentally, whether the member is JCL or not. These niceties could be added to fill this example out to an actual product. That out of the way, let's see how the EXEC performs its job.

The report file is allocated first and a header written to it. Then the argument strings are read from the DDNAME supplied in the argument. Each argument line is processed in the loop from lines 56 to 126. The argument is decoded in lines 61 through 78. This decoding scheme, in particular the parse statement on line 62, will not handle all possible input and needs to be beefed up but will suffice for this sample. We decide whether the job has been submitted already on lines 82 through 88 and bypass the submission if so. We verify that there is JCL in the resubmission data set on lines and submit the job on lines 99 through 102. We trap the output lines from the **submit** command to find the submitted JOBID so we can place it into the report line. The rest of the EXEC detects and processes exception conditions.

Example 3-9 A sample BATCH mode CLIST

```
033 arg DDname
034
035 /*  Allocate a SYSOUT dataset for a report and print the header  */
036
037 "alloc f(#@rpt$#@) sysout(c) reu"
038 queue "---------------- RESUBMIT run on" date( "S" ) "at" time( "N" )
039 queue
040 queue "Jobname  Jobid    Max-RC     Status          NewJobid"
041 queue "-------- -------- ---------- --------------- --------"
042 "execio" queued() "diskw #@rpt$#@"
043
044 /*  Read the batch processing control file  */
045
046 "execio * diskr" DDname"(stem Input. finis"
047 if RC <> 0 then do
048    say "RC" RC "reading" DDname
049    exit 1
050    end
051
052 /*  Process each control record  */
053
054 Submitted. = ""
055
056 do RecX = 1 to Input.0
057    Statement = Input.RecX
058
059    /*  Parse the record to extract all the values  */
```

```
do while Statement <> ""
   parse var Statement Keyword "('" Value "')" Statement

   select
      when Keyword = "JOBNAME" then Jobname = Value
      when Keyword = "JOBID"   then JobID   = Value
      when Keyword = "OWNERID" then Ownerid = Value
      when Keyword = "EXEC"    then Exec    = Value
      when Keyword = "JOBCLS"  then Jobcls  = Value
      when Keyword = "OUTCLS"  then Outcls  = Value
      when Keyword = "FORMS"   then Forms   = Value
      when Keyword = "DEST"    then Dest    = Value
      when Keyword = "MAXCOND" then MaxCond = Value
      when Keyword = "ENDDATE" then EndDate = Value
      otherwise
         say "Keyword" Keyword "is unknown and will be ignored"
      end
   end

/*  Ensure a job is only submitted once per invocation  */

if Submitted.Jobname <> "" then do
   queue left( Jobname, 8 ) left( Jobid, 8 ) left( MaxCond, 10 ) ,
         left( "Duplicate", 15 ) copies( "-", 8 )
   "execio 1 diskw #@rpt$#@"
   iterate
   end
Submitted.Jobname = "YES"

/*  Attempt to find the JCL in the resubmission JCL library  */

call msg "off"
Opinion = sysdsn( "master.jcl("Jobname")" )
call msg "on"

/*  Submit the member if it exists  */

if Opinion = "OK" then do
   call outtrap "Msgs."
   "submit master.jcl("Jobname")"
   SubmitRC = RC
   call outtrap "off"

   /*  Examine the trapped messages and extract the new jobid  */
```

```
         NewJobid = "????????"
         do i = 1 to Msgs.0
            if word( Msgs.i, 1 ) = "JOB"       & ,
               word( Msgs.i, 3 ) = "SUBMITTED" then do
               parse var Msgs.i . "(" NewJobid ")" .
               leave i
               end
            end

         queue left( Jobname, 8 ) left( Jobid, 8 ) left( MaxCond, 10 ) ,
               left( "Submit RC" SubmitRC, 15 ) NewJobid
         end
      else if Opinion = "MEMBER NOT FOUND" then
         queue left( Jobname, 8 ) left( Jobid, 8 ) left( MaxCond, 10 ) ,
               left( "No resubmit JCL", 15 ) copies( "-", 8 )
      else
         queue left( Jobname, 8 ) left( Jobid, 8 ) left( MaxCond, 10 ) ,
               left( "SYSDSN failed", 15 ) copies( "-", 8 )

      "execio 1 diskw #@rpt$#@"
      end

/*  Close the report file and free it  */

"execio 0 diskw #@rpt$#@(finis"
"free f(#@rpt$#@)"
```

3.4 Future development

LISTPROC is an excellent example of how you can use the SDSF API to develop powerful utilities. In addition, you can improve LISTPROC to make it more useful in your environment:

1. You can support more columns by adding one or more panels that represent the scrolled display. Line count might be an especially useful column if you add filtering support for it. Supporting **`Scroll ===> CSR`** and allowing the user to scroll a partial screen's worth of columns is much more difficult and might not be worth the effort.
2. You can support more SDSF commands than O. You can clone LISTPROC and use it to display printers, initiators, JES resources and so forth.

3. You can add filters. Sensitivity to the creation date or FCB might be useful for your installation.
4. You can add support for the hold and release commands.
5. You can implement a command to write a SYSOUT data set to an DASD data set.

4

SDSF support for the COBOL language

Providing REXX programs with an API to access IBM z/OS System Display and Search Facility (SDSF) functions gives you a powerful tool for accessing and controlling JES resources. Although we chose REXX as the language for our installation, you might choose other languages, such as C, COBOL, or assembler.

This chapter presents a running interface that connects a COBOL application program to the REXX with SDSF API. The result is a way for a high-level language, in fact *any* high-level language, to make use of this powerful tool.

For information about how to download the programs in this scenario from the Web, see the instructions in Appendix B, “Additional material” on page 305.

4.1 Understanding the middleware between a REXX exec and another language

We wrote an assembler program, *REXDRIVR*, that acts as *middleware* between a REXX exec and a high-level language. As shown in Figure 4-1, REXDRIVR sits between REX4SDSF, a REXX exec that talks to SDSF through the API and *a high-level language program*, which is any program written in the language of you choice.

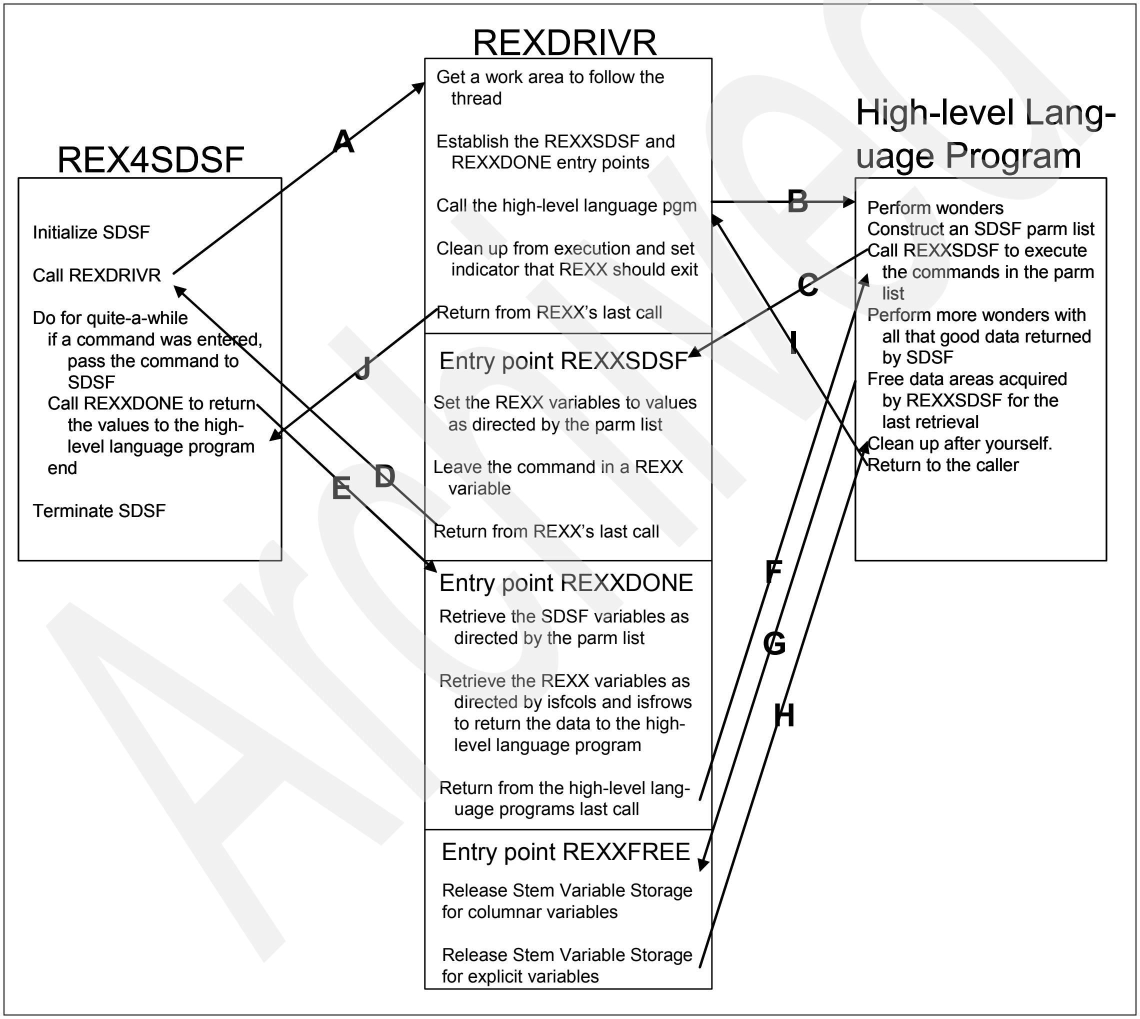


Figure 4-1 The REXDRIVR architecture

Although Figure 4-1 might be a bit overwhelming at first, to understand it you need to remember these three programs work together to accomplish the goals of interfacing (in this case) the COBOL program with SDSF. There is only a single thread of execution (execution is somewhere at all times and never at two places at any one time. REX4SDSF, REXDRIVR, and the C program have slightly different views of the world, including:

- REX4SDSF thinks it is the main program that controls execution. It believes that it is calling a REXX subroutine, REXDRIVR, to retrieve an SDSF command, that REXDRIVR has set the SDSF interface variables properly, and that all it has to do is call SDSF using the **`address SDSF`** interface.
- The COBOL program thinks it is the main program that controls execution. It believes that it is constructing an SDSF command in its parameter list and that all it has to do is call REXXSDSF to interface with SDSF.
- REXDRIVR is the only program that actually knows what is happening. It uses the COBOL program's parameter list to create and update REXX variables and passes control to REX4SDSF to actually drive the REXX with SDSF API. To do all this, REXDRIVR has to be creative in how it handles save areas and registers.

Figure 4-1 includes a series of lines with letters that represent how control passes among the three players. Here is how the thread of execution progresses:

1. REX4SDSF calls **`isfcalls`** to establish the SDSF environment. It gets the name of the COBOL program and any parameters from the command line arguments and calls REXDRIVR (line A).
2. REXDRIVR obtains work storage, IDENTIFYs the REXXSDSF, REXXDONE, and REXXFREE entry points, LOADs itself and the COBOL program into storage, and saves the address of the REXX save area in the work storage area. This address will come in handy in a little while. REXDRIVR then passes control to the COBOL program (line B).
3. The COBOL program performs whatever initialization it requires and constructs a parameter list to request an SDSF service. Note that this parameter list was created just for this example, and there is no reason why the parameters cannot be formatted differently. There is nothing in the SDSF API that requires parameters to be sent in this way. The COBOL program then calls REXXSDSF with the address of the parameter list as an argument to pass control to SDSF (line C).
4. REXXSDSF, a subroutine in the REXDRIVR program, retrieves the address of the work storage and saves the address of the COBOL program's work area. This address will come in handy in a little while. It also saves the address of the COBOL program's parameter list. You never know when you are going to need these things. REXXSDSF then updates the REXX with SDSF variables (**`isfcols`**, **`isfprefix`**, **`isffilter`**, and so forth) under control

of the COBOL program's parameter list and sets a REXX variable to the COBOL program's SDSF command. REXXSDSF then restores REX4SDSF's save area address and returns to REX4SDSF at the point immediately following where REX4SDSF called REXDRIVR (line D).

5. REX4SDSF issues the COBOL program's SDSF command and calls REXXDONE (line E).
6. REXXDONE, a subroutine in the REXDRIVR program, retrieves the address of the work storage and saves the address of REX4SDSF's save area (in the same place it did in item 2 above). It then restores the address of the COBOL program's parameter list and retrieves the values of the REXX with SDSF variables (such as **`isfmsg`**, **`isfcols`**, **`isfulog.x`**, and so forth) under control of the parameter list. It then creates a data area with all the stem variables returned by the SDSF API. It restores the address of the COBOL program's save area and returns to the COBOL program at the point where the COBOL program originally called REXXSDSF (line F).
7. The COBOL program processes the returned values and when done calls REXXFREE to free the data area (line G).
8. REXXFREE frees the data area acquired to hold the stem variables describing the columnar variables and the stem variables explicitly named in the COBOL program's parameter list and returns (line H).
9. The COBOL program completes its cleanup processing and returns (line I).
10. REXDRIVR gets control, frees the work area, restores the address of REX4SDSF's save area, and returns to where REX4SDSF last called REXXDONE (line I).

4.2 The pieces of REXDRIVR and how they work together

To understand how the process works, we need to consider the underlying REXX exec, *REX4SDSF*, and the assembler interface program, REXDRIVR.

4.2.1 The REX4SDSF exec

Example 4-1 shows the driver program, which is the only REXX code that is required for the high-level language interface. The command line arguments are the name of the program optionally followed by arguments to pass in.

REX4SDSF begins by initializing the SDSF environment on line 12. It then calls the REXDRIVR program to invoke the COBOL program at line 20. When control resumes at the next line, it is not because REXDRIVR is returning (that happens when the COBOL program has completed execution), but rather because the COBOL program has called REXXSDSF with an SDSF command or action.

REX4SDSF knows this because the REXXSDSF logic sets a variable, **`R4S_Request`**, to `COMMAND` to tell REX4SDSF to execute a command. In addition, REXXSDSF has taken the COBOL program's command and placed it in variable **`R4S_Cmd`** so REX4SDSF just has to issue the command as shown on line 26. REX4SDSF now calls REXXDONE on line 30 to pass the data back to the COBOL program. Control returns from this call in quite a while.

First REXXDONE passes the results of the call back to the COBOL program. The COBOL program processes it and calls REXXSDSF with another request. Only then does REX4SDSF get control to execute the new command. This process continues, with REX4SDSF staying in the while loop, until the COBOL program completes. At this time COBOL returns to the REXDRIVR program, which cleans up and does not set **`R4S_Request`**. The final return to the call at line 30 happens and control falls out of the loop because the test is unsatisfied. REX4SDSF then terminates the SDSF environment at line 35 and exits.

Example 4-1 The REX4SDSF driver program

```
001 /*  REXX  *************************************************************
002  *                                                                    *
003  *  rex4sdsf <C-program-name> <assembler-program-parms>               *
004  *                                                                    *
005  **********************************************************************/
006
007 parse arg PgmName PgmParms
008 PgmName = translate( PgmName )
009
010 /*  Initialize the SDSF environment  */
011
012 IsfRC = isfcalls( "ON" )
013 if IsfRC <> 0 then do
014    say "RC" IsfRC "returned from isfcalls( ON )"
015    exit IsfRC
016    end
017
018 /*  Call rexdrivr to pass control to the assembler program  */
019
020 call rexdrivr PgmName, PgmParms
021
022 /*  Pass commands to SDSF while the assembler program keeps on returning
023     with more SDSF commands to execute.  */
024
```

```
do while R4S_Request = "COMMAND"
   address SDSF R2S_Cmd
   IsfRC = RC

   R4S_Request = ""
   call rexxdone
   end

/*  Unload the SDSF environment  */

call isfcalls "OFF"

exit 0
```

4.3 The REXDRIVR interface program

REXDRIVR is written in assembler language, which might be a little unfamiliar to some of you. It is written (in part) as a REXX function. However, the concepts are reasonably straightforward, and the REXX function interface is well documented in *IBM z/OS TSO/E REXX Reference*, SA22-7790.

As shown in Figure 4-1 on page 124, REXDRIVR is divided into four sections:

- There is entry point *REXDRIVR* that is called by REXX as though it were a function.
- Next there is entry point *REXXSDSF* that is called by the COBOL program to execute an SDSF call.
- Then there is entry point *REXXDONE* that is called by REX4SDSF to return the results of the executed SDSF command.
- Finally there is entry point *REXXFREE* to free the storage that is acquired to hold variables that are returned by SDSF for processing by the COBOL program.

4.3.1 Entry point REXDRIVR - REX4SDSF function processor

The REXX functions that you normally use are those built into the language, such as `substr`, `date`, and `outtrap`. However, REXX provides a way for you to construct your own functions that you then use in the same way as the REXX functions. You can then call your functions using the **`call`** statement, which does not return a value, or by specifying the function name with arguments immediately following in parentheses, which is replaced by the returned value.

You can write your own functions in REXX and put them into a library in your SYSPROC or SYSEXEC concatenation. However, you can also write your own functions in assembler language and put them into a data set that is in your load library search order.

But why would you want to write your own functions in assembler language? Because by using the function interface, you can access all of the REXX variables through the REXX variable access routines, which means you can retrieve the values of REXX variables and create or update variables with the values that you choose. This function makes the interface natural and intuitive for the REXX programmer. Then, when you have control, all MVS services are available to your program rather than the subset of services that are available to REXX programs. In this case, assembler language provides the layer that connects REXX and COBOL.

The *REXX Reference* manual includes a chapter, TSO/E REXX Programming Services, that discusses how to write REXX functions in assembler language. We discuss a few of the concepts in this section, but we highly recommend that you read the *REXX Reference* manual to get a thorough explanation of how it all works.

Example 4-2 is an extract of the actual REXDRIVR program that we have shortened to simplify the explanation. The code begins with setting up the registers to point to the REXX interface areas on lines 8 through 11.

Example 4-2 The REXDRIVR program (abridged)

```
0001 REXDRIVR CSECT
0002 REXDRIVR AMODE 31
0003 REXDRIVR RMODE ANY
0004          STM   R14,R12,12(R13)         CAN'T USE LINKAGE STACK HERE
0005          LR    R12,R15
0006          USING REXDRIVR,R12
0007
0008          LR    R11,R0                  ->REXX ENVIRONMENT BLOCK
0009          USING ENVBLOCK,R11
0010          LR    R10,R1                  ->PARAMETER LIST
0011          USING EFPL,R10
0012
0013          STORAGE OBTAIN,LENGTH=W$LENGTH
0014          ST    R1,8(,R13)              CHAIN NEW SAVE AREA
0015          ST    R13,4(,R1)
0016          LR    R13,R1
0017          USING WORK,R13
0018
0019          L     R9,EFPLARG              ->ARGUMENT VECTOR
```

```
         USING ARGTABLE_ENTRY,R9
         CLC   ARGTABLE_ENTRY,=8X'FF'   CHECK FOR NO ARGUMENTS
         BE    INVARG                      AND GO IF NOT VALID
         L     R2,ARGTABLE_ARGSTRING_PTR   GET ->ARGUMENT
         L     R3,ARGTABLE_ARGSTRING_LENGTH   AND ITS LENGTH

         CHI   R3,8                     CHECK FOR ROUTINE NAME TOO
         BNH   ARG1OK                      LONG AND GO IF NOT
INVARG   DS    0H
         L     R15,EFPLEVAL             ->->EVALUATION BLOCK
         L     R15,0(,R15)              ->EVALUATION BLOCK
         USING EVALBLOCK,R15

         MVC   EVALBLOCK_EVLEN,=F'1'    SET THE RETURN CODE TO "8"
         MVC   EVALBLOCK_EVDATA(1),=C'8'

         DROP  R15
         B     EXIT0                       AND TERMINATE
ARG1OK   DS    0H
         LA    R0,W$PGMNAM              MOVE THE PROGRAM NAME FROM
         LA    R1,L'W$PGMNAM               THE PARM LIST
         ICM   R3,8,=C' '               BLANK PADDING
         MVCL  R0,R2

         XC    W$PGMPRM(2),W$PGMPRM     INDICATE NO PARMS
         CLC   ARGTABLE_NEXT(8),=8X'FF' CHECK IF THERE ARE PROGRAM
         BE    NOPARMS                     PARMS AND SKIP IF NOT

NEXT     USING ARGTABLE_ENTRY,ARGTABLE_NEXT
         L     R2,NEXT.ARGTABLE_ARGSTRING_PTR
         L     R3,NEXT.ARGTABLE_ARGSTRING_LENGTH
         DROP  NEXT,R9
         CHI   R3,100                   CHECK FOR PARMS TOO LONG AND
         BH    INVARG                      GO ABORT IF SO
         STH   R3,W$PGMPRM              SAVE PARM LENGTH
         LA    R0,W$PGMPRM+2            MOVE PARM TO HOLDING AREA
         LA    R1,100
         ICM   R3,8,=C' '               BLANK PADDING
         MVCL  R0,R2
NOPARMS  DS    0H

*        LOAD OURSELVES TO LOCK THIS PROGRAM IN MEMORY.  WE WILL LOSE
*        CONTROL PERIODICALLY AND DON'T NEED TO INCUR THE OVERHEAD OF
*        RELOADING.

```

```
         LOAD  EP=REXDRIVR             LOAD OURSELVES

*        GET A SPECIAL WORK AREA TO FOLLOW THE THREAD

         STORAGE OBTAIN,LENGTH=P$LENGTH,LOC=ANY
         LR    R9,R1                   ->PERSISTENT WORKAREA
         LR    R14,R9                  CLEAR IT TO BINARY ZEROES
         LHI   R15,P$LENGTH
         SLR   R1,R1
         MVCL  R14,R0
         USING PERSISTW,R9
         MVC   P$IBALL,=C'REXXSDSF'

*        WE SAVE THE REGISTERS OF THE REXX DRIVER SO WE CAN RETURN
*        WHEN THE APPLICATION PROGRAM MAKES AN INTERFACE REQUEST

         L     R15,4(,R13)             ->ORIGINAL RSA
         ST    R15,P$REXSAV            SAVE IN PERSISTENT W/A

*        INITIALIZE THE WORK AREA IN PREPARATION FOR CALLING THE REXX
*        VARIABLE INTERFACE ROUTINE.

         LA    R0,P$PARM_IBALL
         SLR   R1,R1
         SLR   R2,R2
         LA    R3,P$REQ_BLK
         LA    R4,P$ENV_BLK
         STM   R0,R4,P$PARM_LIST
         OI    P$PARM_LIST+16,X'80'
         OI    P$ENV_BLK,X'80'
         MVC   P$PARM_IBALL,=CL8'IRXEXCOM'
         ST    R11,P$ENV_BLK

*        SAVE SOME REXX POINTERS WE NEED TO SET AND RETRIEVE VARIABLES

         ST    R11,P$RXENVB            ->REXX ENVIRONMENT BLOCK
         ST    R10,P$RXEVAL            ->EVALUATION BLOCK

*        BUILD A TOKEN WITH AN EYECATCHER FOLLOWED BY THE ADDRESS OF
*        THE WORK AREA AND REGISTER IT WITH MVS.

         LA    R2,W$NTLVL              -> LEVEL
         MVC   W$NTLVL,=F'1'           SET TASK LEVEL
         LA    R3,W$NTNAME             -> NAME
         MVC   W$NTNAME,=CL16'REXX 4 SDSF API ' UNIQUE IDENTIFIER
```

```
         MVC   W$NTTOKN(12),=CL12'REXX I/F ==>'
         ST    R9,W$NTTOKN+12
         LA    R2,W$NTLVL               -> LEVEL
         LA    R3,W$NTNAME              -> NAME
         LA    R4,W$NTTOKN              -> TOKEN
         LA    R5,W$NTZERO              -> HOT ZERO
         XC    W$NTZERO,W$NTZERO        SET IT
         LA    R6,W$NTRC                -> RETURN CODE FEEDBACK AREA
         STM   R2,R6,W$NTPARM           SET UP P/L
         LA    R1,W$NTPARM              -> P/L
         L     R15,16                   ->CVT
         L     R15,X'220'(,R15)         ->CALLABLE SERVICE REQ TBL
         L     R15,X'14'(,R15)          ->NAME/TOKEN SERVICES VECTOR
         L     R15,X'04'(,R15)          ->IEANTCR
         BALR  R14,R15
         LTR   R15,R15                  GO IF SUCCESSFUL CREATION
         BZ    NTCROK
         ABEND 101                      FATAL ERROR
NTCROK   DS    OH
*        DEFINE ONE ENTRY POINT FOR THE APPLICATION PROGRAM TO REQUEST
*        SDSF SERVICES, A SECOND FOR THE REXX DRIVER PROGRAM TO CALL
*        WHEN THE REQUEST HAS BEEN SERVICED AND A THIRD FOR THE
*        APPLICATION TO CALL TO FREE THE DATA AREA.
         LA    R1,REXXSDSF
         IDENTIFY EP=REXXSDSF,ENTRY=(1)
         LA    R1,REXXDONE
         IDENTIFY EP=REXXDONE,ENTRY=(1)
         LA    R1,REXXFREE
         IDENTIFY EP=REXXFREE,ENTRY=(1)
*        TIME TO GO TO THE APPLICATION PROGRAM
         LA    R1,W$PGMPRM              ->PARM LIST
         ST    R1,W$PGMPTR
         OI    W$PGMPTR,X'80'           SET VL BIT
         LA    R1,W$PGMPTR
         LA    R2,W$PGMNAM              ->NAME OF THE APPLICATION
         MVC   W$LINK(W$LINKL),LINKMFL
         LINK  EPLOC=(R2),SF=(E,W$LINK)
```

```
*        DELETE OURSELVES TO UNLOCK THE PROGRAM IN MEMORY.

         DELETE EP=REXDRIVR            HASTA LA VISTA, BABY

*  THE APPLICATION PROGRAM IS DONE.  GET RID OF THE WORK AREA AND THE
*  NAME/TOKEN PAIR AND RETURN TO THE REXX DRIVER PROGRAM.

         MVC   W$NTLVL,=F'1'           SET TASK LEVEL
         MVC   W$NTNAME,=CL16'REXX 4 SDSF API ' UNIQUE IDENTIFIER

         LA    R0,W$NTLVL              ->LEVEL
         LA    R1,W$NTNAME             ->NAME
         LA    R2,W$NTRC               ->RETURN CODE
         STM   R0,R2,W$NTPARM          SET UP P/L
         OI    W$NTPARM+8,X'80'
         LA    R1,W$NTPARM             ->P/L
         L     R15,16                  ->CVT
         L     R15,X'220'(,R15)        ->CALLABLE SERVICE TABLE
         L     R15,X'14'(,R15)         ->NAME/TOKEN SERVICES VECTOR
         L     R15,X'0C'(,R15)         ->IEANTDL ROUTINE
         CALL  (15)                    DELETE THE NAME/TOKEN PAIR

         L     R3,P$REXSAV             GET ->LATEST REXX RSA
         STORAGE RELEASE,ADDR=(R9),LENGTH=P$LENGTH
         L     R15,EFPLEVAL            ->->EVALUATION BLOCK
         L     R15,0(,R15)             ->EVALUATION BLOCK
         USING EVALBLOCK,R15

         MVC   EVALBLOCK_EVLEN,=F'2'   SET THE RETURN CODE TO "OK"
         MVC   EVALBLOCK_EVDATA(2),=C'OK'
         DROP  R15

EXIT0    DS    0H
         LR    R2,R13                  SAVE ->OUR SAVE AREA
         LR    R13,R3                  ->LATEST REXX SAVE AREA
         STORAGE RELEASE,ADDR=(R2),LENGTH=W$LENGTH
         LM    R14,R12,12(R13)         RESTORE THE ORIGINAL REGISTERS
         SLR   R15,R15                 PEACHY KEEN
         BR    R14                     BACK TO REXDRIVR

LINKMFL  LINK  EPLOC=0,SF=L
         LTORG ,
         DROP  R9,R12,R13
```

The environment on entry to the function processor

When REXX calls its functions, it passes the addresses of two control blocks. The *environment block*, pointed to by R0 at entry, represents the current command processing environment (the one set using the **address** statement) and is an important block because it must be passed to every REXX service routine your program calls. The environment block also includes the address of the Vector of External Entry Points and block that includes the address of REXX service routines your function processor can use. You call one of these routines, IRXEXCOM, to set and retrieve REXX variables. You can call another, IRXSAY, to write messages to the same destination as messages written using **say** in your REXX programs. IRXSAY is used in the program to write debugging messages but was not shown here to simplify the discussion. The parameter list, pointed to by R1 at entry, contains the addresses of all arguments to the function as well as the address of the *evaluation block* which is where you leave the return code from the function. Lines 19 through 59 show how the arguments are retrieved from the REXX program.

Driver logic

On line 65, we use the MVS LOAD macro to increase the use count of REXDRIVR. If we do not do this, REXDRIVR could appear to be free when REX4SDSF regains control following the COBOL program's first interface call and REXX might free the storage. This won't be necessary if the application program issues a LOAD for the REXXSDSF entry point but putting the LOAD here gives us one less thing to worry about. This LOAD will be in effect until the DELETE on line 158 which is only executed after the COBOL program completes execution.

We acquire a work area on line 69, which we call the *persistent work area*, to follow the execution thread and we then use the name/token callable service to have MVS hold the name and token for us on lines 106 to 130. The name is a unique 16 byte identifier associated with the 16-byte token. We set the token to point to the work area and use a name of *REXX 4 SDSF API*. The name/token service saves the token and remembers the name we assigned it. On subsequent entries to the routines inside REXDRIVR, we will be able to retrieve the token by using the name and thus keep track of the work area across all the devious weaving of the thread. Setting a level of 1 makes the name known to all programs running under the same TCB but programs running under different TCBs in the same address space will not be able to retrieve the token. This was done in a belief that the process should all run under the same task and that spanning tasks could be a problem for I/O.

The save areas

On lines 81 and 82 we get the address of REX4SDSF's save area and save it in the persistent work area. When REX4SDSF calls REXDRIVR it passes information in the registers and in the contents of its control blocks which constitute the entire interface with the function processor. When REX4SDSF receives control, it expects the registers and control block contents to be unchanged except as defined in the interface. As long as the content is unchanged, REX4SDSF has no way of knowing, and really no interest in knowing what the function processor has done. Because one routine in the load module receives control from REX4SDSF and a different routine returns control to it is irrelevant.

Similarly, when the COBOL program calls the interface with an SDSF request, the entire interface is embodied in the registers and contents of the control blocks. It has no way of knowing, nor interest in finding out, just what has transpired in the interim. REXDRIVR uses these mutual incuriosities to its advantage. You can save both REXX's and COBOL's register save area addresses as a means of accomplishing this end.

We use the IDENTIFY macros on lines 138, 141, and 144 to make the three other routines known to MVS and to allow them to be called by the COBOL program as needed.

We formally pass control to the COBOL routine using the LINK macro on lines 148 to 154. Control will not return until the COBOL program has completed its work and exits.

When control eventually returns from the COBOL program, all SDSF accessing has been completed. It is now safe to DELETE the REXDRIVR program (it is still held in storage by the last call to REXXDONE from REX4SDSF as will be described later), and we do so on line 158. The name/token pair is deleted on lines 163 through 176. We pick up the address of the register save area last passed to us at REXXDONE and free the persistent work area at lines 178 and 179. Lines 184 and 185 show how a return code is passed back to REXX. The current save area is freed on lines 189 to 191 but r13 is set to the address we picked up on line 178 rather than from the back pointer from the save area we just freed. Setting R/C zero at line 193 convinces REXX that our function succeeded (non-zero implies a failure which will give us an `Incorrect call to routine` message), and we return from the last call to REXXDONE at line 194.

4.3.2 The Application Program's view of SDSF: The parameter list

The COBOL application program that receives control from REXDRIVR gets control as a main routine, not a subroutine. So, it gets the parameter just as though it had been invoked from JCL. In our example, the parameter is a message identifier to look for in SYSLOG and the parameter declaration is shown in Example 4-3.

Example 4-3 Parameter received by the COBOL program

```
linkage section.
  01 msgid.
     03 msgid-l    pic s9(4) usage is binary.
     03 msgid-txt  pic x(8).
...
...
...
procedure division using msgid.
```

The first half word of the parameter is the length in binary and is followed by the actual parameter character string.

The program needs to pass control to REXXSDSF each time that it wants to perform an SDSF API function. When it does so, it uses a parameter area formatted as shown in Figure 4-2.

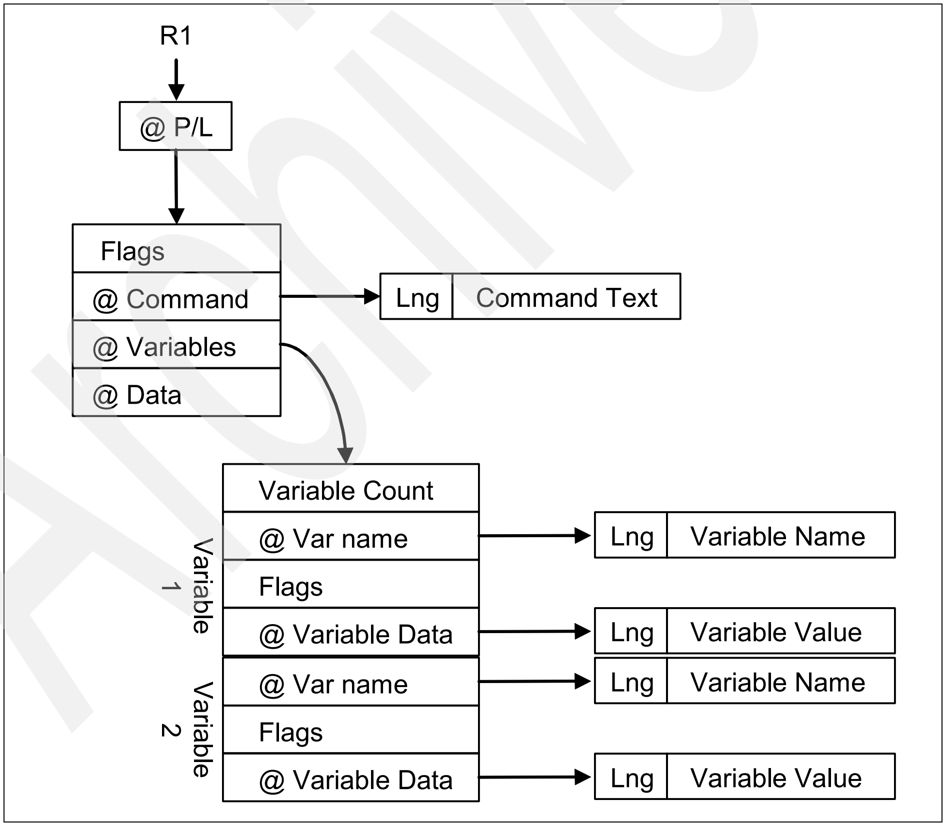


Figure 4-2 COBOL / SDSF Parameter Area (except for returned data and stem variables)

The flags include an indicator that data is returned (as is typical for ISFEXEC or a JDS request through ISFACT). The command address points to a 2-byte length, followed by the actual command text starting with ISFEXEC or ISFACT, and the variable address points to an area that includes a 4-byte count of REXX variables whose values are set or returned by REXXSDSF. The intent was to provide a means to set **isfprefix**, **isfowner**, and the other SDSF-related variables. However, you can set any REXX variable to allow you to communicate with the REX4SDSF EXEC if desired.

The descriptors for each variable immediately follow the count. The variable Flags field describes whether the variable's value is to be set or returned. To both set and return you need to have two entries in the variable list. The data address is ignored when REXXSDSF gets control from the application program and will be set to the address of a stem variable data feedback area when control returns to the application.

Figure 4-3 illustrates how the program can retrieve stem variables other than variables that are set to represent columnar data in the virtual tabular display. The program would use this interface to retrieve variables such as **isfmsg2** and **isfulog**. The second flag byte of the variable descriptor (which is pointed to by the parameter list) is `S` to indicate that this is a stem variable retrieval. The first flag byte is `R` to indicate retrieval because stems can only be retrieved, not set, using this interface. The variable data pointer is to the R4SS area that includes an eye catcher, total area length, the count of stems and the address of the value of each.

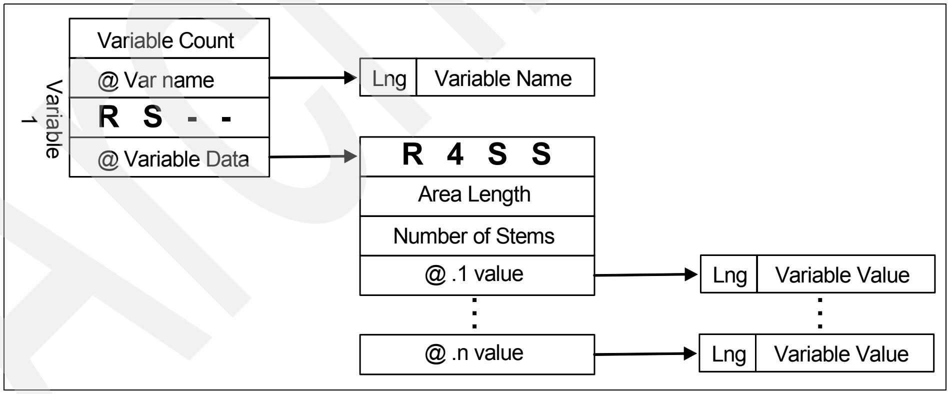


Figure 4-3 COBOL/SDSF - explicit stem variable retrieval

Figure 4-4 shows the format of the area that is returned to the application program. The area begins with a 4-character eye catcher and has the count of stem variables being returned. After the count is a vector of addresses to column descriptors, one for each stem. Each column descriptor includes the 8-character stem variable head (the part before the period), the number of rows (`isfrows`), and one address to the column data for each row. The column data is a 2-byte length, followed by the actual data as returned by SDSF.

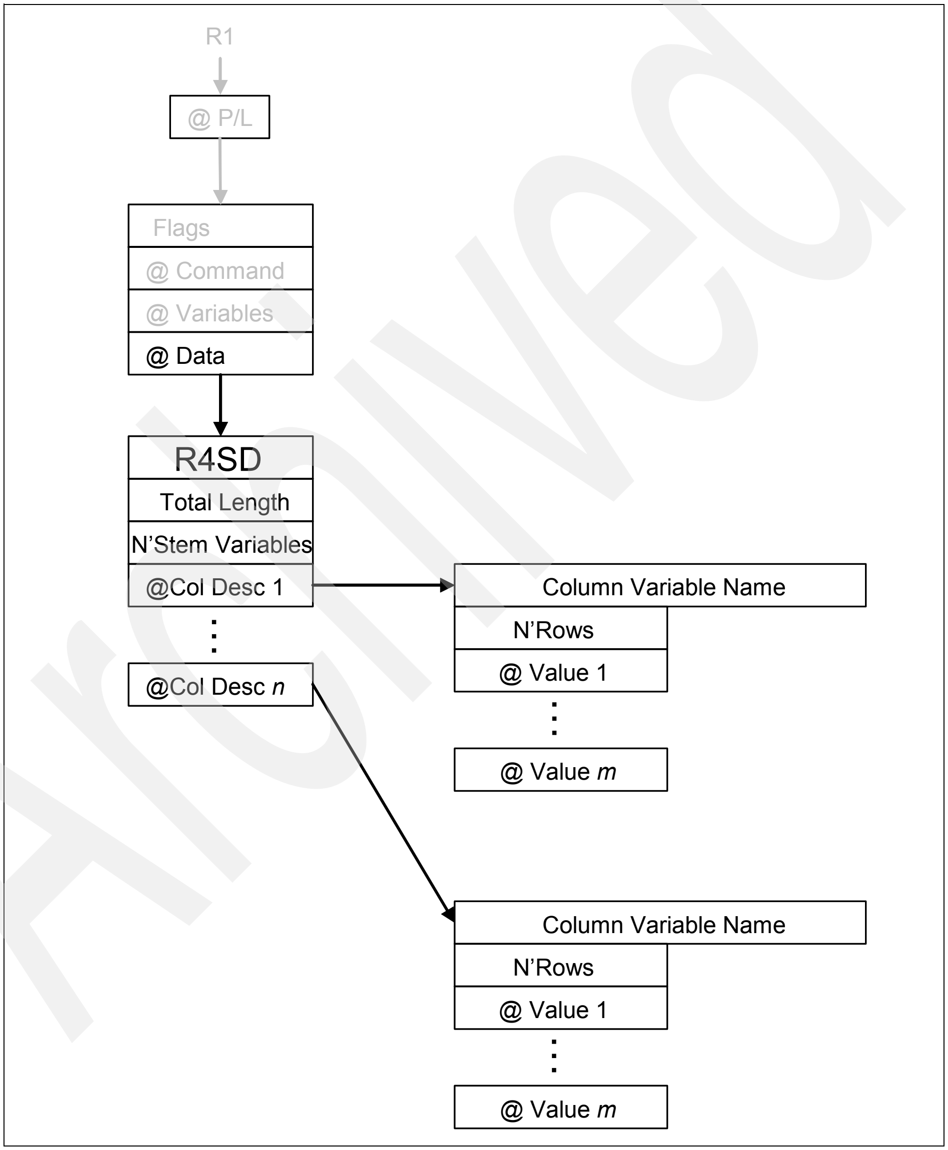


Figure 4-4 C / SDSF Parameter Area returned data

Both the R4SS and R4DD areas are obtained from subpool 0 and must be released back to subpool 0, but this is a non-trivial operation in the COBOL language. In fact, it is not possible. To support programs that, unlike children, clean up after themselves, REXDRIVR has an entry point, *REXXFREE*, that accepts the address of the parameter list and frees the data and variable areas storage for the application program.

4.3.3 Entry point REXXSDSF - Application program service routine

When the application program has received control from REXDRIVR, it needs to create the data area described in 4.3.2, “The Application Program’s view of SDSF: The parameter list”in a non-reentrant area and call entry point REXXSDSF with the parameter list address as an argument to perform SDSF functions. The application makes multiple calls as it traverses the spool data, each time updating the parameter list and processing the output. When done, it executes a return from the main routine, which resumes control in REXDRIVR.

Example 4-4 is an excerpt from the REXXSDSF routine, which we have shortened for clarity. This example gets a temporary save area that it chains to the application’s save area, but this chain is unusable. The application’s save area address is stored away in the persistent area awaiting completion of the SDSF request, and the save area is not used by REXXSDSF.

We retrieve the address of the token on lines 18 through 30 using the name and the IEANTRT routine (name/token retrieval). When we get the token, we pull the persistent area address from the fourth word on line 35.

Example 4-4 The REXXSDSF routine (abridged)

```
0001 REXXSDSF DS    0H
0002          STM   R14,R12,12(R13)          CAN'T USE LINKAGE STACK HERE
0003          LR    R12,R15
0004          USING REXXSDSF,R12
0005          L     R7,0(,R1)                ->INTERFACE AREA
0006          USING R4SAREA,R7
0007
0008 *        GET A SHORT-TERM WORK AREA
0009
0010          STORAGE OBTAIN,LENGTH=W$LENGTH
0011          ST    R1,8(,R13)
0012          ST    R13,4(,R1)
0013          LR    R13,R1
0014          USING WORK,R13
0015
0016 *        RETRIEVE THE WORK AREA ADDRESS
```

```

         LA    R2,W$NTLVL              -> LEVEL
         MVC   W$NTLVL,=F'1'           SET TASK LEVEL
         LA    R3,W$NTNAME             -> NAME
         MVC   W$NTNAME,=CL16'REXX 4 SDSF API ' UNIQUE IDENTIFIER
         LA    R4,W$NTTOKN             -> TOKEN FEEDBACK AREA
         LA    R5,W$NTRC               -> RETURN CODE FEEDBACK AREA
         STM   R2,R5,W$NTPARM
         LA    R1,W$NTPARM             -> P/L
         L     R15,16                  ->CVT
         L     R15,X'220'(,R15)        ->CALLABLE SERVICE REQ TBL
         L     R15,X'14'(,R15)         ->NAME/TOKEN SERVICES VECTOR
         L     R15,X'08'(,R15)         ->IEANTRT
         BALR  R14,R15
         OC    W$NTRC,W$NTRC           GO IF THE NAME WAS RESOLVED
         BZ    NTRESOLV
         ABEND 100                     UNRESOLVED NAME IS FATAL
NTRESOLV DS    0H
         L     R9,W$NTTOKN+12          GET -> PERSISTENT STORAGE
         USING PERSISTW,R9

         L     R11,P$RXENVB            ->REXX ENVIRONMENT BLOCK
         USING ENVBLOCK,R11
         L     R10,P$RXEVAL            ->EVALUATION BLOCK
         USING EVALBLOCK,R10
         ST    R7,P$PGMARG             SAVE -> APPL PGM'S ARG LIST

*        SET REXX VARIABLE R4S_CMD TO THE USER'S COMMAND

         LA    R0,=C'R4S_CMD'          VARIABLE NAME
         LA    R1,7                    L'VARIABLE NAME
         L     R3,R4SACMD              ->VALUE REFERENCE
         USING R4SREFER,R3
         LA    R2,R4SRDATA             ->VALUE
         LH    R3,R4SRLNG              L'VALUE
         DROP  R3
         L     R15,=A(SET_VARIABLE)
         BALR  R14,R15

*        SET ALL THE VARIABLES THE USER WANTS SET

         L     R6,R4SAVARS             ->VARIABLE BLOCK
         USING R4SVARS,R6
         L     R5,R4SVCNT              N'VARIABLES
         LA    R6,R4SVNAME             ->FIRST VARIABLE DESCRIPTOR
```

```
         USING R4SVNAME,R6
SETNEXT  DS    0H
         CLI   R4SVFLGS,R4SVFSET        IS IT A VARIABLE TO BE SET?
         BNE   NEXTDESC                 GO IF NOT

         L     R1,R4SVNAME              ->NAME DESCRIPTOR
         USING R4SREFER,R1
         LA    R0,R4SRDATA              ->NAME ITSELF
         LH    R1,R4SRLNG               L'NAME
         DROP  R1
         L     R3,R4SVVALU              ->VALUE DESCRIPTOR
         USING R4SREFER,R3
         LA    R2,R4SRDATA              ->VALUE ITSELF
         LH    R3,R4SRLNG               L'VALUE
         DROP  R3

         L     R15,=A(SET_VARIABLE)     GO SET THE VARIABLE
         BALR  R14,R15

NEXTDESC DS    0H
         AHI   R6,12                    ->NEXT VARIABLE ENTRY
         BCT   R5,SETNEXT               GO PROCESS IT
         DROP  R6

*        SET REXX VARIABLE R4S_REQUEST TO 'COMMAND' TO TELL REXDRIVR
*        WHAT TO DO

         LA    R0,=C'R4S_REQUEST'       VARIABLE NAME
         LA    R1,11                    L'VARIABLE NAME
         LA    R2,=C'COMMAND'           ->VALUE
         LA    R3,7                     L'VALUE
         L     R15,=A(SET_VARIABLE)
         BALR  R14,R15

*        SAVE THE REGISTERS IN THE PERSISTENT WORK AREA - THEY WILL
*        COME IN HANDY WHEN RETURNING TO THE APPLICATION PROGRAM -
*        AND FREE THE SHORT-TERM WORK AREA

         LR    R2,R13                   ->SHORT TERM W/A
         L     R3,4(,R13)               ->APPLICATION PROGRAM'S RSA
         ST    R3,P$PGMSAV              SAVE IN PERSISTENT W/A
         L     R13,P$REXSAV             GET REXX'S W/A BACK
         STORAGE RELEASE,ADDR=(R2),LENGTH=W$LENGTH  FREE SHORT TERM W/A

*        RESTORE REXDRIVR'S REGISTERS AND RETURN TO IT TO PROCESS THE
```

```
0107 *        APPLICATION PROGRAMS REQUEST
0108
0109          LM    R14,R12,12(R13)         RESTORE REX4SDSF'S REGISTERS
0110          SLR   R15,R15
0111          BR    R14
0112          LTORG
0113          DROP  R7,R9,R12,R13
```

Lines 46 to 54 get the user's command from the parameter list and create the REXX variable **`R4S_CMD`** from it for REX4SDSF.

Example 4-5 shows the **`SET_VARIABLE`** routine. It shows how REXX service IRXEXCOM is used to create REXX variables. **`SET_VARIABLE`** is called a second time on lines 89 through 94 to set **`R4S_REQUEST`** to `COMMAND` to tell REX4SDSF that it is being given control to process an SDSF request rather than being given control to handle the end of the application program's execution.

The logic on lines 58 through 84 examines all the entries in the variable list anchored of the parameter list and call `SET_VARIABLE` to create/update any variable whose flag bytes indicate that it needs to be set. We re-examine the entries looking for variables to be returned later after the SDSF action has been performed.

The application program's save area address is saved in the persistent work area on line 102 for use when the SDSF request has completed and the REXX save area is picked up on line 103. We free the temporary work area on line 104 and return to REX4SDSF.

Example 4-5 SET_VARIABLE routine

```
SET_VARIABLE DS    0H
          BAKR  R14,0
          LR    R12,R15
          USING SET_VARIABLE,R12
          USING PERSISTW,R9

RB        USING SHVBLOCK,P$REQ_BLK
          XC    RB.SHVBLOCK(SHVBLEN),RB.SHVBLOCK
          MVI   RB.SHVCODE,SHVSTORE     IND VARIABLE STORE OPERATION
          STM   R0,R1,RB.SHVNAMA        SAVE NAME ADDRESS AND LENGTH
          STM   R2,R3,RB.SHVVALA        SAVE VALUE ADDRESS AND LENGTH

          LA    R1,P$PARM_LIST
          L     R15,ENVBLOCK_IRXEXTE    ->REXX ROUTINE VECTORS
          USING IRXEXTE,R15
          L     R15,IRXEXCOM
```

```
         DROP  R15
         BALR  R14,R15
         ST    R15,P$RETCODE
         DROP  RB

         PR    ,
         DROP  R9,R12
         LTORG ,
```

4.4 Entry point REXXDONE - REX4SDSF completion routine

When SDSF returns to REX4SDSF after the command has been processed, REX4SDSF calls REXXDONE to process the response. It is in REXXDONE that the actual high-level language interface is implemented.

As shown in Figure 4-5 on page 146, REXXDONE saves the REXX save area address in anticipation of the next REXXSDSF call (or ultimate return of the application program) and picks up the application program's parameter list that was saved by the REXXSDSF routine. REXXDONE then analyzes the variable list to return any variables that were requested. (The parameter list and variable list are shown in Figure 4-2 on page 136.)

Example 4-6 shows the variable return logic. The logic distinguishes between normal variables and stem variables. From the standpoint of the user, a normal variable is returned in an area passed to REXXSDSF in the parameter list but a stem variable, whose total length is unknown until after the call has completed, is returned in storage obtained for the application. Thus, REXXDONE only has to move normal variables to the location contained in the parameter list but must calculate the total size of stem variable values and obtain storage for the user. Example 4-6 includes logic for normal variables and calls `RETURN_STEM` to process stem variables.

Example 4-6 REXXDONE variable return logic

```
         L     R6,R4SAVARS               ->VARIABLE BLOCK
         USING R4SVARS,R6
         L     R5,R4SVCNT                N'VARIABLES
         LA    R6,R4SVNAME               ->FIRST VARIABLE DESCRIPTOR
         USING R4SVNAME,R6
SETNEXT2 DS    0H
         CLI   R4SVFLGS,R4SVFRET         IS IT A VAR TO BE RETRIEVED?
         BNE   NEXTDSC2                  GO IF NOT
```

```
         CLI   R4SVTYPE,R4SVTSTM       IS THIS AN ISFXXX STEM VAR?
         BNE   NOTSTEM                 GO IF NOT
         LR    R1,R6                   ->R4SVARS
         L     R15,=A(RETURN_STEM)     GO RETURN THE STEM ARRAY
         BALR  R14,R15
         ST    R1,R4SVVALU             POINT VARIABLE ENTRY TO R4SSTEM
         B     NEXTDSC2                GO PROCESS NEXT VARIABLE ENTRY
NOTSTEM  DS    0H
         L     R1,R4SVNAME             ->NAME DESCRIPTOR
         USING R4SREFER,R1
         LA    R0,R4SRDATA             ->NAME ITSELF
         LH    R1,R4SRLNG              L'NAME
         DROP  R1
         L     R15,=A(RETRIEVE_VARIABLE) GO GET THE VARIABLE.  ON
         BALR  R14,R15                    RETURN, R0->VALUE, R1 HAS
*                                         ITS LENGTH
         L     R3,R4SVVALU             ->VALUE DESCRIPTOR
         USING R4SREFER,R3
         LA    R2,R4SRDATA             ->VALUE ITSELF
         LH    R3,R4SRLNG              L'VALUE
         DROP  R3
         ICM   R1,8,=C' '              BLANK PAD
         MVCL  R2,R0                   COPY VALUE TO THE AREA POINTED
*                                         TO IN THE INTERFACE AREA
NEXTDSC2 DS    0H
         AHI   R6,12                   ->NEXT VARIABLE ENTRY
         BCT   R5,SETNEXT2             GO PROCESS IT
```

Example 4-6 shows that routine `RETRIEVE_VARIABLE` is called for normal variable processing to get the variable's value from REXX. Example 4-7 shows the routine and is a straightforward implementation of the REXX variable access interface.

Example 4-7   RETRIEVE_VARIABLE routine in REXDRIVR

```
RETRIEVE_VARIABLE DS    0H
                 BAKR  R14,0
                 LR    R12,R15
                 USING RETRIEVE_VARIABLE,R12
                 USING PERSISTW,R9

RB               USING SHVBLOCK,P$REQ_BLK
                 XC    RB.SHVBLOCK(SHVBLEN),RB.SHVBLOCK
                 MVI   RB.SHVCODE,SHVFETCH      IND VAR RETRIEVE OPERATION
                 STM   R0,R1,RB.SHVNAMA         SAVE IN THE PARAMETER LIST

                 LA    R0,P$BUFFER              ->READ BUFFER
                 L     R1,=A(L'P$BUFFER)
                 STM   R0,R1,RB.SHVVALA         SAVE FOR RETRIEVAL
                 ST    R1,RB.SHVBUFL

                 LA    R1,P$PARM_LIST           ->RETRIEVE THE VALUE
                 L     R15,ENVBLOCK_IRXEXTE     ->REXX ROUTINE VECTORS
                 USING IRXEXTE,R15
                 L     R15,IRXEXCOM
                 DROP  R15
                 BALR  R14,R15
                 ST    R15,P$RETCODE            SAVE THE RETURN CODE
                 LM    R0,R1,RB.SHVVALA           AND THE RETURNED VALUE
                 STM   R0,R1,P$VALUE

                 DROP  RB

                 PR    ,
```

Figure 4-5 shows that REXXDONE determines the screen type and whether data was returned right after handling the variables. Determining the screen type is an important factor in returning data to the application and must be done before that process can occur.

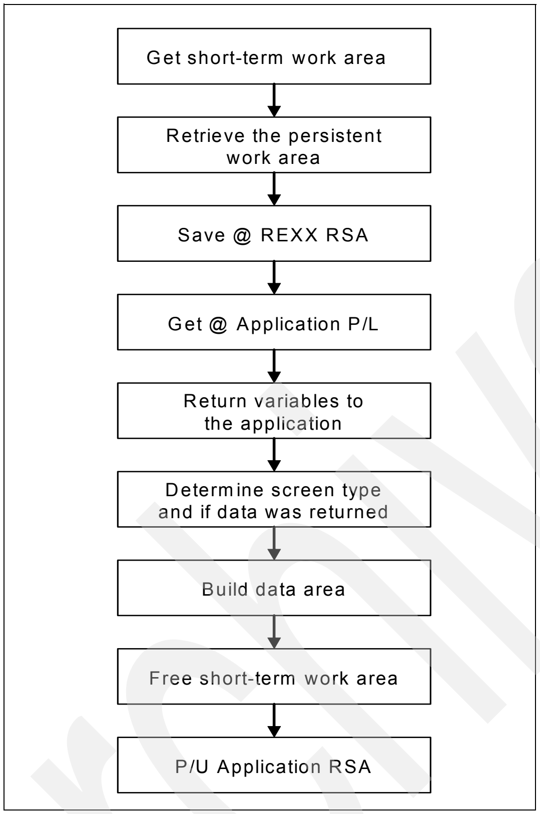


Figure 4-5 Overview of REXXDONE logic (abridged)

When REXDRIVR was written, if was decided to estimate the total variable size rather than retrieve all the variables twice (once to determine their length and again to actually retrieve the values). When SDSF builds the REXX variables to return the results of the SDSF ISFEXEC or ISFACT command, it trims trailing spaces before storing the values. This means that the value lengths can all vary between 0 and the maximum allowed for the variable. To write REXDRIVR, every SDSF panel was examined to determine the maximum length of the data in each column and **`colshelp`** was used to match the internal column names with the column titles. In the process it was discovered that some columns with the same internal name have different lengths on different panels. So, REXDRIVR has a significant amount of space dedicated to a list of column names and lengths organized by screen name. This logic depends on knowing the screen name.

We have not presented the code necessary to scan the command to extract the command name for this chapter because it is long but straightforward.

We use the contents of **`isfcols`** to obtain how many columns have been returned to us in the stem variables and the internal tables to convert this to the bytes necessary to hold all the information. We get the total length from the two most popular of the four basic mathematical operations and get storage using the STORAGE OBTAIN macro. This length is what is required to hold the variable values if every value were as long as could be. In practice, the actual value will be somewhat less.

After the data area is built, we pick up the application's register save area from where it was saved when REXXSDSF was called and return to the application.

4.4.1 Entry point REXXFREE - storage release routine

The last routine in REXDRIVR is REXXFREE, which releases storage dynamically acquired in REXXDONE. We do not show the code here because it is very straightforward. The data area is freed, followed by the stem variable storage areas if any. After the free is done, the parameter list pointers are zeroed.

4.5 The application programs included in the additional materials

Three programs were included in the additional materials to act as templates for you to write your own processing programs:

- ASMPGM, written in assembler
- CPROGRAM, written in C
- COBOLPGM, determining the language in which this last sample was written is left to the reader as an exercise

ASMPGM was written purely for testing the interface and, while functional, is unlikely to provide more than a shell in which you can lay your application. Similarly, CPROGRAM simply displays the results of the operations.

COBOLPGM, however, implements the scenario described in Chapter 9, "JOB schedule and control" on page 201, and might be more illustrative of how you can write an application using the REX4SDSF/REXDRIVR architecture. Similarly, CPROGRAM was written to exercise the interface and to verify that the Language Environment® does not interfere with the flow of control that is imposed by the REXDRIVR logic or does not engage in any potentially contentious behaviors with SDSF.

COBOLPGM, however, was the jewel of the sample world. It implemented the application described in Chapter 6, “Viewing SYSLOG” on page 163 and permitted us a view of how an algorithm implemented in REXX would compare with the same algorithm implemented in COBOL. The results were encouraging. Writing a series of SYSLOG data sets took .90 CPU seconds for the REXX solution but only .11 CPU seconds for COBOL. This difference is solely due to the improved performance of COBOL I/O (standard QSAM/BSAM, we would assume) over that of REXX (EXECIO).

4.6 The COBOL point of view

From the COBOL point of view, the trickiest thing is dealing with the interface variables. Calling the REXX interface is actually straightforward. Setting up the environment that is defined by the variables requires a little more coding and invoking a couple of Language Environment routines to acquire dynamic memory to store them and free it after (as shown in Example 4-8). These routines, CEEGTST and CEEFRST, are explained in Language Environment manuals.

Example 4-8 Acquiring dynamic memory to store variables

```
...
...
move 0 to heap-id.
call "CEEGTST" using heap-id,
                     files-table-size,
                     files-table-ptr,
                     feedback.
if CEE000 of feedback then
   set address of files-table to files-table-ptr
else
...
...
end-if.
```

You must also be methodical, calling REXXFREE after processing the values returned by REXXSDSF, lest you can run in trouble. Example 4-9 shows how this is accomplished.

Example 4-9 Calling REXXSDSF and REXXFREE routines

```
...
...
perform set-isfexec-vars.
perform call-rexxsdsf through call-rexxsdsf-exit.
set address of receiving-area to r4sadata.
perform load-jobs-table.
perform call-rexxfree through call-rexxfree-exit.
...
...
```

Figure 4-6 shows the complete flow chart . After calling REXXSDSF, the information is stored in a data structure. Because we do not know the size of the data in advance, we cannot define working-storage space for it. Thus, we acquire dynamic memory and copy the data there. When the data is copied, the resources acquired by REXXSDSF can be freed by REXXFREE.

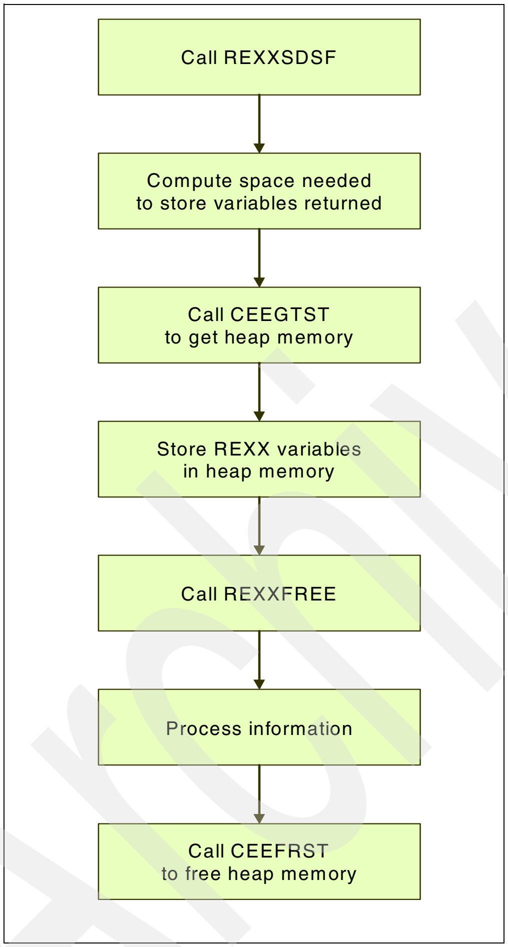


Figure 4-6 Flowchart: Calling REXXSDSF and REXXFREE from COBOL

4.7 Improving the interface

While results from the testing suggested that processing production quantities would be better done in COBOL than REXX, there is always room for improvement. Aesthetically, the interface seems a little grainy to us, requiring the application program to be far more aware of the nitty-gritty of physical reality that is normal in the COBOL world. Perhaps what is needed is a change in metaphor, a way to separate COBOL and SDSF with another layer of abstraction.

It would simplify the COBOL programmer's life, and make the programmer a more productive person, if retrieval requests could be made in more traditional COBOL terms rather than in the assembler terms the interface now requires. It would be possible, for instance, for the programmer to request a status panel and pass a 2-dimensional table to be filled by the interface. The columns would include values for the individual fields, and the rows would represent the individual jobs. The table would be of a fixed size, and the interface could be called repeatedly to fill it until all SDSF-returned data had been passed. Alternatively, perhaps the interface could be similar to that of an E35 exit, repeatedly called for each record where in this case the record is an 01-level structure including all requested fields in fixed locations.

5

Searching for a message in SYSLOG

You can use the power of IBM z/OS System Display and Search Facility (SDSF) combined with the simplicity of the REXX language to solve daily management tasks. In this chapter, we show you how to use REXX with SDSF to search for a message in SYSLOG.

5.1 Scenario description

One installation needs to submit a batch job periodically that scans the system log looking for a particular message. For each occurrence of the message in the system log, the job then needs to issue a system command using all or part of the information that is present in the line of the log where the message is found.

5.2 Solving the issue with REXX with SDSF

REXX with SDSF allows us to develop a very simple utility program that scans the system log and issues a command at each occurrence of the particular message.

5.3 The actual code

The actual code is written entirely in REXX language and will not use any feature of the REXX language other than those present in the REXX language that are supported by SDSF.

5.3.1 Parameters

@SYSLOG accepts and requires only one parameter in the message for which it is searching, *CSV0281*. There is no validation of the parameter correctness.

Example 5-1 Invoking @SYSLOG for message CSV028I

```
@SYSLOG CSV028I
```

5.3.2 Program flow

Figure 5-1 shows the flow of the program.

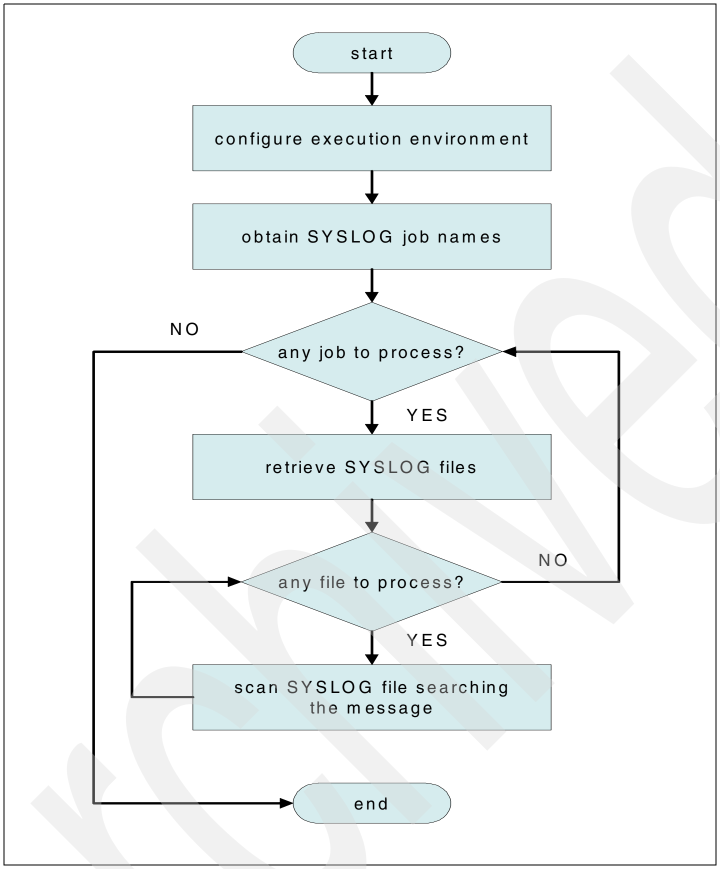


Figure 5-1 Scanning SYSLOG, program flow

The steps in the flow are:

1. Configuring SDSF execution environment.

   The SDSF support for REXX language host command environment must be activated. If the REXX program cannot add this host command environment it must cancel the execution

2. Obtaining SYSLOG job names.

   In a single ISFEXEC call, SDSF returns all the job names of the SYSLOG requested.

3. For each one of the jobs, @SYSLOG must retrieve all the SYSLOG files.
4. The REXX program must then scan each one of the files looking for the exact message it is looking for.
5. Finally, it must issue the command after finding it.

   To issue the command in this sample, we use the REXX sample @SYSCMD that we discuss in Chapter 5, "Searching for a message in SYSLOG" on page 153. @SYSCMD accepts a number of parameters, but we use only two of them:

   – `CMD()`, the system command that we want to submit
   – `QUIET(Y)` to avoid verbose output

5.3.3 Configuring the SDSF execution environment

This step is split in two different functional sets of instructions:

1. Activating the SDSF support for the REXX programming language, as shown in Example 5-2.

Example 5-2 Activating SDSF support for the REXX programming language

```
/*--------------------------------------------------------------------*
/
/* In order to use REXX with SDSF is mandatory to add a host command
*/
/* environment prior to any other SDSF host environment commands
*/
/*--------------------------------------------------------------------*
/
activate_SDSF_REXX_support:

 /*
  * Turn on SDSF "host command environment"
  */
 rc_isf = isfcalls("ON")
 select
  when rc_isf = 00 then return
  when rc_isf = 01 then msg_isf = "Query failed, environment not added"
  when rc_isf = 02 then msg_isf = "Add failed"
  when rc_isf = 03 then msg_isf = "Delete failed"
  otherwise do
    msg_isf = "Unrecognized Return Code from isfCALLS(ON): "rc_isf
  end
 end
```

```
if rc_isf <> 00 then do
   say "Error adding SDSF host command environment." msg_isf
   retcode = rc_isf * 10
   signal finish
end

return
```

2. Establishing the special SDSF variable values to retrieve only SYSLOG job, ordered by date and time ended, in ascending order. The more that SDSF does, the less that the program has to do.

Example 5-3 Setting SDSF special variables

```
/*--------------------------------------------------------------------*/
/* Set SDSF special variables to customize information retrieval      */
/*--------------------------------------------------------------------*/
set_SDSF_special_variables:

 isfprefix = "SYSLOG*"             /* Only syslog jobs               */
 isfowner  = "*"                   /* Owner does not care            */
 isfcols   = "JNAME TOKEN JOBID"   /* Only retrieve certain columns  */
 isfsort   = "DATEE A TIMEE A"     /* Ordered by datetime ending     */
 command   = "ST"                  /* SDSF panel STATUS              */

 return
```

To retrieve only the job whose name conforms to the pattern of SYSLOG in our installation, we must make the next assignment to the variable **isfprefix** that is used to limit the returned variables

```
isfprefix = "SYSLOG*"
```

Reducing the columns to those strictly needed reduces the amount of storage that SDSF must use to return the information to the caller and also reduce slightly the time of processing. To avoid retrieving all the columns, the REXX procedure assigns a string with the name of the desired columns to the special variable **isfcols**:

```
isfcols   = "JNAME TOKEN JOBID"
```

In this scenario, it is mandatory retrieve the jobs, classified by date and time of ending and in ascending order. The REXX program must retrieve first the oldest messages and continue till reaching the current time. This requirement is fulfilled by SDSF with the next variable assignment:

```
isfsort   = "DATEE A TIMEE A"[1]
```

5.3.4 Obtaining the SYSLOG job names

As stated previously, the REXX program only retrieves those job whose prefix is SYSLOG with a single call to the internal REXX procedure **exec_sdsf**, which in turn issues a command address **SDSF "ISFEXEC ST"**. SDSF, using the special variables, tries to satisfy the request. Then, **exec_sdsf** controls the return code and, in case of failure, displays some explanatory messages and cancels the program.

Example 5-4 Obtaining all the SYSLOG job names

```
/*
 * Access the ST display
 */
call exec_sdsf "0 ISFEXEC ST"
```

The program loops through all the job names that are returned by SDSF, issues the SA action (allocates authorized data sets), and processes each of the files that are returned.

Example 5-5 Loop through all the jobs

```
/*
 * Loop for all SYSLOG jobs
 */
do njob = 1 to JNAME.0
   /*
    * Issue the SA action against the row to allocate all
    * data sets in the job.
    */
   call exec_sdsf "0 ISFACT ST TOKEN('"TOKEN.njob"') PARM(NP SA)"
     /*
      * Read the records from each data set and take action
      */
      do loopdd=1 to isfddname.0
        ...
        ...
        ...
      end
end
```

[1] You can find the name of all the columns that you can use for sorting and filtering in the chapter "Columns on the SDSF panels" of *z/OS V1R9.0 SDSF Operation and Customization*, SA22-7670.

On each of the jobs, the REXX exec reads the system log, one line at a time, using the REXX command EXECIO and parses every line looking for the desired message. When it finds the message, @SYSLOG invokes @SYSCMD to execute a command, in this case, to display information about the job.

Example 5-6 Scanning syslog searching a message

```
eof = 'NO'
do while(eof = 'NO')
   "EXECIO 1 DISKR" isfddname.loopdd "(STEM line."
   if (rc = 2) then
      eof = 'YES'
   else do
      parse var line.1,
             20 ldate,
             28 ltime,
             39 .,
             40 jobname,
             48 .,
             59 txtmsg
      currmsg = left(txtmsg,8)
      if currmsg = msgparm then do
         jobtype = left(jobname,1)
         jobid   = substr(jobname,4,5)
         if jobtype <> "" then do
            display_parm = "$DO"jobtype"("jobid")"
            syscmd = "DELAY("3") CMD(“"display_parm"“)"
            call @SYSCMD syscmd
         end
      end
   end
end
```

5.4 Sample output

We submit the batch job showed in Example 5-7 to look for the message `IEA995I`, which is the message that identifies a symptom dump, for all abnormal ends when a SYSABEND, SYSUDUMP, or SYSMDUMP is requested. We use IRXJCL to run the REXX exec in MVS batch.

Example 5-7 Searching the message IEA995I in the system log

```
//REDBOOK1 JOB 'SG24-7419',MSGCLASS=A,CLASS=A,NOTIFY=REDBOOK
//* IRXJCL @SYSLOG
//BATCH   EXEC PGM=IRXJCL,PARM='@SYSLOG IEA995I'
//SYSEXEC  DD  DSN=REDBOOK.TEST.REXX,DISP=(SHR)
//         DD  DSN=MIU.TEST.REXX,DISP=(SHR)
//SYSTSPRT DD  SYSOUT=A
```

In the job log, you can see how the console is activated and deactivated to issue every command (Figure 5-2).

```
 Display  Filter  View  Print  Options  Help
 -------------------------------------------------------------------------------
 SDSF OUTPUT DISPLAY REDBOOK1 JOB26843  DSID   101 LINE 21      COLUMNS 02- 81
 COMMAND INPUT ===>                                            SCROLL ===> CSR
SDSF HCE status   : revoked RC=00
@SYSCMD operands  : DELAY(3) CMD("$DOJ(19868)")
SDSF HCE status   : established RC=00
--------------------------------------------------------------------------------
ISFEXEC options   : (VERBOSE)
Original command  : /$DOJ(19868)

SDSF short message: COMMAND ISSUED
SDSF long  message: ISF754I Command 'SET CONSOLE' generated from associated vari
SDSF long  message: ISF754I Command 'SET DELAY 3' generated from associated vari

SDSF ULOG messages:
SC70      2007109  13:17:06.85             ISF031I CONSOLE REDBOOK ACTIVATED
SC70      2007109  13:17:06.85            -$DOJ(19868)
SC70      2007109  13:17:06.86  JOB19868   $HASP686 OUTPUT(MHLRES10)
                                           $HASP686 OUTPUT(MHLRES10)  OUTGRP=2.1
                                           $HASP686                   FCB=****,F
                                           $HASP686                   FORMS=STD,
                                           $HASP686                   OUTDISP=WR
                                           $HASP686                   PRMODE=LIN
                                           $HASP686                   RECORDS=(6
                                           $HASP686                   ROUTECDE=L
                                           $HASP686                   TSOAVAIL=N
                                           $HASP686                   USERID=MHL
SC70      2007109  13:17:06.86  JOB19868   $HASP686 OUTPUT(MHLRES10)
                                           $HASP686 OUTPUT(MHLRES10)  OUTGRP=1.1
                                           $HASP686                   FCB=****,F
```

Figure 5-2 Activation and deactivation of the console

Figure 5-3 shows the final output. The first lines are the verbose output from @SYSCMD that establish the SDSF REXX host command environment and that execute the system command. Below that is the information returned by the system after issuing the command.

```
 Display  Filter  View  Print  Options  Help
 -------------------------------------------------------------------------------
 SDSF OUTPUT DISPLAY REDBOOK1 JOB26843  DSID   101 LINE 21      COLUMNS 02- 81
 COMMAND INPUT ===>                                            SCROLL ===> CSR
SDSF HCE status   : revoked RC=00
@SYSCMD operands  : DELAY(3) CMD("$DOJ(19868)")
SDSF HCE status   : established RC=00
--------------------------------------------------------------------------------
ISFEXEC options   : (VERBOSE)
Original command  : /$DOJ(19868)

SDSF short message: COMMAND ISSUED
SDSF long  message: ISF754I Command 'SET CONSOLE' generated from associated vari
SDSF long  message: ISF754I Command 'SET DELAY 3' generated from associated vari

SDSF ULOG messages:
SC70      2007109  13:17:06.85             ISF031I CONSOLE REDBOOK ACTIVATED
SC70      2007109  13:17:06.85            -$DOJ(19868)
SC70      2007109  13:17:06.86  JOB19868   $HASP686 OUTPUT(MHLRES10)
                                           $HASP686 OUTPUT(MHLRES10)  OUTGRP=2.1
                                           $HASP686                   FCB=****,F
                                           $HASP686                   FORMS=STD,
                                           $HASP686                   OUTDISP=WR
                                           $HASP686                   PRMODE=LIN
                                           $HASP686                   RECORDS=(6
                                           $HASP686                   ROUTECDE=L
                                           $HASP686                   TSOAVAIL=N
                                           $HASP686                   USERID=MHL
SC70      2007109  13:17:06.86  JOB19868   $HASP686 OUTPUT(MHLRES10)
                                           $HASP686 OUTPUT(MHLRES10)  OUTGRP=1.1
                                           $HASP686                   FCB=****,F
```

Figure 5-3 SYSTSPRT file after executing the REXX exec @SYSLOG

6

Viewing SYSLOG

This chapter describes a simple approach to using the IBM z/OS System Display and Search Facility (SDSF) SYSLOG information with ISPF View or Edit services. Simply browsing the system log from the SDSF panel can be frustrating, because the only command available is **`find`**. Using the **`view`** command also lets you find strings, but you can exclude unwanted lines, hide them, or even incorporate your own macros for viewing only the information in which you are interested.

The example that we describe in this chapter is a starting point that you can modify to use with programs, such as a combination of UNIX **`sort`** and **`grep`**, as is usually done when searching UNIX log files.

6.1 Scenario description

In this scenario, we review the current system log file, using the View ISPF service, not using Browse. In addition, we save the log to a catalogued data set or UNIX file for later analysis if the user calls the REXX exec.

This REXX program might be useful for system programmers and operations support personnel who are looking for a way to tackle the system log.

6.2 Programming caveats

The code for this scenario, whenever possible, uses the facilities that are provided by REXX and avoids similar functionality that might be found in other IBM products.

6.3 Parameters

@BRLOG accepts two parameters that are mutually exclusive and that are passed by name:

- A data set name
- A path name

If neither of these parameters is present, the program copies the SYSLOG data sets to a temporary data set and browses that data set.

If the data set name parameter is present, @BRLOG copies all SYSLOG data sets to the one that is received as a parameter and also browses it.

Example 6-1 Invoking @BRLOG with a data set name

```
@BRLOG DSNAME('B247419.SYSLOG.SC70TS')
```

If the path name parameter is present, @BRLOG copies all SYSLOG data sets to a temporary MVS data set and copies using OCOPY to the path that is specified. Finally, using the TSO command, OEDIT allows the user to view it. Copying the path name does use a Carriage Return/Linefeed (CR/LF) to separate the file records. See Example 6-2.

Example 6-2 Invoking @BRLOG with a path name

```
@BRLOG PATH('/tmp/b247419.syslog.sc70ts')
```

Figure 6-1 shows the result of Example 6-2.

```
 File  Edit  Edit_Settings  Menu  Utilities  Compilers  Test  Help
-------------------------------------------------------------------------------
EDIT /tmp/b247419.syslog.sc70ts                            Columns 00001 00072
Command ===>                                                  Scroll ===> CSR
****** ***************************** Top of Data ******************************
000001 X 0000000 SC70     2007112 14:56:27.80 SYSLOG   00000000  IEE042I SYSTEM
000002 N 0000000 SC70     2007112 14:54:41.96          00000290  IEA630I  OPERA
000003 NC0000000 SC70     2007112 14:54:41.99 INTERNAL 00000290  CONTROL M,UEXI
000004 N 0000000 SC70     2007112 14:53:04.88          00000290  IEA371I SYS0.I
000005 N 0000000 SC70     2007112 14:53:04.88          00000290  IEA246I LOAD
000006 N 0000000 SC70     2007112 14:53:04.88          00000290  IEA246I NUCLST
000007 N 0000000 SC70     2007112 14:53:04.88          00000290  IEA519I IODF D
000008 N 0000000 SC70     2007112 14:53:04.88          00000290  IEA520I CONFIG
000009 N 0000000 SC70     2007112 14:53:04.88          00000290  IEA091I NUCLEU
000010 N 0000000 SC70     2007112 14:53:04.88          00000290  IEA086I EJESSV
000011 N 0000000 SC70     2007112 14:53:04.88          00000290  IEA086I IGC213
```

Figure 6-1 Browsing syslog in a UNIX path file

6.3.1 Program flow

Figure 6-2 illustrates the program flow for this scenario.

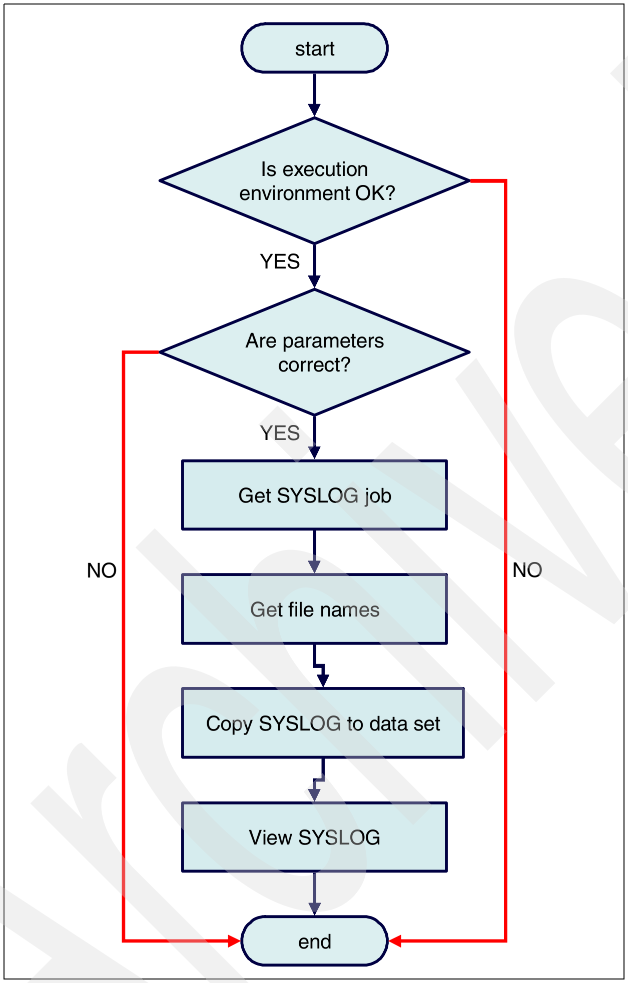


Figure 6-2 Viewing SYSLOG

6.3.2 Testing execution environment

The program uses the ISPF View service, so it must run under interactive ISPF. In any other case, it will cancel.

6.3.3 Parameter verification

The program must verify that it has received the desired data set name. It is only a simple verification and does no name validity checking. If the data sets exists, it is overwritten. If it does not exist, it is created.

6.3.4 Configuring the SDSF execution environment

After establishing a valid REXX with SDSF environment, a call to the internal procedure **activate_SDSF_REXX_support** invokes the **isfcalls** function and analyzes the return code that is obtained, as shown in Example 6-3.

Example 6-3 Activating REXX with SDSF support

```
/*--------------------------------------------------------------------*/
/* In order to use REXX with SDSF is mandatory to add a host command */
/* environment prior to any other SDSF host environment commands     */
/*--------------------------------------------------------------------*/
activate_SDSF_REXX_support:

 /*
  * Turn on SDSF "host command environment"
  */
 rc_isf = isfcalls("ON")
 select
  when rc_isf = 00 then return
  when rc_isf = 01 then msg_isf = "Query failed, environment not added"
  when rc_isf = 02 then msg_isf = "Add failed"
  when rc_isf = 03 then msg_isf = "Delete failed"
  otherwise do
    msg_isf = "Unrecognized Return Code from isfCALLS(ON): "rc_isf
  end
 end

 if rc_isf <> 00 then do
    say "Error adding SDSF host command environment." msg_isf
    signal finish
 end

 return
```

The program modifies the SDSF control variables to obtain a copy of the SYSLOG spool data sets of the system on which it is executed through a call to the internal procedure **set_SDSF_special_vars** (Example 6-4).

Example 6-4 Setting SDSF special variables

```
set_SDSF_special_vars:

 /*
  * Target isfprefix special variable towards own system SYSLOG jobs
  */
 isfprefix = "SYSLOG*"
 isfowner  = "*"
 isfcols   = "JNAME TOKEN JOBID QUEUE ESYSID"
/*
 * We need the output sorted in ascending order by the date and
 * time when the execution began
 */
isfsort   = "DATEE A TIMEE A"
/*
 * ISFFILTER specifies a filter criterion to be applied to the
 * returned variables. Use the column name rather than the column
 * title. Only a single criterion is supported. In this case we
 * use ESYSID (JES2 execution node) column name instead of "Esys"
 * column title
 */
 isffilter = "ESYSID =" mvsvar("SYSNAME")
/*
 * Allocation parameter user by XFC command.
 */
isfPrtDDNAME     = tmpdd

 return
```

Specifying these variables, the REXX program tries to obtain the SYSLOG jobs of the system where it is run, ordered in ascending order by date and time of execution.

> **Note:** It is important to keep in mind that SDSF special variables (such as **isffilter**, **isfsort**, or **isfcols**) must reference *column* names not *panel* names. Refer to *z/OS V1R9.0 SDSF Operation and Customization*, SA22-7670 to make sure that you are using the correct column names.

6.3.5 Obtaining all the SYSLOG jobs

The program gathers all the SYSLOG jobs of the current system by accessing the SDSF ST (STATUS) panel you would from the SDSF command line, as shown in Example 6-5.

Example 6-5 Accessing SDSF STATUS panel from REXX

```
sdsf_command = "ST"
call exec_sdsf "0 ISFEXEC" sdsf_command
```

Each row that is retrieved has its own unique token identifier (`stem TOKEN`) that lets the program copy each one of the spool data sets to the data set whose name it has received as a parameter. For copying, @BRLOG uses the SDSF command XFC. This command prints all data sets to a file (DDNAME) using the attributes that are specified in the special variables. So, for each row, the ISFACT routine must be called, specifying the same panel (ST) and the corresponding row token. See Example 6-6.

Example 6-6 Copying all the syslog data sets

```
do njob = 1 to JNAME.0
   /*
    * Issue the XFC action against each row to copy all the
    * data sets in the job. Maximum return code admitted 0.
    */
    call exec_sdsf "0 ISFACT ST TOKEN('"TOKEN.njob"') PARM(NP XFC)"
end
```

Figure 6-3 shows the output if the parameter received was a cataloged data set, after waiting for all the data sets to be copied.

```
 File  Edit  Edit_Settings  Menu  Utilities  Compilers  Test  Help
-------------------------------------------------------------------------------s
VIEW B247419.SYSLOG.SC70TS                                 Columns 00001 00072
Command ===>                                                  Scroll ===> CSR
****** **************************** Top of Data ******************************
000001 N 0000000 SC70     2007086 13:34:06.65          00000290  IEA630I  OPERA
000002 X 0000000 SC70     2007086 13:35:34.60 SYSLOG   00000000  IEE042I SYSTEM
000003 NC0000000 SC70     2007086 13:34:06.68 INTERNAL 00000290  CONTROL M,UEXI
000004 N 0000000 SC70     2007086 13:30:37.77          00000290  IEA371I SYS0.I
000005 N 0000000 SC70     2007086 13:30:37.77          00000290  IEA246I LOAD
000006 N 0000000 SC70     2007086 13:30:37.77          00000290  IEA246I NUCLST
000007 N 0000000 SC70     2007086 13:30:37.77          00000290  IEA519I IODF D
000008 N 0000000 SC70     2007086 13:30:37.77          00000290  IEA520I CONFIG
000009 N 0000000 SC70     2007086 13:30:37.77          00000290  IEA091I NUCLEU
000010 N 0000000 SC70     2007086 13:30:37.77          00000290  IEA086I EJESSV
000011 N 0000000 SC70     2007086 13:30:37.77          00000290  IEA086I IGC213
000012 N 0000000 SC70     2007086 13:30:37.77          00000290  IEA093I MODULE
000013 S                                                         IFFIOM
000014 N 0000000 SC70     2007086 13:30:37.77          00000290  IEA093I MODULE
000015 S                                                         IEDQATTN
000016 N 0000000 SC70     2007086 13:30:37.77          00000290  IEA093I MODULE
000017 S                                                         IECTATEN
000018 N 8000000 SC70     2007086 13:31:01.56          00080290 *IEA213A DUPLIC
000019 N 8000000 SC70     2007086 13:31:01.58          00000290 *   IEA213A REP
000020 N 0000000 SC70     2007086 13:31:44.09          00000290  IEE600I REPLY
000021 N 4000000 SC70     2007086 13:31:44.11          00000290  IEA313I DEVICE
000022 N 8000000 SC70     2007086 13:31:44.13          00080290 *IEA213A DUPLIC
000023 N 8000000 SC70     2007086 13:31:44.15          00000290 *   IEA213A REP
000024 N 0000000 SC70     2007086 13:31:48.93          00000290  IEE600I REPLY
000025 N 4000000 SC70     2007086 13:31:49.04          00000290  IEA313I DEVICE
000026 N 8000000 SC70     2007086 13:31:49.06          00080290 *IEA213A DUPLIC
```

Figure 6-3 Browsing SYSLOG output data set

6.4 Customization

In a sysplex environment, you might be interested not only in the log of one single system, but in the log of all the systems in the sysplex. If you are running IMS™, it might be appealing to you to reformat and merge the IMS log with the system log to have a more complete view of what is happening.

You could eliminate all the ISPF calls from the code and let the program run in a pure batch environment or as a UNIX command shell script. In this latter case, you can integrate your REXX in a pipe for sorting and searching and then redirect the output to another program for later processing.

Finally, removing the ISPF services lets you use it as a starting point for a client-server pair of programs that allows you to browse your system logs from outside the host system.

7

Reviewing execution of a job

This chapter provides a sample REXX procedure that reviews the execution of a batch job and, depending on some predefined conditions, issues different actions. Writing a similar program without the aid of the IBM z/OS System Display and Search Facility (SDSF) support for the REXX programming language could be difficult and require some assembler programming to accomplish the same tasks.

7.1 Scenario description

In this scenario, a batch job is run every day that produces a report for a group of customers. If the jobs ends normally, the report must be sent. If any abnormal condition occurs, the execution of the job must be analyzed and a summary of the execution sent through e-mail to the programmers.

Of course, the customers report could be sent directly from the sample job, but one of the purposes of this scenario is to extract and process some SYSOUT data from the spool using the facilities that are provided by using REXX with SDSF.

7.2 Solution

The solution provided with this scenario is a REXX program that receives the target job by a parameter, verifies its execution using the REXX interface provided by SDSF, and if the job is found analyzes it. Then, one of the following conditions occurs:

- If the job has ended normally, VERIFJOB searches the report name signaled by the parameter `REPORT()`, builds an e-mail body with it, and sends it to the customers.
- If there is any abnormal condition, a summary of the execution of the job is sent to the programmers team.

The e-mail configuration and the e-mail addresses of both customers and programmers are read from files.

7.2.1 Parameters

There are a small number of parameters that you can use to tailor the behavior of the REXX program:

`JOBNAME(job_name)`	Job name to look for. If no ended job with this name is found, the process fails and a summary must be send to the programmers team.
`JOBID(jobid)`	Job ID that can be specified to locate the correct execution. This parameter is mutually exclusive with parameter `SUBMITTED()`.
`MAXCC(return_code)`	Maximum return code that considered a valid return code for the job.

`REPORT(report_name)`
: The name of DD that the program has to search in the Job Data Set panel (JDS). If the file is not found in the step specified in the parameter `STEP()`, the process fails, and a summary must be send to the programmers team.

`STEP(step_name)`
: Step identifier where the report is created. If the step is not found, the process fails, and a summary is send to the programmers team.

`SUBMITTED(date time)`
: Submit date and time. The first job with the same job name that ended after this time and date will be chosen. This parameter is mutually exclusive with parameter `JOBID()`.

7.2.2 Program logic

Figure 7-1 illustrates the program logic for this scenario.

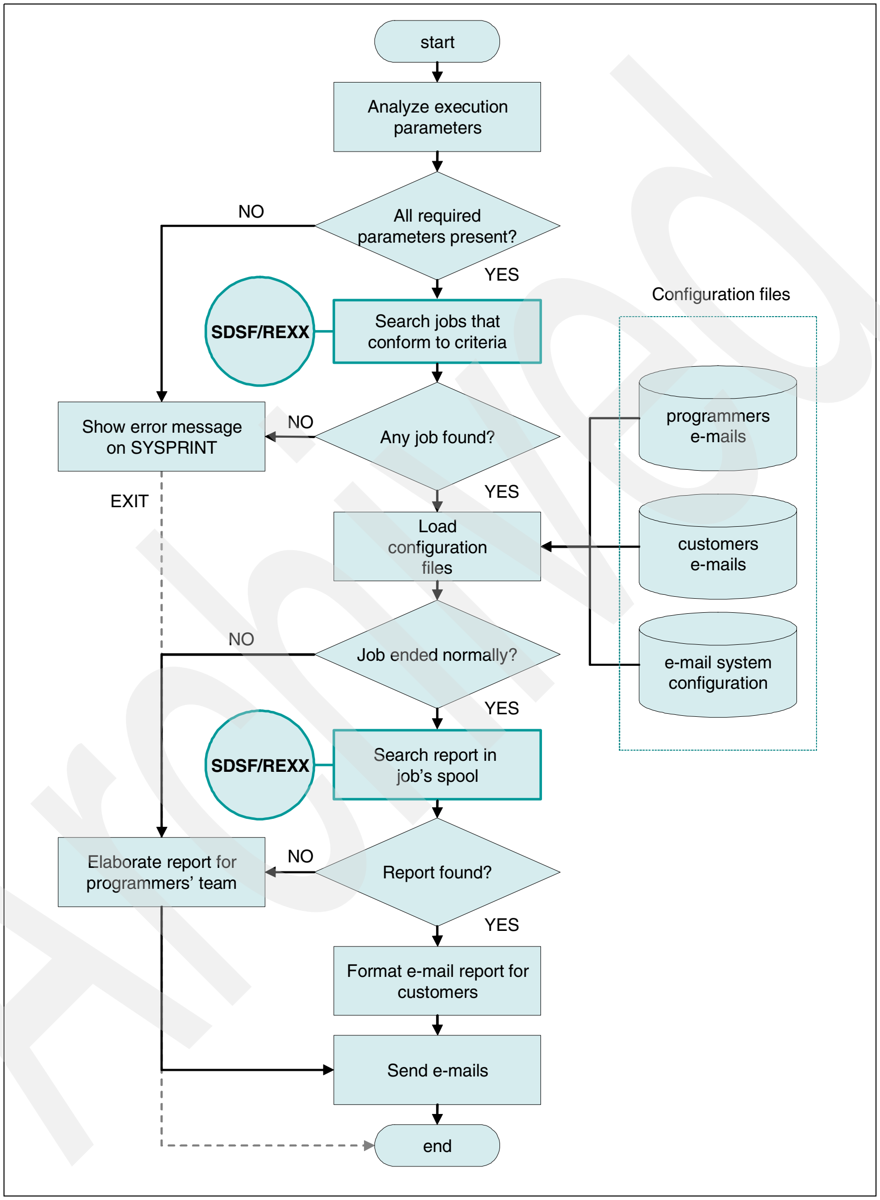


Figure 7-1 Sample scenario program logic

7.2.3 Searching jobs

To retrieve the job that you want, you must first establish the environment. The REXX program calls the internal routine **activate_SDSF_REXX_support** as shown in Example 7-1.

Example 7-1 Activating the SDSF REXX host command environment

```
/*-------------------------------------------------------------------*
/
/* In order to use REXX with SDSF is mandatory to add a host command
*/
/* environment prior to any other SDSF host environment commands
*/
/*-------------------------------------------------------------------*
/
activate_SDSF_REXX_support:

 /*
  * Turn on SDSF "host command environment"
  */
 rc_isf = isfcalls("ON")
 select
  when rc_isf = 00 then return
  when rc_isf = 01 then msg_isf = "Query failed, environment not added"
  when rc_isf = 02 then msg_isf = "Add failed"
  when rc_isf = 03 then msg_isf = "Delete failed"
  otherwise do
    msg_isf = "Unrecognized Return Code from isfCALLS(ON): "rc_isf
  end
 end

 if rc_isf <> 00 then do
    say "Error adding SDSF host command environment." msg_isf
    retcode = rc_isf * 10
    signal finish
 end

 return
```

Setting special variables

When you have established the host command environment, the REXX program must set the special variables that SDSF uses to filter the information that it retrieved. In this case, the variable **isfprefix** is the name of the job received, and the variable **isfowner** is an asterisk (*), meaning that it might be a job that is owned by anyone. See Example 7-2.

Example 7-2 Setting special variables to control SDSF

```
/*--------------------------------------------------------------------*
/
/* Set SDSF special variables to customize information retrieval
*/
/*--------------------------------------------------------------------*
/
set_SDSF_special_variables:

 isfprefix = parm_jobname         /* Only those jobs that matches
*/
 isfowner  = "*"                  /* Owner does not care
*/
 command   = "ST"                 /* SDSF panel STATUS
*/

 return
```

Requesting to SDSF a list of jobs

To request a list of jobs that conform to the criteria established by the special variables the only thing the program has to do is call the ISFEXEC function with the status panel parameter (ST), specifying that it wants also the columns of the alternate panel and those columns that are returned by SDSF if a delay is admitted. See Example 7-3.

Example 7-3 Searching the job

```
/*--------------------------------------------------------------------*
/
/* Search the job received by parameter using the tabular display
*/
/* provided by the SDSF Status panel
*/
/*--------------------------------------------------------------------*
/
search_job:
```

```
if debug > 0 then
   opts_sdsf = "(VERBOSE ALTERNATE DELAYED)"
else
   opts_sdsf = "(ALTERNATE DELAYED)"

call exec_sdsf "0 ISFEXEC ST" opts_sdsf
do ij = 1 to JNAME.0
   ...
   ...
   ...
end

   ...
   ...
   ...
```

The internal subroutine **exec_sdsf** takes care of controlling the return code from ISFEXEC and finishes the program if the return code of received is not equal to zero.

7.2.4 Choosing the desired job

After the program has a list of jobs that satisfy the criteria, it must be filtered again, choosing the first one that meets the additional constraints specified in the parameters:

- The REXX program has used the ST panel, so it must make sure that the job has ended.
- If there is a parameter JOBID(), then the REXX program searches the job whose JES2 job ID equals the parameter received.
- If there is a SUBMITTIME() parameter, the first job executed after the date and time specified is selected

If the correct job is found, the program stores the token returned by SDSF for later use, as shown in Example 7-4.

Example 7-4 Storing the token returned by SDSF for later use

```
do ij = 1 to JNAME.0
   if JNAME.ij = parm_jobname then do
      if RETCODE.ij = "" then do                     /* Not ended */
         if JOBID.ij = parm_jobid then
            leave ij
        else
```

```
            iterate
      end
      if parm_jobid <> "" then do
         if JOBID.ij = parm_jobid then
            job_found = "YES"
      end
      if submit_time <> "" then do
         if later_time(parm_date,parm_time,DATER.ij,TIMER.ij) = 1 then
            job_found = "YES"
      end
      if job_found = "YES" then do
        currtoken = TOKEN.ij
        job_found = "YES"
        leave ij
      end
   end
end
```

Verifying the return code of the job

Using the column RETCODE, the REXX program verifies that the chosen job has a maximum return code less or equal than the parameter received. If this parameter is omitted, then the maximum return code allowed will be zero.

The code in Example 7-5 examines the return code information for the job returned by SDSF in stem RETCODE. Based on it, the code decides whether it has to format and send the report to the customers or whether there is an error and has to analyze the job and send a summary to the application programmers.

Example 7-5 Verifying the return code of the job

```
/*
  * Test if the maximum return code of the job is greater or equal than
  * the parameter MAXCC
  */
 if word(RETCODE.ij,1) = "CC" & word(RETCODE.ij,2) <= parm_maxcc then do
    say " Parameter MAXCC: "parm_maxcc "(OK)"
    report_type  = "CUSTOMER"
    report_title = "Report" parm_report ,
                   "from " JNAME.ij"-"JOBID.ij ,
                   "("DATEN.ij"-"TIMEN.ij")"
 end
 else do
    say " Parameter MAXCC: "parm_maxcc ". Job Max-RC:" RETCODE.ij
    say " Condition not satisfied. The programmers must be notified"
    report_type  = "PROGRAMMER"
    report_title = "Error report of the job" ,
```

```
                    JNAME.ij"-"JOBID.ij ,
                    "("DATEN.ij"-"TIMEN.ij")"
 end
```

7.2.5 Searching the report

To search for one report, the REXX exec must access the Job Data Set panel that allows the user to display information about SYSOUT data sets for a selected job, started task, or TSO user. This task is accomplished issuing the **isfact** command with the **?** action character for the job identified by the token variable, **parm(np ?)** as shown in Example 7-6. In this case, we use the prefix option to ensure unique variables are created, beginning with **J**, then, SDSF returns stems JDDNAME and JSTEPN.

Example 7-6 Searching the report

```
if debug > 0 then
   opts_sdsf = "(VERBOSE PREFIX J)"
else
   opts_sdsf = "(PREFIX J)"

call exec_sdsf "0 ISFACT ST TOKEN('"currtoken"') PARM(NP ?)" opts_sdsf
do ddname = 1 to JDDNAME.0
   /*
    * Look for received parameter
    */
   if JDDNAME.ddname = reportdd then do
      if step_name = "" | JSTEPN.ddname = step_name then do
         report_token = jtoken.ddname
         leave ddname
      end
   end
end

if report_token = "" then do   /* Report not found */
   ...
   ...
   ...
end
```

7.2.6 Processing the report

To process the report, this sample REXX exec adds some HTML code to transform the spool file into an e-mail message (in this case an e-mail message readable by a Lotus® Notes® client). First, the program reads the spool file using the host environment command ISFACT and specifying the allocate authorized data sets command, SA.

After issuing the command, SDSF allocates the spool data set and its ddname is returned in the stem variable `isfddname`.

Example 7-7 Reading a report from the spool

```
email_report:

 parse arg ddtoken

 t = 0
 call exec_sdsf "0 ISFACT ST TOKEN('"ddtoken"') PARM(NP SA)"
 do kx = 1 to isfddname.0
    "EXECIO * DISKR" isfddname.kx "(OPEN FINIS STEM spool.)"
    if rc <> 0 then call error_reading_spool isfddname.kx
    call create_report_header
    do lx = 1 to spool.0
       if t = 1 then
          t = 2
       else
          t = 1
       call store_line '<tr class=t't'><td><pre>',
                     spool.lx'</pre></td></tr>'
    end
    call create_report_footer
 end
```

7.2.7 Analyzing job execution

If the user has implemented IEFACTRT SMF exit and the REXX exec is able to find the termination summary that is formatted by this exit in the spool's JESMSGLG that corresponds to the job, it sends this report to the list of programmers.

If either the user has not implemented IEFACTRT in the installation or the REXX exec cannot find the expected report, it searches the JESYSMSG spool ddname and looks for the system messages that it has read previously from the file SYSMSGS. It then produces a report and sends it to the programmers list. See Example 7-8.

Example 7-8 Analyzing the job execution

```
call search_report "CONTINUE" "JESMSGLG" ""
if report_token <> "" then do
   call locate_IEFACTRT_section report_token
   if IEFACTRT_found = "YES" then do
      call connect_to_smtp_server
      call send_email_message
   end
   return
end
   ...
   ...
   ...
call search_report "CONTINUE" "JESYSMSG" ""
if report_token <> "" then do
   call locate_system_messages report_token
   call connect_to_smtp_server
   call send_email_message
end
```

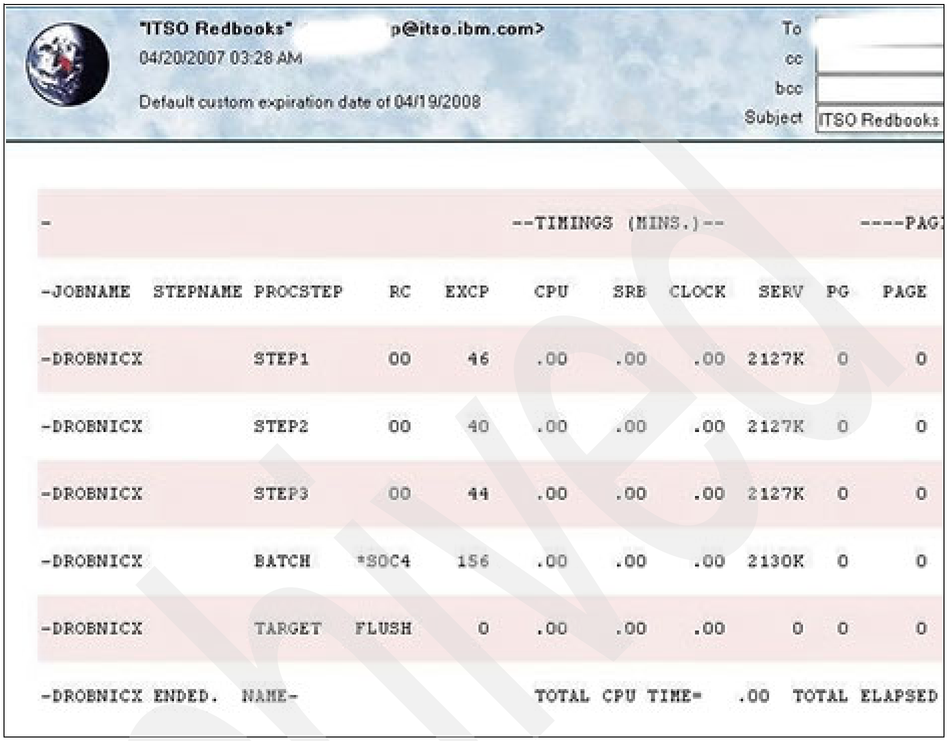
To find JESMSGLG and JESYSMSG, the REXX exec uses the internal subroutine **`search_report`**, which we explain in 7.2.5, “Searching the report” on page 181.

7.2.8 Program output

The output of the program are e-mails. Figure 7-2 shows the report that is sent to the customers. In this case, the report is the compilation listing file of the z/OS XL C/C++ compiler.

Figure 7-2 Sample e-mail report sent by the REXX exec

Figure 7-3 shows the e-mail that is sent to the programmers in the case of an abnormal termination.



"ITSO Redbooks" p@itso.ibm.com>
04/20/2007 03:28 AM
Default custom expiration date of 04/19/2008
To
cc
bcc
Subject ITSO Redbooks

-					--TIMINGS (MINS.)--					----PAG
-JOBNAME	STEPNAME	PROCSTEP	RC	EXCP	CPU	SRB	CLOCK	SERV	PG	PAGE
-DROBNICX		STEP1	00	46	.00	.00	.00	2127K	0	0
-DROBNICX		STEP2	00	40	.00	.00	.00	2127K	0	0
-DROBNICX		STEP3	00	44	.00	.00	.00	2127K	0	0
-DROBNICX		BATCH	*S0C4	156	.00	.00	.00	2130K	0	0
-DROBNICX		TARGET	FLUSH	0	.00	.00	.00	0	0	0
-DROBNICX ENDED.	NAME-				TOTAL CPU TIME=		.00	TOTAL ELAPSED		

Figure 7-3 Job execution summary sent to the programmers team

7.2.9 Possible enhancements

In the sample provided, the information that is collected from the jobs JESMSGLG and JESYSMSG is not used to provide any corrective action. For some simple cases, it might be feasible to analyze the messages that the system gives, take the appropriate corrective actions, and then resubmit the job.

8

Remote control from other systems

This chapter describes a very basic approach to access IBM z/OS System Display and Search Facility (SDSF) from other systems. The aim of this scenario is to show how easily someone can access the SDSF facilities with the REXX interface and how SDSF can be accessed and operated from outside the mainframe system. In this scenario, we are not concerned with security issues, concurrency problems, or performance considerations.

8.1 System structure

The server side of this scenario receives requests from remote clients, connects with SDSF, and processes these requests (Figure 8-1). It then returns the reply from SDSF to the clients. The server side has two main components:

- A communications processor that listens for incoming requests and dispatches them to one of the SDSF command processors.
- SDSF command processor, s whose responsability is to contact SDSF, execute one or more commands, format the reply, and send it back to the client.

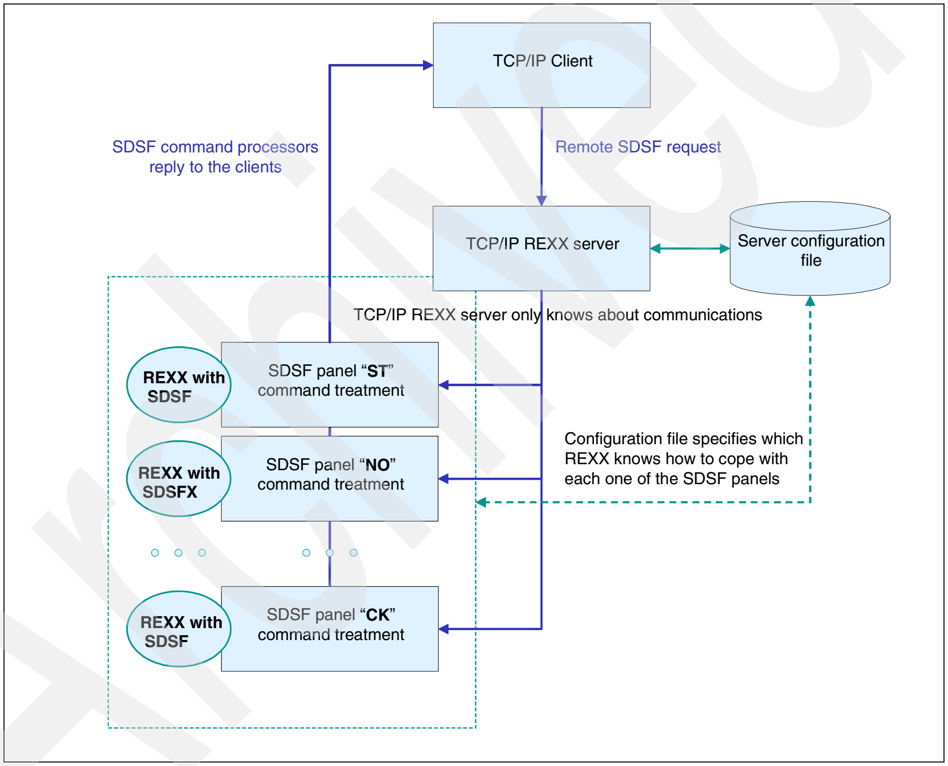


Figure 8-1 Sample client-server system

The TCP/IP REXX server has no knowledge of SDSF. Its responsibility is to receive client requests and to dispatch these requests to the corresponding processors. The code on the server side of this scenario does not have to be written in REXX and can be substituted by a program written in any other programming language.

8.2 The main server

The purpose of the main server is to receive clients requests and dispatch them to the appropriate SDSF command processor. It only replies to the clients if there is no command processor available to process the SDSF command received. In any other case, those command processors take care of the conversation and reply directly to the clients. Example 8-1 shows the JCL to start the server.

Example 8-1 JCL to start the server with inline configuration file

```
//REDBOOKS JOB 'SG24-7419',MSGCLASS=A,CLASS=A,NOTIFY=REDBOOKS
/*JOBPARM S=SC70
//* ------------------------------------------------------------------
*
//* IRXJCL TEST
//* ------------------------------------------------------------------
*
//BATCH   EXEC PGM=IRXJCL,PARM='@SDSFSRV'
//SYSEXEC  DD  DSN=REDBOOKS.TEST.REXX,DISP=(SHR)
//SYSTSPRT DD  SYSOUT=A
//SRVCONF  DD  *
#-----------------------------------------------------------------------
# CONFIGURATION FILE OF THE REXX SDSF SERVER
#-----------------------------------------------------------------------
PORT=24741
#-----------------------------------------------------------------------
# ONE LINE FOR EACH REXX SDSF COMMAND PROCESSOR
#-----------------------------------------------------------------------
NO=@CMDNO
ST=@CMDST
/*
```

All the information needed to start the server is supplied in the inline configuration file: the port number where it listens for incoming conversations as well as the command processors for each one of the SDSF panels.

8.2.1 Main server's program logic

Figure 8-2 illustrates the main server's program logic. On startup, the server reads the configuration file and stores the name of the SDSF command processors that it has to invoke in a stem variable. This stem variable is dependent on the command that it receives from the client. It then starts the TCP/IP REXX interface and listen for any incoming conversation.

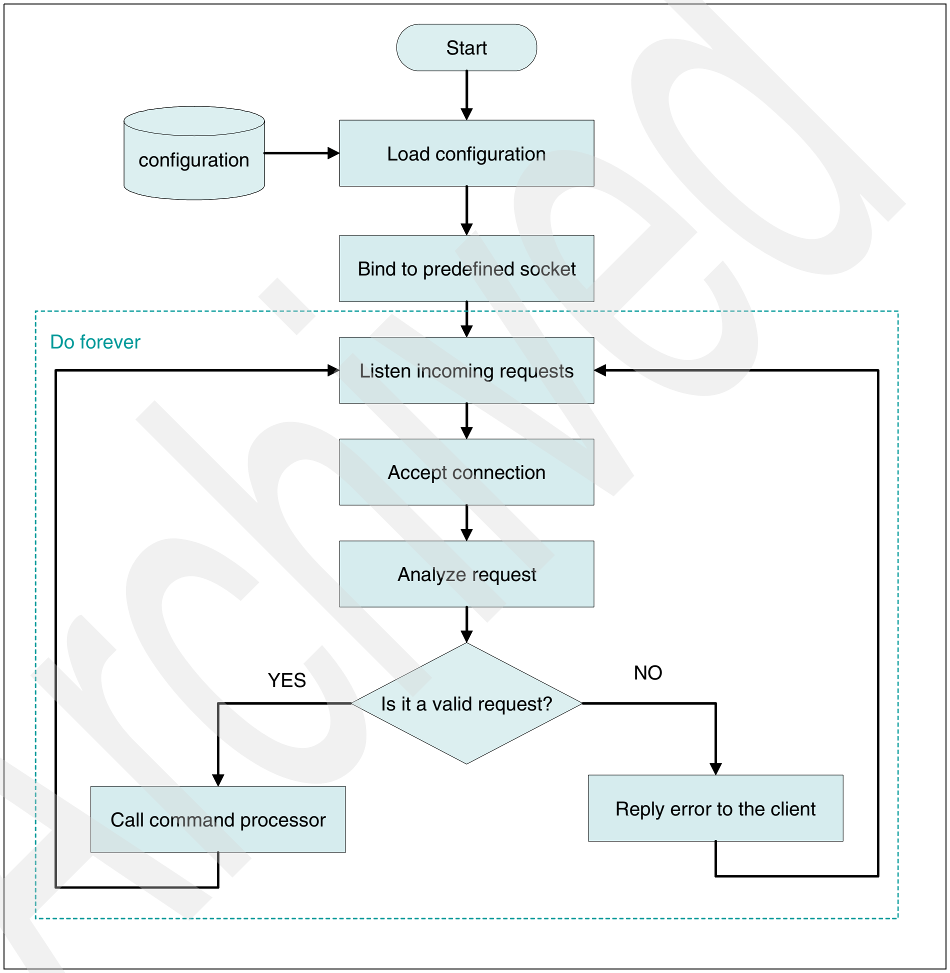


Figure 8-2 Main server logic

8.3 SDSF command processors

SDSF command processors receive the communications socket from the main server. They execute the only command that each one of them knows how to process, and then they reply to the client.

There are two kinds of requests that each command processor can process:

- The request for a tabular display resulting from a main panel command, such a ST (status panel), NO (Nodes panel), and so forth. When a command is received, the sample command processor replies with a series of lines to the client, with all the columns of the SDSF panel, plus the token that is associated with each one of the lines.
- A line command of the tabular display, such as D (Display). To know which command processor has to receive the command and the line to which it refers, the client request needs two additional parameters: the panel command and the token that is associated with that line.

8.3.1 Parameters

The SDSF command processor must receive two parameters, socket and input data, as described here:

Socket: TCP/IP socket attached to the client. The command processor will send its reply using this socket.

Input data: Request sent by the client. It has at least the identifier of the panel.

8.3.2 Program logic

At every call, the SDSF command processor has to start a new REXX with SDSF environment to service client requests and at exit, after the process has been completed, delete it. When calling the REXX interface more than once, it is important to make sure that the cleanup has been correctly performed.

Figure 8-3 illustrates the program logic for the REXX with SDSF command processor.

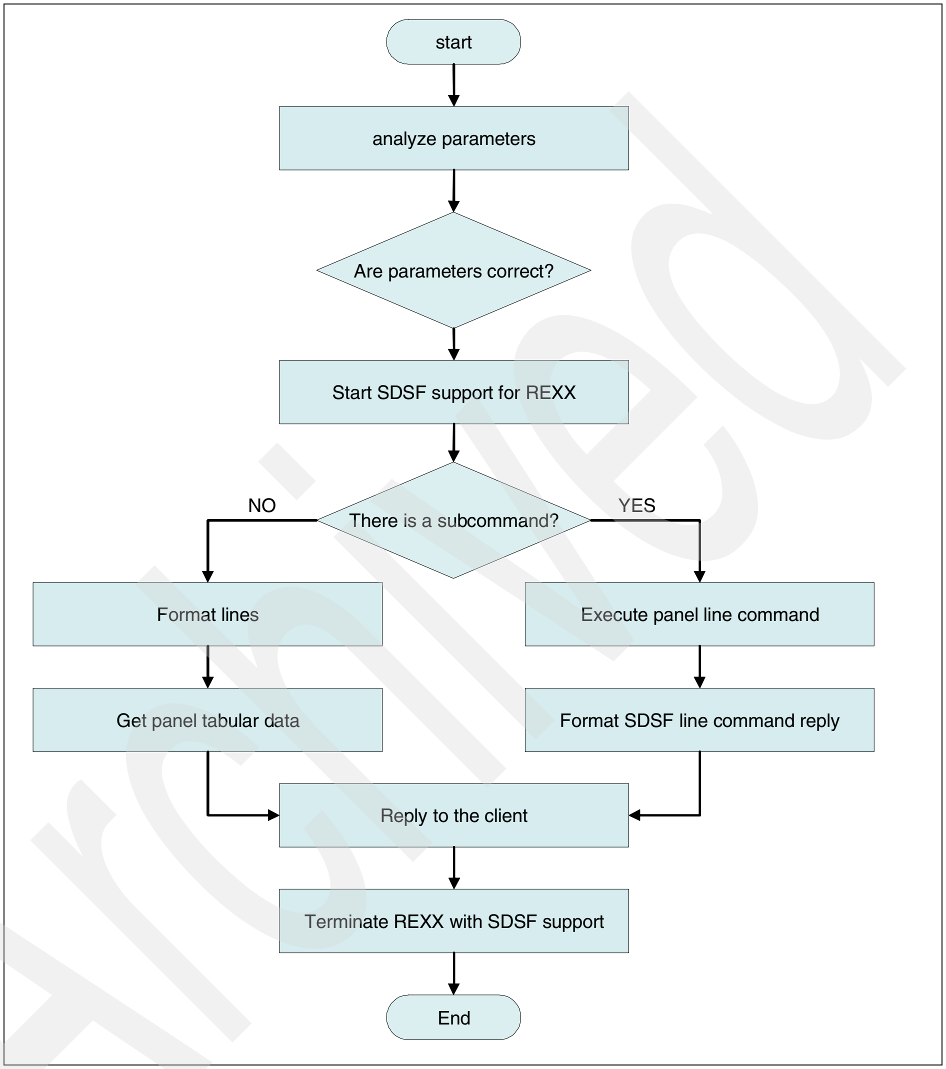


Figure 8-3 REXX with SDSF command processor logic

Activating SDSF support for the REXX language

Every SDSF command processor has to add the SDSF host command environment to the list of available REXX command environments using the REXX internal **activate_SDSF_REXX_support** subroutine. This subroutine takes care of the return codes and, if it cannot add the SDSF environment, cancels the REXX execution. See Example 8-2.

Example 8-2 Activating SDSF support for the REXX language environment

```
activate_SDSF_REXX_support:

 /* Turn on SDSF "host command environment" */
 rc_isf = isfcalls("ON")
 select
  when rc_isf = 00 then return
  when rc_isf = 01 then msg_isf = "Query failed, environment not added"
  when rc_isf = 02 then msg_isf = "Add failed"
  when rc_isf = 03 then msg_isf = "Delete failed"
  otherwise do
    msg_isf = "Unrecognized Return Code from isfCALLS(ON): "rc_isf
  end
 end

 if rc_isf <> 00 then do
    say "Error adding SDSF host command environment." msg_isf
    retcode = rc_isf * 10
    signal finish
 end

 return
```

Terminating SDSF support for the REXX language

To exit the program if any abnormal condition is detected, all the subroutines use the REXX language clause SIGNAL to change the flow of control to the label `finish`, where we free any resource that is used by the program, close all open files and sockets, and terminate the REXX with SDSF environment. See Example 8-3.

Example 8-3 Exiting the program

```
finish:
  rc = isfcalls('OFF')
  socktxt = exec_socket("*","Shutdown",s,"BOTH")
  exit retcode
```

It is the client's responsibility to reply, finish any SDSF command processing, and close the communication socket on both normal and abnormal exits.

Requesting services from SDSF

To call REXX with SDSF, all the code in the REXX uses the internal subroutine **`exec_sdsf`** (Example 8-4). This subroutine accepts two parameters:

- The *maximum* return code must be numeric, but if it takes the special nonnumeric value of an asterisk ("*"), it means, *do not care about the result*.
- The *SDSF* command is a string of characters. SDSF tries to execute and analyze it to provide a meaningful message if it is incorrect.

> **Note:** This subroutine uses the maximum return code permitted and controls whether the return code that is received from SDSF is admissible.

Example 8-4   Internal subroutine exec_sdsf

```
/*--------------------------------------------------------------------*
/
/* Subroutine to execute an SDSF REXX command testing its return
code */
/*--------------------------------------------------------------------*
/
exec_sdsf:

 parse arg max_SDSF_rc exec_SDSF_command

 /*
  * Drop SDSF msg standard variable in order to not get confused by
  * any previous value
  */
 if symbol("ISFMSG") = "VAR" then
    drop isfmsg

 sdsf = "OK"
 address SDSF exec_SDSF_command "(VERBOSE ALTERNATE DELAYED)"
 if (max_SDSF_rc = "*") then
    return rc

 if (rc > max_SDSF_rc | rc < 0) then do
    call SDSF_msg_rtn exec_SDSF_command
    sdsf = "KO"
 end

 return 0
```

Requesting tabular data

Example 8-5 shows how the SDSF command processors obtain data from SDSF and format it to send a reply to the client. In the code, after calling the subroutine **`exec_sdsf`** with the parameter ISFEXEC ST and a maximum return code of 0, we know that the call succeeded, the subroutine has issued the ST command, and SDSF has created variables for the alternate field list, including delayed-access columns because the final request made to SDSF has been:

```
address SDSF "ISFEXEC ST (VERBOSE ALTERNATE DELAYED)"
```

Specifying the option VERBOSE, if there is any problem, the stem variable isfmsg2. will includes SDSF numbered messages.

Example 8-5 Formatting information about jobs

```
/*-------------------------------------------------------------------*/
/* Display information about Jobs                                    */
/*-------------------------------------------------------------------*/
display_status:

 parse arg token

 call exec_sdsf "0 ISFEXEC ST”
 if sdsf = "KO" then return

 /*
  * Loop through all JOBS
  */
 do njob = 1 to JNAME.0
    line = "<JOB>"
    do nw = 1 to words(isfcols)
       vnam = word(isfcols,nw)
       if left(vnam,4) = "DATE" then
          call reformat_date vnam njob
       interpret "line = line'<"vnam">'||"vnam".njob||'</"vnam">'"
    end
    line = line"</JOB>"
    nbytes = exec_socket("*","Send",s,line)
 end
 nbytes = exec_socket("*","Send",s,"/*EOD")

return
```

The `loop 1 ... JNAME.0` retrieves all the rows for the job names that are returned by SDSF that satisfy the conditions expressed in the SDSF special variables such as **`isfowner`** or **`isfprefix`**, formats all the variables returned, and sends them back to the client.

8.4 A sample client

The sample client is an ISPF client. Its only purpose is to illustrate how to construct a remote application to control a z/OS sysplex.

Here are the menu entries:

File The only option is Exit.

SDSF Panel This lists all the SDSF panels that are available remotely. The sample provided has only two: ST and NO. The list of valid SDSF panels must match the server configuration file.

Communications This point of the menu, shows a window where the you can specify the address of the remote system and the port of communications used. The ISPF window is shown in Figure 8-4.

```
  File  SDSF Panel  Communication
 - +----------------------------------------------------------------------------+
   ¦ Customize communication                                                    ¦
 C ¦ Command ===>                                                               ¦
   ¦                                                                            ¦
 N ¦ Enter the following fields:                                                ¦
 _ ¦                                                                            ¦
 _ ¦ System name/address .... wtsc70oe.itso.ibm.com                             ¦
 _ ¦ Port number ............ 24741                                             ¦
 _ ¦                                                                            ¦
 _ +----------------------------------------------------------------------------+
 ____ REDBOOK1 JOB25704 A CC 0000  2007/04/13 11:50:59 2007/04/13 11:51:00
 ____ REDBOOK1 JOB25706 A CC 0000  2007/04/13 11:55:12 2007/04/13 11:55:13
 ____ REDBOOK1 JOB25707 A CC 0000  2007/04/13 11:57:47 2007/04/13 11:57:48
 ____ REDBOOK1 JOB25710 A CC 0000  2007/04/13 13:12:26 2007/04/13 13:12:27
 ____ REDBOOK1 JOB25711 A CC 0000  2007/04/13 13:14:03 2007/04/13 13:14:04
 ____ REDBOOK1 JOB25712 A CC 0000  2007/04/13 13:15:53 2007/04/13 13:15:54
 ____ REDBOOK1 JOB25713 A CC 0000  2007/04/13 13:17:42 2007/04/13 13:17:43
 ____ REDBOOK1 JOB25714 A CC 0000  2007/04/13 13:17:55 2007/04/13 13:17:56
 ____ REDBOOK  TSU25673   CC 0000  2007/04/13 07:24:29 2007/04/13 13:19:06
 ____ REDBOOK1 JOB25716 A CC 0000  2007/04/13 13:21:16 2007/04/13 13:21:17
 ____ REDBOOK1 JOB25717 A CC 3632  2007/04/13 13:23:39 2007/04/13 13:23:39
 ____ REDBOOK1 JOB25718 A CC 0000  2007/04/13 13:24:11 2007/04/13 13:24:12
 ____ REDBOOK1 JOB25720 A CC 0000  2007/04/13 13:28:51 2007/04/13 13:28:52
 ____ REDBOOK1 JOB25722 A CC 0000  2007/04/13 13:30:35 2007/04/13 13:30:36
 ____ REDBOOK1 JOB25723 A CC 3632  2007/04/13 13:41:39 2007/04/13 13:41:39
 ____ REDBOOK1 JOB25724 A CC 3632  2007/04/13 13:42:44 2007/04/13 13:42:44
 ____ REDBOOK1 JOB25725 A CC 0000  2007/04/13 13:43:57 2007/04/13 13:43:58
 ____ REDBOOK1 JOB25735 A CC 0020  2007/04/13 14:20:08 2007/04/13 14:20:08
 ____ REDBOOK1 JOB25736 A CC 0020  2007/04/13 14:20:38 2007/04/13 14:20:38
 ____ REDBOOK1 JOB25741 A CC 0099  2007/04/13 15:01:56 2007/04/13 15:01:56
 ____ REDBOOK1 JOB25742 A CC 0099  2007/04/13 15:03:57 2007/04/13 15:03:58
```

Figure 8-4 Configuring communications in the client panel

Figure 8-5 shows the ISPF window with the ST panel selected.

```
 File  SDSF Panel  Communication
---------------------------------------------------------------------------
                               SDSF Client                Row 1 to 26 of 91
Command ===>                                             Scroll ===> CSR

NP    Jobname  Jobid    C Max-RC   St-Date    St-Time  End-Date   St-Time
____  REDBOOKS JOB26219 A          2007/04/17 09:45:21            00:00:00
____  REDBOOK  TSU26233            2007/04/17 10:24:40            00:00:00
____  REDBOOK1 JOB25700 A CC 3632  2007/04/13 11:38:41 2007/04/13 11:38:41
____  REDBOOK1 JOB25702 A CC 0501  2007/04/13 11:44:48 2007/04/13 11:44:48
____  REDBOOK1 JOB25703 A CC 3632  2007/04/13 11:47:21 2007/04/13 11:47:22
____  REDBOOK1 JOB25704 A CC 0000  2007/04/13 11:50:59 2007/04/13 11:51:00
____  REDBOOK1 JOB25706 A CC 0000  2007/04/13 11:55:12 2007/04/13 11:55:13
____  REDBOOK1 JOB25707 A CC 0000  2007/04/13 11:57:47 2007/04/13 11:57:48
____  REDBOOK1 JOB25710 A CC 0000  2007/04/13 13:12:26 2007/04/13 13:12:27
____  REDBOOK1 JOB25711 A CC 0000  2007/04/13 13:14:03 2007/04/13 13:14:04
____  REDBOOK1 JOB25712 A CC 0000  2007/04/13 13:15:53 2007/04/13 13:15:54
____  REDBOOK1 JOB25713 A CC 0000  2007/04/13 13:17:42 2007/04/13 13:17:43
____  REDBOOK1 JOB25714 A CC 0000  2007/04/13 13:17:55 2007/04/13 13:17:56
____  REDBOOK  TSU25673   CC 0000  2007/04/13 07:24:29 2007/04/13 13:19:06
____  REDBOOK1 JOB25716 A CC 0000  2007/04/13 13:21:16 2007/04/13 13:21:17
____  REDBOOK1 JOB25717 A CC 3632  2007/04/13 13:23:39 2007/04/13 13:23:39
____  REDBOOK1 JOB25718 A CC 0000  2007/04/13 13:24:11 2007/04/13 13:24:12
____  REDBOOK1 JOB25720 A CC 0000  2007/04/13 13:28:51 2007/04/13 13:28:52
____  REDBOOK1 JOB25722 A CC 0000  2007/04/13 13:30:35 2007/04/13 13:30:36
____  REDBOOK1 JOB25723 A CC 3632  2007/04/13 13:41:39 2007/04/13 13:41:39
____  REDBOOK1 JOB25724 A CC 3632  2007/04/13 13:42:44 2007/04/13 13:42:44
____  REDBOOK1 JOB25725 A CC 0000  2007/04/13 13:43:57 2007/04/13 13:43:58
____  REDBOOK1 JOB25735 A CC 0020  2007/04/13 14:20:08 2007/04/13 14:20:08
____  REDBOOK1 JOB25736 A CC 0020  2007/04/13 14:20:38 2007/04/13 14:20:38
____  REDBOOK1 JOB25741 A CC 0099  2007/04/13 15:01:56 2007/04/13 15:01:56
____  REDBOOK1 JOB25742 A CC 0099  2007/04/13 15:03:57 2007/04/13 15:03:58
```

Figure 8-5 Image of the client program with ST panel selected

8.5 Extending to more complex environments

On the server side it is necessary to ensure the following precautions:

- Provide the same security that SDSF does, which implies receiving a valid user ID and an encrypted password from the remote client, validating them in RACF, and submitting the commands to SDSF on behalf of this received user ID.
- To avoid sending and receiving a user ID and an encrypted password with each request, the server might, for example, return a security token that uniquely identifies the client in each subsequent request.
- The only way to stop the sample server is to cancel the job. It is necessary to provide a way of stop the server gracefully, letting the command processors reply to their clients before quitting.
- The command processors might be more *aware* of the replies that are returned by SDSF. In the sample code that we provide, the only thing that the command processors do is reformat the dates and provide a basic color indication of the row returned.
- It is mandatory to provide some kind of parallelism to cope with more than one request at a time. Some of the requests made to SDSF can take a while to complete, and the queue of requests can grow large, to the point that system performance is affected.

On the client side, you might need to modify the extremely simple ISPF client to provide a more sophisticated look and a more pleasant user experience. It could be a workstation or Web application that provides access not only to a single sysplex system but to a group of geographically dispersed sysplex systems.

9

JOB schedule and control

In this chapter, we describe a simple application that acts as a job scheduler by executing a group of jobs according to some rules without any manual intervention.

The application allows the user to group jobs, to define the dependencies of the jobs that compose the group, and to run them unattended. The jobs are also monitored during their execution. When one job ends, its return code is checked to define which job of the group is the next job to be executed. Determining which job is the next job to be executed is done by following the dependencies that are specified by the user. While a job is running, you can use the application to monitor the system resources used by the running job.

To perform these tasks, the application needs to interact with the system to start the jobs and to monitor their execution. This interaction can be done easily using the services that are provided by the REXX with SDSF interface. Of course, the system provides many other different ways to do this, but the use of REXX with SDSF makes this interaction very simple.

The scenario that we describe here is based on a client/server model to take advantage of the graphics capability that is available on a workstation, because we want to track job submissions and executions using a graphical interface.

9.1 Scenario description

Imagine that you have a group of jobs that you want to execute on your system. Usually, without the use of an automation tool, you must start these jobs one at time and check the return code to decide which will be the next job to be executed. In addition, the sequence in which the jobs are executed depends on the logic of the application and on the behavior of the job at run time. We call these types of issues *dependencies*.

Some dependencies are hard coded by the application logic that runs the jobs in a predefined sequence. As an example, one job can write a file that a subsequent job needs to read. This type of dependency is a *solid dependency* that can be defined before the jobs run. However, there are other dependencies that can be seen only during the runtime of the jobs. For example, one job can end with a different return code depending on its input. This kind of dependency is only *visible* when one job ends. Again, on an application logic basis, we might need to run a different job depending on the return code of the preceding one.

Figure 9-1 represents a logical flow of a group of jobs where *JOB_x* is the name of a job.

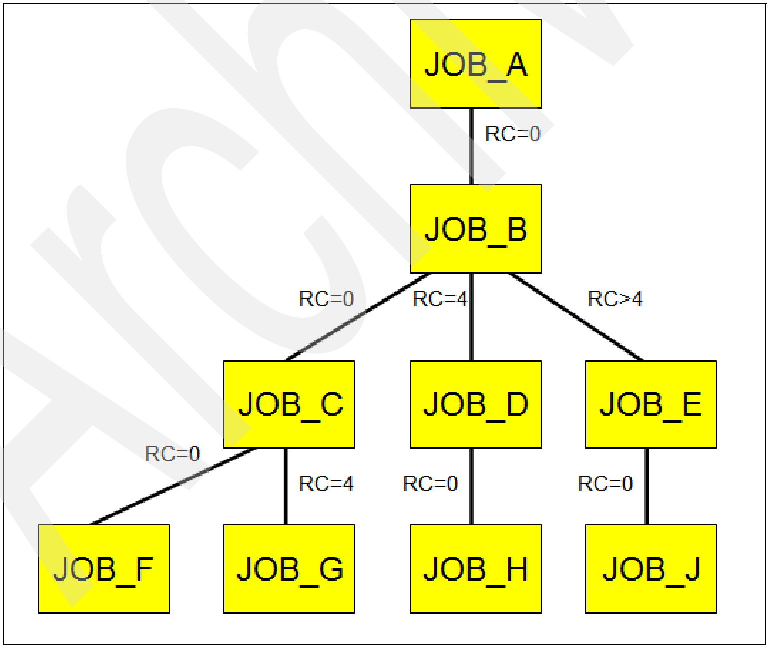


Figure 9-1 Example flow of a group of jobs

JOB_A is the first job that we want to execute. If the execution completes with a return code of 0 (RC=0), then we want to start JOB_B. Again, when JOB_B completes, we want to start JOB_C, JOB_D, or JOB_E, *depending on the return code of JOB_B*. If RC=0, then we start JOB_C. If RC=4, then we start JOB_D. If RC>4, then we start JOB_E, and so forth.

We can also decide to check the job execution and take some actions based on its behavior. For example, we might decide to cancel the job if it consumes too much CPU time or if the paging activity for this job is too high. In this case, we can notify someone through e-mail when one job completes with an unacceptable return code.

9.2 Implementation

The scheduler in this scenario is implemented as a client/server application. The server part is a REXX procedure that runs on a z/OS host system. It is the same server code that we use in Chapter 10, “SDSF data in graphics” on page 223. The client part is a Java™ program that runs, in our case, on a Windows workstation.

9.2.1 Server program

This scenario uses the same server program described in Chapter 10, “SDSF data in graphics” on page 223. You can refer to that chapter for further information about it.

This sample application makes two assumptions.

- The jobs to be executed must be submitted before the application is started. These jobs must be available as an input class, and `no init` must be available to take jobs for this class. In our scenario, the jobs are submitted, specifying as execution `class=I`. We do not have any `init` available to pick up jobs from this class in our installation.
- When finished, the jobs must be available on the `Held` output queue.

9.2.2 Client program

We can divide the client program into three parts:

- The first part allows the user to create a logical flow chart of a group of jobs and to describe the dependencies between them. We can also describe the actions to take when some limits are reached at execution time for a specific job.

- The second part includes the code and functions that are needed to submit a job, check its execution, and check the maximum return code of the job.
- The third part includes code to monitor the execution of a job and to take actions based on some rules that are defined by the user.

When you start the program, you are presented with the window as shown in Figure 9-2.

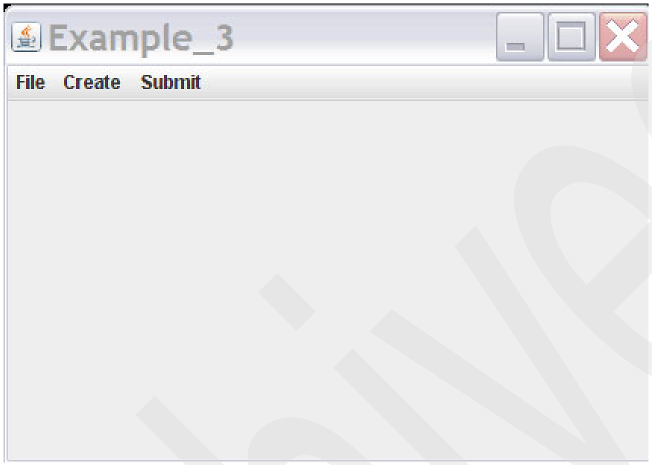


Figure 9-2 Program control window

If you were to run a client program such as this, you would proceed in the following fashion:

1. If you need to change the IP address or the port number of your server, click **File** → **Configure** (Figure 9-3). Then, click **Save**.

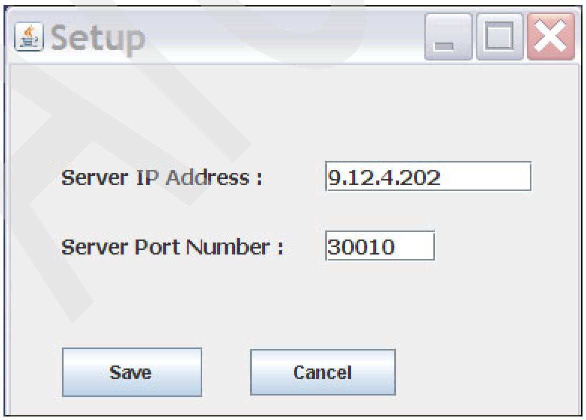


Figure 9-3 Configuring the server IP address and port number

2. You are then presented with the opportunity to describe the dependencies and execution limits for a group of jobs. To do this, you will need to identify the dependencies and the execution limits for a group of jobs. You need to identify a group of jobs that you want to execute and then describe to the application the relations and dependencies of the jobs within this group.

   As an example, assume that you want to group and run the jobs with names *DARIO100* to *DARIO910*. These jobs compose a batch application, and you want to group them because they are expected to be executed in a certain order. The execution of one of them also depends on the return code of the preceding job.

   Figure 9-4 shows the dependencies of these jobs.

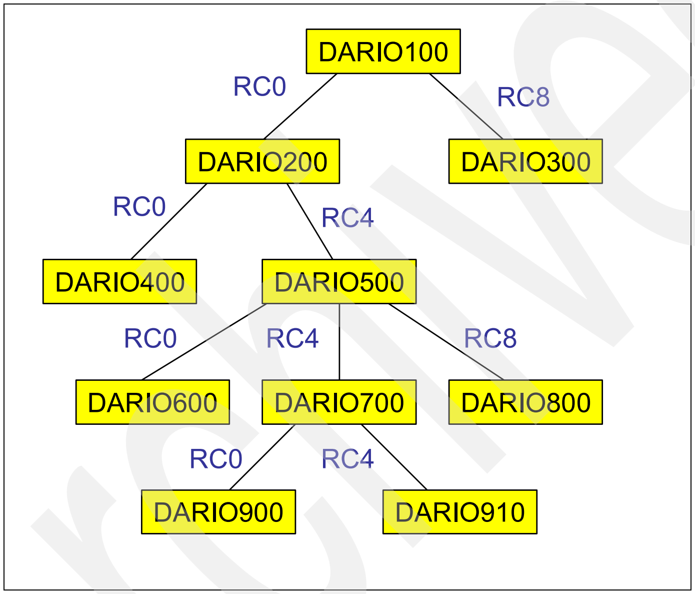


Figure 9-4 Example job flow

3. To describe the job flow shown in Figure 9-4, you need to create a job group. To do this click **Create** → **Job Group**. You are presented with the window shown in Figure 9-5.

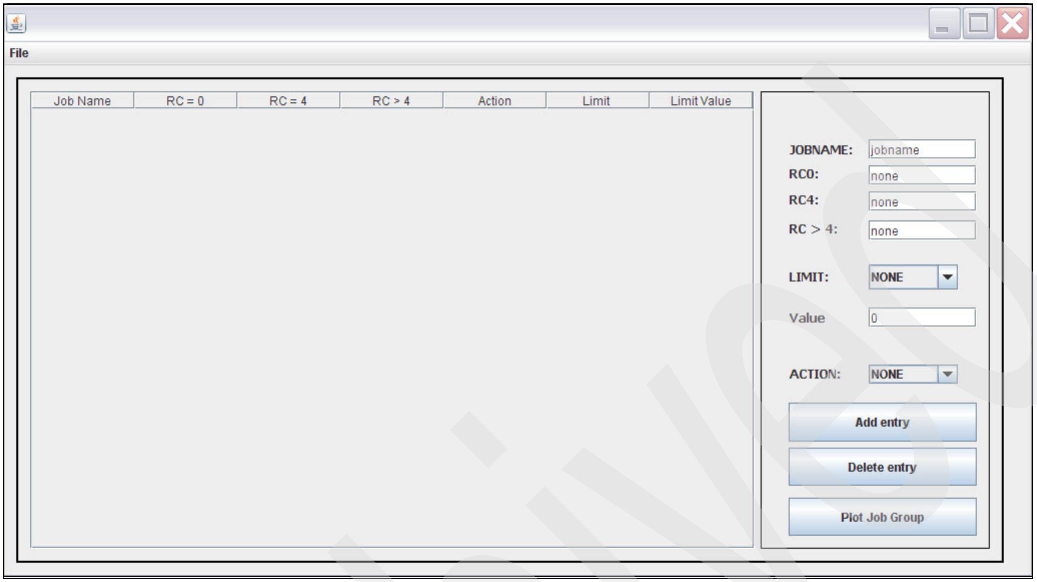


Figure 9-5 Creating a job group

On the right side of the window, there are some entry fields and list boxes that are used to describe the jobs in the group. Now, you need to describe how to reflect the dependencies of the group of jobs that are reported in Figure 9-4 on page 205.

4. In our example, we start with job DARIO100. It is the first job in the chain. So, it does not have any predecessors. Figure 9-6 shows a close up of this job. Complete the JOBNAME: entry field with the jobname `DARIO100`.

5. Now, you have to describe what to do when each job ends:

   – If the job ends with `RC=4`, you want to start the job DARIO200. Complete the entry field RC0: with `DARIO200`.
   – If the job ends with `RC=8`, you want to start the job DARIO300. Complete the entry field RC > 4 with `DARIO300`.
   – Where there is no entry, type the word `none`.
   – For now, skip the remaining columns (limits and actions) of the window. We will explain these columns later.

6. When you have completed describing your first job in the chain, press the **ADD entry** button (Figure 9-6) to accept this data.

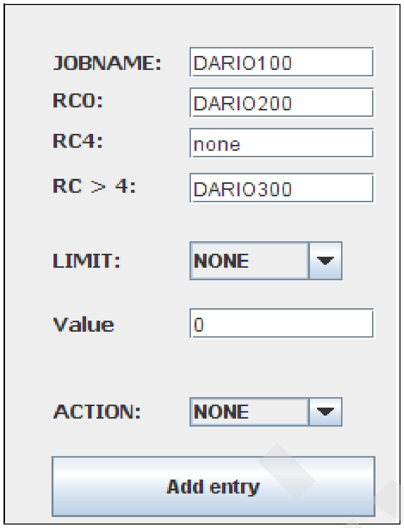
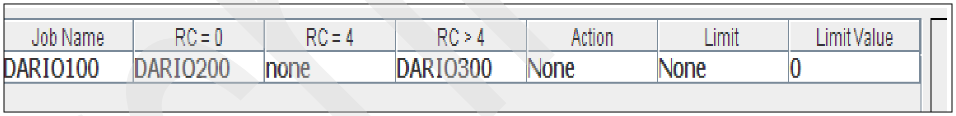


Figure 9-6 Adding entries to the job group

The windows is updated as shown in Figure 9-7.



Job Name	RC = 0	RC = 4	RC > 4	Action	Limit	Limit Value
DARIO100	DARIO200	none	DARIO300	None	None	0

Figure 9-7 Updated job group

7. Continue adding job entries to the group in this fashion by typing over the last entry in each field and clicking the Add Entry button. Insert the data for all the jobs in the group. When you have completed these steps for all of the jobs, your list will look something like that shown in Figure 9-9.

Job Name	RC = 0	RC = 4	RC > 4	Action	Limit	Limit Value
DARIO100	DARIO200	none	DARIO300	None	None	0
DARIO300	none	none	none	None	None	0
DARIO400	none	none	none	None	None	0
DARIO500	DARIO600	DARIO700	DARIO800	None	None	0
DARIO600	none	none	none	None	None	0
DARIO800	none	none	none	None	None	0
DARIO700	DARIO900	DARIO910	none	None	None	0
DARIO900	none	none	none	None	None	0
DARIO910	none	none	none	None	None	0
DARIO200	DARIO400	DARIO500	none	None	None	0

Figure 9-8 List of jobs in the job group

As mentioned previously, you might want to look at some system indicators while a specific job is running. You can put a limit on some system resources that this job is using.

In this scenario, we decided to monitor the EXCP count and the CPU time used by a job. You can define a limit for these resources on a job basis and decide to take an action when the limit is reached.

You can define a limit when you define the group of jobs. At this time, you can define which limit you have to monitor for a job and which action you take when this limit is reached. Figure 9-9 shows that a CPU time limit for the job with the name *DARIO200*.

DARIO910	none	none	none	None	None	0
DARIO200	DARIO400	DARIO500	none	Mail	CPU-Time	8

Figure 9-9 CPU time limit placed on a job

For this scenario, we also want the application to send an e-mail message when the job is absorbing more that eight seconds of CPU time during its execution. To set a limit, you can use the box on the right side of the job group windows:

If you want to have a graphics representation of your job dependencies, select **Plot Job Group**. The window shown in Figure 9-10 opens. You can use the button on the small control window to move the graph, zoom on it, and so forth.

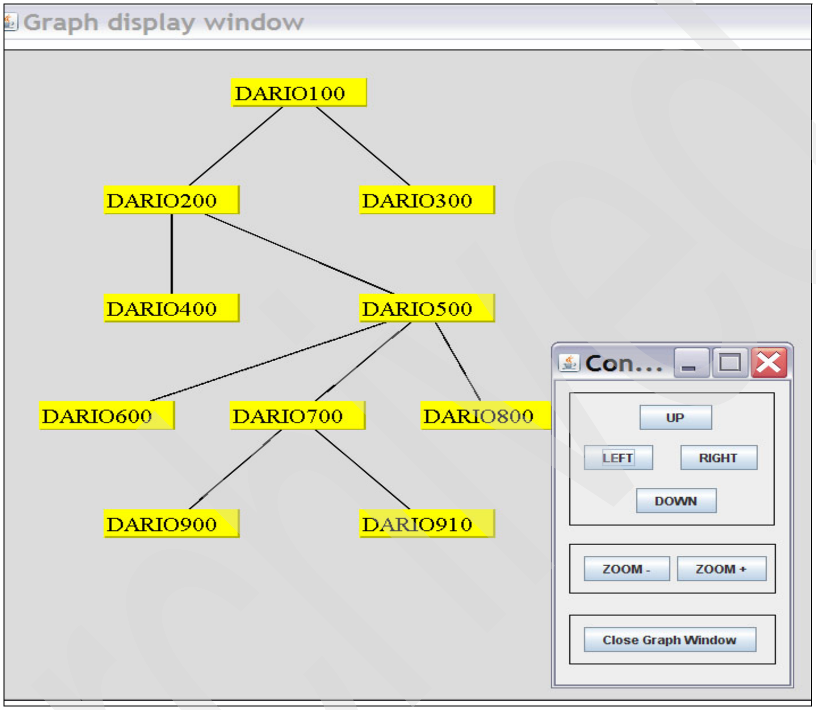


Figure 9-10 Plotting job groups graphically

After you have successfully described the group of jobs to your scheduler, you can save it by clicking **File** → **Save**. You can also recall an already saved job group description by clicking **File** → **Load**.

You are now ready to submit your group of jobs which will follow the sequence you have defined for the job group. The application expects to find the jobs already submitted with a specific execution class assigned to them (that is, `CLASS=I`).

Note: You cannot have any initiator available to pick up a job from this input class.

When the job is submitted, it sits in the input queue waiting for someone to change its execution class. As an example, we decided to submit jobs with an execution `CLASS=I`, because we do not have an initiator that selects this class in our system. When our sample scheduler program wants to start a job, it simply asks SDSF to change the execution class of the job from `I` to `A`, because we have initiators available for this execution class.

At this point, you should have all the jobs that comprise your group already submitted and available in a JES2 queue. In our test, they look as shown in Example 9-1.

Example 9-1 Input queue displaying all classes

```
SDSF INPUT QUEUE DISPLAY ALL CLASSES                    DATA SET
DISPLAYED
COMMAND INPUT ===>                                            SCROLL
===>
NP   JOBNAME  JobID    Owner    Prty C  Pos  PrtDest              Rmt
Nod

     DARIO100 JOB28385 DARIO      5 I     1 WTSCPLX2
     DARIO200 JOB28386 DARIO      5 I     2 WTSCPLX2
     DARIO300 JOB28387 DARIO      5 I     3 WTSCPLX2
     DARIO400 JOB28388 DARIO      5 I     4 WTSCPLX2
     DARIO500 JOB28389 DARIO      5 I     5 WTSCPLX2
     DARIO600 JOB28390 DARIO      5 I     6 WTSCPLX2
     DARIO700 JOB28391 DARIO      5 I     7 WTSCPLX2
     DARIO800 JOB28392 DARIO      5 I     8 WTSCPLX2
     DARIO900 JOB28393 DARIO      5 I     9 WTSCPLX2
```

Close the job group windows and the graphic windows.

From the main windows of the sample application click **Submit** → **Submit Group**. A window similar to Figure 9-11 opens.

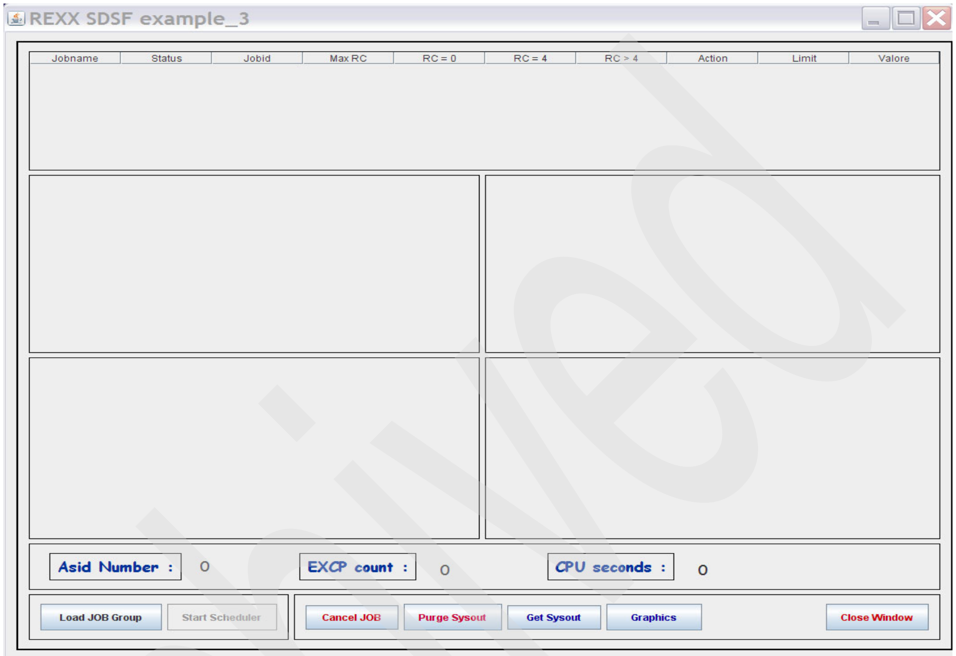


Figure 9-11 Submitting the group

Select **Load JOB Group** and, from the load dialog box, select the file that includes the description of the job dependencies, which you created previously. The list box in the upper part of the window is updated as shown in Figure 9-12.

Jobname	Status	Jobid	Max RC	RC = 0	RC = 4	RC > 4	Action	Limit	Valore
DARIO100	none	none	none	DARIO200	none	DARIO300	None	None	0
DARIO300	none	none	none	none	none	none	None	None	0
DARIO400	none	none	none	none	none	none	None	None	0
DARIO500	none	none	none	DARIO600	DARIO700	DARIO800	None	None	0
DARIO600	none	none	none	none	none	none	None	None	0
DARIO800	none	none	none	none	none	none	None	None	0
DARIO700	none	none	none	DARIO900	DARIO910	none	None	None	0
DARIO900	none	none	none	none	none	none	None	None	0
DARIO910	none	none	none	none	none	none	None	None	0

Figure 9-12 Updating the list box by loading the job group

The Status field of the list box includes the status of the jobs during their execution. You just loaded the list at this point, and so status of the jobs is `none`. Select **Start Scheduler** to continue. When you select this button, you ask the application to check whether all the jobs listed in the list box are really present in the input queue in class `I`. Figure 9-13 shows the list box contents change.

Jobname	Status	Jobid	Max RC	RC = 0	RC = 4	RC > 4	Action	Limit	Valore
DARIO100	Input Queue	JOB28385	none	DARIO200	none	DARIO300	None	None	0
DARIO300	Input Queue	JOB28387	none	none	none	none	None	None	0
DARIO400	Input Queue	JOB28388	none	none	none	none	None	None	0
DARIO500	Input Queue	JOB28389	none	DARIO600	DARIO700	DARIO800	None	None	0
DARIO600	Input Queue	JOB28390	none	none	none	none	None	None	0
DARIO800	Input Queue	JOB28392	none	none	none	none	None	None	0
DARIO700	Input Queue	JOB28391	none	DARIO900	DARIO910	none	None	None	0
DARIO900	Input Queue	JOB28393	none	none	none	none	None	None	0
DARIO910	none	none	none	none	none	none	None	None	0

Figure 9-13 List box change after starting the scheduler

When you select Start Scheduler, the client sends a request to the server to list all the jobs that are actually present in the input queue `class I` that belong to user *DARIO*. This is the user ID that submitted the jobs. Its name is hard coded into the client code. You can change it to meet your installation specification. The server calls SDSF to retrieve this data.

Figure 9-2 is a fragment of code extracted from the REXX clist `my_task`.

Example 9-2 Listing all jobs in a particular input class with a specific owner

```
call_sdsf_get_i_data: procedure expose peer_socket

  isfprefix = "*"
  isfowner = "dario"
  isfcols = "jname jobid"
  /* isffilter = "jclass eq i" */

  Address SDSF "ISFEXEC I"
  isf_rc = rc
  call check_isf_rc isf_rc
```

When the response to this request comes from the server, the client code checks whether all the jobs included in the job group in use are in the input queue. If the job is found in the JES input queue, its status changes to `Input Queue`, and the color of the list box cell changes to yellow. If the job is not found in the JES2 input queue, the status of the job remains `none`, and the list box cell is painted in cyan to indicate that this job is missing from the input queue.

Submit the job DARIO910 to make it available in the spool. Again, select **Start Scheduler**. This time, all the jobs are available, and the scheduler starts to submit jobs.

To submit a job, the sample scheduler application changes the job's class in the input queue from I to A. Example 9-3 is an extract of the code that is executed by the server program when it receives a request to submit a job.

Example 9-3 Request received to submit a job

```
submit_jobid: procedure expose peer_socket
arg my_jobid

isfprefix = "dario*"
isfowner = "dario"
isfcols = "jname jclass jobid"
/* isffilter = "jobid eq " || jobid */

Address SDSF "ISFEXEC I"
isf_rc = rc
call check_isf_rc isf_rc

do a = 1 to JNAME.0
  if (jobid.a = my_jobid) then do
    Address SDSF "ISFACT I TOKEN('"token.a"') PARM(JCLASS A)"
    isf_rc = rc
```

The client passes the JOBID to the server of the job to be executed. The server looks at the I queue and when it finds the job, it issues an ISFACT command to change the class of the job.

Figure 9-14 shows the status of the first job (DARIO100) while it is executing.

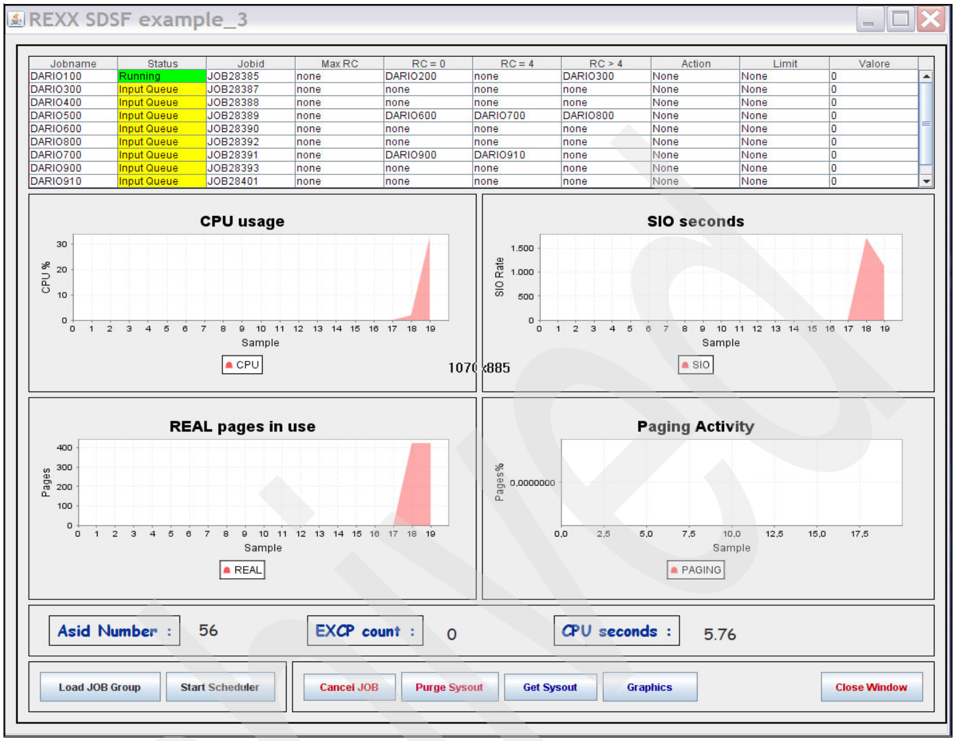


Figure 9-14 Job scheduler in action after submission of first job

In the status column, the status of the job `DARIO100` is changed to `Running`, and the list box cell is green, indicating that the job is running. Every five seconds, the client issues a request to the server to check the status of the job. The response also includes some information about the system resources that are used by the job that is executed. Example 9-4 is an extract of the code that is used by the server whenever the server receives a request to check the status of the job from the client.

Example 9-4 Request to check status of jobs

```
check_if_jobid_is_executing: procedure expose peer_socket
  arg jobid

  jobid = word(jobid,1)
  isfcols = "jname jobid paging excprt cpupr asid cpu excp real"
  isffilter = "jobid eq " || jobid

  Address SDSF "ISFEXEC DA"
  isf_rc = rc
  call check_isf_rc isf_rc
```

Four graphs are included in the middle of Figure 9-14:

- The upper-left graph represents the CPU use of the job that is executed. The values plotted are gathered from the `cpupr` field of the DA panel.
- The upper-right graph represents the number of **`sio`** issue by the job that is executed. The values for this graph are obtained from the `excprt` field of the DA panel.
- The lower-left graph represents the number of real storage pages in use by the job that is executed. These values are obtained from the `real` field of the DA panel.
- The lower-right graph represents the paging activity of the job that is executed. This value is obtained from the `paging` field of the DA panel.

Below the graph, you might see the **`asid`** number in which the job is running, the number of EXCPs issued by the job since the start of its execution, and the number of CPU seconds used. These value are obtained from the fields `asid`, `excp`, and `cpu` fields of the DA panel.

When the application ends, all the jobs listed in the job group that was loaded have been executed. Figure 9-15 shows the end of the execution.

Figure 9-15   End of execution

The status field of the window is changed to `Ended` for those jobs that have been started and are complete. The color of the list box cell changes to blue. The `Max RC` value also changes to reflect the **`maxrc`** value for the job as reported by the `retcode` field into the H panel. When the job no longer appears in the DA output, the client assumes that the job has ended, and it issues an H command to retrieve the return code value. Example 9-5 is an extract of the server code that is executed when the client requests to check the return code of a job that is completed.

Example 9-5   Checking the return code of a job that is complete

```
get_job_maxrc: procedure expose peer_socket
arg jobid

isfprefix = "DARIO*"
isfowner = "DARIO"
isfcols = "jname jobid retcode"
```

```
isffilter = "jobid eq " || jobid

Address SDSF "ISFEXEC H"
isf_rc = rc
call check_isf_rc isf_rc
```

Looking at the list box values in Figure 9-15 on page 216, you can check the job while it is executing. You can also follow the scheduling of the job by selecting **Graphics** on the lower side of the window. Figure 9-16 is a graphical representation of the jobs as they are executed.

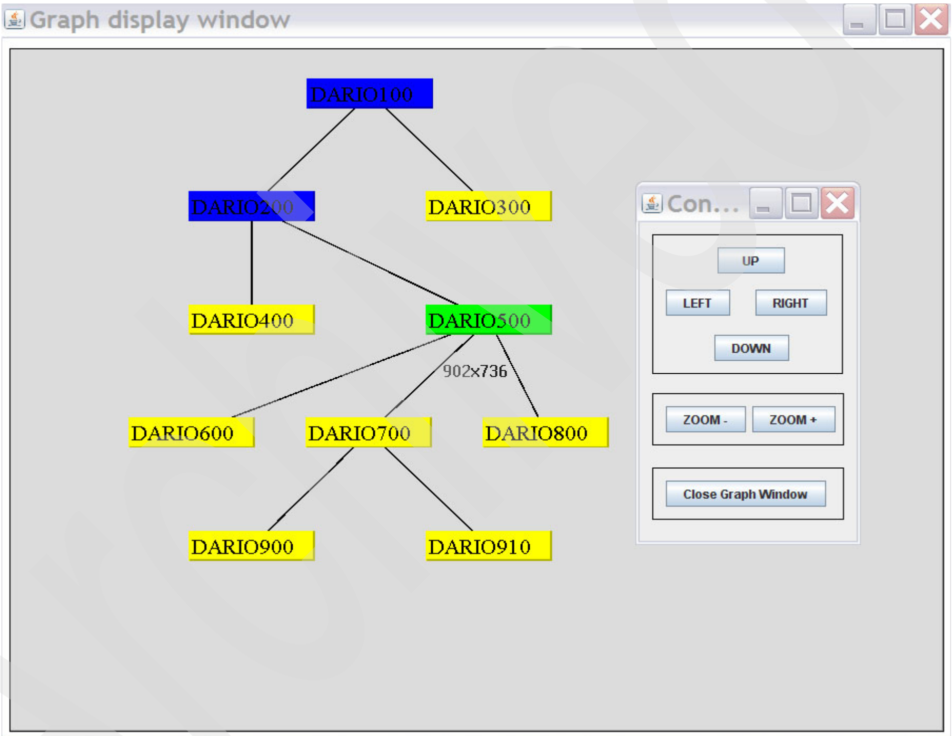
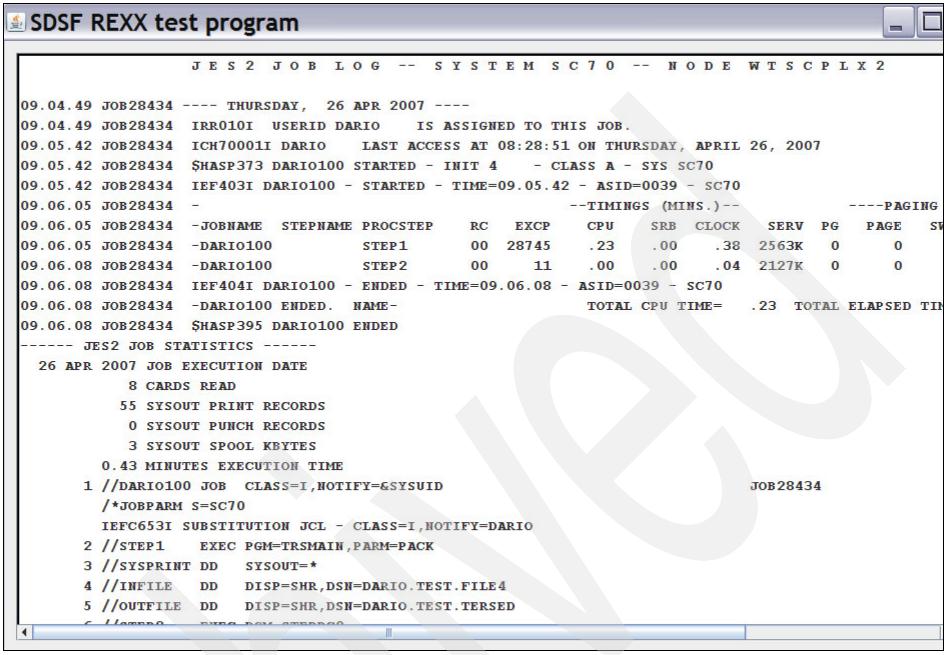


Figure 9-16 Graphical display of jobs running

In the graphical display window, as in the list box, the job that is executed is marked in green, and the jobs that have ended are marked in blue.

When all the jobs in the group end their execution, you can also look at their system output (SYSOUT) by selecting a job in the list box and selecting **Get Sysout**. Note that the sample application expects to find the system output of a job in the `Held` queue. If this method is not acceptable in your organization, change the source code to reflect your requirement. (For information about how to obtain the source code, see Appendix B, “Additional material” on page 305.) Figure 9-17 shows the SYSOUT of a job that has completed.



```
SDSF REXX test program
                   J E S 2  J O B  L O G  --  S Y S T E M  S C 7 0  --  N O D E  W T S C P L X 2

09.04.49 JOB28434 ---- THURSDAY,  26 APR 2007 ----
09.04.49 JOB28434  IRR010I  USERID DARIO    IS ASSIGNED TO THIS JOB.
09.05.42 JOB28434  ICH70001I DARIO    LAST ACCESS AT 08:28:51 ON THURSDAY, APRIL 26, 2007
09.05.42 JOB28434  $HASP373 DARIO100 STARTED - INIT 4    - CLASS A - SYS SC70
09.05.42 JOB28434  IEF403I DARIO100 - STARTED - TIME=09.05.42 - ASID=0039 - SC70
09.06.05 JOB28434  -                                      --TIMINGS (MINS.)--            ----PAGING
09.06.05 JOB28434  -JOBNAME  STEPNAME PROCSTEP    RC   EXCP    CPU    SRB  CLOCK   SERV  PG   PAGE   SW
09.06.05 JOB28434  -DARIO100          STEP1       00  28745    .23    .00    .38  2563K   0      0
09.06.08 JOB28434  -DARIO100          STEP2       00     11    .00    .00    .04  2127K   0      0
09.06.08 JOB28434  IEF404I DARIO100 - ENDED - TIME=09.06.08 - ASID=0039 - SC70
09.06.08 JOB28434  -DARIO100 ENDED.  NAME-                     TOTAL CPU TIME=   .23  TOTAL ELAPSED TIM
09.06.08 JOB28434  $HASP395 DARIO100 ENDED
------ JES2 JOB STATISTICS ------
  26 APR 2007 JOB EXECUTION DATE
            8 CARDS READ
           55 SYSOUT PRINT RECORDS
            0 SYSOUT PUNCH RECORDS
            3 SYSOUT SPOOL KBYTES
         0.43 MINUTES EXECUTION TIME
       1 //DARIO100 JOB  CLASS=I,NOTIFY=&SYSUID                                  JOB28434
         /*JOBPARM S=SC70
         IEFC653I SUBSTITUTION JCL - CLASS=I,NOTIFY=DARIO
       2 //STEP1    EXEC PGM=TRSMAIN,PARM=PACK
       3 //SYSPRINT DD   SYSOUT=*
       4 //INFILE   DD   DISP=SHR,DSN=DARIO.TEST.FILE4
       5 //OUTFILE  DD   DISP=SHR,DSN=DARIO.TEST.TERSED
```

Figure 9-17 System output of a completed job

When you select Get Sysout, the server program allocates the SYSOUT data set and reads it using EXECIO. For additional information about this program, refer to Chapter 10, “SDSF data in graphics” on page 223. The program can retrieve a small number of rows from SYSOUT. In our testing, we observed that a maximum of 300 rows provides an acceptable response time.

You can purge a SYSOUT by selecting a job from the list box and selecting **Purge Sysout**. For additional information, see Chapter 10, “SDSF data in graphics” on page 223.

9.2.3 Personalizing the server code

As discussed previously, the sample application expects the jobs that are executed to be available in `class I` with a specific user ID assigned. The user ID in our case is *DARIO*. If you need to change any of these values in our sample for your own use, you must do so in the server code Example 9-6 is an extract of code from the **`my_task`** REXX program. We have highlighted the SDSF calls where you need to modify the code for your environment in bold.

Example 9-6 Personalizing the server code

```
/* This routine submits the job with jobid passed as parm          */
/* It means that we change the class of the job from I to A        */
/* We assume that the job is actually in the inout queue class=I   */
/* To submit the job we simply change its class from I to A        */
/*-----------------------------------------------------------------*/
submit_jobid: procedure expose peer_socket
arg my_jobid

isfprefix = "dario*"
isfowner = "dario"
isfcols = "jname jclass jobid"
/* isffilter = "jobid eq " || jobid */

Address SDSF "ISFEXEC I"
isf_rc = rc
call check_isf_rc isf_rc

do a = 1 to JNAME.0
  if (jobid.a = my_jobid) then do
    Address SDSF "ISFACT I TOKEN('"token.a"') PARM(JCLASS A)"
    isf_rc = rc
    call check_isf_rc isf_rc
    if (isf_rc = 0) then do
      call send_response_to_peer "submitted"
    end
    leave
  end
end

return
```

9.3 Compile and customize the sample programs

In this section we describe how to install and compile the sample applications that we implement in this chapter.

Before you begin, you need to download the compressed file that includes the source code of the programs. For information about how to download this file, see Appendix B, “Additional material” on page 305. The .zip file includes both the server and the client code. Follow the instructions to extract the source code, go to the ch9and10 directory, and find the directories example_3 and rexx. In these directories are the class files of the client Java programs and the server REXX code that you need to upload to the host system.

Next, upload the REXX code using your preferred method (FTP, PC3270 file transfer, or another method) into a UNIX System Services file system. You might need to verify the TCP/IP port availability with your network organization because the code uses TCP/IP sockets. For sample JCL that you can use to start the server, see Chapter 10, “SDSF data in graphics” on page 223.

We compiled the Java sample programs using JDK1.6.0 (Standard Edition) on a Windows XP workstation. As for the example in Chapter 10, “SDSF data in graphics” on page 223, this program uses a library from JFreeChart, an open source graphics rendering package (see `http://www.jfree.org/`), to draw graphics. You can refer to the notes in that chapter about how to download and install the library from JFreeChart. In addition, the sample program in that chapter also uses JavaMail™ classes to send e-mail messages. You can download JavaMail from the following Web site:

`http://java.sun.com/products/javamail/`

Follow the instructions to download and install the JavaMail classes on your computer. When done, you also need to update your CLASSPATH variable to include two jar files that are supplied by JFreeChart and two jar files that are supplied with JavaMail. You need these jar files to compile the programs.

Your classpath variable should include an entry for the following .JAR files:

- jcommon-1.0.8.jar
- jfreechart-1.0.4.jar
- activation.jar
- mail.jar

> **Note:** The activation.jar file is part of the JavaBeans™ Activation Framework (JAF). You can download it (if needed) from the following Web site:
>
> `http://java.sun.com/products/javabeans/jaf/downloads/index.html`

> **Note:** If you are running an enterprise edition of the Java Developer Kit (JDK™), you might have the JavaMail and JAF jar file already installed.

The Java programs have some variables with hard coded values in them. You need to update these values to adapt them to your installation.

The **send_an_email** method in the submit_worker class includes all the information needed to send an e-mail. See Example 9-7.

Example 9-7 Sending an e-mail using send_an_email

```
//--------------------------------------------------
  private void send_an_email(int tipo, String jobname)
  {
    email.mail_subject = "Jobname: " + jobname + " limit exceeded.";
    email.from_address = "example3@it.ibm.com";
    email.to_address = "destinatio@it.ibm.com";
    email.set_mail_server("emea.relay.ibm.com");
    if (tipo == 0) email.mail_text = "The job in subject hit the EXCP 
limit imposed.";
    if (tipo == 1) email.mail_text = "The job in subject hit the CPU 
limit imposed.";
    email.run();
  }
```

The name of the methods used are self explanatory. You need to update or change them to reflect your installation needs. You need to at least update the to_address and the set_mail_server methods. The parameter that is passed to the to_address method is the name of the designatory of the e-mail that is sent when a limit is hit. The set_mail_server parameter is the name of the SMTP server in your organization, that you can use to send your e-mail.

10

SDSF data in graphics

Using REXX with SDSF support, you can access IBM z/OS System Display and Search Facility (SDSF) functions through REXX variables and read SYSOUT data sets using EXECIO. With this interface, a REXX procedure can easily obtain, manipulate, and reduce SDSF data. You can run the REXX procedure in batch mode or under TSO. In the latter case, as an example, a REXX procedure can use the ISPF facilities to create panels to present structured data to the user.

Instead of using a 3270 screen to present the user with the data gathered by the REXX interface, we decided to proceed in a different way. We collected the SDSF data on the host system and sent the data to a workstation. Using this method, we can plot or display the data graphically on a local personal computer.

This process is accomplished through the implementation of three small client-server applications that use the REXX socket functions that are provided by TCP/IP for communication between the workstation and the host system. These applications use the new REXX support to access SDSF functions and data on the host side.

- The first sample application gathers CPU consumption data from SDSF DA panel and plots them graphically on a workstation screen.
- The second one lists, reads, and cancels a SYSOUT data set.
- The third sample application issues some system commands, summarizes the data, and presents the output to the user in a graphical mode.

Note: We built these sample applications for test purposes. They do not have the completeness, performance, and reliability of production applications. The main objective of these sample applications is to show what can be done with the new REXX with SDSF interface.

The client programs use the JFreeChart Java library to make graphics. This library is covered by the GPL GNU Lesser General Public Licence. For further details, refer to `http://www.jfree.org/` under the link to JFreeChart where you can find information about how to download and install JFreeChart.

10.1 TCP/IP socket communications

Before we start to describe in detail what our application does, let us take a moment to describe how the TCP/IP socket communication works, assuming a basic knowledge of TCP/IP.

Sockets are a mechanism provided by TCP/IP that allow two programs to communicate with each other using a defined set of functions. The two programs can reside on the same computer (the same TCP/IP stack) or on different computer systems in the network.

In TCP/IP, every machine, or host, can have one or more *IP addresses* assigned to it. A machine can have more addresses assigned to it, for example, depending, on the number of hardware adapters that connect the machine to the network.

In our context, for simplicity, we assume that our machines have a unique IP address assigned to them.

You can think of an IP address as a phone number. A family usually has a single phone number assigned to it, but a company or a hotel might have several phone numbers assigned to it. When you want to call a specific person, you dial that person's number. With a company that has many phone numbers assigned to it, you can choose to call a phone number for the company that might direct you to an operator or receptionist, you might choose to dial a direct phone number to talk with a specific person in the company. In any case, you always connect to this specific company first.

The *port* is a unique number in a specific host that identifies a program or a service available on this host. This program or this service can be contacted by any client in the network that can reach this host.

In our phone number example, you can think of the port number as a company phone extension. When you want to contact, using a phone, a specific person in a company, you dial the company number first (IP address) and then the person's extension number (port).

The *socket* (talking about TCP/IP) is basically the combination of the IP address and the port number. With our phone example in mind, this is the invisible wire that connects two phone users when communications have been established between them. The socket, like a phone connection, allows two programs to communicate with each other. In the telephone example, you must first make a phone call and, when the other party answers, you can then establish your conversation (socket).

Figure 10-1 show two hosts called A and B, respectively. Host A has the IP address *10.10.10.100* assigned and host B has the IP address *10.10.10.5* assigned.

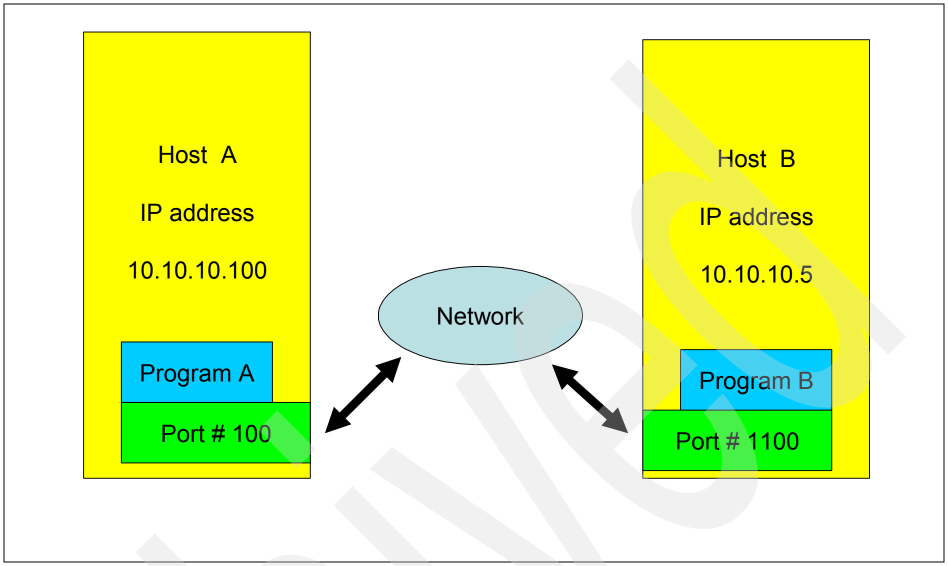


Figure 10-1 Communication between two host systems

Assuming that program A in host A wants to communicate with program B in host B, host A has to connect to `10.10.10.5` on port `1100`. In TCP/IP socket terminology, we issue a `connect()` function that specifies the IP address of the host name and the port number to which we want to connect.

After program B agrees to receive the call from program A, both the programs can talk to each other in a bi-directional way using respectively port `100` (program A in host A) and port `1100` (program B in host B).

10.1.1 TCP/IP socket functions

This section describes how our simple client server application is using the functions that are supplied by TCP/IP for the communication. In our implementation, the server application can work with more than one client at time. It uses the REXX built-in function *RXSOCKET* to access the TCP/IP socket interface and UNIX System Services syscalls to spawn child tasks.

Note: In this section, we describe the TCP/IP functions at a basic level, because discussing these functions in depth is beyond the scope of this book.

Figure 10-2 shows the sequence of TCP/IP socket functions that are used to establish a communications between a client and a server program. From top to bottom, you can follow the flow of the functions on the client and on the server side.

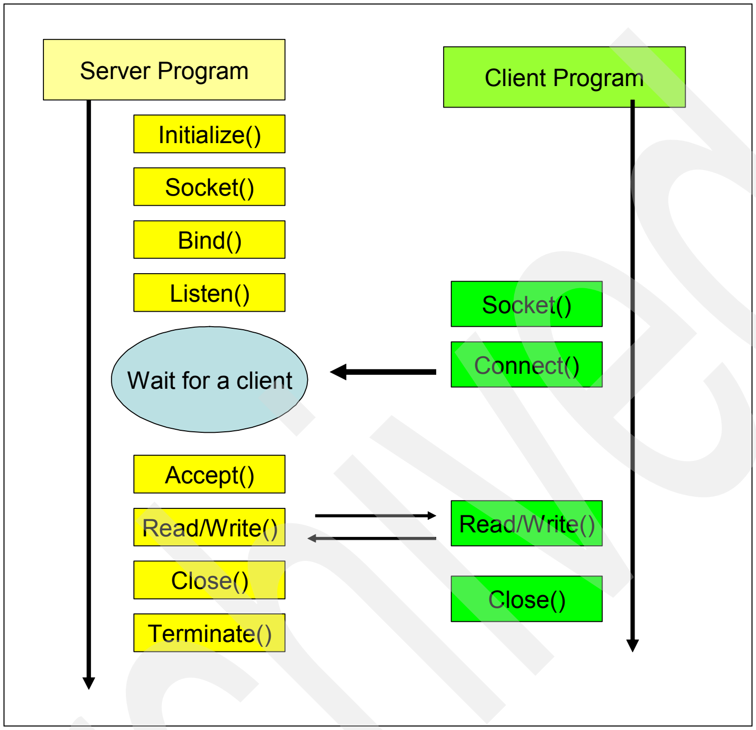


Figure 10-2 TCP/IP socket functions used to establish communication

Referring to Figure 10-2, our sever program uses the following REXX socket functions:

`socket()` Creates an endpoint for communication and returns a socket descriptor (a number) that represents that endpoint. TCP/IP supplies various types of sockets with different characteristics. A programmer can select one of these based on the characteristics of the communication to establish. In our case, we use a sock stream connection that provides a reliable 2-way connection between two peers.

`setsockopt()` Changes various options of a specific socket. As an example, in our case, we used it to translate data to ASCII when we send or receive data to or from our client program. Because the application runs on two different platforms that use two different character sets, we need to translate the data from EBCDIC to ASCII and vice versa.

`bind()` Binds a unique local name to the socket using the descriptor received from a socket call. Basically, we inform our TCP/IP stack that our program intents to use a specific port number for our communication with our client program.

`listen()` Notifies TCP/IP that we are ready to accept connection from the client program. After this function is invoked, the program will wait until a client `connect()` request comes in. In other words, the program stops waiting to be awaken by the TCP/IP stack until a client requests a connection to the server program.

`accept()` Used by the server program to accept the connection request from a client. When the function completes, the two programs are ready to send and receive data between them.

`connect()` The client invokes this function to establish a connection with the server program. Generally speaking, a server program can refuse a connection request on the basis of some rules that the server might observe. In our implementation, our server always accepts connection requests coming from the client.

`read() / write()` Used to read data from a socket and to write data to a socket.

`close()` Shuts down the socket and frees the resources allocated to it.

`initialize()` Sets up the REXX socket environment for us.

`terminate()` Terminates the REXX socket environment.

For further details about these functions, refer to *IP Programming Application Interface Guide*, SC31-8788.

10.2 Description of the server program

The server program is written in the REXX language and uses the TCP/IP REXX socket functions to access the network and uses the REXX with SDSF interface to issue SDSF commands. In addition, it uses REXX UNIX System Service (**`syscalls`**) to create child tasks. Because these REXX interfaces do not need TSO, you can also run the server program as a batch job.

The REXX implementation on z/OS does not allow, by itself, the establishment of a multi-thread programming environment. However, using the REXX **syscalls** facilities that is provided by the UNIX System Services, we can create a multitasking environment for our applications. In our implementation, we use the spawn **syscall** to have a separate process that runs on the server side for every client connected. In this way, a single server can handle multiple client connections.

We recommend that you start the server program as a batch job using the UNIX System Service BPXBATCH facility. Example 10-11 gives an example of how to invoke the facility in batch.

The server is composed of two programs:

- A main program (the parent) with name MY_SRV
- A child program with name MY_TASK

We show some extracts of the code to understand the logic of the server program. For information about how to obtain the code, refer to Appendix B, “Additional material” on page 305.

10.2.1 Initializing the program

The server side of the application is composed of two programs, although it can be considered as a single application. The server program requires one input parameter, the *port number*. This port number is where the program listens for a connection request coming from the client. If a port number is not specified, the server program terminates with a return code of 8, as shown in Example 10-1.

Example 10-1 Coding for return codes

```
data = date()
ora = time()
say "MY_SRV: Started at" data "," ora

arg port_no
if (port_no = '') then do
  say 'MY_SRV: Invalid port number'
  exit(8)
end
```

In the server code, you need to activate the SDSF host command environment to access REXX with SDSF. Because SDSF is an optional component of z/OS, the default host environment that is shipped with REXX does not including SDSF by default.

We invoke the **isfcalls(on)** function to establish the environment as shown in Example 10-2. We also call to the routine **setup_tcp** to set up the TCP/IP REXX interface.

Example 10-2 Invoking isfcalls(on)

```
rc = isfcalls('ON')                    /* Need to set up first */
if (rc <> 0) then do
  say mmm "Error activating the isfcalls() environment"
  select
    when (rc = 0) then say mmm "REXX SDSF HCE established"
    when (rc = 1) then say mmm "REXX SDSF HCE not established. RC=1"
    when (rc = 2) then say mmm "REXX SDSF HCE not established. RC=2"
    when (rc = 3) then say mmm "REXX SDSF HCE delete failed. RC=3"
    otherwise say mmm 'Unknown Return Code from isfcalls(on). RC=' rc
  end
  exit(8)
end

call setup_tcp
```

The subroutine **setup_tcp** initializes TCP/IP and calls the REXX socket functions to obtain a socket, bind it, and change some socket options.

In Example 10-3, the TCP/IP socket functions are shown in bold. We have described some of them briefly in Chapter 9, “JOB schedule and control” on page 201. The **gethostid()** and **gethostname()** methods are used just to printout the IP address and the host name of the server’s program machine. They do not have any interactions with the application’s logic.

Example 10-3 Routine to set up the TCP socket interface

```
/*-------------------------------------------------------------*/
/* Routine to setup the tcp socket interface                  */
/*-------------------------------------------------------------*/
setup_tcp: procedure expose my_socket port_no

s_rc= Socket('Initialize', 'MY_SRV')
if word(s_rc, 1) <> 0 then do
  say 'MY_SRV: Unable to initialize the TCP/IP socket interface'
  say 'MY_SRV: rc: ' word(s_rc, 1) word(s_rc,2)
  exit(8)
end

s_rc = Socket('GetHostId')
if word(s_rc, 1) = 0 then do
```

```
  my_ip_address = word(s_rc, 2)
  say 'MY_SRV: IP address of this host: ' my_ip_address
end

s_rc = Socket('Gethostname')
if  word(s_rc, 1) <> 0 then do
  say 'MY_SRV: Host name is: ' word(s_rc, 2)
end

s_rc = Socket('Socket')
if  word(s_rc, 1) <> 0   then call errore 'Socket()' s_rc
my_socket = word(s_rc, 2)

s_rc = Socket('Setsockopt', my_socket, 'SOL_SOCKET','SO_ASCII','ON')
if word(s_rc, 1 ) <> 0  then call errore 'Setsockopt()' s_rc

s_rc = Socket('Setsockopt', my_socket,
'SOL_SOCKET','SO_REUSEADDR','ON')
if word(s_rc, 1 ) <> 0 then call errore 'Setsockopt()' s_rc

s_rc = Socket('Bind', my_socket, 'AF_INET' port_no)
if word(s_rc, 1 ) <> 0 then call errore 'Bind()' s_rc

return
```

When this internal procedure terminates, the REXX socket interface has been initialized, and we are ready to accept requests coming from the client program. Example 10-4 shows the calls to the **listen()** and **accept()** functions. The **getpeername()** function returns the IP address of the client after a connection has been established. It does not have any interactions with the core logic of the application.

Example 10-4 Calls to the listen, accept and Getpeername functions

```
s_rc = Socket('Listen', my_socket, 10)
if  word(s_rc, 1) <> 0  then call errore 'Listen()' s_rc

s_rc = Socket('Accept', my_socket)
if  word(s_rc, 1) <> 0  then call errore 'Accept()' s_rc
peer_socket = word(s_rc, 2)

s_rc = Socket('Getpeername', peer_socket)
if  word(s_rc, 1) <> 0  then call errore 'Getpername()' s_rc
say 'MY_SRV: Incoming Connection from: ' word(s_rc, 4)
```

After the program has invoked the **accept()** function, it waits to be notified by TCP/IP of an incoming client request. When notified, it wakes up and issues the **accept()** function to accept the connection request coming from the client. Then, it prints out the IP address of the client, obtained through the **getpeername()** function.

When the request for a connection comes in, we accept it and spawn a separate task for every client. This task carries on the connection and the data traffic between the client and the server. This part of process has been realized using the REXX UNIX System Service syscall interface. Refer to Example 10-5 for further details.

Example 10-5 Calling the UNIX System Service spawn syscall

```
map.0=-1
map.1=1
map.2=2
parm.0=4
parm.1= '/u/dario/my_task'        /* name of the child program     */
parm.2= my_jobname                /* jobname of the parent task    */
parm.3= peer_socket               /* socket number we want to pass */
parm.4= '/'
Address syscall 'spawn /u/dario/my_task 3 map. parm. __environment.'
```

We start the program /u/dario/my_task that calls the UNIX System Service spawn **syscall**. This **syscall**, creates a separate process in which the program **my_task** runs.

In our sample, we used a stream socket connection. This type of connection provides a reliable, order preserving, flow controlled 2-way communication.

Because TCP/IP stream sockets send and receive information in streams of data, it can take more than one **read()** or **write()** function call to transfer all of the data. It is up to the client and the server to agree on some rules to signal that all of the data has been transferred. So, we adapted a simple protocol to exchange data between our two peers. The communication is always initiated by the client that is sending a request (or command) to the server and expecting, where needed, a response.

In detail:

1. The client sends a request (or command) to the server as a packet of 10 bytes.
2. The server, based on the request coming in, issues SDSF commands and sends back the result to the client

3. It sends first a 10 bytes packet that includes the length of the data to be sent back to the client.
4. After, it sends the real data.

Figure 10-3 illustrates this process.

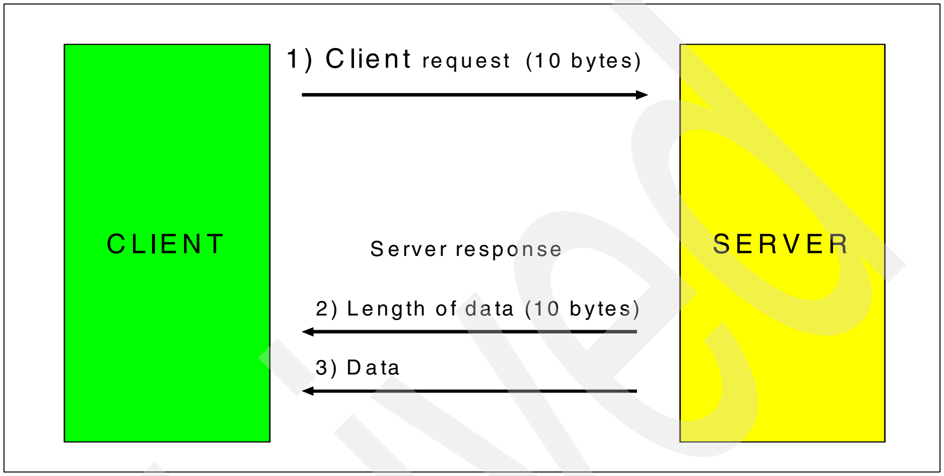


Figure 10-3 Sending and receiving TCP/IP data

10.2.2 Commands accepted by the server

The client sends the server a packet of 10 bytes that includes the request. The packet can include a simple command or a command in the first byte, followed by one parameter. The server, based on the command received issues in turn the appropriate SDSF or system command on the host system. Where needed, it also sets up some SDSF special variables, to limit the size of the response produced by the SDSF commands.

The server recognizes the following commands coming from the client:

`get_data` Informs the server to issue the ISFEXEC O or H command depending on the setting of the **`sdsf_queue`** variable.

D*parm* Informs the server to issue an SDSF DA command. It is used by the **`example2`** program to gather CPU consumption data from SDSF.

C*parm* Informs the server to issue a C command for a SYSOUT belonging to a specific JOBID. The ***parm*** field passed by the client contains the JOBID of the job selected. Note that the behavior of the C command, changes depending on the queue (`Held` or `Output`) selected.

`O`*`parm`* — Tells the server to set the name of the SYSOUT file owner. The input **`parm`** value, is moved into the `sdsf` special variable **`isfowner`**. It is used to filter the row produced by the subsequent ISFEXEC commands.

`P`*`parm`* — Passes to the server the value (**`parm`**) that has to be put into the SDSF special variable **`isfprefix.`** It it used to limit the number of rows produced by the ISFEXEC command.

`G`*`parm`* — Used to read the SYSOUT for a specific JOBID. The ***`parm`*** field include the value of the JOBID.

`Q`*`parm`* — Selects the `Output` or `Held` queue. The parameters recognized are O or H , to indicate `Output` or `Held`. Again, this is used as filter for the subsequent SDSF ISFEXEC call.

`end_end` — Sent to the server when the client program is terminating. It indicates that the client program is going to disconnect, and the server is free to accept other connection requests.

`sysplex` — Tells the server to issue a D XCF command to retrieve the name of the systems belonging to the sysplex

`S`*`parm`* — Informs the server to set the special variable **`isfsysname`**. This is used to identify the target system of subsequent SDSF commands.

`L`*`parm`* — Sent to the server when we want to cancel a running job. The parm field is the JOBID that identifies the job that has to be cancelled.

`K`*`parm`* — Sent to the server when we want to have information about a specific job that is running in the system . If the job is running when the command is issued, we have back some information about the resources used by the job. It it is not running, we simply get back an indication then notify this. The `parm` field identify the `jobid` of the job.

`M`*`parm`* — Sent to the server when we want to start a specific job. In order to make a job running, the server program changes the execution class from I to A. You can change these classes as you prefer. Also in this case, the `parm` field identifies the job ID of the job.

X*parm*
Sent to the server when we want to have the maximum return code of the job. The `parm` field contains the job ID of the job.

dm_chp
Sent to the server when we want to issue a D M=CHP system command.

J*parm*
Sent to the server when we want to issue a D M=DEV(parm) system command. The `parm` field includes the device address.

H*parm*
Sent to the server when we want to issue a D M=CHP(parm) system command. The `parm` field includes the `chpid` address.

10.2.3 REXX with SDSF function call

The ISFEXEC command is called from different places in the program, depending on which SDSF command we want to issue. Some special SDSF for REXX variables are also used in the program to limit and control the data that is received back from the command invocation.

Usually, SDSF formats only rows and columns that are visible.

In the REXX environment, the full complement of columns and rows are formatted and thus the max number of variables are created. For a large number of rows and columns, this could lead to a very high number of variables which can consume both CPU cycles and storage.

In detail we use ISFEXEC to obtain DA, O, and H data from SDSF. As an example, the DA data are used by the client example1 to gather CPU consumption information for all the active address spaces (`asid`) in the system. The O and H commands are used to get or purge SYSOUT data for the job in the Held or Output queue.

Before any ISFEXEC command is issued, the program assigns a value to some SDSF special variables in order to control and limit the data returned by the command. Some of these values are passed by the client while sending a request such as O, P, G, and Q. Other values are imposed by the program depending on the routine being executed. When the SDSF command completes, the values extracted from the SDSF special variables (that is **`isfcols`** or **`isfrows`**) are reduced and sent to the client in order to be graphically showed on the workstation screen.

Example 10-6, shows an extract of the code where the call to the procedure **`isfexec_call`** is done. You can refer to the source code of the sever program for further details.

In this example, we issue a command using the REXX for SDSF interface. We defined four special variables earlier to control the behavior of the command an its response:

`isfprefix` Names the jobname prefix to be used when filtering the row.

`isfowner` Names the owner to be used when filtering the rows.

`isfcols` Names the list of columns used for the display.

`isfsort` Names a sort field to be used when building the returned variables.

On return from the command, the SDSF special variables **isfrows** and **isfcols** are filled respectively with the number of rows returned and the column names and contents.

Example 10-6 Calling isfexec_call

```
call_sdsf_get_da_data: procedure expose peer_socket


isfprefix = "*"
isfowner  = "*"
isfcols = "jname cpupr"
isfsort = "cpupr d"

call isfexec_call "DA"
if (rc <> 0) then do
  say mmm 'Error invoking ISFEXEC. rc:' rc
end

colonne = words(isfcols)

resp = ''
do riga = 1 to isfrows
  do colonna = 1 to colonne
    nome_colonna = word(isfcols, colonna)
    contenuto = value(nome_colonna"."riga)
    if (nome_colonna = 'TOKEN') then iterate
    resp = resp || ' ' || ' ' || contenuto
  end
end
```

Example 10-7 shows that we issue an O or H command setting up special SDSF variables for them. We again set the same special variable used in Example 10-6, but in this case we use different values.

Note that the **isfprefix** and **isfowner** variable are set by the client sending the commands **Pparm** and **Oparm**. You might want to review the routine **set_owner** and **set_prefix** for further details.

Example 10-7 Issuing SDSF commands

```
call_sdsf_get_data:

isfcols = "jname jobid dsdate reccnt ownerid retcode"
isfsort = "dsdate a"

if (sdsf_queue = 0) then call isfexec_call "O"
if (sdsf_queue = 1) then call isfexec_call "H"
```

In Example 10-7, **sdsf_queue** passes either a 0 or a 1 to the routine named **isfexec_call**. This routine then issues the **isfexec** command and handles the return code, as shown in Example 10-8.

Example 10-8 Issuing the isfexec command

```
isfexec_call:

arg cmd
Address SDSF "ISFEXEC" cmd
if (rc <> 0) then do
  say 'MY_SRV: ERROR: Error calling ISFEXEC'
  say 'MY_SRV: ERROR: command:' cmd
  say 'MY_SRV: ERROR: return code:' rc
  exit(-1)
end

return
```

Another routine that might be of interest is the **call_sdsf_read_SYSOUT**. It is called when we want to read the SYSOUT for a defined job ID. The `jobid` parameter passed as input parameter to the routine identifies the job for which we want to read the SYSOUT.

The client program sends the ***Gparm*** request to the server when it wants to read system output. Before the client sends a ***Gparm*** request, it also sends a ***Qparm*** request to select which queue the user wants to scan: `Held` or `Output`. After we issued the desired ISFEXEC (O or H) command to SDSF, we scan the answer returned to find the entry for our JOBID. When we find our job, we read the entire SYSOUT data set for this job through EXECIO.

The data read is packed into a variable and sent to the client. Note that we mark the end of every record read, with the sequence of characters E_O_R. This is done to enable the client to find the end of a line to print the data correctly.

Example 10-9 uses the ISFACT command to issue a SDSF action. We can see in this extract of code, how the server use ISFACT to allocate all the SYSOUT data sets for the job represented by TOKEN.

Remember that for a tabular display (O or H in this case) an additional column variable named TOKEN contains a string that uniquely identifies the row and is used as an input on the ISFACT command if the row is to be modified. The token contains special characters, must not be modified by the user and must be passed as is to ISFACT.

Example 10-9 Using the ISFACT command

```
call_sdsf_read_sysout:

arg jobid

if (sdsf_queue = 0) then address SDSF "ISFEXEC O"
if (sdsf_queue = 1) then address SDSF "ISFEXEC H"
if (rc <> 0) then do
  say mmm 'Error invoking ISFEXEC. RC:' rc
  exit(8)
end

sysout_buf = ""

do a = 1 to JNAME.0
  if JOBID.a = jobid then do
    if (sdsf_queue = 0) then
      Address SDSF "ISFACT O TOKEN('"TOKEN.a"') PARM(NP SA)"
    if (sdsf_queue = 1) then
      Address SDSF "ISFACT H TOKEN('"TOKEN.a"') PARM(NP SA)"

    /* we read in the sysout data set     */
    /* Need to mark the end of every rows */
    /*  We use the char sequence E_O_R    */

    do b=1 to isfddname.0
       "EXECIO * DISKR" isfddname.b "(STEM s_tab. FINIS"
       if (rc <> 0) then do
```

```
        say 'MY_SRV: Error reading spool data'
        exit(-1)
      end

      do c=1 to s_tab.0
        sysout_buf = sysout_buf || s_tab.c || "E_O_R"
      end

    end /* do b */
    leave
  end  /* if jobid */
end
```

The routine, **call_sdsf_cancel_sysout** (as shown in Example 10-10) is called to cancel the SYSOUT for a specific JOBID.

The `jobid` is passed as an input parameter to the routine. The request to purge a SYSOUT comes from the client as a ***Cparm*** command, where ***parm*** is the job ID selected. We issue an SDSF O or H command to retrieve the list of the job in the Output or Held queue. Note that as reported previously some special variables are set to limit and control the output of the command.

When we find a job with the corresponding `jobid` assigned, we pick up its token and call ISFACT to cancel its SYSOUT as follows:

```
Address SDSF "ISFACT O TOKEN('"TOKEN.a"') PARM(NP C)
```

The special variable NP is used to represent the column for action character. In our case the action character is C (see the parm field).

This command notifies SDSF to issue an O command. Then, using the token that identifies our job (the row in the O command output) we ask SDSF to issue the C action command. This result is the cancel of the SYSOUT for the defined job.

Example 10-10 Canceling SYSOUT

```
call_sdsf_cancel_sysout:

arg jobid

if (sdsf_queue = 0) then address SDSF "ISFEXEC O"
if (sdsf_queue = 1) then address SDSF "ISFEXEC H"
if (rc <> 0) then do
  say mmm 'Error invoking ISFEXEC. RC:' rc
  exit(8)
end
```

```
sysout_buf = ""

do a = 1 to JNAME.0
  if JOBID.a = jobid then do
    if (sdsf_queue = 0) then
      Address SDSF "ISFACT O TOKEN('"TOKEN.a"') PARM(NP C)"
    if (sdsf_queue = 1) then
      Address SDSF "ISFACT H TOKEN('"TOKEN.a"') PARM(NP C)"
    return
  end
end

return
```

10.2.4 Running the server program

You can find the REXX code of the server program together with the client Java programs in the .zip files for this book. For information about how to obtain these files, see Appendix B, “Additional material” on page 305.

You need to move the code into a UNIX System Service file system directory. In our example, the code is loaded into the directory /u/dario.You can use your preferred method to do this.

The server program can run both under a TSO UNIX System Service session or as a batch job. You need to specify the port number to use it. We use, as an example, the JCL shown in Example 10-11 to start it from a batch job. Note the port number value 20030 passed as a parameter.

Example 10-11 Starting a TSO UNIX System Service session in batch

```
//DARIOS   JOB CLASS=A,NOTIFY=&SYSUID
/*JOBPARM S=SC70
//STEP1    EXEC PGM=BPXBATCH,PARM='sh /u/dario/my_srv 20030'
//SYSPRINT DD  SYSOUT=*
//STDOUT   DD  PATH='/u/dario/stdout'
//STDERR   DD  PATH='/u/dario/stderr'
//STDENV   DD  *
```

You need to create the two files, stdout and stderr, using the shell command **touch** (for example, **touch stdout**). When started, the server program issues the messages shown in Example 10-12 and then waits for a connection request coming from a client.

Example 10-12 Example touch stdout

```
MY_SRV ==> Started at 5 Apr 2007 , 10:37:53
MY_SRV ==> Listening on port: 20030
MY_SRV ==> IP address of this host:  9.12.4.202
MY_SRV ==> Host name is:  WTSC70
```

When a client connects to the server as shown in Example 10-13. The IP address of the client being connected is reported in the message.

Example 10-13 Client connecting to a server

```
MY_SRV ==> Incoming Connection from: 9.57.138.152
```

When the client disconnects, we can see the message displayed in Example 10-14. Again, the IP address of the client being disconnected is reported in the message text.

Example 10-14 Client disconnected

```
MY_SRV ==> Disconnect received from: 9.57.138.152
```

10.2.5 Configuration of the server program

In the server code, there are some variables initialized with hard coded values. Example 10-15 has been extracted from the **my_task** REXX program. You need to update the **isfsysname** variable content, in order to reflect your default system name. In the example, the system name of our default system is SC70.

Example 10-15 Server code with hard coded isfsysname

```
peer_socket = 0                  /* init some variables */
 peer_name = ''
 isfprefix = "*"
 isfowner  = "*"
 isfsysname = "SC70"
 sdsf_queue = 0
```

You need to update some lines of code in the **my_srv** REXX program in order to reflect the UNIX System Service directory name where the code resides. In Example 10-16, we loaded the code into /u/dario.

Example 10-16 UNIX System Service directory with server code

```
-rwx------   1 DARIO    SYS1        6192 Apr 26 10:50 my_srv
-rwx------   1 DARIO    SYS1       20764 May  2 04:04 my_task
```

You need to update the code, specifying the directory where you loaded the server code. Example 10-17 has highlighted in bold, the places where you need to modify the code.

Example 10-17 Updating parameters with your directory structure

```
map.0=-1
map.1=1
map.2=2
parm.0=4
parm.1= '/u/dario/my_task'        /* name of the child program     */
parm.2= my_jobname                /* jobname of the parent task    */
parm.3= peer_socket               /* socket number we want to pass */
parm.4= '/'
Address syscall 'spawn /u/dario/my_task 3 map. parm. __environment.'
if (errno <> 0) then do
    say mmm 'Error in the spawn() syscall'
    say mmm 'retval : ' retval
    say mmm 'errno  : ' errno
    say mmm 'errnojr: ' errnojr
end
say mmm 'Child spawned. PID: ' retval
```

10.3 First client program

The first client program, renders the CPU consumption data of the ASIDs that are running currently in the system graphically. The program is written in Java and uses the JFreeChart library to display graphics.

The server program gathers the data by issuing an ISFEXEC DA command and sends the data to the client to be plotted in graphical format. The program is just a sample, written to show a possible use of the new REXX with SDSF interface.

We describe what the client program does and how it interacts with the server in 10.2, “Description of the server program” on page 228.

Example 10-18, shows an extract of the code taken from the Java class `thread_1`.

Example 10-18 Java client code

```
public void run()
  {
    int   delay_value, i;
    host = new Host();
    Grafico_1 graf_1 = new Grafico_1();
    String job_selected = "";
    String sysname_selected = "";
    String s_w;

    JOptionPane.showMessageDialog(null,
            "\n Connecting to the Server Program. " +
            "\n This may take a few seconds. Please Wait..." +
            "\n\n Press OK to continue\n\n");

    get_server_info();
    host.connect(host_addr, host_port);

    host.cmd("sysplex");
    st = host.get_data_from_host();
    graf_1.set_sysnames(st);

    for(;;)
    {
      sysname_selected = graf_1.get_sysname_selected();
      s_w = "S";
      s_w = s_w + sysname_selected;
      boolean b = s_w.equals("S");
      if (! b)
      {
        host.cmd(s_w); // send a Sxxx command to select a system
      }

      host.cmd("D");
      st = host.get_data_from_host();
      job_selected = graf_1.draw(st, job_selected);
      graf_1.setVisible(true);
      delay_value = graf_1.get_delay_value();
      delay_value = delay_value * 500;
      try {
        Thread.sleep(delay_value);
```

```
        } catch (InterruptedException e) {
          System.out.println("Thread: Error in thread.");
          System.out.println("e:" + e.getMessage());
        }
    }

 }
```

The client program connects to the server through a call to the **connect()** method in the class `host`. This method builds a socket and initializes a TCP/IP stream connection with the server. The IP address of the server and the related port number can be specified in a configuration panel.

The first command issued by the server **host.cmd("sysplex")** asks the server to return a list of the systems belonging to the sysplex. A list box is then filled in with this information. In this way, the user can choose to retrieve DA information for a specific system.

Then, it calls the method **cmd()** in class `host` to issue a D request to the server. Before this, it calls **host.cmd(s_w)** to select the system chosen by the user. It retrieves the output data gathered by the server calling the **get_data_from_host()** method in the class `host`. The data is plotted by calling **graf_1.draw()**.

The program waits for a specified amount of time and reissues the D command again until the program terminates. The wait time can be changed dynamically while the program is executing.

Example 10-4 shows a high-level overview of the program logic. On the left is the server program, and on the right is the client program.

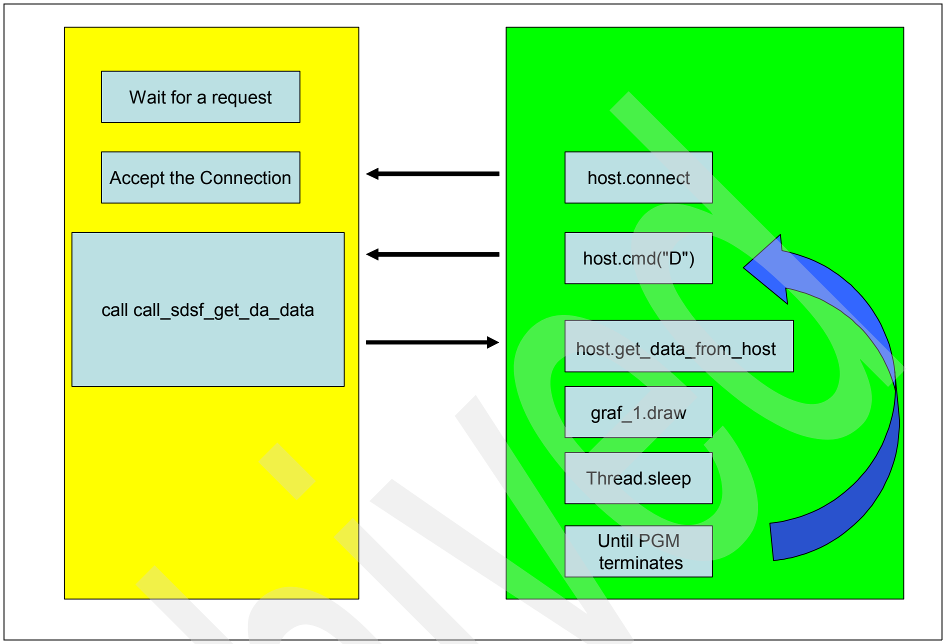


Figure 10-4 Program logic

10.3.1 Use of the program

To use the program, follow these steps:

1. Start the program by typing `Java example1` from the Command Prompt window of a workstation. When started, the program shows a window like that shown in Figure 10-5.

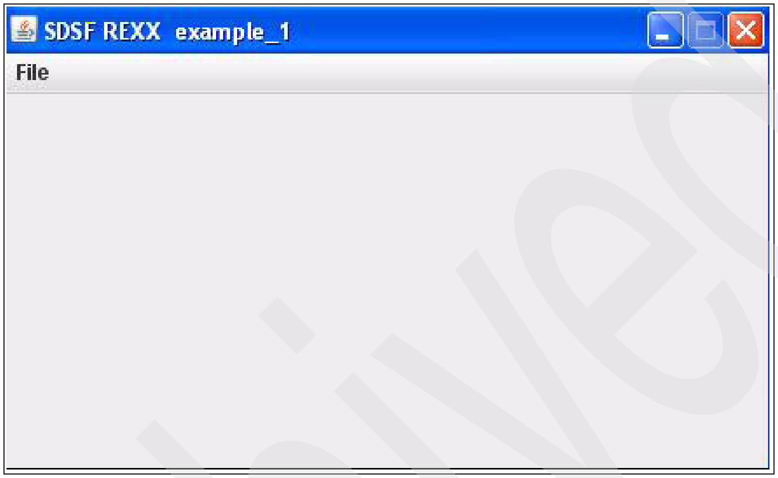


Figure 10-5 Running run.bat

2. Click **File** → **Configure** to enter the configuration windows (Figure 10-6). Enter the IP address of the host where the server program resides and the related port number.

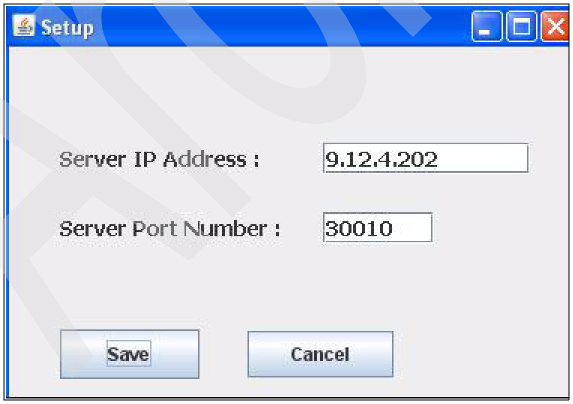


Figure 10-6 Setup window

3. Close the configuration window and click **File → Start**. After a few seconds, the window shown in Figure 10-7 opens.

Figure 10-7 Connecting message

4. The client program is connecting to the server. Press **OK** to continue.

   The server, accepts the connection from the client and retrieves the list of the active systems into the sysplex. This step can take a few seconds. When this process is complete, a window opens that looks similar to that shown in Figure 10-8.

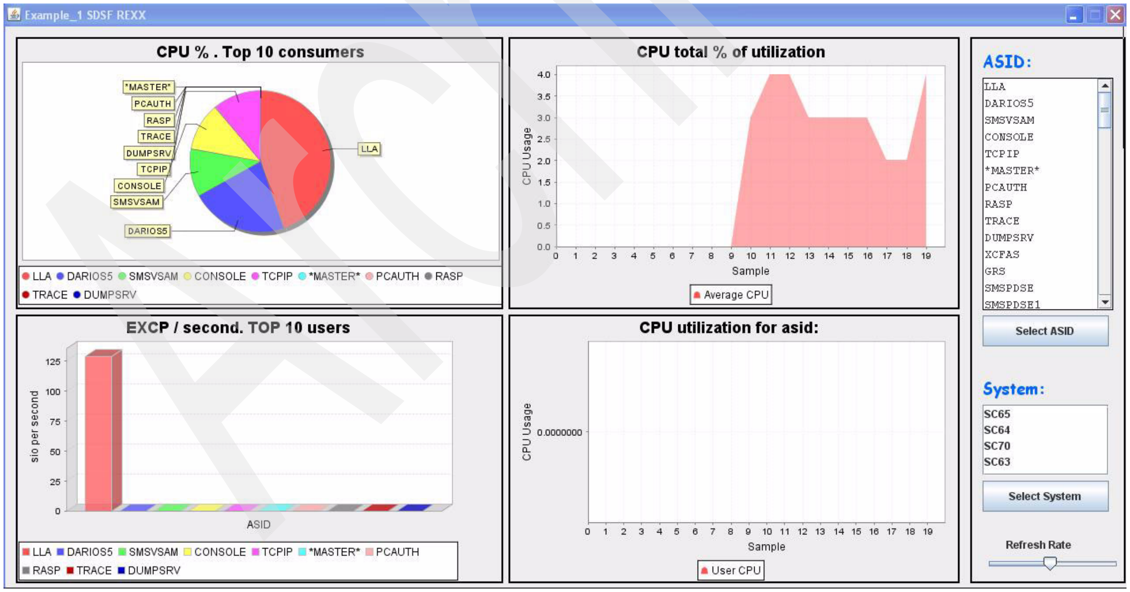


Figure 10-8 Dashboard, first glance

Figure 10-8 shows the following information:

- The report on the upper-right, shows the total CPU consumption of all the active Address Spaces (ASID) in the system. This value is calculated by adding together the `cpupr` values for all the active ASID as returned by the DA command.
- The pie chart on the left, reports the `cpupr` value for the 10 highest CPU consumers in the last interval. Before the server issues the DA command, it sets the following SDSF special value:
  - **`isfcols`** = “jname cpupr”
  - **`isfsort`** = “cpupr d”

  So, when the command is executed, we receive as output a job name and the CPU absorbed, sorted on the `cpupr` value in descending order. We pick up the `cpupr` of the first 10 jobs reported into the list and create the pie chart.
- The graph in the lower-right area reports the CPU consumption for a selected ASID. You can select a specific ASID from the list box and by pressing the Select ASID button below it. The list box is repopulate every interval and is sorted on the `cpupr` value. The asids listed are reported on a `cpupr` consumption order.
- The graph on the lower-left area reports the `excprt` value for the 10 most active ASIDs in term of EXCP per seconds issued in the interval. They are sorted on the `excprt` value.
- The slider on the bottom-right of the window is used to change the wait time interval between the DA commands. As shown in Example 10-18, the client program issues the DA command, plots the data received back and then waits for a specific amount of time.
- The lower list box can be used to select the system that we want to interrogate. To determine the system, you can select it from the list box and press **Select system**.

For both the `sysname` and `asid` selection, the program takes one interval to make the data available. When you select the button, you need to wait for the next cycle of the program to have the data updated and refreshed.

10.4 Second client program

The second Java client program, shows how to list and read SYSOUT data. It uses the same server program. To obtain the SYSOUT data, the client program sends a Q command to the server to select the `Held` or `Output` queue, then issues a **`get_data`** command to retrieve the list of job in `Held` or `Output` queue.

You can select a job from the list box presented by the program and decide to retrieve or purge the SYSOUT of the job selected.

At start up, the program shows the window displayed in Figure 10-9.

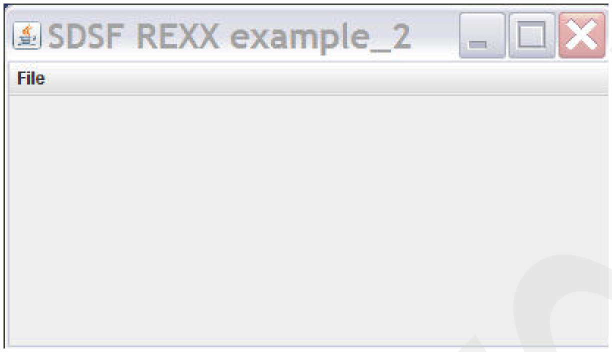


Figure 10-9 Startup panel for example 2 Java program

To use the program, follow these steps:

1. Click **File** → **Start** and you get the panel shown in Figure 10-10.

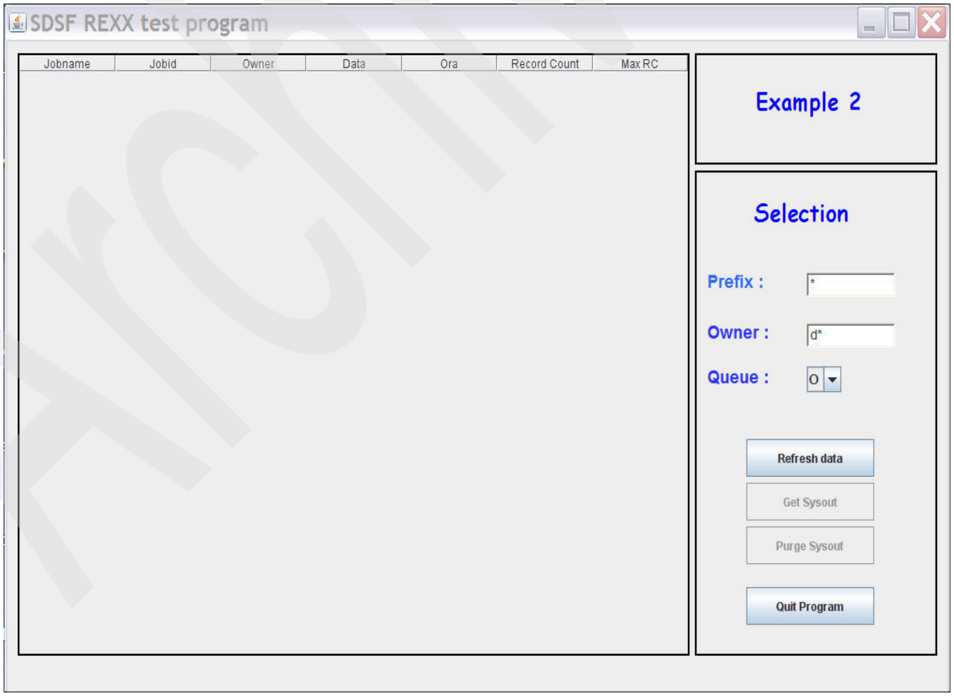


Figure 10-10 Example 2 program start

This window allows you to see the status of the jobs in the `Output` (O) or `Held` (H) queue to retrieve the SYSOUT of a job and to purge a SYSOUT.

2. To select the queue, click **QUEUE** and select **O** or **H**.

   Complete the Prefix and Owner fields to represent the current selection. It is recommended to uses these filters to limit the bandwidth used by the application.

3. When done, click **Refresh data**. Based on the selection made, the panel updates as shown in Figure 10-11.

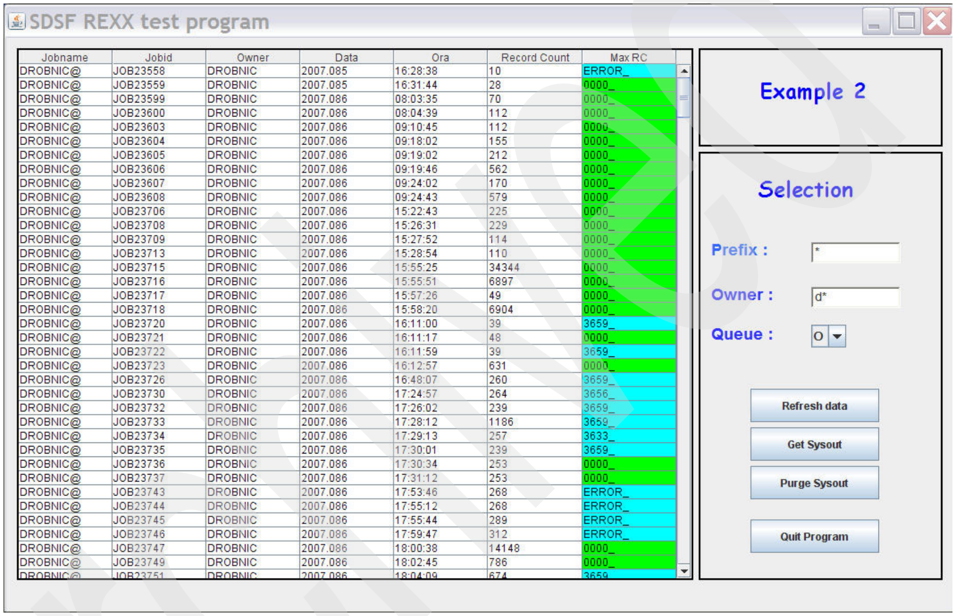


Figure 10-11 Status of jobs in the output queue

In this example, we asked for the list of job in the `Output` queue with a owner beginning with `D*`. The window includes some information about the job, including the `maxrc` value. The cell of the table that include this value can have different colors, depending on the `maxrc` value:

GREEN	The maximum return code (`maxrc`) of the job is equal 0
YELLOW	The `maxrc` of the job is equal 4
RED	The `maxrc` of the job is equal 8
CYAN	The `maxrc` of the job is none of the above

You can now select a job from the list and have the SYSOUT exported to a pc window. Select a row and click Get Sysout. You should receive a window similar to the one shown in Figure 10-12.

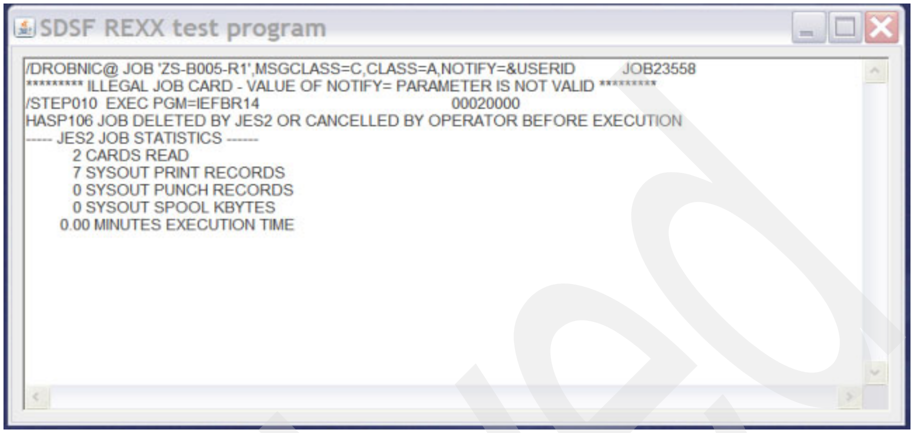


Figure 10-12 Getting the system output

Because this is a sample program, we tested and verified that it works when the number of rows returned in not high. We recommend that you set up the Owner and Prefix input fields to limit the output produced and to export to a PC SYSOUTs with no more then 200 to 300 output lines to avoid performance problems. You can also purge the SYSOUT for a specific job by selecting the job and selecting the Purge Sysout button.

10.5 Third client program

The idea behind this example is to show how to reduce the data provided by some system commands and show their output on a workstation in graphic mode. There are some system commands whose output spans more than one page. You need to scroll the output pages to see all the output.

As an example, we implemented our sample application around the D M system command. We issue the command through the SDSF REXX interface, reduce the output, and display the results using some graphics. We implemented the program, as in the other cases, using a client server approach. It uses the same server as the other two samples.

Figure 10-13 shows the window that opens when the program starts.

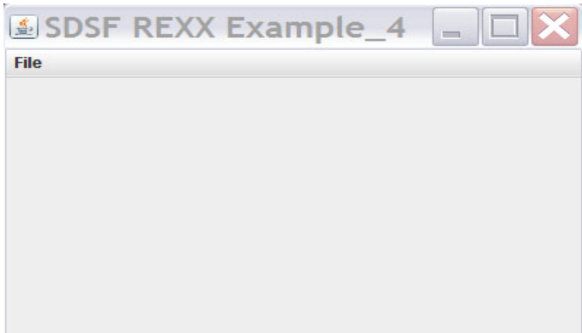


Figure 10-13 Startup window for example 4

To run the program, follow these steps:

1. If you click **File** → **Configure**, you receive the configuration window where you can select the IP address of the machine where the server program reside and its port number. Refer to the description of the previous sample for further information.
2. Click **File** → **Start** to start the program. You receive the window shown in Figure 10-14.

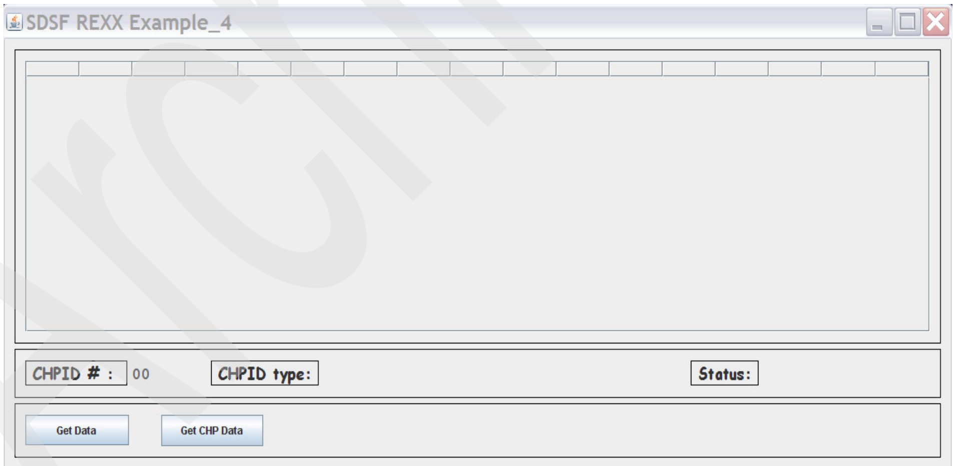


Figure 10-14 Starting the program

3. Press the **GetData** button to proceed. When you press this button, the program sends to the server a request to issue a D M=CHP command to collect information about all the `chpid` connected to the partition in which z/OS is running. Example 10-19 is an extract of the code executed by the server.

Example 10-19 Server code extract - example 4

```
issue_dm_chp_cmd:

isfdelay = 5
address SDSF "ISFEXEC '/D M=CHP' (WAIT"
isf_rc = rc
call check_isf_rc isf_rc

dm_buf = ''
sw = 0
do a = 1 to isfulog.0
  if word(isfulog.a, 1) = 'CHANNEL' then do
    sw = 1
    iterate
  end
  s = word(isfulog.a, 1)
 l = length(s) l = length(s)
  if (l <> 1) then iterate        /* discard some output lines.. */
  if (s = '*') then iterate
  if (s = '+') then iterate
  riga = isfulog.a
  if (a = 5) | (a = 26) then do
    riga = '* ' || riga
  end
  dm_buf = dm_buf || riga || ' '
end

if (sw = 0) then dm_buf = "no_data"
call send_response_to_peer dm_buf
```

As we can see, it invokes the D M=CHP system command through the SDSF REXX slash command. Before the command is send, it sets the **`isfdelay`** special variable to `5`. This causes a five second wait for the command response. When the command is issued, we pull out the output of the command from the **`isfulog.`** special variable (it is a REXX stem), extract some output lines and send the data to the client program.

When this process completes, we can see window shown in Figure 10-15 on the workstation.

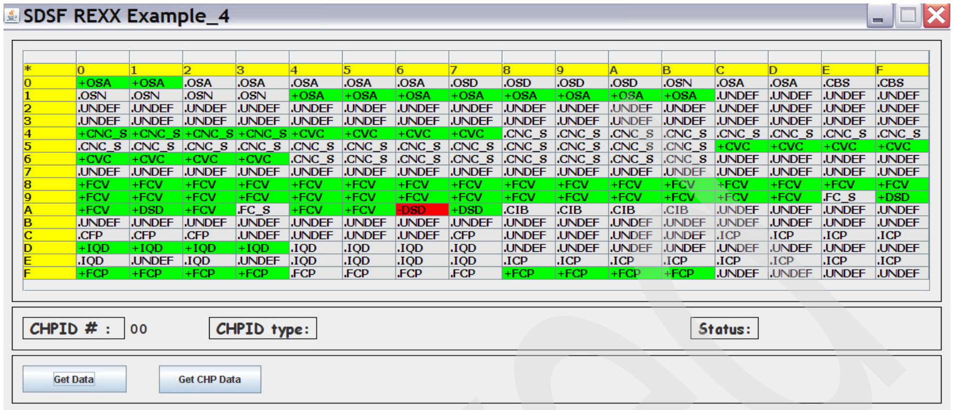
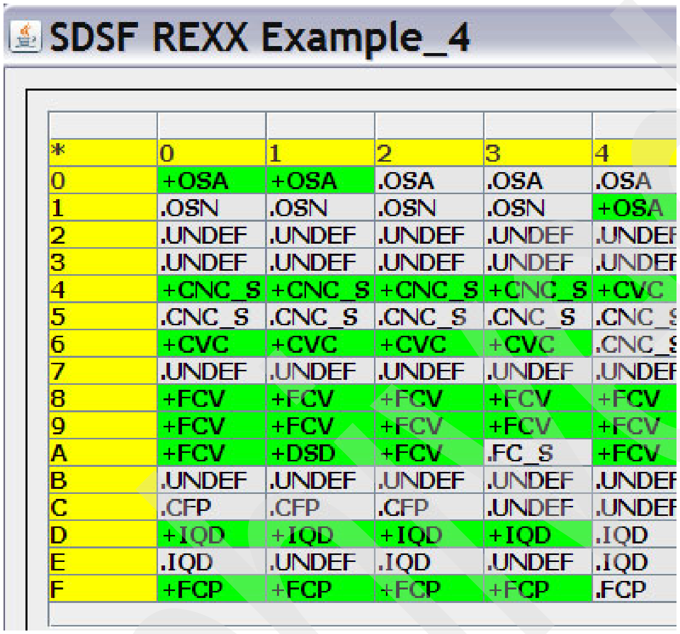


Figure 10-15 Data rendered to client workstation - example 4

If you try a D M=CHP system command from a console or through SDSF, you see that the output spans four pages. Moreover, you might have to scroll left or right on the screen to see all the information in it. Reducing the data in the way we did, we might have all the information condensed onto one single page. In addition we use color to identify the status of the chpid.

The Figure 10-16, shows a portion of the window. Every cell of the table, contains the information belonging to a specific chpid. As an example, the chpid with address 01 is an OSA system adapter card. The plus sign (**+**) indicates that this chpid is online to our system. We painted the cells of the table containing online chpid in green. We painted in red, the cells that describe offline chpid.



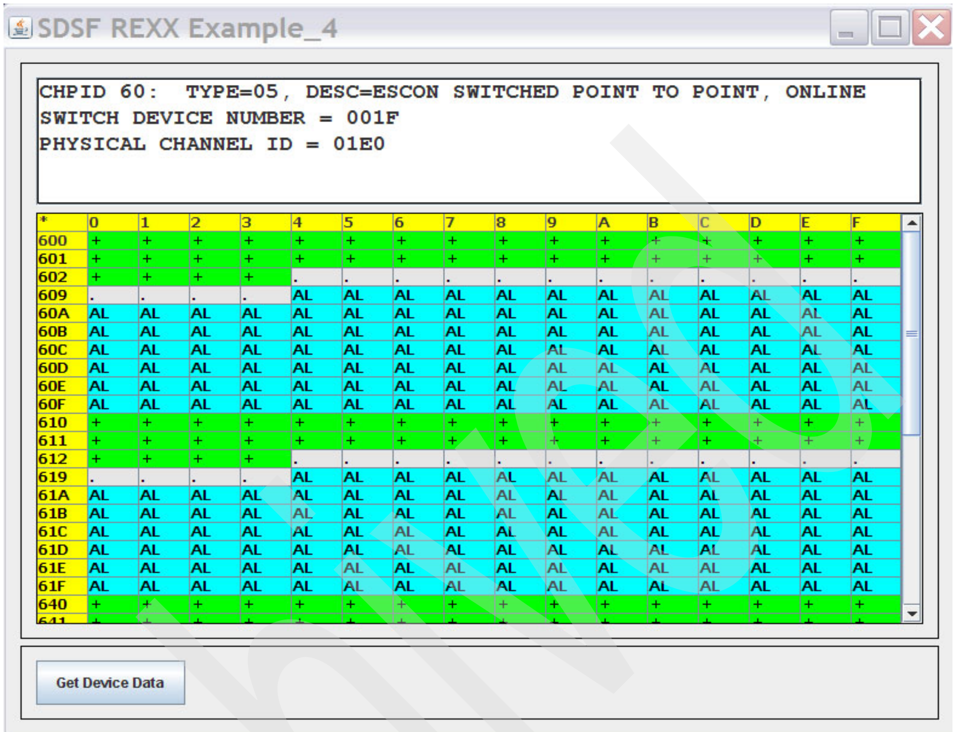
*	0	1	2	3	4
0	+OSA	+OSA	.OSA	.OSA	.OSA
1	.OSN	.OSN	.OSN	.OSN	+OSA
2	.UNDEF	.UNDEF	.UNDEF	.UNDEF	.UNDEF
3	.UNDEF	.UNDEF	.UNDEF	.UNDEF	.UNDEF
4	+CNC_S	+CNC_S	+CNC_S	+CNC_S	+CVC
5	.CNC_S	.CNC_S	.CNC_S	.CNC_S	.CNC_
6	+CVC	+CVC	+CVC	+CVC	.CNC_
7	.UNDEF	.UNDEF	.UNDEF	.UNDEF	.UNDEF
8	+FCV	+FCV	+FCV	+FCV	+FCV
9	+FCV	+FCV	+FCV	+FCV	+FCV
A	+FCV	+DSD	+FCV	.FC_S	+FCV
B	.UNDEF	.UNDEF	.UNDEF	.UNDEF	.UNDEF
C	.CFP	.CFP	.CFP	.UNDEF	.UNDEF
D	+IQD	+IQD	+IQD	+IQD	.IQD
E	.IQD	.UNDEF	.IQD	.UNDEF	.IQD
F	+FCP	+FCP	+FCP	+FCP	.FCP

Figure 10-16 Detail of chpid data

Assume, as an example, that we want to see more information for the chpid with address 60. As shown in Figure 10-15 on page 254, it is a CVC chpid and its status in online.

So, click on the cell of the table that describes this chpid and press **Get CHP Data**. When you click this button, the client sends to the server a request to issue a D M=CHP(60) command. You can refer to the routine with name **`issue_dm_chp_specific_cmd`** in the **`my_task`** REXX program for further details. We do not report the code here because the logic is very similar to the one described in the Example 10-19 on page 253.

You should receive the window shown in Figure 10-17.



*	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
600	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
601	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
602	+	+	+	+	.	.	.	.	.	.	.	.	.	.	.	.
609	.	.	.	.	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL
60A	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL
60B	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL
60C	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL
60D	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL
60E	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL
60F	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL
610	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
611	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
612	+	+	+	+	.	.	.	.	.	.	.	.	.	.	.	.
619	.	.	.	.	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL
61A	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL
61B	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL
61C	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL
61D	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL
61E	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL
61F	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL	AL
640	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
641	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+

Figure 10-17 Status of all devices - example 4

This window reports the status of all the devices connected to the chpid that we selected in the previous step (chpid 60). Every cell into the table represent a device. We paint the cells in different colors depending on the status of the device described by the cell. We just copy into the cell the status of the device as reported back by the D M command output. You can refer to the *z/OS System Commands and z/OS Messages and Codes* manuals for a complete description of the keyword used to describe the device and chpid status.

We want now to retrieve all the information for the device with address `6001`. We click the device cell and press **Get Device Data**. After a while we should receive the window shown in Figure 10-18.

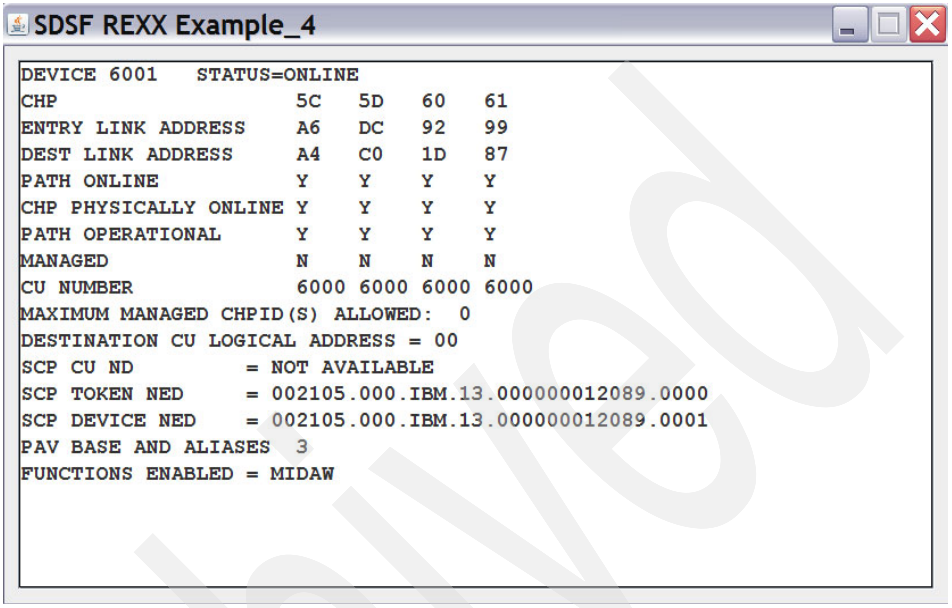


Figure 10-18 Getting device data - example 4

When we press **Get Device Data**, the client send to the server a command **J6001** (in this case). **J** represent the command type and 6001 is the parameter for this command. The server code, in turn issue a system command D M=DEV(6001) using the SDSD slash command and send back to the client the response.

10.6 Extending the examples

The three simple examples described in this chapter, can be further modified and improved. Our goal is to provide you with ideas about what it is possible to do with REXX for SDSF. We used in the server code the ISFEXEC, ISFACT and 'slash' commands.

The client applications are written in JAVA and can be run on different platforms. For our sample programs, we decided to implement a client/server application to use the graphics capability of a workstation.

We decided to plot the CPU% consumption as reported by the SDSF DA command. Using the same mechanism, you can collect, plot, and mix many other data that can be obtained from the SDSF DA command.

As an example, you might decide to plot the following DA fields:

- Real To track the current utilization of real storage in frames
- SIO To track the address space's EXCP rate in EXCPs per second
- CPU-Time To track the accumulated CPU time (TCB plus SRB) consumed on behalf of the address space, for the current job

You can also decide not to plot graphically the data, but use the raw data from the server. In this case, any application able to use TCP/IP services can act as a client program. Furthermore, a palm handled computer or a mobile phone can also be used for this scope writing a J2ME™ (Java2 Mobile Edition) client program.

Adding additional logic to the client program, you can also monitor and control from a remote location the status and vitality of a system. You might decide, as an example, to take some actions based on the health of some system values returned by the server program.

You can expand the third sample, as an example, to issue some system VARY command to put online/offline devices and chpid.

The idea to reduce the system command output and send the result to a graphic workstation, can be applied to other system commands. We can have a graph representation, as an example, of the status of system data sets (such as, DUMP, PAGE, LNKLST, and so forth).

10.7 How to compile the Java programs

In this section we describe how to install and compile the sample applications.

First, you need to download from the ITSO Java site, the compressed file that includes the source code of the programs. The compressed file includes both the server and the client code. Decompress the file, and you have four directories created with the names example_1, example_2, example_3, and ch9and10. In these directories, you find the source code of the client Java programs and the server REXX code that has to be uploaded to the host system.

Upload the REXX code, using your preferred method (FTP, PC3270 file transfer, and so forth) into a UNIX System Service file system. When loaded, you can use the JCL shown in Example 10-11 on page 240 to start the server side of the application.You might need to verify the networking definition and the TCP/IP port availability because the code uses TCP/IP sockets.

The Java sample programs have been compiled using JDK1.6.0. In our case, we compiled them on Windows XP. The Example 10-20 shows the **java -version** command that you can use to verify the level of the Java available on the workstation.

Example 10-20 Checking your Java version

```
C:\SG24-7419\addmat\java_code\example_1>java -version
java version "1.6.0"
Java(TM) SE Runtime Environment (build 1.6.0-b105)
Java HotSpot(TM) Client VM (build 1.6.0-b105, mixed mode)
```

You also need to update your CLASSPATH variable to include two jar files supplied by JFreeChart, jfreechart-1.0.4.jar and jcommon-1.0.8.jar, that are needed for the compilation of the programs. In our case the JFreeChart top level directory in located in C:\$user\java.sources\jfreechart-1.0.4.

We use a .bat file to add the two jar files to the existing Java CLASSPATH contents, as shown in Example 10-21.

Example 10-21 Example classpath statement

```
set
CLASSPATH=%CLASSPATH%;C:\$user\java.sources\jfreechart-1.0.4\lib\jfreec
hart-1.0.4.jar

set
CLASSPATH=%CLASSPATH%;C:\$user\java.sources\jfreechart-1.0.4\lib\jcommo
n-1.0.8.jar
```

We are now ready to compile the programs using these commands from their respective directories: **javac Example_1.java**, **javac Example_2.java**, and **javac Example_3.java**. You can ignore the warning message that you receive.

At this point you should have the Java workstation programs compiled and the REXX code loaded into the host system. Start first the server program submitting the provided JCL. After this, start the workstation part of the application invoking **java Example_1**, **java Example_2**, or **java Example_3** from a command prompt window.

11

Extended uses

In this chapter we describe some uses of the REXX with SDSF interface that we were unable to explore fully during the residency for this book but might be considered of interest to you and your installation.

11.1 A different desktop for each role

Only a few users would ever use all of the features provided by SDSF. The interface of the product can become too cumbersome for someone who only has to accomplish a small set of tasks related to the system.

For instance, in their day-to-day work, operators might only manage output queues and devices such as readers, printers, lines, or punches and would not be interested in system tasks or initiators, and even in those former panels, not all the columns shown by the tabular displays might be meaningful for their specific needs. Alternatively, application programmers might be interested only in viewing the jobs they submit and not in the nodes of the system, the punches or the WLM enclaves.

To help you, SDSF only shows on the primary menu the options to which you are authorized, but even with this feature, programmers might want to see abended jobs in a different color or might need an additional column telling them to which application their jobs belong or which subsystems they used as shown in Figure 11-1.[1]

```
 File   Compilers   Utilities  SDSF   Help
-------------------------------------------------------------------------------
Command ==>                                                      Scroll ==> CSR
Cmd  Member    Lang  Max-RC      Library                           Application
-------------------------------------------------------------------------------
___  CBLPGM1   CBL   CC 0000     ITSO.DEVELOP.CBL                  PAYROLL
___  CBLPGM2   CBL   CC 0012     ITSO.PAYROLL.CBL                  ACCOUNTING
___  ASMPGM1   ASM   CC 0004     ITSO.DEVELOP.ASM                  ACCOUNTING
___  PAYROLL1  JCL   ABEND SOC1  ITSO.STRESS.TEST.JCL              PAYROLL
___  TESTCBL6  JCL   CC 0000     ITSO.PAYROLL.CBL                  PAYROLL
___  CBLPGM9   CBL   CC 0000     ITSO.PAYROLL.CBL                  PAYROLL
___  PLIPGM1   PLI   CC 0000     ITSO.PAYROLL.PLI                  PAYOLL
___  PLIPGM1   PLI   CC 0012     ITSO.PAYROLL.PLI                  REDBOOKS
___  PAYROLLX  JCL   CC 0000     ITSO.LEVEL1.TEST.JCL              PAYROLL
___  EMPLOYEE  JCL   CC 0000     ITSO.LEVEL1.TEST.JCL              PAYROLL
___  STRESJCL  JCL   ABEND SOC4  ITSO.STRESS.TEST.JCL              REDBOOKS
___  PAYROLLW  JCL   CC 0000     ITSO.LEVEL1.TEST.JCL              REDBOOKS
___  ASMMOD7   ASM   CC 0000     ITSO.SDSF9.ASM                    PAYROLL
___  CBLPGM8   CBL   CC 0000     ITSO.PAYROLL.CBL                  ACCOUNTING
___  CBLPGM9   CBL   CC 0000     ITSO.PAYROLL.CBL                  REDBOOKS
___  CBLPGMA   CBL   CC 0000     ITSO.DEVELOP.CBL                  REDBOOKS
     ***End***
```

Figure 11-1 Sample programmers desktop

Operators might want to see a list in one screen of the last issued commands, a quick way to retype them and a journal of the ulog (user log) from session to session. You can refer to Chapter 1, “Issuing a system command” on page 37 for

[1] Changing the color of rows in an ISPF table display dynamically requires using dynamic areas to show the actual screen.

more information about how to do something like this. Figure 11-2 shows an example of this.

```
 Menu  Ulog   List  Mode  Functions  Utilities  Help
------------------------------------------------------------------------------
                            Operators Command Shell
Enter JES2 or system commands below:

===> _________________________________________________________________________
______________________________________________________________________________
______________________________________________________________________________

Place cursor on choice and press enter to Retrieve command

=> /PJ01589
=> /D ASM,PLPA
=> /CMDS REMOVE,CMD=VARY,JOB=JOB1111
=> /DUMP PARMLIB=NJ,SYMDEF=(&PAGING1.='AQFT',&CICS.='CICS1')
=> /M 282,VOL=(SL,222222),USE=PRIVATE
=>
=>
=>
=>
=>
```

Figure 11-2 Sample operators command entry facility

With the REXX with SDSF, all these tasks can be accomplished easily. You can build a task-oriented desktop for every role in your organization that needs the services SDSF offers. These desktops can have the look you want and the functionality you want.

11.2 Control your subsytems

You might want to periodically monitor your subsystems: DB2®, IMS, or CICS®, verify that they are up and running and analyze their logs, looking for errors that could affect system performance.

If you know the name of the started tasks of your subsystems, with the help of the REXX interface for SDSF, you can implement a batch control of the subsystems status: verifying CPU time used during the last interval, looking for system messages in JESMSGLG and JESYSMSG, or reading each subsystem log searching symptoms of an abnormal behavior. In Chapter 7, “Reviewing execution of a job” on page 173, there is a sample of how review these files.

Figure 11-3 depicts how SDSF support for REXX fits into the larger system picture.

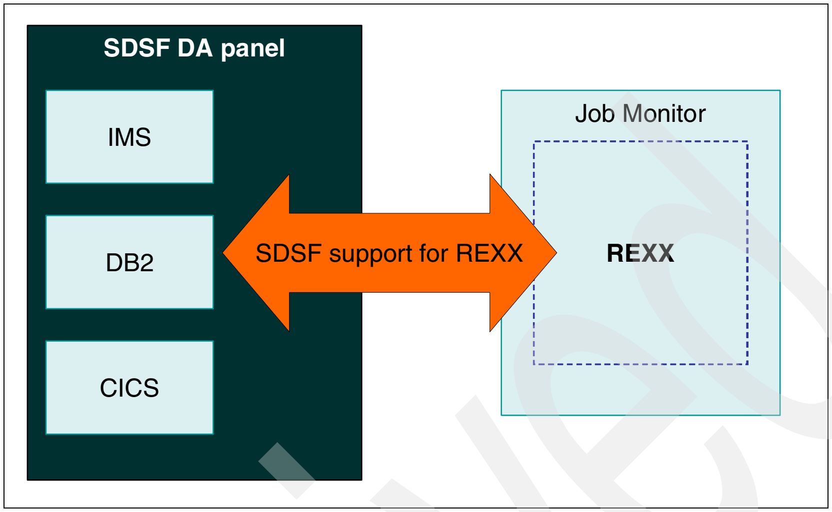


Figure 11-3 Controlling subsystems using SDSF support for the REXX language

If you want a more graphical presentation of your subsystem's health, your application can run in a workstation and get the data using the techniques presented in the previous chapter (see Chapter 10, "SDSF data in graphics" on page 223).

11.3 Application point of view of the system

SDSF presents the system from the point of view of physical resources and physical names. It, obviously, does not know anything about your applications, the subsystems they use, the jobs they run or the STCs that are needed. With REXX interface of SDSF, you might obtain a completely different view of your system. For example, you might view all the jobs and started tasks of your application together, review them, change their SYSOUT destination, their priority, and so forth.

Figure 11-4 shows an example of the system from an application point of view.

```
 Display  Filter  View  Print  Options  Help
-------------------------------------------------------------------------------
INSTALLATION STATUS DISPLAY                          LINE 1-25 (2195)
COMMAND INPUT ===>                                              SCROLL ===> CSR
APPLICATION=PAYROLL  DEST=(ALL)  OWNER=*  SORT=  SYSNAME=
NP    JOBNAME  JobID    Owner    Prty Queue      C  Pos  SAff  ASys Status
____  IMS9EXPX STC69069 STC        15 PRINT             1
____  PAYROL1  JOB25151 LEVEY       1 PRINT      B  1360
____  PAYROL1  JOB25152 LEVEY       1 PRINT      D  1361
____  PAYROL1  JOB25153 LEVEY      15 EXECUTION  U  1362 SCHI  SCHI
____  PAYROL2  JOB15219 MIU         1 PRINT      A    35
____  LOCALTAX JOB20116 DARIO       1 PRINT      A   530
____  PAYROL2  JOB21730 DARIO       1 PRINT      A  1006
____  PAYROL2  JOB21731 DARIO       1 PRINT      A  1007
____  PAYROL2  JOB23514 DARIO       1 PRINT      A  1324
____  PAYROL2  JOB23515 DARIO       1 PRINT      A  1325
____  PAYROL2  JOB23516 DARIO       1 PRINT      A  1326
____  LOCALTAX JOB23517 DARIO       1 PRINT      A  1327
____  PAYROL2  JOB24311 DARIO       1 PRINT      A  1345
____  PAYROL2  JOB24333 DARIO       1 PRINT      A  1346
____  PAYROL2  JOB24337 DARIO       1 PRINT      A  1347
____  FEDTAXES JOB24338 DARIO       1 PRINT      A  1348
____  PAYROL2  JOB24389 DARIO       1 PRINT      A  1350
____  PAYROL2  JOB24390 DARIO       1 PRINT      A  1351
____  DB9BIRLM STC23492 STC        15 PRINT      A  1352 SMIL  SMIL
____  DB9BDBM1 STC23499 STC        15 PRINT      A  1352 SMIL  SMIL
____  PAYROL2  JOB29496 MIU         1 PRINT      A    15
____  PAYROL2  JOB29499 MIU         1 PRINT      A    17
____  FEDTAXES JOB29504 MIU        15 EXECUTION  A    19 STOR  STOR
____  PAYROL3  JOB24435 DARIO       1 PRINT      A  1355
____  PAYROL3  JOB24436 DARIO       1 PRINT      A  1356
```

Figure 11-4 Application point of view of the system

11.4 Verify the service level agreement of your batch jobs

You can control the batch service level agreement of your installation using the REXX interface for SDSF to gather statistics of the jobs that are executing and also in the different queues: how long have been them in the input queue, how long took them to complete, what percentage of them has ended abnormally.

For example, if your installation service level agreement is that the turnaround of 98% of the jobs submitted in class Q must be 15 minutes or less, and 5 minutes or less for 95% of jobs in class P, you can write a small REXX exec that uses the facilities provided by SDSF to control if this agreement is being accomplished, and if not throw take a predefined action or throw an alert.

11.5 Remote control of your system

If your mainframe accepts TELNET or SSH connections, you can let your trusted applications access SDSF in order to execute queries or some predefined command easily.

For example, if an AIX® server controls whether a Web application is running correctly, and if it finds some problem, your control application can query SDSF if the WebSphere® started task is up and running or even read the logs to see if something unusual has been happening.

Another example might be that your ATM network is controlled in a dedicated computer. This application would periodically query, executing through SSH, a small SDSF REXX exec. The application could search for any of a set of batch jobs that might be running on the mainframe, and if found, could put an order in the network to not allow those operations (Figure 11-5).

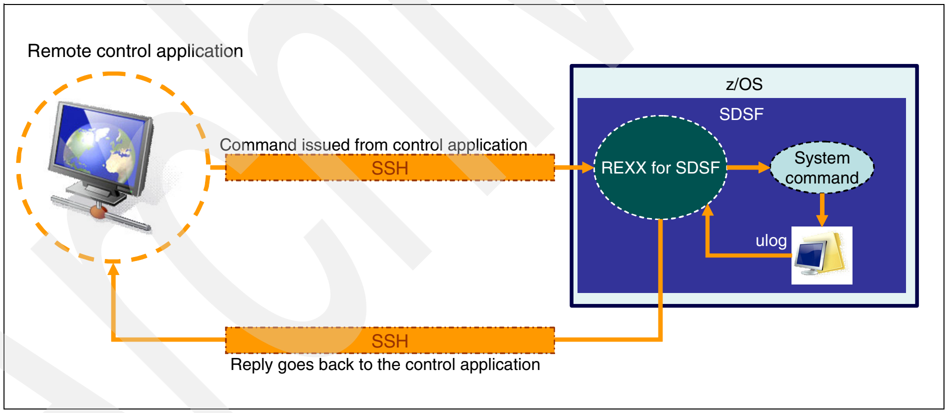


Figure 11-5 Remote control application through SSH

11.6 Add SDSF commands and data to your own tools

If your installation has a set of tools which are now familiar to the users and you were planning to add some data from SDSF to them or need to submit some authorized system command and control the reply, REXX with SDSF is a perfect solution.

Note: In writing the examples that we present in this book, we found the interactive `rexxhelp` panels quite useful. They have some small but good examples. The `rexxhelp` provides information on using REXX with SDSF. It is available in SDSF online help. The help includes links to descriptions of commands, action characters and overtypable columns. To display the online help on using REXX with SDSF, enter `rexxhelp` on any command line in SDSF when using SDSF under ISPF.

11.7 Create a personalized Workload Manager

Using the INIT and JC panels, you could write a workload manager to process a large number of different kinds of jobs. For instance, if you were running DB2 image copies as well as other maintenance for a large number of volumes, you could set up two jobclass and a set of initiators to work off both types of work. You could then monitor the number of jobs queued to each class to control which initiators would be set to which class, the purpose being to ensure that the two jobstreams end at the same time to minimize the total maintenance window.

A

REXX variables for SDSF host commands

This appendix shows some tables on the REXX variables that are required to use SDSF functions in the host environment. See Chapter 1, “Issuing a system command” on page 37 for more information about invoking SDSF host commands.

REXX variables

SDSF defines several REXX variables to supplement host environment commands or to provide request feedback. These special variables all begin with the prefix ISF. They are divided into two groups:

- General variables
- Print related variables

We present the following tables in this appendix:

- Table 1, “SDSF commands and the supported REXX interface”
- Table 2, “SET command and the supported REXX interface”
- Table 3, “CK panel and the supported REXX interface”
- Table 4, “DA panel and the supported REXX interface”
- Table 5, “ENC panel and the supported REXX interface”
- Table 6, “H panel and the supported REXX interface”
- Table 7, “I panel and the supported REXX interface”
- Table 8, “INIT panel and the supported REXX interface”
- Table 9, “JC panel and the supported REXX interface”
- Table 10, “JDS panel and the supported REXX interface”
- Table 11, “JDS panel (when accessed from H panel) and the supported REXX interface”
- Table 12, “JDS panel (when accessed from DA, I or ST panel) and the supported REXX interface”
- Table 13, “LI panel and the supported REXX interface”
- Table 14, “MAS panel and the supported REXX interface”
- Table 15, “NO panel and the supported REXX interface”
- Table 16, “O panel and the supported REXX interface”
- Table 17, “OD panel and the supported REXX interface”
- Table 18, “PR panel and the supported REXX interface”
- Table 19, “PS panel and the supported REXX interface”
- Table 20, “PUN panel and the supported REXX interface”
- Table 21, “RDR panel and the supported REXX interface”
- Table 22, “RES panel and the supported REXX interface”
- Table 23, “SE panel and the supported REXX interface”

- Table 24, “SO panel and the supported REXX interface”
- Table 25, “SP panel and the supported REXX interface”
- Table 26, “SR panel and the supported REXX interface”
- Table 27, “ST panel and the supported REXX interface”
- Table 28, “General REXX variables”
- Table 29, “REXX variables related to the tabular requests”
- Table 30, “REXX variables related to the filter commands for tabular requests”
- Table 31, “REXX variables related to browse function.”
- Table 32, “REXX variables related to printing to a SYSOUT”
- Table 33, “REXX variables related to printing to a data set”
- Table 34, “REXX variables related to printing to a SYSOUT”
- Table 35, “REXX variables related to console processing”
- Table 36, “REXX variables for diagnosing host command problems”

REXX variables for SDSF commands.

Table 1 lists the SDSF commands and the corresponding REXX interface required to perform the same function in the host environment.

Table 1 SDSF commands and the supported REXX interface

Command	Function	Use on ISFEXEC?	Use on ISFACT?	REXX variable	Notes
/	Issue a system command	Yes	No		
?	Switch between the primary and the alternate panels	Yes	No		Use the ALTERNATE and DELAYED options of the ISFEXEC command
ABEND	Force SDSF to abend	No	No		No REXX support
ACTION	Control the display of the SYSLOG WTORs	No	No		Syslog is not supported
AFD	Invoke SDSF with the ISFAFD program	No	No		No REXX support
APPC	Control the display of the transaction data	No	No	ISFAPPC	
ARRANGE	Control the order of the panel columns	No	No		No REXX support
BOOK	Invoke BookManger	No	No		No REXX support
BOTTOM	Scroll to the bottom	No	No		No REXX support
CK	Display the CK panel	Yes	Yes	See Table 29 for the REXX variables	
COLS	Display the scale line	No	No		No REXX support
DA	Display the DA panel	Yes	Yes	See Table 29 for the REXX variables	Requires RMF™
DEST	Filter the display by destination	No	No	ISFDEST	Destinations list can exceed 42 characters
DOWN	Scroll down	No	No		No REXX support

Command	Function	Use on ISFEXEC?	Use on ISFACT?	REXX variable	Notes
ENC	Display the ENC panel	Yes	Yes		
END	Return to the previous panel	No	No		No REXX support
FILTER	Filter the display	No	No	ISFFILTER & ISFFILTER2	Only 1 filter criteria can be specified
FIND	Find a data string	No	No		No REXX support
FINDLIM	Set the number of lines to search	No	No		No REXX support
H	Display the H panel	Yes	Yes	See Table 29 for the REXX variables	
I	Display the I panel	Yes	Yes	See Table 29 for the REXX variables	
I/	Issue a system command with an internal console	Yes	No		Issue a system command with the INTERNAL option of the ISFEXEC command
INIT	Display the INIT panel	Yes	Yes	See Table 29 for the REXX variables	
INPUT	Control the inclusion of the input data sets in browse	No	No	ISFINPUT	
JC	Display the JC panel	Yes	Yes	See Table 29 for the REXX variables	
LEFT	Scroll left	No	No		No REXX support
LI	Display the LINES panel	Yes	Yes	See Table 29 on page 299 for the REXX variables	
LOCATE	Locate a line or a column	No	No		No REXX support

Command	Function	Use on ISFEXEC?	Use on ISFACT?	REXX variable	Notes
LOG	Display the syslog or the operlog	No	No		Syslog and operlog are not supported
LOGLIM	Limit the operlog display	No	No		Operlog is not supported
MAS	Display the MAS panel	Yes	Yes	See Table 29 for the REXX variables	
NEXT	Skip to the next data set	No	No		No REXX support
NO	Display the NODES panel	Yes	Yes	See Table 29 for the REXX variables	
O	Display the O panel	Yes	Yes	See Table 29 for the REXX variables	
OWNER	Filter the display by owner ID	No	No	ISFOWNER	
PANELID	Display the panel ID	No	No		No REXX support
PR	Display the PR panel	Yes	Yes	See Table 29 for the REXX variables	
PREFIX	Filter the display by JOBNAME	No	No	ISFPREFIX	
PREV	Display the previous data set	No	No		No REXX support
PRINT	Print data or screen	No	No		No REXX support
PS	Display the PS panel	Yes	Yes	See Table 29 for the REXX variables	
PUN	Display the PUN panel	Yes	Yes	See Table 29 for the REXX variables	

Command	Function	Use on ISFEXEC?	Use on ISFACT?	REXX variable	Notes
QUERY	List the SDSF data	Yes	No		Support is on the QUERY AUTH command but not the QUERY MOD command; output returned in the ISFRESP stem variable
RDR	Display the RDR panel	Yes	Yes	See Table 29 for the REXX variables	
RES	Display the RES panel	Yes	Yes	See Table 29 for the REXX variables	
RESET	Clear pending actions	No	No		No REXX support
RIGHT	Scroll right	No	No		No REXX support
RM	Display the RM panel	Yes	Yes	See Table 29 for the REXX variables	
RSYS	Filter the SYSLOG WTORs by system	No	No		Syslog and operlog are not supported
SE	Display the SE panel	Yes	Yes	See Table 29 for the REXX variables	
SELECT	Display the selected rows	No	No		No REXX support
SET	Set the SDSF options	No	No	See Table 2 for the REXX variables	
SO	Display the SO panel	Yes	Yes		
SORT	Sort the display	No	No	ISFSORT & ISFSORT2	
SP	Display the SP panel	Yes	Yes	See Table 29 for the REXX variables	

Command	Function	Use on ISFEXEC?	Use on ISFACT?	REXX variable	Notes
SR	Display the SR panel	Yes	Yes	See Table 29 for the REXX variables	
ST	Display the ST panel	Yes	Yes	See Table 29 for the REXX variables	
SYSID	Filter the SYSLOG data by system ID	No	No		Syslog is not supported
SYSNAME	Filter the display by system name	No	No	ISFSYSNAME	
TOP	Scroll to the top	No	No		No REXX support
TRACE	Enable SDSF tracing	No	No	ISFTRACE & ISFTRMASK	
TUTOR	Invoke the SDSF tutorial	No	No		No REXX support
ULOG	Display the ULOG panel	No	No	ISFULOG stem variable	Issue a system command with the WAIT option of the ISFEXEC command; or issue an action character with the WAIT option of the ISFACT command
UP	Scroll up	No	No		No REXX support
W/	Issue a system command with the WAIT option	Yes	No		Issue a system command with the WAIT option of the ISFEXEC command
WHO	List environmental data	Yes	No		Output returned in the ISFRESP stem variable

Table 2 lists the SET command and its corresponding REXX interface that is required to perform the same function in the host environment.

Table 2 SET command and the supported REXX interface

Command	Function	Supported by REXX?	Input REXX variable	Output REXX variable
SET ACTION	Control the display of valid action characters	Yes	ISFACTIONS	ISFRESP stem
SET APPC	Control the display of the transaction data			
SET BROWSE	Set the default browse characters	No		
SET CONFIRM	Control the display of confirmation prompt for actions	No		
SET CONSOLE	Set the console name for ULOG	Yes	ISFCONS	
SET CURSOR	Set the cursor position	No		
SET DATE	Set the date and time format	No		
SET DELAY	Set the timeout limit for command responses	Yes	ISFDELAY	
SET DISPLAY	Control the display on current active filters	Yes		ISFDISPLAY
SET HEX	Control the display of browse data in hex	No		
SET LANG	Set the default panel language	No		
SET LOG	Set the default log type	No		
SET SCHARS	Set the search characters for the FIND command	Yes	ISFSCHARS	
SET SCREEN	Set the screen display characteristics	No		
SET SHELF	Set the bookshelf name	No		
SET TIMEOUT	Set the timeout limit for the sysplex displays	Yes	ISFTIMEOUT	

Table 3 to Table 27 show the supported action characters on the ISFACT command.

Table 3 CK panel and the supported REXX interface

Action	Description	Supported by REXX?
//	Block repeat; type // on the first row and another // on the last row to be processed	No
=	Repeat previous action character or overtype	No
+	Expand the NP column. (Use RESET to reset)	No
A	Activate	Yes
D	Display	Yes
DL	Display long	Yes
DP	Display policies	Yes
DPO	Display policies that are outdated and not applied	Yes
DS	Display status	Yes
E	Refresh	Yes
H	Deactivate	Yes
P	Delete	Yes
PF	Delete force	Yes
R	Run	Yes
S	Browse	Yes
SB	Browse using ISPF BROWSE	No
SE	Browse using ISPF EDIT	No
U	Remove all categories for the check	Yes
X	Print the check output	Yes
XC	Print the check output and close the print file	
XD	Display the Open Print Dataset panel	Yes
XDC	Display the Open Print Dataset panel and close the print file	Yes
XF	Display the Open Print File panel	Yes

Action	Description	Supported by REXX?
XFC	Display the Open Print File panel and close the print file	Yes
XS	Display the Open Print panel	Yes
XSC	Display the Open Print panel and close the print file	Yes

Table 4 DA panel and the supported REXX interface

Action	Description	Supported by REXX?
//	Block repeat; type // on the first row and another // on the last row to be processed	No
=	Repeat previous action character or overtype	No
+	Expand the NP column. (Use RESET to reset)	No
A	Release a held job	Yes
C	Cancel a job	Yes
CA	Cancel a job that is defined to Automatic Restart Manager (ARM)	Yes
CDA	Cancel a job that is defined to ARM and take a dump	Yes
D	Display job information in the log	Yes
DL	Display job information in the log, long form	Yes
E	Process a job again	Yes
EC	Process a job again, but cancel and hold it prior to execution	Yes
H	Hold a job	Yes
K	Cancel a start task system cancel)	Yes
KD	Cancel a started task and take a dump (system cancel)	Yes
L	List output status of a job in the log	Yes
P	Cancel a job and purge its output	Yes
PP	Cancel a protected job and purge its output	Yes
R	Reset and resume a job	Yes
RQ	Reset and quiesce a job	Yes

Action	Description	Supported by REXX?
Q	Display out put descriptors for all of the data sets in an output group	No
S	Browse	No
SA	Browse using data set allocation	Yes only
SB	Browse using ISPF BROWSE	No
SE	Browse using ISPF EDIT	No
SJ	JCL Edit	No
SJA	JCL edit using data set allocation	Yes only
W	Cause job and message logs to spin	Yes
X	Print the job output	Yes
XC	Print the job output and close the print file	Yes
XD	Display the Open Print Dataset panel	Yes
XDC	Display the Open Print Dataset panel and close the print file	Yes
XF	Display the Open Print File panel	Yes
XFC	Display the Open Print File panel and close the print file	Yes
XS	Display the Open Print panel	Yes
XSC	Display the Open Print panel and close the print file	Yes
Y	Stop a started task (system stop)	Yes
Z	Cancel a started task (system force)	Yes
?	Display a list of data sets for a job. (Access the Job Dataset panel)	Yes

Table 5 ENC panel and the supported REXX interface

Action	Description	Supported by REXX?
//	Block repeat; type // on the first row and another // on the last row to be processed	No
=	Repeat previous action character or overtype	No
+	Expand the NP column.	No

Action	Description	Supported by REXX?
I	Display additional information about the enclave	No
M	Match the enclave by export token, to display on the instances of a multisystem enclave. Valid only for multisystem enclaves, as indicated in the Scope column. To see all enclaves again, reaccess the panel	No
R	Reset and resume an enclave	Yes
RQ	Reset and quiesce an enclave	Yes

Table 6 H panel and the supported REXX interface

Action	Description	Supported by REXX?
//	Block repeat; type // on the first row and another // on the last row to be processed	No
=	Repeat previous action character or overtype	No
+	Expand the NP column.	No
?	Display a list of the data sets for an output group. (Access the Job Data Set Panel)	Yes
A	Release a job's output	Yes
C	Purge a job's output	Yes
H	Hold a job's output	Yes
L	List a job's output in the log	Yes
LL	List a job's output in the log, long form	Yes
O	Release output to be printed, then purged	Yes
OK	Release output to be printed and kept	Yes
P	Purge a job's output	Yes
Q	Display output descriptors for all of the data sets for an output group	No
S	Browse	No
SA	Browse using data set allocation	Yes only
SB	Browse using ISPF BROWSE	No

Action	Description	Supported by REXX?
SE	Browse using ISPF EDIT	No
SJ	JCL Edit	No
SJA	JCL edit using data set allocation	Yes only
X	Print the job output	Yes
XC	Print the job output and close the print file	Yes
XD	Display the Open Print Dataset panel	Yes
XDC	Display the Open Print Dataset panel and close the print file	Yes
XF	Display the Open Print File panel	Yes
XFC	Display the Open Print File panel and close the print file	Yes
XS	Display the Open Print panel	Yes
XSC	Display the Open Print panel and close the print file	Yes

Table 7 I panel and the supported REXX interface

Action	Description	Supported by REXX?
//	Block repeat; type // on the first row and another // on the last row to be processed	No
=	Repeat previous action character or overtype	No
+	Expand the NP column.	No
?	Display a list of the data sets for a job. (Access the Job Data Set panel)	Yes
A	Release a held job	Yes
C	Cancel a job	Yes
CA	Cancel a job that is defined to Automatic Restart Manager (ARM)	Yes
CD	Cancel a job and take a dump	Yes
CDA	Cancel a job that is defined to ARM and take a dump	Yes
D	Display job information in the log	Yes
DL	Display job information in the log, long form	Yes

Action	Description	Supported by REXX?
E	Process a job again	Yes
EC	Process a job again, but cancel and hold it prior to execution	Yes
H	Hold a job	Yes
I	Display job delay information	Yes
J	Start a job immediately (WLM-managed classes only)	Yes
L	List output status of a job in the log	Yes
P	Cancel a job and purge its output	Yes
PP	Cancel a protected job and purge its output	Yes
Q	Display output descriptors for all of the data sets for an output group	No
S	Browse	No
SA	Browse using data set allocation	Yes only
SB	Browse using ISPF BROWSE	No
SE	Browse using ISPF EDIT	No
SJ	JCL Edit	No
SJA	JCL edit using data set allocation	Yes only
W	Cause job and message logs to spin	Yes
X	Print the job output	Yes
XC	Print the job output and close the print file	Yes
XD	Display the Open Print Dataset panel	Yes
XDC	Display the Open Print Dataset panel and close the print file	Yes
XF	Display the Open Print File panel	Yes
XFC	Display the Open Print File panel and close the print file	Yes
XS	Display the Open Print panel	Yes
XSC	Display the Open Print panel and close the print file	Yes

Table 8 INIT panel and the supported REXX interface

Action	Description	Supported by REXX?
//	Block repeat; type // on the first row and another // on the last row to be processed	No
=	Repeat previous action character or overtype	No
+	Expand the NP column.	No
D	Display information about an initiator	Yes
DL	Display the long form of information about an initiator	Yes
P	Stop an initiator when the current job completes	Yes
S	Start an initiator	Yes
Z	Halt an initiator when the current job completes. This suspends, rather than stops, the initiator	Yes

Table 9 JC panel and the supported REXX interface

Action	Description	Supported by REXX?
//	Block repeat; type // on the first row and another // on the last row to be processed	No
=	Repeat previous action character or overtype	No
+	Expand the NP column.	No
D	Display information about a job class in the logs and ULOG	Yes
ST	Display the ST panel for all jobs in the class	No

Table 10 JDS panel and the supported REXX interface

Action	Description	Supported by REXX?
//	Block repeat; type // on the first row and another // on the last row to be processed	No
=	Repeat previous action character or overtype	No
+	Expand the NP column.	No
C	Purge an output data set	Yes

Action	Description	Supported by REXX?
H	Hold an output data set	Yes
O	Release an output data set	Yes
P	Purge an output data set	Yes
Q	Display output descriptors for the data set	No
S	Browse	No
SA	Browse using data set allocation	Yes only
SB	Browse using ISPF BROWSE	No
SE	Browse using ISPF EDIT	No
SJ	JCL edit	No
SJA	JCL edit using data set allocation	Yes only
V	View page mode output	No
X	Print the job output. Add C to close the print file after printing (XC)	No
XD	Display the Open Print Data Set panel (XD or XDC)	No
XF	Display the Open Print Data Set panel (XF or XFC)	No
XS	Display the Open Print panel (XS or XSC)	No

Table 11 JDS panel (when accessed from H panel) and the supported REXX interface

Action	Description	Supported by REXX?
//	Block repeat; type // on the first row and another // on the last row to be processed	No
=	Repeat previous action character or overtype	No
+	Expand the NP column.	No
C	Purge an output data set	Yes
O	Release an output data set	Yes
P	Purge an output data set	Yes
Q	Display output descriptors for the data set	No
S	Browse	No

Action	Description	Supported by REXX?
SA	Browse using data set allocation	Yes only
SB	Browse using ISPF BROWSE	No
SE	Browse using ISPF EDIT	No
SJ	JCL edit	No
SJA	JCL edit using data set allocation	Yes only
V	View page mode output	No
X	Print the job output. Add C to close the print file after printing (XC)	No
XD	Display the Open Print Data Set panel (XD or XDC)	No
XF	Display the Open Print File panel (XF or XFC)	No
XS	Display the Open Print panel (XS or XSC)	No

Table 12 JDS panel (when accessed from DA, I or ST panel) and the supported REXX interface

Action	Description	Supported by REXX?
//	Block repeat; type // on the first row and another // on the last row to be processed	No
=	Repeat previous action character or overtype	No
+	Expand the NP column	No
Q	Display output descriptors for the data set	No
S	Browse	No
SA	Browse using data set allocation	Yes only
SB	Browse using ISPF BROWSE	No
SE	Browse using ISPF EDIT	No
SJ	JCL Edit	No
SJA	JCL edit using data set allocation	Yes only
V	View page mode output	No
X	Print the job output. Add C to close the print file after printing (XC)	No
XD	Display the Open Print Data Set panel (XD or XDC)	No

Action	Description	Supported by REXX?
XF	Display the Open Print File panel (XF or XFC)	No
XS	Display the Open Print panel (XS or XSC)	No

Table 13 LI panel and the supported REXX interface

Action	Description	Supported by REXX?
//	Block repeat; type // on the first row and another // on the last row to be processed	No
=	Repeat previous action character or overtype	No
+	Expand the NP column.	No
C	Cancel a transmitter or receiver	Yes
D	Display a line, transmitter or receiver in the log	Yes
E	Restart a line, transmitter or receiver	Yes
I	Interrupt a line	Yes
P	Drain a line, transmitter or receiver	Yes
Q	Quiesce a line	Yes
S	Start a line, transmitter, or receiver	Yes
SN	Start network communications	Yes

Table 14 MAS panel and the supported REXX interface

Action	Description	Supported by REXX?
//	Block repeat; type // on the first row and another // on the last row to be processed	No
=	Repeat previous action character or overtype	No
+	Expand the NP column.	No
D	Display a member of the MAS in the log	Yes
E	Restart a member of the MAS	Yes
ER	Reset a member of the MAS	Yes

Action	Description	Supported by REXX?
J	Display the current state of monitor subtasks	Yes
JD	Display monitor details	Yes
JH	Display resource history	Yes
JJ	Display the current state of JES2	Yes
JS	Display the current status of JES2	Yes
P	Stop a member of the MAS	Yes
PA	Stop a member of the MAS (abend)	Yes
PQ	Stop a member of the MAS, ignoring cross system activity	Yes
PT	Stop a member of the MAS, ignoring active programs	Yes
PX	Stop scheduling of jobs for the member of the MAS	Yes
S	Start a member of the MAS	Yes
SX	Start scheduling of jobs for a member of the MAS	Yes
ZM	Stop the JES2 monitor	Yes

Table 15 NO panel and the supported REXX interface

Action	Description	Supported by REXX?
//	Block repeat; type // on the first row and another // on the last row to be processed	No
=	Repeat previous action character or overtype	No
+	Expand the NP column.	No
D	Display information about a node in the log	Yes
DC	Display information about network connections for a node in the log	Yes
DP	Display information about paths in the log	Yes
SN	Start node communication on a line	Yes

Table 16 O panel and the supported REXX interface

Action	Description	Supported by REXX?
//	Block repeat; type // on the first row and another // on the last row to be processed	No
=	Repeat previous action character or overtype	No
+	Expand the NP column.	No
?	Display a list of the data sets for an output group	Yes
A	Release held output data sets. (If job has been held, it must be released from the Status panel).	Yes
C	Purge a job's output (do not cancel the job)	Yes
H	Hold output	Yes
L	List a job's output status in the log	Yes
LL	List a job's output status in the log, long form	Yes
P	Purge output data sets	Yes
Q	Display output descriptors for all of the data sets for an output group	No
S	Browse	No
SA	Browse using data set allocation	Yes only
SB	Browse using ISPF BROWSE	No
SE	Browse using ISPF EDIT	No
SJ	JCL Edit	No
SJA	JCL edit using data set allocation	Yes only
X	Print the job output	Yes
XC	Print the job output and close the print file	Yes
XD	Display the Open Print Data Set panel	Yes
XDC	Display the Open Print Data Set panel and close the print file	Yes
XF	Display the Open Print File panel	Yes
XFC	Display the Open Print File panel and close the print file	Yes
XS	Display the Open Print panel	Yes

Action	Description	Supported by REXX?
XSC	Display the Open Print panel and close the print file	Yes

Table 17   OD panel and the supported REXX interface

Action	Description	Supported by REXX?
E	Erase an output descriptor. E is valid only when the Output Descriptors panel is accessed from the: ► Output Queue panel ► Held Ooutput Queue panel ► Job Data Set panel if it was accessed from the Output Queue panel or the Held Output Queue panel	No
?	Display a list of data sets	No
S	Browse	No
SB	Browse using ISPF BROWSE	No
SE	Browse using ISPF EDIT	No
SJ	JCL edit	No
X	Print job output	No
XC	Print job output and close the print file	No
XD	Display the Open Print Data Set panel	No
XDC	Display the Open Print Data Set panel and close the print file	No
XF	Display the Open Print File panel	No
XFC	Display the Open Print File panel and close the print file	No
XS	Display the Open Print panel	No
XSC	Display the Open Print panel and close the print file	No

Table 18   PR panel and the supported REXX interface

Action	Description	Supported by REXX?
//	Block repeat; type // on the first row and another // on the last row to be processed	No
=	Repeat previous action character or overtype	No

Action	Description	Supported by REXX?
+	Expand the NP column.	No
B(x)	Backspace a printer. ‘x’ can be: ► number of pages ► D (top of the current data set) ► C (most recent checkpoint) ► C,number (pages before the most recent checkpoint)	Yes
C	Purge output printing on a printer	Yes
D	Display information about a job	Yes
DL	Display the long form of information about a job	Yes
E	Restart a printer	Yes
F(x)	Space a printer forward. x can be: ► number of pages ► D (top of the current data set) ► C(most recent checkpoint) ► number,C(pages before the most recent checkpoint)	Yes
I	Interrupt a printer	Yes
K	Force termination of the FSS	Yes
N	Print another copy of the output	Yes
P	Stop a printer	Yes
S	Start a printer	Yes
Z	Halt an active printer	Yes

Table 19 PS panel and the supported REXX interface

Action	Description	Supported by REXX?
//	Block repeat; type // on the first row and another // on the last row to be processed.	No
=	Repeat previous action character or overtype	No
+	Expand the NP column	No
C	Cancel the address space that owns the process	Yes
D	Display information about processes	Yes

Action	Description	Supported by REXX?
K	Kill the process (SIGKILL)	Yes
T	Kill the process (SIGTERM)	Yes

Table 20 PUN panel and the supported REXX interface

Action	Description	Supported by REXX?
//	Block repeat; type // on the first row and another // on the last row to be processed.	No
=	Repeat previous action character or overtype	No
+	Expand the NP column	No
Bx	Backspace a punch. x can be: - number of pages - D (top of the current data set) - C (most recent checkpoint) - C, number (pages before the most recent checkpoint)	Yes
C	Purge output being processed by a punch	Yes
D	Display information about a job	Yes
DL	Display the long form of information about a job	Yes
E	Restart a punch	Yes
Fx	Space a punch forward. x can be: - number of pages - D (top of the current data set) - C (most recent checkpoint) - C, number (pages before the most recent checkpoint)	Yes
I	Interrupt a punch	Yes
N	Punch another copy of the output	Yes
P	Stop a punch	Yes
S	Start a punch	Yes
Z	Halt an active punch	Yes

Table 21 RDR panel and the supported REXX interface

Action	Description	Supported by REXX?
//	Block repeat; type // on the first row and another // on the last row to be processed	No
=	Repeat previous action character or overtype	No
+	Expand the NP column	No
C	Cancel a job being processed by a reader	Yes
D	Display information about a job	Yes
DL	Display the long form of information about a job	Yes
P	Stop a reader	Yes
S	Start a reader	Yes
Z	Halt a reader	Yes

Table 22 RES panel and the supported REXX interface

Action	Description	Supported by REXX?
//	Block repeat; type // on the first row and another // on the last row to be processed	No
=	Repeat previous action character or overtype	No
+	Expand the NP column	No
D	Display information about the resource	Yes

Table 23 SE panel and the supported REXX interface

Action	Description	Supported by REXX?
//	Block repeat; type // on the first row and another // on the last row to be processed	No
=	Repeat previous action character or overtype	No
+	Expand the NP column	No
D	Display scheduling environments in the log. This issues the MVS D command	Yes

Action	Description	Supported by REXX?
R	Display resources for a scheduling environment	No
ST	Display the ST panel for all jobs requiring the scheduling environment	No

Table 24 SO panel and the supported REXX interface

Action	Description	Supported by REXX?
//	Block repeat; type // on the first row and another // on the last row to be processed	No
=	Repeat previous action character or overtype	No
+	Expand the NP column	No
C	Cancel a transmitter or receiver	Yes
D	Display an offloader, transmitter, or receiver in the log	Yes
E	Restart a transmitter	Yes
P	Drain an offloader, transmitter or receiver	Yes
S	Start a transmitter or receiver	Yes
SR	Start an offloader to receive jobs and SYSOUT	Yes
ST	Start an offloader to transmit jobs and SYSOUT	Yes

Table 25 SP panel and the supported REXX interface

Action	Description	Supported by REXX?
//	Block repeat; type // on the first row and another // on the last row to be processed	No
=	Repeat previous action character or overtype	No
+	Expand the NP column	No
D	Display the status of a spool volume	Yes
DL	Display the long form of status	Yes
J	Display all jobs using the spool volume	Yes
P	Drain a spool volume	Yes

Action	Description	Supported by REXX?
PC	Drain a spool volume and cancel all jobs that have used it	Yes
S	Start a spool volume, adding or reactivating it to the spool configuration	Yes
Z	Halt a spool volume, deallocating it after active work completes its current phase of processing	Yes

Table 26 SR panel and the supported REXX interface

Action	Description	Supported by REXX?
//	Block repeat; type // on the first row and another // on the last row to be processed	No
=	Repeat previous action character or overtype	No
+	Expand the NP column	No
C	Remove an action message	Yes
D	Display a message in the logs or ULOG	Yes
R(command)	Reply to the message. R by itself displays a pop-up on which you can complete the command	Yes

Table 27 ST panel and the supported REXX interface

Action	Description	Supported by REXX?
//	Block repeat; type // on the first row and another // on the last row to be processed	No
=	Repeat previous action character or overtype	No
+	Expand the NP column	No
?	Display a list of the data sets for a job	Yes
A	Release a held job	Yes
C	Cancel an active job or a job waiting to be processed	Yes
CA	Cancel a job that is defined to Automatic Restart Manager (ARM)	Yes
CD	Cancel a job and take a dump	Yes

Action	Description	Supported by REXX?
CDA	Cancel a job that is defined to ARM, and take a dump	Yes
D	Display job information in the log	Yes
DL	Display job information in the log, long form	Yes
E	Process a job again	Yes
EC	Process a job again, but hold it prior to execution	Yes
H	Hold a job	Yes
I	Display job delay information	No
J	Start a job immediately (WLM-managed classes only)	Yes
L	List a job's output status in the log	Yes

Table 28 lists the general REXX variables and their corresponding SDSF functions. For more details about each variable, refer to *z/OS V1R9.0 SDSF Operation and Customization*, SA22-7670.

Table 28 General REXX variables

Variable	Description	Associated online command	Input or output	Stem variable?
ISFACTIONS	Controls return of valid action characters	SET ACTION	Input	No
ISFAPPC	Controls the display of the transaction data	SET APPC	Input	No
ISFCOLS	Sets the columns to be returned for the primary panel; Returns the columns for the primary panel		Input Output	No
ISFCOLS2	Sets the columns to be returned for the secondary panel; Returns the columns for the secondary panel		Input Output	No
ISFCONS	Sets the console name for ULOG	SET CONSOLE	Input	No
ISFDCOLS	Returns the delayed access columns for the primary panel		Output	No
ISFDCOLS2	Returns the delayed access columns for the secondary panel		Output	No

Variable	Description	Associated online command	Input or output	Stem variable?
ISFDDNAME	Returns the ddnames within the requested row entry		Output	Yes
ISFDELAY	Sets the timeout limit for command responses	SET DELAY	Input	No
ISFDEST	Sets the destinations for filtering	DEST	Input	No
ISFDISPLAY	Returns the current active filters	SET DISPLAY	Output	No
ISFDSNAME	Returns the spool data set names within the requested row entry		Output	Yes
ISFFILTER	Sets a single filter criterion for the primary panel	FILTER	Input	No
ISFFILTER2	Sets a single filter criteria for the secondary panel	FILTER	Input	No
ISFINPUT	Controls the inclusion of the input data sets in browse	SET INPUT	Input	No
ISFJESNAME	Sets the JES2 subsystem to be processed		Input	No
ISFLINE	Returns the lines of data in response to a browse request		Output	Yes
ISFMSG	Returns SDSF short message		Output	No
ISFMSG2	Returns SDSF messages associated with a request, especially when the VERBOSE option is specified on a tabular request		Output	Yes
ISFOWNER	Sets the owner ID for filtering	OWNER	Input	No
ISFPREFIX	Sets the jobname prefix for filtering	PREFIX	Input	No
ISFRCOLS	Returns the related field columns for the primary panel		Output	No
ISFRCOL2	Returns the related field columns for the secondary panel		Output	No
ISFRESP	Returns the command responses from the WHO and the QUERY AUTH commands; returns the output when ISFACTIONS is set to ON	QUERY AUTH & WHO & SET ACTION	Output	Yes

Variable	Description	Associated online command	Input or output	Stem variable?
ISFROWS	Returns the number of rows created by a tabular request for the primary panel		Output	Yes
ISFROWS2	Returns the number of rows created by a tabular request for the secondary panel		Output	Yes
ISFSCHARS	Sets the search characters for the FIND command		Input	No
ISFSERVER	Sets the SDSF server to process the host command		Input	No
ISFSORT	Sets the criteria for sorting the primary panel		Input	No
ISFSORT2	Sets the criteria for sorting the secondary panel		Input	No
ISFSYSNAME	Sets the system name for filtering	SYSNAME	Input	No
ISFTIMEOUT	Sets the timeout limit for the sysplex displays	SET TIMEOUT	Input	No
ISFTITLES	Returns the column names associated with the variables on the primary panel		Output	No
ISFTITLES2	Returns the column names associated with the variables on the secondary panel		Output	No
ISFTLINE	Returns the title line from the tabular request		Output	No
ISFTRACE	Enables SDSF tracing	TRACE	Input	No
ISFTRMASK	Sets the trace mask	TRACE	Input	No
ISFUCOLS	Returns the modifiable columns for the primary panel		Output	No
ISFUCOLS2	Returns the modifiable columns for the secondary panel		Output	No
ISFULOG	Returns the console activation message, system command echo and command responses		Output	Yes

Table 29 lists the REXX variables related to the tabular requests.

Table 29 REXX variables related to the tabular requests

REXX Variable	Description	Input or Output	Stem variable?
ISFACTIONS	Controls return of valid action characters	Input	No
ISFCOLS	Sets the columns to be returned for the primary panel; Returns the columns for the primary panel	Input Output	No
ISFCOLS2	Sets the columns to be returned for the secondary panel; Returns the columns for the secondary panel	Input Output	No
ISFCONS	Sets the console name for slash command and actions	Input	No
ISFDCOLS	Returns the delayed access columns for the primary panel	Output	No
ISFDCOLS2	Returns the delayed access columns for the secondary panel	Output	No
ISFDELAY	Set the timeout limit for command responses	Input	No
ISFDISPLAY	Returns the current active filters	Output	No
ISFMSG	Returns SDSF short message	Output	No
ISFMSG2	Returns SDSF messages associated with a request, especially when VERBOSE option is specified on a tabular request	Output	Yes
ISFRCOLS	Returns the related field columns for the primary panel	Output	No
ISFRCOL2	Returns the related field columns for the secondary panel	Output	No
ISFROWS	Returns the number of rows created by a tabular request for the primary panel	Output	Yes
ISFROWS2	Returns the number of rows created by a tabular request for the secondary panel	Output	Yes
ISFSCHARS	Set the search characters for the FIND command	Input	No

REXX Variable	Description	Input or Output	Stem variable?
ISFSORT	Sets the criteria for sorting the primary panel	Input	No
ISFSORT2	Sets the criteria for sorting the secondary panel	Input	No
ISFTIMEOUT	Sets the timeout limit for the sysplex displays	Input	No
ISFTITLES	Returns the column names associated with the variables on the primary panel	Output	No
ISFTITLES2	Returns the column names associated with the variables on the secondary panel	Output	No
ISFTLINE	Returns the title line from the tabular request	Output	No
ISFUCOLS	Returns the modifiable columns for the primary panel	Output	No
ISFUCOLS2	Returns the modifiable columns for the secondary panel	Output	No

Table 30 lists the REXX variables related to the filter commands for tabular requests.

Table 30 REXX variables related to the filter commands for tabular requests

Variable	Associated SDSF filter command	Input or Output	Stem variable?
ISFAPPC	SET APPC command	Input	No
ISFDEST	DEST command	Input	No
ISFFILTER	FILTER command for the primary panel	Input	No
ISFFILTER2	FILTER command for the secondary panel	Input	No
ISFINPUT	INPUT command	Input	No
ISFJESNAME	No corresponding SDSF command but sets the JES2 subsystem to be processed	Input	No
ISFOWNER	OWNER command	Input	No
ISFPREFIX	PREFIX command	Input	No

Variable	Associated SDSF filter command	Input or Output	Stem variable?
ISFSERVER	No corresponding SDSF command but sets the SDSF server to process the host command	Input	No
ISFSYSNAME	SYSNAME command	Input	No

Table 31 lists the REXX variables related to the browse function

Table 31 REXX variables related to browse function.

Variable	Description	Input or Output	Stem variable?
ISFDDNAME	Returns the ddnames within the requested row entry	Output	Yes
ISFDSNAME	Returns the Spool data set names within the requested row entry	Output	Yes
ISFLINE	Returns the lines of data in response to a browse request	Output	Yes

SDSF can print a SYSOUT to another SYSOUT, to a data set, and to a ddname. Table 32 list the REXX variables related to print a SYSOUT to another SYSOUT.

Table 32 REXX variables related to printing to a SYSOUT

Variable	Description	Input or Output	Stem variable?
ISFPRTCLASS	Sets the output class	Input	No
ISFPRTCOPIES	Sets the number of copies	Input	No
ISFPRTDEST	Sets the destination	Input	No
ISFPRTFCB	Sets the FCB	Input	No
ISFPRTFORMDEF	Sets the formdef	Input	No
ISFPRTFORMS	Sets the forms	Input	No
ISFPRTOUTDESNAME	Sets the output descriptor name	Input	No
ISFPRTPAGEDEF	Sets the pagedef for the SYSOUT	Input	No

Variable	Description	Input or Output	Stem variable?
ISFPRTPRTMODE	Sets the process mode	Input	No
ISFPRTUCS	Sets the UCB	Input	No

Table 33 lists the REXX variables related to print a SYSOUT to a data set.

Table 33 REXX variables related to printing to a data set

Variable	Description	Input or Output	Stem variable?
ISFPRTBLKSIZE	Sets the block size	Input	No
ISFPRTDATACLAS	Sets the data class	Input	No
ISFPRTDIRBLKS	Sets the number of directory blocks	Input	No
ISFPRTDISP	Sets the allocation disposition	Input	No
ISFPRTDSNAME	Sets the data set name	Input	No
ISFPRTLRECL	Sets the logical record length	Input	No
ISFPRTMEMBER	Sets the member name	Input	No
ISFPRTMGMTCLAS	Sets the management class	Input	No
ISFPRTPRIMARY	Sets the primary space	Input	No
ISFPRTRECFM	Sets the record format	Input	No
ISFPRTSECONDARY	Sets the secondary space	Input	No
ISFPRTSPACETYPE	Sets the allocation space units for the data set	Input	No
ISFPRTSTORCLAS	Sets the storage class for the data set	Input	No
ISFPRTUNIT	Sets the allocation unit for the data set	Input	No
ISFPRTVOLSER	Sets the volume serial for the data set	Input	No

Table 34 lists the REXX variables related to the print a SYSOUT to a ddname.

Table 34 REXX variables related to printing to a SYSOUT

Variable	Description	Input or Output	Stem variable?
ISFPRTDDNAME	Sets the ddname of the output file	Input	No

Table 35 lists the REXX variables related to the console functions.

Table 35 REXX variables related to console processing

Variable	Description	Input or Output	Stem variable?
ISFCONS	Sets the console name for ULOG	Input	No
ISFDELAY	Sets the timeout limit for command responses	Input	No
ISFULOG	Returns the system command echo and the system responses generated by the host command	Output	Yes

Table 36 lists the REXX variables for debugging the host command problems.

Table 36 REXX variables for diagnosing host command problems

Variable	Description	Input or Output	Stem variable?
ISFMSG	Returns SDSF short message	Output	No
ISFMSG2	Returns SDSF messages associated with a request, especially when VERBOSE option is specified on a tabular request	Output	Yes
ISFTRACE	Enables SDSF tracing	Input	No
ISFTRMASK	Sets the trace mask	Input	No

B

Additional material

This appendix describes the additional material to which this book refers that you can download from the Internet.

Locating the Web material

The Web material that is associated with this book is available in softcopy on the Internet from the IBM Redbooks Web server. Point your Web browser to:

`ftp://www.redbooks.ibm.com/redbooks/SG247419`

Alternatively, you can go to the IBM Redbooks Web site at:

ibm.com/redbooks

Select **Additional materials** and open the directory that corresponds with the IBM Redbooks form number, SG24-7419.

Using the Web material

The additional Web material that accompanies this book includes the following files:

File name	*Description*
SG247419.zip	Compressed Code Samples for each chapter

System requirements for downloading the Web material

The following system configuration is recommended:

Hard disk space:	2 MB minimum
Operating System:	At least Windows XP
Processor:	1700 MHz or higher
Memory:	At least 500 MB

How to use the Web material

Create a subdirectory (folder) on your workstation, and decompress the contents of the Web material compressed file into this folder.

Glossary

3270 Display devices made by IBM since 1972. Unlike common serial ASCII terminals, the 3270 minimizes the number of I/O interrupts that are required by accepting large blocks of data known as data streams and uses high-speed proprietary communications interface.

AIX (Advanced Interactive Executive) A UNIX operating system developed by IBM that is designed and optimized to run on POWER™ microprocessor-based hardware such as servers, workstations, and blades. Based on UNIX System V, AIX was introduced in 1986.

AMRF (Action Message Retention Facility) A z/OS facility that, when active, retains all action messages, except those specified by the installation.

ATM An automated teller machine is a computerized telecommunications device that provides a financial institution's customers a method of financial transactions in a public space without the need of a human clerk or bank teller.

BPXBATCH MVS utility that can be used to run shell commands or shell scripts and to run executable files through the MVS batch facility. BPXBATCH can be invoked from a batch job or from the TSO/E environment.

CK Health Checker panel (CK) The CK panel shows information from IBM Health Checker for z/OS about the active checks.

client/server Pertaining to the model of interaction in distributed data processing in which a program on one computer sends a request to a program on another computer and awaits response. The requesting program (often an application that uses a graphical user interface) is called a client; the answering program is called a server. Each instance of the client software can send requests to the server.

COBOL (Common Business Oriented Language) High level third-generation programming language that is used primarily for commercial data processing. Its primary domain is in business, finance and administrative systems. It is one of the oldest programming languages still in active use.

console A display station from which an operator can control and observe the system operation.

DA Display Active Users panel (DA) The DA panel shows information about MVS address spaces (jobs, started tasks, and TSO users) that are running. SDSF obtains the information from RMF when it is installed. Columns for which RMF is required are indicated by RMF.

Eclipse An open-source initiative that provides ISVs and other tool developers with a standard platform comprised of extensible frameworks for developing plug-in compatible application development tools.

ENC Enclaves panel (ENC) The Enclaves panel allows authorized users to display information about WLM enclaves.

H Held Output panel (H) The Held Output panel shows the user information about SYSOUT data sets for jobs, started tasks, and TSO users on any held JES2 output queue.

Health Checker An IBM licensed program that provides a foundation to help simplify and automate the identification of potential configuration problems before they impact system availability.

HLL (High Level Language) A programming language that provides some level of abstraction from assembler language and independence from a particular type of system. Rather that dealing with registers, memory addresses and call stacks, high-level languages deal with variables, arrays, boolean expressions or complex arithmetic.

I Input Queue panel (I) The Input Queue panel allows the user to display information about jobs, started tasks, and TSO users on the JES2 input queue or executing.

SR SDSF System Requests panel (SR) The System Requests panel allows authorized users to display information about reply and action messages.
If AMRF is not active, the panel shows only reply messages. This is controlled by the AMRF parameter in PARMLIB member CONSOLxx.

IMS (Information Management System) System environment with a database manager and transaction processing that are capable of managing complex databases and terminal networks.

INIT Initiator panel (INIT) The Initiator panel allows users to display information about JES2 initiators that are defined in the active JES2 on their CPUs.

IRXEXEC Routine to run REXX execs from any MVS address space.running in any address space. IRXEXEC routine gives you more flexibility than IRXJCL. For example, you can preload the REXX exec in storage and pass the address of the preloaded exec to IRXEXEC. This is useful if you want to run an exec multiple times to avoid the exec being loaded and freed whenever it is invoked. You might also want to use your own load routine to load and free the exec.

IRXJCL Routine to run REXX execs from any MVS address space.

ISFACT REXX command, belonging to the SDSF host command environment, used to execute SDSF commands such as the commands to SDSF panels.

ISFEXEC REXX command, belonging to the SDSF host command environment, used for action characters and overtyping columns.

ISPF (Interactive System Productivity Facility) IBM licensed program that serves as a full-screen editor and dialog manager. Used for writing application programs, it provides a IBM 3270 terminal interface, and a means of generating standard screen panels and interactive dialogs between the application programmer and terminal user. ISPF is frequently used to manipulate z/OS data sets through its Product Development Facility (PDF). ISPF is user extensible and it is often used as an application program interface (API).

Java An object-oriented programming language for portable interpretive code that supports interaction among remote objects Java was developed and specified by Sun Microsystems, Incorporated.

JC Job Class panel (JC) The Job Class (JC) panel allows authorized users to display and control the job classes in the MAS. It shows both JES2 and WLM managed job classes.

JDK (Java Development Kit) The name of the software development kit that Sun Microsystems provides for the Java platform.

JDS Job Data Set panel (JDS) The Job Data Set panel allows the user to display information about SYSOUT data sets for a selected job, started task, and TSO user.

JES (Job Entry Subsystem) An IBM licensed program that receives jobs into the system and processes all output data that is produced by jobs.

JES2 An MVS subsystem that receives jobs into the system, converts them to internal format, selects them for execution, processes their output, and purges them from the system. In an installation with more than one processor, each JES2 processor independently controls its job input, scheduling and output processing.

JES3 An MVS subsystem that receives jobs into the system, converts them to internal format, selects them for execution, processes their output, and purges them from the system. In complexes that have several loosely coupled processors so that the global processor exercises centralized control over the local processors and distributes jobs to them using a common job queue.

JESMSGLG DDNAME where JES and operator messages for the job are written.

JESYSMSG DDNAME where system messages for the job are written.

JOE Information that describes a unit of work for the JES output processor and represents that unit of work for queuing purposes.

JQE A control block containing a summary information from a Job Control Table (JCT) entry. JQEs move from queues as work moves through each stage of processing. JQEs are used instead of JCT entries for the scheduling of work.

LI The Lines panel allows the user to display information about JES2 lines and their associated transmitters and receivers.

LPAR (Logically Partitioned Mode) A capability provided by the Processor Resource/System Manager (PR/SM™) that allows a single processor to run multiple operating systems using separate sets of system resources, or logical partitions.

MAS (Multi-Access Spool) A multiple-processor complex that consists of two to more processors ate the same physical location, which share the same spool and checkpoint data sets.

MAS Multi-Access Spool panel (MAS) The Multi-Access Spool (MAS) panel simplifies the display and control of members in a JES2 MAS.

MVS (Multiple Virtual Storage) An IBM operating system that accesses multiple address spaces in virtual storage. It was the most commonly used operating system on the System/370™ and S/390® series. By design, programs written for MVS can still run on z/OS without modification.

NO Nodes panel (NO) The Nodes panel allows the user to display information about JES2 nodes.

O Output Queue panel (O) The Output Queue panel allows the user to display information about SYSOUT data sets for jobs, started tasks, and TSO users on any nonheld JES2 output queue.

OCOPY Copy an MVS data set member or z/OS UNIX file to another file or member. COPYcommand can be used to copy data between an MVS data set and the z/OS UNIX file system.

OD Output Descriptors panel (OD) The OD panel allows the user to display SYSOUT data sets before they are printed.
Columns can be overtyped only if you accessed the OD panel from the O or H panel, or from a JDS panel that was accessed from the O or H panel.

OEDIT Edit an z/OS UNIX file system file. OEDIT enables users to edit a file in the z/OS UNIX file system. This command uses the ISPF/PDF Edit facility.

port In TCP/IP, a separately addressable point to which an application can connect. For example, by default HTTP uses port 80 and Secure HTTP (HTTPS) uses port 443.

PR Printer panel (PR) The Printer panel allows the user to display information about JES2 printers printing jobs, started task, and TSO user output.

PS Processes panel (PS) The PS panel displays information about z/OS UNIX System Services processes.

PUN Punch panel (PUN) The PUN panel allows the user to display information about JES2 punches.

RACF (Resource Access Control Facility) IBM licensed program that provides control by identifying users to the system; verifying users of the system; authorizing access to protected resources; logging unauthorized attempts to enter the system; and logging assesses to protected resources.

RC A REXX special variable set to the return code from any executed host command or subcommand. It is also set to the return code when the conditions ERROR, FAILURE, and SYNTAX are trapped.

RDR Reader panel (RDR) The RDR panel allows the user to display information about JES2 readers.

RES Resource panel (RES) The RES panel allows users to display information about WLM resources in a scheduling environment, or in the sysplex.

RESULT A REXX special variable that is set by the RETURN instruction in a called routine. The RESULT variable is dropped if the called routine does not return a value.

REXX (Restructured Extended Executor) A general-purpose, interpreted high level programming language which was developed at IBM and was designed to be both easy to learn and easy to read. Designed originally by Mike Cowlishaw as a scripting programming language, it is particularly well suited for writing quickly small utility programs.

rexxhelp Rexxhelp provides information about using REXX with SDSF. It is available in SDSF's online help. The help includes links to descriptions of commands, action characters and overtypable columns. To display the online help on using REXX with SDSF, type rexxhelp on any command line in SDSF when using SDSF under ISPF.

rlogin UNIX software utility that allows users to log in another host through a network, communicating through TCP port 513. rlogin has serious security problems

RM Resource Monitor (RM) panel The Resource Monitor panel shows information about JES2 resources such as JOEs, JQEs and BERTs.

RMF (Resource Measurement Facility) A feature of z/OS that measures selected areas of system activity and presents the data collected in the format of printed reports, System Management Facility (SMF) records, or display reports.

scheduler In this book, a computer program that performs functions such as scheduling, initiation, and termination ofr jobs.

scheduling environment A list of resource names along with their required states. If an MVS image satisfies all of the requirements in the scheduling environment associated with a given unit of work, then that unit of work can be assigned to that MVS image. If any of the requirements are not satisfied, then that unit of work cannot be assigned to that MVS image.

SDSF (System Display and Search Facility) IBM licensed program that provides a menu-driven full-screen interface that is used to obtain detailed information about jobs and resources in a system.

SE Scheduling Environment panel (SE) The SE panel allows authorized users to display information about scheduling environments in the MAS or the sysplex.

SMF (System Management Facility) A component of z/OS operating system that collects and records a variety of system and job related information, including I/O, network activity, software usage, error conditions, processor utilization, and so forth. SMF forms the basis for nearly all the monitoring and automation utilities.

SO Spool Offload panel (SO) The Spool Offload panel allows authorized users to display information about JES2 spool offloaders and their associated transmitters and receivers.

socket A means of directing data to an application in a TCP/IP network using a unique identifier that is a combination of an IP address and a port number. It is the communication end-point unique to a system.

SP Spool Volumes panel (SP) The Spool Volumes panel lets authorized users to display and control JES2 spool volumes.

spool data set A data set written on an auxiliary storage device and managed by Job Entry Subsystem (JES).

SSH (Secure Shell) Set of standards and network protocol used to establish a secure channel between to computers on the Internet or local area network (LAN) connections.

ST The SDSF Status panel allows the user to display information about jobs, started tasks, and TSO users on the JES2 queues.

stem In REXX, that part of a compound symbol up to and including the first period. It contains just one period, which is the last character. It cannot start with a digit or a period. A reference to a stem can also be used to manipulate all variables sharing that stem.

sysplex A set of z/OS systems that communicate with each other through certain multisystem hardware components and software services.

tail In REXX, that part of a compound symbol that follows the stem. A tail can consist of constant symbols, simple symbols and periods.

TCP (Transmission Control Protocol) A communications protocol used in the Internet and in any network that follows the Internet Engineering Task Force (IETF) standards for internetwork protocol. TCP provides a reliable host-to-host protocol in packet-switched communication networks and in interconnected systems for such networks.

TELNET (Teletype Network) Network protocol used on the Internet or local area network (LAN) connections. The term also refers to the software that implements the client part of the protocol, and this is the meaning in this book. The TELNET command enables remote users to log on to a foreign host that supports TCP/IP using a telnet client. The z/OS UNIX telnet server is started for each user by the inetd listener program.

JESJCL DDNAME where JCL statements of the job are written.

MCS (Multiple Console Support) A feature of MVS that permits selective message routing to multiple consoles. MCS consoles are either output-only devices like printers or input/output devices like a 3279 display console.
To extend the number of consoles on MVS systems or to allow applications and programs to access MVS messages and send commands, an installation can use extended MCS consoles. The use of these consoles can help alleviate the constraint of the 99 MCS console limit. Moving to an extended MCS console base from a subsystem-allocatable console base will allow for easier expansion in a sysplex.
You can define a TSO/E user to operate an extended MCS console from a TSO/E terminal. The user issues the TSO/E CONSOLE command to activate the extended MCS console.
An installation can also write an application program to act as an extended MCS console. An authorized program issues the MVS authorized macro MCSOPER to activate and control the extended MCS console and uses other MVS macros and services to receive messages and send commands.

TSO (Time Sharing Option) Base element of z/OS operating system with which users can interactively work with the system. It fills the same purpose as the login sessions used by users on UNIX or Windows. It was originally introduced in the 1960s, time-sharing was considered then an “optional feature”, and hence TSO was offered as an optional feature of OS/360. It became a standard part of the system with the introduction of MVS in 1974.

UNIX System Services An element of z/OS that creates a UNIX environment that conforms to XPG4 UNIX 1995 specification and that provides two open-system interfaces on the z/OS operating system: an application programming interface (API) and an interactive shell interface.

UNIX A highly portable operating system, originally developed in the 1960s and 1970s by a team of AT&T employees at Bell Labs, that features multiprogramming in a multiuser environment. The UNIX operating system was originally developed for use on minicomputers, but was adapted for mainframes and microcomputers. The AIX operating system is the implementation of the UNIX operating system from IBM. The owner of the trademark UNIX is *The Open Group*, an industry standards consortium. Operating systems can only use the UNIX trademark if they have been certified to do so by *The Open Group*. UNIX-compatible operating systems that are not certified by *The Open Group* are typically referred to as “UNIX-like”. For instance, Linux® is a UNIX-like operating system.

Web server A software program that is capable of servicing Hypertext Transfer Protocol (HTTP) requests. It accepts HTTP requests from clients which are generally known as Web browsers, and serving them HTTP responses along with optional data contents.

WebSphere An IBM brand name that encompasses tools for developing e-business applications and middleware for running Web applications.

WLM (WorkLoad Manager) Component of z/OS that provides the ability to run multiple workloads at the same time within one z/OS image or across multiple images.

WLM enclave An enclave is an anchor for a transaction that can be spread across multiple dispatchable units in multiple address spaces. These multiple address spaces can even span across multiple systems in a parallel sysplex.

z/OS An operating system for the IBM eServer™ product line that uses 64-bit real storage. It is the successor to the mainframe operating system OS/390, combining MVS and UNIX System Services.

Index

X

Z

Implementing REXX Support in SDSF

(0.5” spine)
0.475”<->0.875”
250 <-> 459 pages

Implementing REXX Support in SDSF

Harness the power of SDSF with the versatility of REXX

Write powerful REXX code to manage your environmen

Access SDSF outside of your mainframe

The Restructured Extended Executor (REXX) language is a procedural language that allows you to write programs and algorithms in a clear and structural way. It is an interpreted and compiled language, and you do not have to compile a REXX command list before executing it.

With IBM z/OS V1.9, you can harness the versatility of REXX to interface and interact with the power of SDSF. A new function called REXX with SDSF is available that provides access to SDSF functions through the use of the REXX programming language. This REXX support provides a simple and powerful alternative to using SDSF batch.

This IBM Redbooks publication describes the new support and provides sample REXX execs that exploit the new function and that perform real-world tasks related to operations, systems programming, system administration, and automation. This book complements the SDSF documentation, which is primarily reference information.

The audience for this book includes operations support, system programmers, automation support, and anyone with a desire to access SDSF using a REXX interface.

INTERNATIONAL TECHNICAL SUPPORT ORGANIZATION

BUILDING TECHNICAL INFORMATION BASED ON PRACTICAL EXPERIENCE

IBM Redbooks are developed by the IBM International Technical Support Organization. Experts from IBM, Customers and Partners from around the world create timely technical information based on realistic scenarios. Specific recommendations are provided to help you implement IT solutions more effectively in your environment.

For more information:
ibm.com/redbooks

SG24-7419-00 ISBN 073848914X