A Broadcom Company

RLX for z/OS

REXX Language Xtensions

User Guide and Reference

Version 9.1

June 2010

Publication RLX-101-7

Relational Architects Intl

This Guide: (RAI Publication RLX-101-7)

This document applies to RLX for z/OS Version 9 Release 1 (June 2010), and all subsequent releases, unless otherwise indicated in new editions or technical newsletters. Specifications contained herein are subject to change and will be reported in subsequent revisions or editions.

Purchase Orders for publications should be addressed to:

Documentation Coordinator
Relational Architects Intl.
Riverview Historic Plaza
33 Newark Street
Hoboken NJ 07030 USA

Tel: 201 420-0400
Fax: 201 420-4080
Email: Sales@relarc.com

Reader comments regarding the product, its documentation, and suggested improvements are welcome. A reader comment form for this purpose is provided at the back of this publication.

Ed 11F10

Preface

This manual is intended for application developers, database administrators, technical professionals and users working with REXX in the z/OS environment. The reader is assumed to be familiar with REXX along with the programming and operations standards in use at the installation site.

This RLX for z/OS User Guide and Reference (publication RLX-101) is a composite manual comprised of the following product documentation:

- RLX Product Family: Guide and Reference RPF-001
- RLX/VSAM Reference RRV-001
- RLX/SDK Reference.. RDK-001
- AcceleREXX User Guide RCX-001

Related Publications

The RLX for DB2 User Guide and Reference (publication RLX-102) is a composite manual comprised of the following product documentation:

- RLX Installation and Customization Guide RLX-001
- RLX Getting Started Guide RLX-002
- RLX/Translate Guide and Reference RCT-001
- RLX for DB2 User Guide and Reference RLX-102

For information related to IBM's REXX implementation for TSO/E refer to the following IBM publications:

- TSO/E REXX User's Guide
- TSO/E REXX Reference

For information related to DB2 / MVS refer to the following IBM publications:

- DB2 Administration Guide
- DB2 Application Programming & SQL Guide
- DB2 SQL Reference
- DB2 Messages and Codes
- DB2 Command Reference
- DB2 Utility Guide and Reference

For information related to ISPF / PDF see the following IBM publications:

- ISPF Dialog Developer's Guide and Reference
- ISPF Edit and Edit Macros
- ISPF User's Guide

For information related to the MVS/DFP implementation of VSAM please refer to the following IBM publications:

- DFSMS Macro Instructions for Data Sets
- DFSMS Access Method Services
- DFSMS Using Data Sets
- DFSMS Managing Catalogs

Notational Conventions

The following notational conventions are used in this Guide:

➢ Uppercase commands and their operands should be entered as shown but need not be entered in uppercase.

➢ Operands shown in lower case are variable; a value should be substituted for them.

➢ Operands shown in brackets [ ] are optional, with a choice indicated by a vertical bar |. One or none may be chosen; the defaults are underscored.

➢ Operands shown in braces { } are alternatives; one must be chosen.

➢ An ellipsis (...) indicates that the parameter shown may be repeated to specify additional items of the same category.

➢ In the Messages section, contents within arrowheads < > will be replaced at run-time with the actual value.

RLX

REXX Language Xtensions

Product Family Reference

Version 9.1

June 2010

Publication RPF-001-7

Relational Architects Intl

This Guide: (RAI Publication RPF-001-7)

This document applies to Version 9 Release 1 (June 2010) of both RLX for DB2/z and RLX for z/OS, and all subsequent releases, unless otherwise indicated in new editions or technical newsletters. Specifications contained herein are subject to change and will be reported in subsequent revisions or editions.

Purchase Orders for publications should be addressed to:

Documentation Coordinator
Relational Architects Intl.
Riverview Historic Plaza
33 Newark Street
Hoboken NJ 07030 USA

Tel: 201 420-0400
Fax: 201 420-4080
Email: Sales@relarc.com

Reader comments regarding the product, its documentation, and suggested improvements are welcome. A reader comment form for this purpose is provided at the back of this publication.

Ed 13F10

Preface

Related Publications

This Introductory Section provides descriptive material applicable to the set of components which comprise the RLX for z/OS product. This material is augmented by the more extensive RLX product library which at present includes the following publications:

- RLX Installation and Customization Guide RLX-001
- RLX Getting Started Guide RLX-002
- RLX/Translate Guide and Reference RCT-001
- RLX for DB2 User Guide and Reference RLX-102

For information related to IBM's REXX implementation for TSO/E refer to the following IBM publications:

- TSO/E REXX User's Guide
- TSO/E REXX Reference

For information related to DB2 / MVS refer to the following IBM publications:

- DB2 Administration Guide
- DB2 Application Programming & SQL Guide
- DB2 SQL Reference
- DB2 Messages and Codes
- DB2 Command Reference
- DB2 Utility Guide and Reference

For information related to ISPF / PDF see the following IBM publications:

- ISPF Dialog Developer's Guide and Reference
- ISPF Edit and Edit Macros
- ISPF User's Guide

For information related to the MVS/DFP implementation of VSAM please refer to the following IBM publications:

- DFSMS Macro Instructions for Data Sets
- DFSMS Access Method Services
- DFSMS Using Data Sets
- DFSMS Managing Catalogs

Notational Conventions

The following notational conventions are used in this Guide:

- Uppercase commands and their operands should be entered as shown but need not be entered in uppercase.
- Operands shown in lower case are variable; a value should be substituted for them.
- Operands shown in brackets [ ] are optional, with a choice indicated by a vertical bar |. One or none may be chosen; the defaults are underscored.
- Operands shown in braces { } are alternatives; one must be chosen.
- An ellipsis (...) indicates that the parameter shown may be repeated to specify additional items of the same category.
- In the Messages section, contents within arrowheads < > will be replaced at run-time with the actual value.

RLX Product Family Reference

Table of Contents

Chapter 1

What's New in RLX for z/OS

Summary of Changes to RLX Version 9.1

Version 9 Release 1 of RLX for z/OS supports all functions available in prior releases of the product plus all fixes and enhacements provided in interim software releases.

- The RLX Product Family includes a new component named ***RDX*** which supports SQL and pureXML at the latest levels of DB2 for z/OS. RDX also provides a set of REXX extension services through its REXX Function Package.

- RDX is integrated with RLX, included in the RLX distribution libraries and installed together with RLX. In addition, RDX is available as a standalone product with its own distribution libraries and manual (RDX Guide and Reference: Publication RDX-001).

- RLX Version 9 now supports both ADDRESS RLX and ADDRESS RDX as host command environments. In addition, both RLX and RDX make use of same DB2 application plans. However, RDX establishes its own DB2 thread, independent from RLX, when both RLX and RDX are active in the same REXX exec. Thus, two discrete COMMIT / Unit-of-Recovery scopes will be concurrently active – a practice RAI strongly discourages.

- RAI recommends that RLX and RDX code be segregated in separate execs that use either ADDRESS RLX or ADDRESS RDX exclusively. ADDRESS RLX allows you to continue using your existing RLX applications without modification. However, when new DB2 facilities (such as pureXML support) are required, RAI recommends you develop a new REXX exec and use ADDRESS RDX.

Summary of Changes to RLX Version 8.1

Version 8 Release 1 of RLX supports all functions available in prior releases of the product. Certain revisions have been made to the documentation.

New in Version 7.1

RLX for MVS V7.1 contains all program fixes and enhacements distributed in previous RLX for MVS product, release and modification levels.

New in Version 6.2

- Introduced the RAI Server address space to support RLX facilities that require APF authorization. The server address space is an alternative to the RAI SVC.
- Full year 2000 compliance for all members of the RLX product family

New in Version 5.2.2

- The V5R2M2 release of RLX contains many product quality improvements and enhancements in the RLX/SDK product.
- New Date-Time conversion functions: SDKSTCK, STCKCONV, and CONVTOD will help you convert date-time values in STCK format to various Time and Date formats. These tools can be very useful in your year 2000 conversion efforts.
- The SDKDSNU function is enhanced and now can be invoked from the TSO/ISPF foreground to execute DB2 utilities (e.g. RUNSTATS, REORG, etc.) This function requires an RAI authorization mechanism.
- RLX now provides an interface to the System Authorization Facility which allows you to manage access to RLX authorized facilities via RACF or other Resource Access Control products.
- The new REXX function SDKSAF allows you to control access to your own sensitive REXX applications.
- The new REXX function VSMUTIL helps you to identify virtual storage utilization by reporting allocated memory both above and below the 16MB line.
- The new SDKNTS function provides native REXX access to MVS Name-Token services. With name/token services you can establish persistent anchor points for your control blocks on Task and System levels

- The new REXX function SDKJLKUP allows you to search the Job Pack Queue and System Nucleus by load module address or name.

- The new REXX function TPG implements the TSO TPG macro which allows you to read a terminal buffer from a TSO supported terminal. The TPG function is useful for determining terminal charcteristics by issuing a QUERY command.

- The new REXX function SDKDIV is a full implementation of MVS Data-In-Virtual facilities. It allows you to use VSAM Linear Datasets to develop very fast I/O interfaces natively from REXX.

New in Version 5.1.1

- SDKSORT now supports up to 5 sort fields. Performance is improved 40x (on a 26000 element array) by implementing a new mergesort algorithm.

- The REXX functions D2F and F2D allow you to convert between decimal and Internal Floating point numbers (both single and double precision. See the RXDHEXT member of RLXPLIB for description of the syntax of these functions.

- The RLX region size must be at least 2.2M -- an increase of 200K in comparison with RLX V4.2.

New in Version 5.1

- You can now invoke REXX execs from any compiled or assembled program through the callable RLX/SDK service RFPH2R.

- The RPFRFI routine lets you dynamically define RAI REXX functions

- A new facility of the RLX/SDK allows you to invoke a REXX exec from a COBOL program. See the following source members:

  COB2RTST - A sample COBOL source program (in the RLXCNTL library)

  COB2REXE - A REXX exec invoked from the sample COBOL program (in the RLXEXEC library)

  COB2RJCL - JCL to run the sample COBOL program and REXX exec (in the RLXCNTL library)

New in Version 4.1

Packaging

The product called AcceleREXX prior to this release is now called RLX for MVS -- a set of components that includes a REXX VSAM interface, Software Development Kit and REXX compiler. AcceleREXX now refers to the REXX compiler only. The RLX components for MVS are now distributed on the same cartridge as the other components of the **REXX Language Xtentions** family of products. This lets you install and trial all RLX products using a single, simple installation procedure.

REXX Compiler

The AcceleREXX compiler now produces a sorted cross reference of symbols and labels that improves program readability, helps with debugging and increases programmer productivity. The sorted cross reference is especially useful for large REXX applications comprised of many subroutines and variables.

The AcceleREXX preprocessor is also enhanced with preprocessor directives that are specified as REXX comments. These directives let you include frequently used sections of REXX code, exclude sections of code and format your REXX program to clarify its structure and functionality.

The following AcceleREXX catalogued procedures are introduced in this release:

> RCXP invoke the AcceleREXX precompiler
> RCXC invoke the AcceleREXX compiler
> RCXLKED link edit compiled REXX programs

The AcceleREXX dialogs have been enhanced to include discrete and merge compile options.

> **Compile:** This dialog lets you select REXX programs from up to 4 libraries and build a jobstream to compile the selected execs as discrete load modules.

> **Merge Compile:** This dialog lets you select REXX programs from up to 4 libraries, merge them into a single composite source module and build a jobstream that invokes the precompiler, the compiler and the linkage editor.

RLX/SDK

The RLX Software Development Kit (RLX/SDK) is extended with new functions while many existing functions have been improved. SDK functions are grouped in the following categories:

> Memory Management Functions
(GETMAIN, FREEMAIN, XSTORAGE, DUMP, LOAD, DELETE)

> Resource Control Functions (ENQ, DEQ, GQSCAN, ENQSUM, ENQUEUE)

> Global Variable Services (GVPUT, GVGET, GVDEL)

> Operator console and log interaction (WTL, WTO, WTOR, DOM, MVSCMD)

> REXX host command definition, SDK functions
(SUBCOM, SDKINIT, SDKTERM)

> TSO I/O (TGET, TPUT)

3270 Data stream functions: ($CMD, $SBA, $SF, $RA, $ATTR, etc)

> Miscellaneous REXX extensions
(SDKREAD, SDKBRIF, SDKEDIT, SDKSCAN, SDKSORT,
SDKWSORT, SDKRSVC, P2D, D2P, A2E, E2A, SVC)

> Multi-tasking and Interprocess Communication
(WAIT, POST, XPOST, ATTACH, DETACH, IPCSEND, IPCRECE)

> The **SDK dialog** provides an on-line demonstration of RLX/SDK functions and a tutorial on function usage.

> Manage REXX Function Packages (RFP Directory)
This dialog lets you create and manage REXX function packages and their corresponding directories

> VSAM File allocation
This dialog is an extension of the ISPF 3.2 option which lets you allocate new VSAM datasets or show the attributes of an existing VSAM cluster. This dialog helps you generate IDCAMS control statements to delete and/or define VSAM clusters, alternate indices and paths. The dialog also serves to tune and monitor VSAM performance.

New in Version 2.1 of RLX for MVS (formerly AcceleREXX)

RLX/VSAM

RLX/VSAM provides a simple and intuitive VSAM interface that's native to REXX and supports all read, write and delete operations on VSAM ESDS, KSDS and RRDS datasets -- on a record and control interval basis. RLX/VSAM supports entry sequenced data access as well as direct access by key value, relative record number or relative byte address. RLX/VSAM includes a full screen facility with which to display and process VSAM data.

RLX/SDK

The RLX/SDK bundles an evolving set of powerful functions to boost developer productivity and augment the functionality of your applications. The productivity pack includes:

A robust suite of MVS supervisor services, native to REXX, that include:

> memory management (GETMAIN, FREEMAIN)

> process synchronization
(WAIT, POST, ENQUEUE, DEQUEUE, STIMER)

> Security interface to RACF, ACF2 and/or Top Secret
(SAF and RACROUTE)

> Partitioned dataset support that's more robust and efficient than EXECIO. The SDKREAD function lets you read multiple PDS members and directory entries without repeatedly freeing and reallocating the file.

> Global Variable Services let you develop more complex and sophisticated applications in REXX than would otherwise be possible. Global variables allow REXX routines to return complex structures and arrays to their callers. This added flexibility makes REXX the equal of compiled languages for developing generic, reusable routines.

The RLX/SDK also includes:

> Built-in functions that handle data conversions between packed decimal and character (C2P and P2C) and between ASCII and EBCDIC (A2E and E2A).

> Functions that sort and display REXX stemmed variables.
(SDKSORT and SDKBRIF)

> The SDKSCAN function that parses a string into its constituent tokens using delimiter characters you supply.

> A SUBCM function lets you define new Host Command Environments dynamically, without the need for assembly language programming.

> Support for developing host command environment replaceable routines -- in REXX!

New in Version 1.1 of AcceleREXX

> The restriction concerning invocation of recursive EXECs is eliminated. AcceleREXX permits ***any*** EXEC to be compiled and executed;

> Compiled EXECs which make use of ISPF dialog services can be run without the need for the RA front-end EXEC. ISPF/REXX EXECs can be invoked directly as commands, subroutines and/or functions;

> ***NOTE:*** *In the ISPF environment, the RLXEXEC dataset should be preallocated to either SYSEXEC, SYSPROC or an ALTLIB dataset. This latter ALTLIB option applies only to OS/390 and MVS/ESA environments.*

> The AcceleREXX compile dialog has been enhanced to support the selection of multiple EXECs for compilation and link edit in a single pass. Moreover, you can select previously compiled EXECs for inclusion in the link edited composite load module.

> The compiler's `INVOKED` parameter is obsolete. AcceleREXX now dynamically and implicitly determines at execution time whether an EXEC has been invoked as a command, subroutine or function;

> The compiler's `RLX` parameter has been removed and migrated to the RLX/Compile product;

> The compiler's `CONDENSE` option can now be used without restriction;

> The new `SPECIAL` compile option lets you specify whether the compiled exec will issue ISPF dialog service requests or execute in the NetView environment. This option signals AcceleREXX to conduct the special initialization processing required by these two environments;

> New AcceleREXX Installation Verification Procedure EXECs include:

RCXTISPF This new sample EXEC demonstrates AcceleREXX support for REXX EXECs which make use of ISPF dialog services.

RCXTED This compiled REXX PDF/EDIT macro is invoked within EDIT by entering the !RCXTED command on the Edit command line.

NOTE: *Source code for these sample EXECs are present in the RLXEXEC distribution library.*

RLX/Translate provides an automated means to translate your TSO CLISTs into REXX procedures. Once translated, all your command procedures can be maintained as REXX execs which can be further extended and compiled with the rest of the RLX product family.

Chapter 2

Concepts and Facilities

2.1 What is RLX?

RLX designates a family of ***REXX L***anguage e***X***tensions that endow REXX with additional capabilities. Together they make REXX a viable replacement for compiled host languages so you can `Do it all in REXX'. RLX is designed to let you tackle the kinds of sophisticated applications that previously had to be coded in Assembler, PL/1 or C. Its features include:

- native support for SQL (RLX/SQL and RLX/TSO)
- native support for SQL and pureXML (RLX/RDX)
- native access to CAF and RRSAF services (RLX/CAF)
- native access toDB2 commands and IFI requests (RLX/CAF)
- a native VSAM interface (RLX/VSAM)
- a REXX Software Development Kit that provides developers with numerous functions such as native access to MVS supervisor services and interprocess communications (RLX/SDK)
- special composite objects for DB2/ISPF processing (RLX/ISPF)
- REXX compiler (AcceleREXX)
- SQL compiler (RLX/Compile)
- CLIST to REXX translator (RLX/Translate)

RLX is designed to run everywhere REXX does. Supported z/OS environments include TSO, ISPF, batch TSO, batch ISPF, native batch, DB2 Stored Procedure address spaces, ISPF panels, SDSF, z/OS Health Checker, NetView and System REXX. RLX ***exploits*** TSO, ISPF and NetView facilities when they are available but ***does not require*** them to operate.

RLX provides an integrated development environment for REXX which lets you develop applications very quickly. Later you can deploy `industrial strength' solutions that exhibit the security and performance characteristics of applications written in compiled languages.

2.2 The RLX Product Family

The RLX Product Family includes the following REXX Language Extensions:

RLX components for z/OS

RLX/VSAM provides a simple and intuitive VSAM interface that's native to REXX. RLX/VSAM supports all read, write and delete operations on VSAM ESDS, KSDS and RRDS files -- on a record and control interval basis. It also supports control interval processing of VSAM linear datasets. RLX/VSAM is modeled after the DFP macro interface so programmers familiar with VSAM coding conventions can make use of RLX/VSAM almost immediately.

The RLX/SDK provides an evolving Software Development Kit (***SDK)*** that affords native access to MVS system services, as well as facilities for partitioned dataset I/O, resource control, parsing, tokenizing, sorting and many other functions.

The ***AcceleREXX*** compiler can dramatically improve the performance of your REXX EXECs while protecting them from modification. AcceleREXX provides an ISPF dialog framework to compile and link multiple EXECs together and even combine them into REXX function packages.

RLX/Translate provides an automated means to translate your TSO CLISTs into REXX procedures. Once translated, all your command procedures can be maintained as REXX execs which can be further extended and compiled with the rest of the RLX product family.

RLX components for DB2

RLX/RDX **R**EXX **D**B2 e**X**tensions (**RDX**) is the newest RLX component. It lets you code SQL and SQL/XML statements natively within REXX execs. RDX supports both embedded SQL syntax used in compiled host languages like COBOL as well as the dynamic SQL syntax used by DSNREXX. REXX execs that combine SQL statements, XPath and XQuery expressions, DB2 commands, IFI requests and RRSAF/CAF services can be developed rapidly and tested immediately. Without compile and link edit steps. Without preprocess and bind delays. You can edit and test RDX execs directly within ISPF Edit and get instant feedback.

RLX/SQL provides embedded SQL support for REXX that is as robust and flexible as that for COBOL. RLX EXECs can be developed rapidly and tested immediately ***without*** the preprocess, compile, linkedit and bind steps normally required by DB2 applications written in compiled languages. RLX/SQL includes a DECLARE REXXSTEM service which lets you copy SQL result tables directly into REXX stemmed arrays.

RLX/TSO provides a REXX SQL implementation that, in REXX parlance, is integrated into TSO/E and exploits available ISPF facilities. For example, when errors occur RLX/TSO uses ISPF services to display full screen, scrollable diagnostic panels.

Both RLX/SQL and RLX/TSO include a DB2 attachment facility (***RLX/CAF)*** which allows user-written applications to create, manage and switch among multiple, concurrently active DB2 plan threads. RLX/CAF enables you to connect to multiple DB2 subsystems simultaneously and lets you maintain multiple threads for each subsystem.

RLX/ISPF provides a pair of powerful composite objects that address common processing requirements in the DB2/ISPF environment. The RLX DECLARE ISPFTABLE service provides the means to load SQL query results directly into ISPF tables while the RLX TBDISPL service lets you display and process those results on scrollable ISPF table display panels.

RLX/Net extends REXX SQL support to NetView and endows dialogs running in that environment with the look and feel of ISPF.

RLX/Compile extracts and compiles the SQL statements embedded in your REXX EXECs to produce static DB2 application plans and their associated load modules. RLX/Compile interoperates with both the AcceleREXX compiler and the REXX/370 Compiler from IBM.

2.3 The RLX Product Library and Examples

The RLX product library at present includes the following publications:

- RLX Installation and Customization Guide RLX-001
- RLX Getting Started Guide RLX-002
- RLX/Translate Guide and Reference RCT-001

The RLX for z/OS User Guide and Reference (publication RLX-101) is a composite manual comprised of the following product documentation:

- RLX Product Family Reference RPF-001
- RLX/VSAM Reference RRV001
- RLX/SDK Reference RDK-001
- AcceleREXX User Guide RCX-001

The RLX for DB2 User Guide and Reference (publication RLX-102) is a composite manual comprised of the following product documentation:

• RLX/SQL and RLX/TSO Reference	RLX-003
• RLX/CAF and RRSAF Reference	CAF-001
• RLX/IFI Reference	RIF-001
• RLX/ISPF Reference	RTS-001
• RLX/Net Reference	RNV-001
• RLX/Compile Reference	RCS-001

The RLX product library is designed to provide an appropriate level of detail for audiences whose requirements may differ.

In addition, the RLXEXEC library contains many examples of RLX usage. The various components of the RLX product family are further documented as follows:

- REXX **VSAM support** is described in the RLX/VSAM section of the RLX for z/OS Guide and Reference. Examples of REXX VSAM usage appear in members of the RLXEXEC library whose names start with the letters 'RXDV'.

- The REXX **Software Development Kit** (SDK) is described in the RLX/SDK section of the RLX for z/OS Guide and Reference. SDK functions are exercized by members of the RLXEXEC library whose names start with 'RXD'.

- The **AcceleREXX compiler** is described in the AcceleREXX section of the RLX for z/OS Guide and Reference and its examples are prefixed by the letters 'RCXT'.

- **RLX/Translate** is described in the RLX/Translate Guide and Reference and its examples are prefixed by the letters 'RCT'.

- The **REXX SQL** interface is described in the RLX for DB2 Guide Reference. Members of the RLX EXEC library whose names start with RLXE and RMV provide examples of RLX/TSO and RLX/SQL usage. Additional examples of support for **RXSQL** syntax (IBM's REXX SQL implementation for SQL/DS) appear in members whose names start with RXS.

- The Call Attach Facility services provided by **RLX/CAF** are described in the RLX/CAF section of the RLX for DB2 Guide and Reference. Member RMVDCAF provides a complete and annotated example of RLX/CAF usage.

- The DB2/ISPF composite functions that comprise **RLX/ISPF** are described in the RLX/ISPF section of the RLX for DB2 Guide and Reference. Examples of RLX/ISPF usage appear in those members of the RLX EXEC library whose names start with RTD.

- **RLX/Net** is described in the RLX/Net section of the RLX for DB2 Guide and Reference. Examples of RLX/Net usage appear in those members of the RLX EXEC library whose names are prefixed by the letters RNV.

- **RLX/Compile** is described in the RLX/Compile section of the RLX for DB2 Guide and Reference and its examples are prefixed by the letters 'RCSD'.

2.4 Rapid Application Development for Mission Critical Applications

RLX provides the best of both worlds. Development is quick and easy with interpretive REXX and dynamic SQL. Once you are satisfied with an application's functionality and correctness, RLX/Compile lets you deploy compiled 'industrial strength' applications into production. These static SQL applications exhibit qualitatively different behavior from their dynamic SQL counterparts. They not only run faster, they are more secure. Administrators can grant execute authority for compiled plans without granting users access to the DB2 tables and views referenced by a plan. This prevents unauthorized access to sensitive data through dynamic SQL facilities like QMF and SPUFI. **Bottom Line: RLX provides rapid application development for mission critical applications.**

RLX/Compile includes a complete application development system to compile, link edit and bind 'real world' applications comprised of multiple modules. This is accomplished within an ISPF dialog framework which eliminates the tedious, error prone aspects of program preparation.

2.5 RLX Support for Embedded SQL

RLX offers comprehensive SQL support (at the level of the most current DB2 for OS/390 release) while strictly adhering to standard syntax for embedded SQL. RLX fully supports all SQL data definition, manipulation, and control statements which may be embedded in programs, subject to the user's Authorization ID. RLX users can

- Issue queries (SELECT, DECLARE, OPEN, FETCH, CLOSE)
  Applications can process individual rows as well as result tables.
- Manipulate data (INSERT, UPDATE, DELETE)
- Manage data consistency and referential integrity (COMMIT and ROLLBACK)
- Invoke DB2 stored procedures and pass them parameters (CALL)
- Process query result(s) returned by DB2 stored procedures (ASSOCIATE LOCATORS and ALLLOCATE CURSOR)
- Define and revise DB2 objects (CREATE, ALTER, DROP)
- Administer authorizations and privileges (GRANT and REVOKE)
- Process dynamic SQL (PREPARE, DESCRIBE, EXECUTE)
- Connect to remote servers (CONNECT and RELEASE)
- Set and obtain values of SQL special registers (various forms of the SET statement)

2.6 RLX/SQL Features

- Full compatibility with DB2 security and authorization mechanisms.
- Full compatibility with ISPF Dialog Management Services.
- Automatic conversion between DB2 internal and REXX external datatypes.
- Feedback from the variables comprising the SQL Communications Area are returned to the RLX EXEC after each SQL statement executes.
- Interactive facilities to pinpoint errors and speed their correction through extensive diagnostics, error reporting and context sensitive help.
- RLX provides profile facilities to customize the product according to the preferences and requirements of an individual, group, department, or an entire installation.
- Extensive features for enhanced reliability, availability and serviceability.

Chapter 3

Setting Up the Environment to use RLX

This chapter describes the dataset allocations required to run RLX in the various environments it supports -- which include MVS batch as well as batch and foreground versions of TSO and ISPF. (The unique requirements of the NetView environment are described in Chapter 7 of the RLX Installation Guide).

RLX may *already* be installed and deployed for general use in which case this material serves mainly as reference. Ask your RLX product administrator whether any further allocations are necessary to use RLX in the TSO/ISPF foreground. Section 3.4 provides sample JCL that illustrates how to run RLX execs natively in batch and Section 3.7 describes how to run RLX in the TSO/ISPF background.

Section 3.2 describes the basic dataset allocations required for all environments. Section 3.3 discusses the additional allocations required to exploit the ISPF environment. Section 3.4 describes and illustrates how the RLX libraries can be ***preallocated*** before the ISPF environment is entered while Section 3.5 describes the use of the ISPF LIBDEF service and the ALTLIB facility of TSO/E to ***dynamically*** allocate and release the RLX libraries on an application basis.

3.1 RLX Libraries

Figure 3.1 lists the RLX libraries and the DDnames to which they are allocated. The names of the RLX libraries in the figure are those restored from the distribution tape. They may be different in your environment. Check with your RLX product administrator to determine if these are the actual dataset names in use at your installation.

All RLX libraries can be shared among multiple users and multiple DB2 subsystems. These include the RLX load module library, the RLXEXEC and RLXCLIST libraries and the ISPF panel, message and skeleton libraries.

Throughout this publication, Tvrm refers to the target libraries created at your site during installation. Thus, all references to Tvrm in this manual should be read as referring to the actual node name of the target datasets created at your site -- e.g. V9R9M0.

```
File Name   Dataset               Contents
---------   ------------------    ----------------------------------------
RLXLOAD     RLX.Tvrm.RLXLOAD    RLX load module library
STEPLIB     RLX.Tvrm.RLXLOAD    RLX load module library
SYSPROC     RLX.Tvrm.RLXCLIST   RLX CLIST library
            RLX.Tvrm.RLXEXEC    RLX EXEC library (or allocate to SYSEXEC)
SYSEXEC     RLX.Tvrm.RLXEXEC    RLX EXEC library
ISPLLIB     RLX.Tvrm.RLXLOAD    RLX load module library
ISPMLIB     RLX.Tvrm.RLXMLIB    RLX ISPF message library
ISPPLIB     RLX.Tvrm.RLXPLIB    RLX ISPF panel library
ISPSLIB     RLX.Tvrm.RLXSLIB    RLX file tailoring skeleton library
RLXTLIB     RLX.Tvrm.RLXTLIB    RLX permanent ISPF tables
```

Figure 3.1

3.2 Basic file allocations for all environments (batch, TSO, TSO/ISPF and NetView)

Two RLX library allocations -- the RLXLOAD and RLXMLIB datasets -- are required for all environments in which RLX procedures will run. The next section describes the additional library allocations that exploit the TSO/ISPF environment.

Ask your RLX product administrator whether the RLXLOAD and RLXMLIB datasets have been ***preallocated*** during installation. If so, you need not allocate them now. Otherwise, the RLX libraries can be allocated through a TSO LOGON procedure or an EXEC or CLIST used to initialize the environment.

The RLX supplied versions of the REXX parameter modules IRXISPRM, IRXPARMS and IRXTSPRM must be accessible ***ahead*** of any other versions of these modules. However, this requirement *does not apply*

The load modules of the RLXLOAD library should be accessible through the standard MVS search order. In addition, unless you the RLXS command processor front end to invoke an RLX exec or CLIST within ISPF (as described in Chapter 3.2), ***you must ensure that the REXX parameter modules (IRXPARMS, IRXTSPRM and IRXISPRM) supplied with RLX are loaded from the RLXLOAD dataset and not from some other library.*** Otherwise, your RLX procedures will not work properly.

> ***NOTE:*** *One other option is to customize the REXX parameter modules to include definitions for the Host Command Environments and REXX function packages employed by RLX. Appendix D of the RLX Installation Guide describes the customization procedure for REXX parameter modules in detail.*

MVS conducts its search for the requested module in the following sequence:

1. The job pack area queue
3. A task library (such as ISPLLIB) if one is allocated
3. A private STEPLIB or JOBLIB (If defined in the TSO LOGON procedure)
4. Link Pack Area
5. The Link list concatenation

The simplest and most reliable method is to allocate the RLXLOAD dataset to the ISPLLIB DDname. This file defines the task library from which load modules are fetched during an ISPF session. Alternatively, you can allocate a private STEPLIB or JOBLIB in your TSO LOGON procedure if you have the authority. ***Be cautioned*** that any dynamic STEPLIB facility you have available to you may not work properly for purposes of setting up the environment for RLX. This is because the appropriate REXX parameter modules are ***already loaded -- before*** you issue a command to alter the STEPLIB concatenation.

The RLX message library should be allocated to the DDname ISPMLIB -- in both ISPF and non-ISPF environments. RLX messages are maintained in an ISPF format but can be processed even if ISPF is not active. RLX has its own facilities to access the message library and conduct variable substitution to produce runtime message text. This allows the same messages to be utilized in all the environments that RLX supports such as NetView, MVS batch, TSO and TSO/ISPF.

3.3 Additional library allocations to exploit the TSO/ISPF environment

In addition to the RLXMLIB dataset (the RLX message library), the RLX panel library (RLXPLIB dataset) should be allocated in ISPF environments so RLX can display diagnostic messages on scrollable ISPF panels.

```
ISPMLIB    RLX.Tvrm.RLXMLIB    RLX ISPF message library
ISPPLIB    RLX.Tvrm.RLXPLIB    RLX ISPF panel library
ISPTLIB    RLX.Tvrm.RLXTLIB    RLX permanent ISPF tables
```

The ISPF skeleton library (DDname ISPSLIB) and table library (DDnames ISPTLIB and ISPTABL) supplied with RLX are more typically of interest to users of RLX/Compile and/or AcceleREXX (both of which are documented in their own sections) or to product installers and administrators who will run the RLX installation, maintenance and administrative dialogs.

3.4 Preallocating the RLX Libraries

This section describes how to preallocate the RLX libraries for the ISPF environment. Perhaps the easiest and most reliable way to allocate the RLX datasets is with a REXX exec or TSO CLIST which can be executed from the TSO READY prompt, ***before*** ISPF is entered. This option lets you concatenate your own run-time libraries as well.

The RLXEXEC library contains a member named RLXAPA that provides an illustrative -- but ***not executable*** -- procedure that issues TSO ALLOCATE commands to setup the TSO/ISPF environment for use with RLX. The RLXAPA exec appears below in Figure 3.3. In the figure, the datasets named `base.xxxx.libraries` refer to the set of TSO/ISPF libraries which are always allocated in your environment. The numbers in the figure's right margin correspond to the numbered, annotating paragraphs that follow.

> ***NOTE:*** *The RLX Installation Guide describes how to pre-allocate the RLX libraries via a variety of methods.*

```
Exec RLXAPA

/* REXX */
Address TSO

'free dd(sysproc sysexec ispllib ispmlib ispplib ispslib)'

"alloc dd(ispllib) da('rlx.Tvrm.rlxload'",                (1)
                     "'base.load.libraries'",               (2)
                      ") shr reuse"

"alloc dd(ispmlib) da('rlx.Tvrm.rlxmlib'",                (3)
                      "'base.message.libraries'",
                      ") shr reuse"

"alloc dd(ispplib) da('rlx.Tvrm.rlxplib'",                (4)
                      "base.panel.libraries'",
                      ") shr reuse"

"alloc dd(ispslib) da('rlx.Tvrm.rlxslib'",                (5)
                      "base.skeleton.libraries'",
                      ") shr reuse"

"executil searchdd(yes)"                                  (6)

"alloc dd(sysexec) da('rlx.Tvrm.rlxexec'",                (7)
                    "'base.exec.libraries'",
                      ") shr reuse"

"alloc dd(sysproc) da('rlx.Tvrm.rlxclist'",               (8)
                    "'base.clist.libraries'",
                      ") shr reuse"
```

Figure 3.2 File Allocations for TSO/ISPF

(1) The DDname ISPLLIB is used as a task library when fetching load modules. It is searched prior to the system link libraries and the link pack area. The RLXLOAD library is concatenated ***ahead*** of other dialog datasets allocated to DDname ISPLLIB to ensure the RLX versions of the REXX parameter modules are loaded when referenced. In addition, be sure the system libraries that contain DB2 executable code are either in the link list or are also pre-allocated.

(2) The dataset referred to as `'base.load.libraries'` refers to the set of load libraries that are typically allocated in your environment to the DDname ISPLLIB.

(3) The RLX message library is allocated to the file ISPMLIB

(4) The RLX panels are allocated to the file ISPPLIB

(5) The RLX skeleton library is allocated to the file ISPSLIB

(6) In the TSO environment, the EXECUTIL SEARCHDD command governs the search for execs that are implicitly invoked. Specifying SEARCHDD(YES) directs the system to first search the datasets allocated to the SYSEXEC DDname. If not found, the search continues with the libraries allocated to SYSPROC. If you make use of both execs and CLISTs, this technique will shorten the search for REXX execs.

(7) Exec libraries must be allocated before their members can be invoked implicitly. The option illustrated here is to allocate them to the DDname SYSEXEC if the EXECUTIL SEARCHDD(YES) command is specified as in note (6). Alternatively in the TSO environment, exec libraries can be allocated along with TSO CLISTs to the SYSPROC file described in note (8). Lastly, TSO users in OS/390 and MVS/ESA environments can use the ALTLIB command (described in Section 3.6 of this manual and in the TSO/E Command Reference) to control the search for execs and CLISTs.

(8) CLIST libraries should be allocated to the DDname SYSPROC before CLISTs can be invoked implicitly. An alternative in the OS/390 and MVS/ESA environments is to use the ALTLIB command (described in Section 3.6). The SYSPROC concatenation can reference both exec and CLIST libraries.

3.5 How to dynamically allocate and free the RLX libraries (on an application basis)

You can use the RLXDYN exec in the ISPF environment to first dynamically allocate the RLX dialog libraries via a series of ISPF LIBDEF services and TSO ALTLIB commands and then call your exec or CLIST. Ask your RLX product administrator about the availability and use of the RLXDYN exec.

3.6 The ALTLIB command of TSO/E

The ALTLIB command, a TSO/E feature available in OS/390 and MVS/ESA environments provides users with greater speed and flexibility in specifying command procedure libraries from which REXX execs and TSO CLISTs can be implicitly executed.

REXX execs and TSO CLISTs may be invoked implicitly, simply by specifying the command procedure's name. (When invoked implicitly, the fully qualified dataset name of the library containing the target EXEC or CLIST is omitted.) Implicit execution is simpler to code and easier to read. In addition, implicit execution is ***required*** if the REXX exec or CLIST is invoked by the ISPF SELECT service or the RLXS command frontend (described in Chapter 3 of the RLX/SQL Reference).

Ordinarily, in order for REXX execs and TSO CLISTs to be invoked implicitly, they must reside in a library preallocated to file SYSEXEC (for REXX execs) or file SYSPROC (for either execs or TSO CLISTs).

However, pre-allocating files SYSPROC or SYSEXEC can be problematic if, for example:

- your TSO logon procedure cannot be modified
- the SYSEXEC and SYSPROC datasets are protected, or
- you need to mix exec or CLIST libraries with different dataset attributes.

The ALTLIB command solves such problems because with it you can:

- Easily activate and deactivate additional REXX exec and CLIST libraries for implicit execution without disturbing the list of datasets concatenated to files SYSEXEC and/or SYSPROC. Furthermore, you can issue the ALTLIB command from within the ISPF environment. There is no need to return to the TSO READY prompt.

- You can continue to access installation-wide REXX execs and CLISTs, since the SYSEXEC and SYSPROC file allocations remain undisturbed.

- Search time for the EXEC and CLISTs will be reduced since the system searches the ALTLIB libraries first, using the Virtual Lookaside Facility (VLF).

- The ALTLIB command lets you specify EXEC and CLIST libraries whose dataset characteristics (record format, record size, etc.) are different from those of the libraries already allocated to files SYSEXEC and SYSPROC.

3.6.1 ALTLIB Functions

The ALTLIB command offers four functions, which use the following operands:

- ACTIVATE — Allows implicit execution of REXX execs and CLISTs
- DEACTIVATE — Removes libraries from the search order
- DISPLAY — Displays the current search order
- RESET — Resets the search order to just the SYSEXEC and SYSPROC datasets

The ACTIVATE and DEACTIVATE functions permit you to divide execution libraries into 3 groups (or levels) termed USER, APPLICATION and SYSTEM. The system conducts the search for an implicitly named EXEC in that order.

The USER level permits searching of datasets previously allocated to SYSUEXEC and SYSUPROC while the SYSTEM level affects the search of libraries allocated to files SYSEXEC and SYSPROC.

> ***NOTE:*** *Since SYSEXEC and SYSPROC libraries are already available for implicit execution, the primary advantage is increased library search speed for the EXECs and CLISTs.*

The APPLICATION level lets you dynamically specify a list of command procedure libraries or a DDNAME of your choice, which should be searched for an implicitly named procedure. For this reason, the APPLICATION level is (arguably) the most useful and most-often used ALTLIB group.

Additional parameters on the ALTLIB command allow you to activate and/or deactivate REXX EXECs, TSO CLISTs or both EXECs and CLISTs at once.

Stacking ISPF Applications

The NEWAPPL parameter of the ISPF SELECT command creates a new ISPF "application" environment with its own variable pools. Such ISPF applications may be nested. In order to pass ALTLIB definitions from one ISPF application to another, ***you must specify the PASSLIB parameter on the SELECT service request***. For example:

```
ISPEXEC SELECT CMD(xyzexec) NEWAPPL(xyz) PASSLIB
```

After the new ISPF application completes and returns control to the invoking application, ISPF restores the original ALTLIB definitions, regardless of any ALTLIB changes made by the lower-level application.

Stacking ALTLIB Requests

Within an ISPF application you can stack up to 8 ACTIVATE requests on a "Last-In-First-Out" (LIFO) basis. Only the top, or last request is active.

Conditional Requests

A COND parameter may also be specified when activating application-level libraries. This will prevent the activation of any application-level definitions should they already exist for the same type (CLIST or EXEC).

3.6.2 ALTLIB Examples

Issue the following ALTLIB command to let you implicitly execute REXX execs which reside within library 'MY.REXX.EXEC':

```
ALTLIB ACTIVATE APPLICATION(EXEC) DATASET('MY.REXX.EXEC')
```

Issue the following ALTLIB command to allow implicit execution of REXX execs and CLISTs within libraries MYDATA.EXEC and YOURDATA.CLIST. However, ALTLIB activation should only take place if no previous application-level CLIST requests exist.

```
ALTLIB ACTIVATE APPLICATION(CLIST) DATASET('mydata.exec' 'yourdata.clist') COND
```

A return code of 8 is returned if previous definitions do exist.

3.6.3 ALTLIB Usage Notes

- Datasets referenced in an ALTLIB command must be both catalogued and partitioned

- Up to 15 application-level datasets may be listed in one ACTIVATE command. For all 3 levels, datasets concatenated within the same level must have the same logical record length (LRECL) and record format. Physical block sizes (BLKSIZE) may differ, but the dataset with the largest block size must be listed first.

- Datasets activated with the CLIST parameter may contain both CLISTs and REXX EXECs, but those activated with the EXEC parameter may contain only REXX execs.

- 'ALTLIB DEACTIVATE ALL' and 'ALTLIB RESET' are ***not*** equivalent. 'DEACTIVATE ALL' deactivates all 3 levels while 'RESET' deactivates the user and application levels and activates the system level (SYSEXEC and SYSPROC). RESET restores the library search order in effect before any ALTLIB commands were issued.

3.7 Verify that RLX is accessible

It may be advisable to verify that you have access to RLX before you try to use it. To determine whether you can get to RLX, you can key in and invoke an exec such as:

```
/*  rexx */
address rlx                          (1)

"RLX term'                           (2)
say 'Return Code ='rc                (3)
return
```

where

(1) ADDRESS RLX instructs REXX to pass host commands to RLX for execution. The first section of Chapter 3 describes ADDRESS RLX and the RLX host command environment in detail.

(2) Call the RLX TERM service immediately to terminate the RLX environment. The purpose of calling RLX TERM here is to verify whether RLX is accessible in your environment.

(3) Display the value of the REXX special variable RC. ***If the return code from the 'RLX TERM' command is -3, it means the command was not found and you do not have access to RLX***. In this case ask your RLX product administrator for assistance.

3.8 RLX User Facility

Figure 3.5 illustrates the RLX User Facility menu with which you can explore and exercise various RLX product components. For example, the RLX for MVS option provides dialogs, sample execs and tutorial information on its REXX VSAM interface, Software Development Kit and REXX compiler.

Ask your RLX product administrator whether the RLX User Facility is available in your environment as an ISPF selection menu option. Alternatively, you can invoke the RLX dialog that displays this menu with the implicit EXEC command %RLXUSER.

```
----------------------------- RLX User Facilities -----------------------------
Option ===>

   1  Job / Parm   - Specify job statement(s) and JCL parameters
   2  DB2 Libs     - Specify SDSNEXIT and SDSNLOAD libraries
   3  Prof Refresh - Update your RLX profile with RLX system defaults
   4  User Profile - Maintain your RLX user profile for DB2 access
   5  RLX for z/OS - REXX VSAM,Software Development Kit and AcceleREXX compiler
   6  RLX/Compile  - Program preparation for RLX SQL source modules
   7  RLX/Translate- CLIST to REXX translation
   8  Samples      - Run (and examine) additional RLX procedure samples
   9  RLXS         - Invoke the RLXS front end command
  10  Tutorial     - RLX interactive help
   X  Exit         - Leave RLX User Menu

Specify target DB2 subsystem parameters
   Subsystem name    ===> DSN1
   Version / Release ===> 7.1          (for example: 6.1, 7.1, 8.1)

Enter END to exit
```

Figure 3.5

Chapter 4

REXX Language Summary

This chapter provides a summary of the REXX language tokens, operators, instructions, and functions. It also describes the REXX extension functions supplied with the RLX/SDK (Software Development Kit). For complete details on the TSO/E REXX language see the IBM publication: TSO Extensions - REXX Reference.

Tokens

Literal strings

Strings are sequences of characters delimited by single or double quotes.

```
"ABCDE"
'ABCDE'
```

Use two consecutive quotes to denote an embedded quote, as in the following example.

```
"ABC""DEF"
'ABC''DEF'
```

You can also embed a quote by delimiting the string with apostrophes:

```
"ABC'DEF"
'ABC"DEF'
```

The empty or null string contains no characters and its length is 0.

```
""
''
```

Hexadecimal strings end with an X following the closing delimiter and may include a blank at byte boundaries -- as in the following examples:

```
"lDECF8"X
"lD EC F8"X
```

Binary strings end with a B following the closing delimiter and may include a blank between every four bits -- as in the following examples.

```
"11110000"B
"1111 0000"B
```

Symbols

Symbols are single words comprised of the letters A-Z, a-z, 0-9, the period (.), exclamation mark (!), question mark (?), and underscore (_). Lower case letters are internally translated to upper case. A variable's uninitialized value, therefore, is its own name -- translated to upper case.

Examples:

```
able
ABLE
!name
23.0blart
```

Constants begin with 0-9 or a period. Constants that form valid numbers can be used in arithmetic expressions. Numbers may also be in exponential form.

Examples:

```
0IAmAConstant
3.ButNotANumber
99
99.0
99.0e+0
```

Simple variables do not start with 0-9 and do not contain periods.

Examples:

```
A
A9
!name
?name
```

Compound variables are variables that contain at least one period. If a compound variable contains only one period it must not be the last character in the symbol.

Examples:

```
student.i
student.i.j
record.!name
```

A **stem** is the root name of a compound variable, including the trailing period.

Examples:

```
student.
record.
```

Operators

Some operators have more than one form. Only the most commonly used forms are included in this summary.

Arithmetic

`+`	Add (or unary plus)
`-`	Subtract (or unary minus)
`*`	Multiply
`/`	Divide
`%`	Integer divide (integer part or division result)
`//`	Remainder (may be negative)
`**`	Power (exponent must be a whole number)

Comparative

`=`	Equal
`==`	Strictly equal
`\=`	Not equal
`< >`	Not equal
`\= =`	Not strictly equal
`>`	Greater than
`>>`	Strictly greater than
`<`	Less than
`<<`	Strictly less than
`> =`	Greater than or equal
`>> =`	Strict greater than or equal
`< =`	Less than or equal
`<< =`	Strictly less than or equal

Logical

`&`	And
`|`	Or
`&&`	Exclusive or
`\`	Not

Concatenation

`<blank>`	Concatenate terms with one blank between
`||`	Concatenate terms without blank between
`<abuttal>`	Concatenate terms without blank between

Instructions

address environment expression

address value expression

arg [template]

call name [expression] [, expression] ...

call on trapName [name label]

call off trapName

do [for] [control] [;]

  [instruction-list]

end [loopSymbol]

where `for` is one of the following:

forVar = initial [tofinal] [byincrement] [forcount]

repetitions

forever

and `control` is one of

while termination

until termination

drop symbols

exit [expression]

if expression [;] then [;] instruction [else[;] instruction]

interpret expression

iterate [loopSymbol]

leave [loopSymbol]

nop

numeric digits expression

numeric form scientific

numeric form engineering

numeric form [value] expression

numeric fuzz expression

options expression

parse [upper] arg [template]

parse [upper] linein [template]

parse [upper] pull [template]

parse [upper] source [template]

parse [upper] value [expression] with [template]

parse [upper] var symbol [template]

parse [upper] version [template]

procedure [expose symbols]

pull [template]

push [expression]

queue [expression]

return [expression]

say [expression]

select [;]

when expression [;] then[;] instruction

. . .

otherwise[;]

[instruction list]

end

signal label

signal [value] expression

signal on trapName [namelabel]

signal off trapName

trace traceSetting

trace [value] expression

Built-in functions

abbrev(target, prefix [, minimum_length])

abs(number)

address()

arg([number [, verification]])

bitand(left, right [,fill])

bitor(left, right [,fill])

bitxor(left, right [,fill])

b2x(bitstring)

center(string, length [,fill])

charin([stream_name] [,starting_position] [, num_chars]])   (available in OS/2 only)

charout([stream_name] [, [value] [, starting_position]])   (available in OS/2 only)

chars([stream_name])   (available in OS/2 only)

compare(left, right [,fill])

condition([information type])

copies(string, number_copies)

c2d(string [, size])

c2x(string)

datatype(value [, verification type]

date([formatting_option])

delstr(source, starting_position [, length])

delword(source, starting_position [, length])

digits()

d2c(numeric_value [, result length])

d2x(numeric_value [, result length])

errortext(message_number)

find(string, phrase)

form()

format(numeric_value [, [leading_digits] [, decimal_places]])

fuzz()

getmsg(msgetm, [,msgtype[,cart [,mask [,time]]]]) (available in TSO/E only)

index(haystack,needle [,start])

insert(source, target [, [insertion_point] [, [length] [,fill]]])

justify(string,length [,pad])

lastpos(looking_for, looking_in [, starting_position])

left(string, length [,fill])

linesize() (available in TSO/E and VM only)

length(string)

listdsi([dsname [location] | filename file] [directory] [recall] (available in TSO/E only)

linein([stream_name] [, [line_to_read] [, number_lines]]) (available in OS/2 only)

lineout([stream_name] [, [value] [, position]]) (available in OS/2 only)

lines([stream_name]) (available in OS/2 only) max(numeric_value [, [numeric_value] [, ...]])

min(numeric_value [, [numeric_value] [, ...]])

msg(ON | OFF) (available in TSO/E only)

outtrap(off | [varname [,max] [,concat]]) (available in TSO/E only)

overlay(source, target, [, [insertion_point] [, [length] [,fill]]])

pos(lookingfor, looking in [, starting_position])

prompt(ON | OFF) (available in TSO/E only)

queued()

random([lower] [, [upper] [, seed]])

random(upper)

reverse(string)

right(string, length [,fill])

sign(numeric_value)

setlang(CHS | CHT | DAN | DEU | ENP | ENU | ESP | FRA | JPN | KOR | PTB)

sign(number) (available in TSO/E only)

sourceline([line_number])

space(string [, [numberfill_chars] [,fill]])

stream(stream_name [, operation [, command]]) (available in OS/2 only)

storage(address [,length] [,data]) (available in TSO/E only)

strip(string [,B | ,L | ,R] [,char])

substr(string, starting_position [, [length] [,fill]])

subword(string, starting_position [, length])

symbol(identifier)

sysdsn(datasetname) (available in TSO/E only)

sysvar(var_name) (available in TSO/E only)

time([format])

trace([trace_type])

translate(string [, [output_table] [, [input_table] [,fill]]])

trunc(numeric_value [, decimal_places])

value(identifier [, [value] [, variable_pool]])

verify(string, alphabet [, [match] [, starting_position]])

word(string, word_number)

wordindex(string, word_number)

wordlength(string, word_number)

wordpos(sought, source [, starting_word])

words(string)

xrange([starting_char] [, ending_char])

x2b(hex_string)

x2c(hex_string)

x2d(hex_string [, size])

RLX / VSAM

The REXX VSAM Interface

User

Guide

Version 2.2

June 2010

Publication   RRV-001-3

Relational Architects Intl

This Guide: (RAI Publication RRV-001-3)

This document applies to RLX/VSAM Version 2.2 and RLX for z/OS Version 9 Release 1 (June 2010), and all subsequent releases, unless otherwise indicated in new editions or technical newsletters. Specifications contained herein are subject to change and will be reported in subsequent revisions or editions.

Purchase Orders for publications should be addressed to:

Documentation Coordinator
Relational Architects Intl.
Riverview Historic Plaza
33 Newark Street
Hoboken NJ 07030 USA

Tel: 201 420-0400
Fax: 201 420-4080
Email Sales@relarc.com

Reader comments regarding the product, its documentation, and suggested improvements are welcome. A reader comment form for this purpose is provided at the back of this publication.

Ed 21F10

RLX / VSAM

Table of Contents

Chapter 1

The REXX VSAM Interface

1.1 Concepts and Facilities

RLX/VSAM gives REXX applications the ability to access the full range of DFSMS VSAM facilities. These include record and CI level input and output services for ESDS, KSDS, and RRDS VSAM clusters -- and control interval access to VSAM linear (LDS) datasets. The developer can choose from the full spectrum of VSAM dataset MACRF options and RPL OPTCD processing options, enabling comprehensive, low level control over the various data access strategies supported by VSAM.

The syntax for invoking these VSAM facilities is designed to parallel standard MVS VSAM macro conventions. A programmer familiar with the use of standard MVS VSAM facilities can use RLX/VSAM almost immediately. The RLX/VSAM documentation details the invocation syntax for each function although this manual is not a substitute for IBM's VSAM documentation. In addition, Chapter 2 of this Guide includes several sample REXX execs that illustrate read, write and update access to VSAM datasets and also serves to familiarize the reader with RLX/VSAM services.

RLX/VSAM services may be requested through subroutine calls or function references, according to user preference. RLX/VSAM operates in the TSO foreground as well as in TSO batch and MVS batch environments.

1.2 RLX / VSAM Invocation Overview

From an application's perspective, RLX/VSAM is a set of services which a REXX application can access via a REXX CALL statement or function reference. Invocation via a CALL statement is coded as follows:

```
call OPERATION parm1,parm2, ...
```

while invocation as a REXX function is coded like this:

```
RC = OPERATION(parm1,parm2,...)
```

With function invocation, when OPERATION executes, it returns a value that REXX assigns in the above example to the variable RC. This value of RC is the same as the value of the RVRC variable returned as feedback by RLX/VSAM and discussed in Section 1.2.1.

The examples of RLX/VSAM command syntax presented below are all invoked via REXX CALL statements. All parameters coded in the CALL are positional and appear either as variables or as keywords. Variable parameters appear in lowercase between arrows (for example `< value >`) while keywords appear in uppercase and must be coded exactly as shown (for example REFRESH). VSAM services and their associated parameters are summarized below. Each VSAM function is discussed in detail in its own subsection of Section 1.6.

```
call RVOPEN   <ddname>, <acb_options>

call RVGET    <ddname>, <key>, <rpl_options>

call RVPUT    <ddname>, <record>, <key>, <rpl_options>

call RVERASE  <ddname>

call RVENDREQ <ddname>

call RVPOINT  <ddname>, <key>, <rpl_options>

call RVVERIFY <ddname> [,REFRESH]

call RVCLOSE  <ddname> [,TEMP]

call RVMSG    <routecde>, <msg_group_level>, <ddname>

call RVTERM   <files>
```

Each VSAM related call must identify the file it references. The number of concurrently OPEN VSAM files is limited only by available memory. Several positional parameters are optional since they have meaning only in certain contexts. For example, a `<key>` is typically required only when a file is accessed directly. All keyword parameters are optional. When omitted, RLX/VSAM assumes either the VSAM defaults or the last user specified option for that file. Some parameters depend on the type of dataset being accessed (e.g. ESDS, KSDS or RRDS).

Parameters may be specified as standard REXX variables or as literal strings. Information returned to a caller (such as the contents of a record retrieved by the RVGET service) is placed into the set of REXX feedback variables discussed next.

1.2.1 Feedback

After each function call, RLX/VSAM returns information related to that call into one or more of the REXX feedback variables listed below:

RVRC	RVMACRC	RVREASON	RVCISZE
RVSKEY	RVKEYOF	RVHRBA	RVURBA
RVRECNM	RVRECORD	RVAVSPAC	RVBUFND
RVBUFNI	RVFTYPE	RVRBA	RVRRN
RVKEYL	RVRECL	RVMAXLR	

The user may reference these feedback variables to determine the success of an operation and obtain any data returned by the function. Each REXX VSAM service returns feedback into a subset of these variables. The subsections of Section 1.6 describe each VSAM service in turn, along with their associated feedback variables.

1.2.2 Processing Overview

Your exec must allocate any VSAM file it will access before you request an RLX/VSAM service for that file. A standard ALLOCATE command can be issued in the TSO environment for this purpose while JCL allocation serves in batch. The first RLX/VSAM service referencing a file should be an RVOPEN call. The exception is the RVVERIFY service which may be invoked ***without*** a prerequisite RVOPEN.

The RVOPEN service establishes a connection to the VSAM file, allocates I/O buffers, creates the VSAM ACB and RPL control blocks and issues the MVS OPEN SVC. An optional <acb_options> parameter may be passed to RVOPEN to request specific VSAM processing options. These options correspond exactly to the MACRF options defined in the standard IBM VSAM documentation. Similarly, the default settings are the same as the default values defined by IBM.

Following a successful RVOPEN (RVRC = 0), the application is free to issue other requests (such as RVGET, RVPUT, RVPOINT, etc.) for the now open file. These calls let you retrieve, insert, update and delete VSAM records or control intervals. In addition, record position may be manipulated to achieve 'skip sequential' access. The <rpl_options> parameter corresponds to the VSAM RPL OPTCD options described in the IBM VSAM documentation. As in the native VSAM interface, the RPL OPTCD settings can be altered between calls to RLX/VSAM. For example, you can change RPL options between one RVGET request and another or between an RVGET and an RVPUT.

Feedback from each call is returned in the set of REXX variables enumerated in Section 1.2.1. Your exec can reference these variables to obtain such information as the key, RBA or RRN of the record just accessed; the contents of the RECORD or control interval (CI) just read; its length; etc.

When processing is complete, the RVCLOSE function terminates processing of the dataset. If desired, the application may request a temporary RVCLOSE which flushes VSAM buffers but allows processing to resume without another RVOPEN. The RVTERM function should be used to terminate the RLX/VSAM environment and close any RLX/VSAM managed files that remain open.

1.3 ACB options

The Application Control Block or ACB specifies the attributes and processing options for a VSAM dataset. The list below enumerates the various ACB options, described in the MVS VSAM macro reference, which may be specified on the RVOPEN call. These ACB attributes remain in effect while the file is open. In the following list, mutually exclusive options are separated by a vertical bar | .

```
<acb_options> = (ADR|CNV|KEY
                ,DFR|NDF
                ,DIR|SEQ|SKP
                ,IN|OUT
                ,NIS|SIS
                ,NRM|AIX
                ,NRS|RST)
```

Table 1.1 describes the various ACB options:

Control class	Option	Description
Key	ADR	Specifies Addressed access to a KSDS or ESDS dataset. RBA is used as a search argument
	CNV	Defines access to the entire contents of a Control Interval
	KEY	Defines keyed access to a KSDS or access by relative record number to an RRDS
Access	DIR	Direct access to a KSDS, ESDS or RRDS
	SEQ	Sequential access to a KSDS, ESDS or RRDS
	SKP	Skip-sequential access to a KSDS or RRDS (only with keyed access in a forward direction).
Update Intent	IN	VSAM dataset is being opened for read only
	OUT	VSAM dataset is being opened for write
Write Deferment	DFR	The buffer pool will not be immediately written by the RVPUT function request
	NDF	The buffer pool will be immediately written by the RVPUT function request
CI Split control	NIS	Control intervals are split at the midpoint
	SIS	Split CI and CA at the insertion point rather than at the midpoint when doing direct RVPUT
Access object	NRM	By the specified DDname with the RVOPEN request
	AIX	Alternate Index of the path specified by DDname
Reuse	NRS	Data set is not reusable
	RST	Data set is reusable

Table 1.1

1.4 RPL options

The Request Parameter List (or RPL) specifies how a record in a VSAM file will be accessed: sequentially or directly, with or without update intent, etc. RPL options are specified in RPL based VSAM calls and complement ACB processing options. The possible RPL options are shown below in Table 1.2 with mutually exclusive options separated by a vertical bar | .

```
<rpl_options> = (ADR|CNV|KEY
                ,DIR|SEQ|SKP
                ,ARD|LRD
                ,FWD|BWD
                ,ASY|SYN
                ,NSP|NUP|UPD
                ,KEQ|KGE
                ,FKS|GEN
                ,LOC)
```

Control class	Option	Description
Key	ADR	Addressed access to a KSDS or ESDS using an RBA as a search argument
	CNV	Access to the entire contents of a Control Interval
	KEY	Access by key to a record of a KSDS file or by relative record number to a record of an RRDS file
Access	DIR	Direct access to a KSDS, ESDS or RRDS
	SEQ	Sequential access to a KSDS, ESDS or RRDS
	SKP	Skip-sequential access to a KSDS or RRDS (only with keyed access in a forward direction).
Record	ARD	User argument determines the record to be processed
	LRD	Last record in the data set is to be processed by the RVPOINT or RVGET request
Processing direction	FWD	Forward direction
	BWD	Backward direction
Data record processing	NSP	For direct processing requests, VSAM should remember the position of the record being processed
	NUP	A data record that is being retrieved will not be updated or deleted
	UPD	A data record that is being retrieved may be updated or deleted
Search key	KEQ	User's search key must be equal to the key or relative record number
	KGE	User's request applies to the record that has the next greater or equal key
	FKS	A full key is provided as a search argument
	GEN	A generic key is provided as a search argument

Table 1.2

1.5 Working with RLX / VSAM

Examples of RLX/VSAM service invocations are presented in this section and summarized in Table 1.3. They are categorized by the first column labeled 'Processing Intent' (either Read, Update or Point) and secondarily by VSAM file type: ESDS, KSDS, or RRDS. In these examples, REXX variables with lowercase names -- such as `DDdname`, `key`, `rrn` and `rba` -- are assumed to have been set to appropriate values before a call is issued.

Insert and delete processing:

`Insert` See Update section, modify ACB 'IN,OUT' parameters to specify just the 'OUT' option on the RVOPEN call
e.g. `call RVOPEN ddname,'(ADR,DIR,OUT)'`

`Delete` See Update section, instead of RVPUT function use RVERASE
e.g. `call RVERASE ddname.`

NOTE*: The RVERASE request is invalid for a VSAM ESDS cluster.*

```
===========================================================================
Processing | Access     | VSAM | RLX/VSAM function syntax
Intent     | mode       | type |
-----------+------------+------+-------------------------------------------
Read       | Sequential | KSDS | call RVOPEN  ddname,'(KEY,SEQ,IN)'
           |            | RRDS | call RVGET   ddname,,'(NUP,LOC)'
           |            |      | call RVCLOSE ddname
           |            +------+-------------------------------------
           |            | KSDS | call RVOPEN  ddname,'(ADR,SEQ,IN)'
           |            | ESDS | call RVGET   ddname,,'(NUP,LOC)'
           |            |      | call RVCLOSE ddname
           +------------+------+-------------------------------------
           | Direct     | KSDS | call RVOPEN  ddname,'(KEY,DIR,IN)'
           |            |      | call RVGET   ddname, key,'(NUP,LOC)
           |            |      | call RVCLOSE ddname
           |            +------+-------------------------------------
           |            | RRDS | call RVOPEN  ddname,'(KEY,DIR,IN)'
           |            |      | call RVGET   ddname,rrn,'(NUP,LOC)'
           |            |      | call RVCLOSE ddname
           |            +------+-------------------------------------
           |            | ESDS | call RVOPEN  ddname,'(ADR,DIR,IN)'
           |            |      | call RVGET   ddname,rba,'(NUP,LOC)'
           |            |      | call RVCLOSE ddname
-----------+------------+------+-------------------------------------
Update     | Sequential | KSDS | call RVOPEN  ddname,'(KEY,SEQ,IN,OUT)'
           |            | RRDS | call RVGET   ddname,,'(UPD,LOC)'
           |            |      | call RVPUT   ddname,updrec,,'(LOC)'
           |            |      | call RVCLOSE ddname
           |            +------+-------------------------------------
           |            | KSDS | call RVOPEN  ddname,'(ADR,SEQ,IN,OUT)'
           |            | ESDS | call RVGET   ddname,,'(UPD,LOC)'
           |            |      | call RVPUT   ddname,updrec,,'(LOC)'
           |            |      | call RVCLOSE ddname
           +------------+------+-------------------------------------------
           | Direct     | KSDS | call RVOPEN  ddname,'(KEY,DIR,IN,OUT)'
           |            |      | call RVGET   ddname,SearchKey,'(UPD,LOC)'
           |            |      | call RVPUT   ddname,updrec,SearchKey,'(LOC)'
           |            |      | call RVCLOSE ddname
           |            +------+-------------------------------------------
           |            | RRDS | call RVOPEN  ddname,'(KEY,DIR,IN,OUT)'
           |            |      | call RVGET   ddname,RRNO,'(UPD,LOC)'
           |            |      | call RVPUT   ddname,Updrec,rrn,'(LOC)'
           |            |      | call RVCLOSE ddname
           |            +------+-------------------------------------
           |            | ESDS | call RVOPEN  ddname,'(ADR,DIR,IN,OUT)'
           |            |      | call RVGET   ddname,RBA,'(UPD,LOC)'
           |            |      | call RVPUT   ddname,Updrec,rba,'(LOC)'
           |            |      | call RVCLOSE ddname
-----------+------------+------+-------------------------------------
Point      |            |      | call RVPOINT ddname,key,'(FWD,KGE)'
           |            |      | call RVPOINT ddname,rrn,'(BWD,KEQ)'
           |            |      | call RVPOINT ddname,rba,'(FWD,KGE)'
Pointed to |            |      | call RVPOINT ddname,,'(LRD,BWD)'
the last   |            |      |
record     |            |      |
===========================================================================
```

Table 1.3

1.6 Function Call Reference.

1.6.1 RVCLOSE - Close a VSAM file

`CALL RVCLOSE <ddname>[,TEMP]` where

`<ddname>`	is a required parameter, coded as either a REXX variable or literal string within quotes, whose value is the DDname of the file to which the VSAM dataset is allocated.
`TEMP`	Indicates that a temporary close should be issued for the file.

Usage:

The RVCLOSE function is used to terminate your application's access to the dataset and to free resources associated with the file. The <ddname> parameter identifies the name of the file used to allocate the VSAM dataset. In addition to releasing RLX/VSAM related control areas and buffers, RVCLOSE also issues a standard VSAM CLOSE before completing its processing.

The optional TEMP parameter requests a 'temporary' VSAM CLOSE. Such a temporary (TYPE=T) CLOSE allows all outstanding I/O operations to complete before purging all VSAM buffers associated with the file and updating VSAM catalog statistics. However, the temporary CLOSE does not disconnect your application from the file. Rather, the file remains open with its record position unaltered such that subsequent functions like RVGET, RVPUT, and RVERASE may be performed without re-issuing an RVOPEN.

The following REXX Feedback Variables are set on completion of RVCLOSE:

RVRC	Contains a general return code from RVCLOSE. It is set to zero when all VSAM and RLX/VSAM functions complete successfully.
RVMACRC	Contains a code returned to RLX/VSAM by the VSAM CLOSE macro. Note that this is independent of the RVRC variable setting.
RVREASON	Is set to the reason code returned in the ACB by the VSAM CLOSE macro in the event of VSAM CLOSE failure. When successful, RVREASON is set to zero.

Example of Coding:

```
/* Rexx */
...
File = 'ESDSFILE'
CALL RVOPEN  File,'(SEQ,IN)'
Do While RVEOF <> 'EOF'
   CALL RVGET File,'(LOC)'
   Say 'Record = ' RVRECORD
End
CALL RVCLOSE File
...
```

In this example, a VSAM ESDS allocated to DDname ESDSFILE is opened. Then within a loop, records are sequentially accessed and then printed until end-of-file is reached. Finally, the dataset is closed.

1.6.2 RVENDREQ - Release a VSAM control interval

`CALL RVENDREQ <ddname>` where

`<ddname>` is a required parameter, coded as either a REXX variable or literal string within quotes, whose value is the DDname of the file to which the VSAM dataset is allocated.

Usage:

The RVENDREQ function issues an VSAM ENDREQ macro in order to release an application's exclusive hold on the current VSAM control interval. In addition, all buffers for the specified file are externalized and record positioning is lost. Subsequent RLX/VSAM functions must reestablish file positioning (e.g. via RVPOINT).

The following REXX Feedback Variables are set on completion of RVENDREQ:

RVRC Indicates a general return code from the RVENDREQ service. It is set to zero when all VSAM and RLX/VSAM functions complete successfully.

RVMACRC Is set to the return code passed back to RLX/VSAM by the VSAM ENDREQ macro. Note that this is independent of the RVRC variable setting.

RVREASON Is set to the reason code returned by the VSAM ENDREQ macro in the RPL. When successful, RVREASON is set to zero.

Example of Coding:

```
/* Rexx */
...
CALL RVOPEN   'KSDSFILE','(KEY,DFR,IN,OUT)'
CALL RVGET    'KSDSFILE','KEYØØ5','(DIR,UPD,LOC)'
CALL RVPUT    'KSDSFILE','KEYØØ5 NEW RECORD',,'(LOC)'
CALL RVENDREQ 'KSDSFILE'
...
```

In this example, a VSAM KSDS allocated to DDname KSDSFILE is opened with the Deferred Write option (DFR). Next, a record with a key value of 'KEY005' is first accessed and then updated in-place with a new record value. Finally, RVENDREQ instructs VSAM to complete all outstanding I/O operations.

1.6.3 RVERASE - Delete a record from a VSAM file

`CALL RVERASE <ddname>` where

<ddname>	is a required parameter, coded as either a REXX variable or literal string within quotes, whose value is the DDname of the file to which the VSAM dataset is allocated.

Usage:

The RVERASE function requests that VSAM delete the record that was most recently retrieved via an RVGET that specified an <rpl-option> of UPD. Both KSDS and RRDS files permit such record deletions. When these records are deleted, the space they occupied is freed. RVERASE is not available for entry sequenced datasets. Your application can only logically delete records within an ESDS by setting flags within the records themselves. No actual increase in free space or free record slots occurs.

The following REXX Feedback Variables are set on completion of RVERASE:

RVRC	Indicates a general return code from the RVENDREQ service. It is set to zero when all VSAM and RLX/VSAM functions complete successfully.
RVMACRC	Is set to the return code passed back to RLX/VSAM by the VSAM ERASE macro. Note that this is independent of the RVRC variable setting.
RVREASON	Is set to the reason code returned by the VSAM ERASE macro via the RPL in case of VSAM ERASE failure. When successful, RVREASON is set to zero.
RVSKEY	Is set to the value of the key of the record just erased. Note that this may be an embedded key, an RBA or an RRN, depending on the file type and access mode.
RVRBA	Is set to the RBA of the record just erased. This is true for any file type or access mode.
RVRRN	Is set to the RRN of the record just erased. This variable is set only when an RRDS is accessed.
RVKEYL	Is set to the length of the key used for the erase. This variable only has meaning when accessing by embedded key.

1.6.4 RVGET - Read a record from a VSAM file

`CALL RVGET <ddname>, <key>, <rpl_options>` where

`<ddname>` Is a required parameter, coded as either a REXX variable or literal string within quotes, whose value is the DDname of the file to which the VSAM dataset is allocated.

`<key>` Is a character string that represents the RBA, RRN, full key or generic key of an ESDS, RRDS or KSDS.

> For an ESDS which is processed directly, use an RBA as the key.

> For a KSDS, the key must be an RBA when <rpl-options> contains either the ADR or CNV option. Use an embedded key (full or generic) when <rpl--options> specifies the KEY option.

> For an RRDS, the key must be an RRN.

The <key> parameter specifies a search key used to retrieve a particular record or CI when processing in *direct* or *skip sequential* mode. The <key> must be specified whenever the VSAM processing mode necessitates the use of a retrieval key. When the key is included but not required, it is ignored by RLX/VSAM.

`<rpl_options>` Specifies the VSAM RPL OPTCD processing options with syntax analogous to that described in the IBM VSAM documentation. The processing options must be separated by commas. The string of options may optionally be enclosed in parentheses. Option selection rules are defined by IBM VSAM RPL OPTCD specifications.

Usage:

The RVGET function retrieves a record or control interval from a VSAM file. The record area is returned in the REXX variable RVRECORD. For sequential processing, no key is required. For direct processing, use the <key> parameter to specify the key of the record to be retrieved.

The key is a character string that represents the RBA (for a KSDS or ESDS), an RRN (for an RRDS) or a full or generic embedded key for a KSDS. Use an RBA as a key for an ESDS that is processed directly. For a KSDS, the key must be a character representation of an RBA when <rpl_options> specifies either the ADR or CNV options. When the KEY option is specified for a KSDS, an embedded full or generic key should be supplied on the call. Lastly, for an RRDS, the key must be a character representation of a relative record number (RRN).

Your REXX exec can detect a VSAM end-of-file condition when processing either sequentially or skip sequentially when the value of the RVEOF variable is set to 'EOF'. Alternatively, when feedback variable RVMACRC is equal to 8 ***and*** the RVREASON variable is set to 4, the two, together, signal an end-of-file condition.

Similarly, an end-of-file condition is signaled when a search key supplied on a direct access request has a higher key than the current high key of the dataset.

The retrieved record or CI is returned to your REXX exec in the REXX variable RVRECORD.

> ***NOTE****: The use of VSAM* **move** *mode (MVE option in <rpl-options>) is not appropriate and should not be used. The <rpl_options> should specify or default to the LOC option.*

The following REXX Feedback Variables are set on completion of RVGET:

RVRC	Indicates a general return code from the RVGET service. It is set to zero when all VSAM and RLX/VSAM functions complete successfully.
RVMACRC	Is set to the return code passed back to RLX/VSAM by the VSAM GET macro. Note that this is independent of the RVRC variable setting.
RVREASON	Is set to the reason code passed back to RLX/VSAM by the VSAM GET macro via the RPL in case of VSAM GET failure. When successful, RVREASON is set to zero.
RVRECORD	Contains the record or CI that was retrieved by RVGET.
RVRECL	Contains the length of the record just retrieved.
RVEOF	Is set to 'EOF' whenever an end-of-file condition is sensed.
RVSKEY	Is set to the value of the key of the record just retrieved. Note that this may be an embedded key, an RBA or an RRN, depending on the file type and access mode.
RVRBA	Is set to the RBA of the record just retrieved. This is true for any file type or access mode.
RVRRN	Is set to the RRN of the record just retrieved. This variable will only be set when accessing RRDS VSAM datasets.
RVKEYL	Is set to the length of the key used for the retrieval. This variable only has meaning when access is through an embedded key.

1.6.5 RVMSG - Control Error / Information message output

`CALL RVMSG <routecde>, <msg_group_level>, <ddname>,` where

`<routecde>`

Specifies the destination for RLX/VSAM error and informational messages. One of the values shown below must be selected. The default value is 'TSO'.

- `'TSO'` — This option indicates that all messages are to TSO via the standard TPUT service. (This is the default).
- `'WTO'` — This option indicates that all messages will be directed to MCS with a WTO routing code of 11 (WTP).
- `'NONE'` — This option indicates that no messages will be issued except for catastrophic errors. These errors will be reported via WTO with a routing code of 11.
- `'FILE'` — This option causes all messages to be routed to the file named in the <ddname> parameter. The file must be allocated before issuing the RVMSG function. This log file must be allocated as a standard sequential file with a fixed or variable RECFM and an LRECL greater than or equal to 80.

`<msg_group_level>`

Specifies how selective RLX/VSAM should be in issuing messages. One of the following values must be selected:

- `'TERSE'` — This option indicates that only error level messages are displayed.
- `'VERBOSE'` — This option indicates that error, warning and informational messages are displayed.
- `'DEBUG'` — This option indicates that error, warning and informational messages are displayed. In addition, debugging messages and traces are displayed.

`<ddname>`

Is a required parameter, coded as either a REXX variable or literal string within quotes, whose value is the DDname of the file to which the VSAM dataset is allocated. This parameter is only relevant when <routecde> is set to FILE; otherwise, it is ignored. The DDname TVRLOG is used as a default unless a preceding call to RVMSG established another DDname as the RLX/VSAM message file.

Usage:

The RVMSG function controls the destination and volume of RLX/VSAM messages. Messages may be issued to communicate error and warning conditions, to provide feedback regarding normal events, and to provide trace data for debugging purposes. Each of these three groups of messages can be selectively enabled or disabled via the RVMSG <msg_group_level> parameter.

The destination for the messages is controlled via the <routecde> parameter. Messages may be routed

- > to TSO via standard TPUT
- > to MCS using WTP (WTO with ROUTCDE of 11)
- > to a sequential file for logging

Alternatively, message issuance may be suppressed. When using the file logging option, ensure that the file has been allocated before the RVMSG call.

The RVMSG function may be invoked at any time. It may be reissued whenever new display options are desired. Please note that when using the NONE <routecde> option, only messages that report catastrophic errors are issued -- as WTOs using ROUTCDE 11.

When using the FILE option, the message log file is not closed until one of the following events occurs: either all VSAM activity has quiesced (i.e. all VSAM files are closed) or an RVTERM is issued. When VSAM activity resumes, the log is re-opened and messages are again directed to the log file. It is the application's responsibility to allocate the message log with a disposition of MOD rather than OLD or SHR. Unless MOD is specified, the log dataset is repositioned to the beginning for each such re-OPEN. Refer to the RVTERM call for more information regarding the log file.

The following REXX Feedback Variables are set on completion of RVMSG:

RVRC — Indicates a general return code from the RVMSG service. It is set to zero when all VSAM and RLX/VSAM functions complete successfully.

Example of Coding:

Only severe error messages will be issued.

```
/* Rexx */
...
CALL RVMSG ,'NONE'
...
```

1.6.6 RVOPEN - Open a VSAM file

`CALL RVOPEN <ddname>, <acb_options>` where

`<ddname>`	Is a required parameter, coded as either a REXX variable or literal string within quotes, whose value is the DDname of the file to which the VSAM dataset is allocated.
`<acb_options>`	Specifies the VSAM ACB MACRF processing options with syntax analogous to that described in the IBM VSAM documentation. The processing options must be separated by commas. The string of options may optionally be enclosed in parentheses. Option selection rules are defined by the IBM VSAM ACB MACRF specifications.

Usage:

The RVOPEN function requests that RLX/VSAM establish a processing environment for the file identified by the DDname parameter. When RLX/VSAM receives an RVOPEN request, it constructs VSAM ACB and RPL control blocks, obtains buffers, builds RLX/VSAM control areas and invokes the appropriate VSAM service routines for the type of file requested. **Remember that the file must be allocated before issuing RVOPEN.** All other RLX/VSAM functions (except RVVERIFY) require a successful RVOPEN *before* their invocation.

The <acb_options> positional parameter is used to specify VSAM processing options for the file. The options specified must be sufficiently broad so as to allow any of the functions that may be requested by subsequent RLX/VSAM function calls. For example, if both RVGET and RVPUT calls are issued, the <acb_options> should include both the 'IN' and 'OUT' options. These processing options correspond exactly to the VSAM ACB MACRF options documented in the IBM VSAM publications.

The following REXX Feedback Variables are set on completion of RVOPEN:

RVRC	Indicates a general return code from the RVOPEN service. It is set to zero when all VSAM and RLX/VSAM functions complete successfully.
RVMACRC	Is set to the return code passed back to RLX/VSAM by the VSAM OPEN macro. Note that this is independent of the RVRC variable setting.
RVREASON	Is set to the reason code passed back to RLX/VSAM by the VSAM OPEN macro via the RPL in case of VSAM OPEN failure. When successful, RVREASON is set to zero.
RVCISZE	Is set to the dataset's CI size as specified by the VSAM DEFINE command.

RVKEYOF — Is set to the offset from the beginning of the record to the start of the key field in the record.

RVKEYSZ — Is set to the length of the key field.

RVMAXLR — Is set to the maximum record length specified in the VSAM DEFINE command.

RVFTYPE — Is set to a character string that indicates the type of VSAM file just opened. Possibilities are 'ESDS', 'KSDS', 'RRDS' and 'LDS'.

RVHRBA — Is set to the value of the high allocated RBA for the dataset just opened.

RVURBA — Is set to the value of the high used RBA for the dataset just opened.

RVRECNM — Is set to the number of records currently in the dataset. Note that this variable is meaningful only for certain VSAM dataset types.

RVAVSPC — Is set to the number of available bytes remaining within the existing file allocation extents.

RVBUFNI — Is set to the number of index buffers defined for the file. Note that this variable is meaningful only for certain VSAM dataset types.

RVBUFND — Is set to the number of data buffers defined for the file.

Example of Coding:

```
/* Rexx */
...
CALL RVOPEN   'ESDSFILE','(SEQ,IN)'
...
```

In this example, a VSAM ESDS is opened for sequential input processing.

1.6.7 RVPOINT - Position at a record in a VSAM file

`CALL RVPOINT <ddname>, <key>, <rpl_options>` where

`<ddname>` Is a required parameter, coded as either a REXX variable or literal string within quotes, whose value is the DDname of the file to which the VSAM dataset is allocated.

`<key>` Is a required parameter which is either a REXX variable or a character string that represents an RBA (for a KSDS or ESDS), an RRN (for an RRDS) or a full or generic embedded key for a KSDS. Use an RBA as a key for an ESDS that is processed directly. For a KSDS, the key must be a character representation of an RBA when <rpl_options> specifies either the ADR or CNV options. When the KEY option is specified for a KSDS, an embedded full or generic key should be supplied on the call. Lastly, for an RRDS, the key must be a character representation of a relative record number (RRN).

`<rpl_options>` Specifies the VSAM RPL OPTCD processing options, with the syntax analogous to that described in the IBM VSAM documentation. The processing options must be separated by commas. The string of options may optionally be enclosed in parentheses. Option selection rules are defined by the IBM VSAM RPL OPTCD specifications. Refer to Section 1.4 for more details.

Usage:

The RVPOINT function requests that file positioning be altered for subsequent sequential processing. Once an RVPOINT has completed successfully, the application may issue an RVGET in order to retrieve that record. RVPOINT is useful when sequential retrieval is desired but the set of consecutive records does not start at the logical or physical beginning of the dataset.

The following REXX Feedback Variables are set on completion of RVPOINT:

RVRC Indicates a general return code from the RVPOINT service. It is set to zero when all VSAM and RLX/VSAM functions complete successfully.

RVMACRC Is set to the return code passed back to RLX/VSAM by the VSAM POINT macro. Note that this is independent of the RVRC variable setting.

RVREASON Is set to the reason code passed back to RLX/VSAM by the VSAM POINT macro via the RPL in the event of VSAM POINT failure. When successful, RVREASON is set to zero.

RVSKEY	Contains the value of the key of the currently positioned record. Note that this may be an embedded key, an RBA or an RRN depending on the file type and access mode.
RVRBA	Is set to the RBA of the record at which position is currently established. This is true for any file type or access mode.
RVRRN	Is set to the RRN of the positioned record. This is only set when an RRDS is accessed.
RVKEYL	Is set to the length of the key used for positioning. This variable only has meaning when accessing by embedded key.

Example of Coding:

```
/* Rexx */
...
CALL RVPOINT 'KSDSFILE','KEY012','(FWD,KGE)'
...
```

In this example, a KSDS allocated to the DDname KSDSFILE is positioned to the record with a key value equal to or greater than 'KEY012'. The search for a qualified record proceeds in a forward direction.

1.6.8 RVPUT - Write a record in a VSAM file

`CALL RVPUT <ddname>,<record>,<key>,<rpl_options>` where

`<ddname>` Is a required parameter, coded as either a REXX variable or literal string within quotes, whose value is the DDname of the file to which the VSAM dataset is allocated.

`<record>` Specifies either a REXX variable or a literal whose value is the string used to add, insert or update a record of the requested VSAM file. The record may be of a variable length, as long as this is appropriate for the VSAM file type and processing options in effect. For example, the length of records in an RRDS must be consistent with the RECORDSIZE sub parameter of the IDCAM's DEFINE function.

`<key>` Is a required parameter which is either a REXX variable or a character string that represents an RBA (for a KSDS or ESDS), an RRN (for an RRDS) or a full or generic embedded key for a KSDS. Use an RBA as a key for an ESDS that is processed directly. For a KSDS, the key must be a character representation of an RBA when <rpl_options> specifies either the ADR or CNV options. When the KEY option is specified for a KSDS, an embedded full or generic key should be supplied on the call. Lastly, for an RRDS, the key must be a character representation of a relative record number (RRN).

The <key> keyword parameter specifies the search key used to insert or update a record or CI. The key must be specified in most cases. The exceptions are when adding new records sequentially or when relying on a preceding RVGET to position to the required record by key (update mode). The key may be an RBA, RRN or embedded record key depending on the VSAM file type and processing options in effect. Any key field specified when it is not required is ignored.

`<rpl_options>` Specifies the VSAM RPL OPTCD processing options, with syntax analogous to that described in the IBM VSAM documentation. The processing options must be separated by commas. The string of options may optionally be enclosed in parentheses. Option selection rules are defined by the IBM VSAM RPL OPTCD specifications. Refer to Sections 1.3 and 1.4 for more details on ACB and RPL processing options.

Usage:

The RVPUT function writes a record or CI to a VSAM file. The record to be written is specified via the <record> parameter. For sequential processing no <key> parameter is required. The <key> specifies the key of the record to be inserted or updated. When updating, an RVGET call with the UPD <rpl_option> must precede the RVPUT call. In this case, the key need not be respecified on the RVPUT request.

The following REXX Feedback Variables are set on completion of RVPUT:

RVRC — Indicates a general return code from the RVPUT service. It is set to zero when all VSAM and RLX/VSAM functions complete successfully.

RVMACRC — Is set to the return code passed back to RLX/VSAM by the VSAM PUT macro. Note that this is independent of the RVRC variable setting.

RVREASON — Is set to the reason code passed back to RLX/VSAM by the VSAM PUT macro via the RPL in case of VSAM PUT failure. When successful, RVREASON is set to zero.

RVSKEY — Is set to the value of the key of the record just written. Note that this may be an embedded key, an RBA or an RRN depending on the file type and access mode.

RVRBA — Is set to the RBA of the record just written. This is true for any file type or access mode.

RVRRN: — Is set to the RRN of the record just written. This is only set when an RRDS is accessed.

RVKEYL — Is set to the length of the key used for the write. This variable only has meaning when accessing by embedded key.

Example of Coding:

```
/* Rexx */
...
CALL RVOPEN   'RRDSFILE','(KEY,DIR,IN,OUT)'
CALL RVPUT    'RRDSFILE','NEW RECORD',3,'(UPD,LOC)'
...
```

In this example, the third record slot in the RRDS is updated with a new value.

1.6.9 RVTERM - Terminate RLX/VSAM processing

`CALL RVTERM <files>` where

`<files>` Specifies which types of RLX/VSAM managed files are to be included in termination processing. The following are supported:

- `'VSAM'` any currently open RLX/VSAM managed VSAM files will be closed
- `'LOG'` closes the RLX/VSAM message log.
- `'ALL'` closes any open VSAM files as well as the message log. This is the default.

Usage:

The RVTERM function closes all currently open VSAM files that are being managed by RLX/VSAM. In addition, when the RVMSG FILE option is in effect, the message log is also closed. The RVTERM service should be called to cleanup after an error condition.

> ***NOTE:*** *Unless RVTERM is called, the last few log messages may be lost if FILE logging is in effect. The number of messages lost depends on the block size of the message file.*

The following REXX Feedback Variables are set on completion of RVTERM:

RVRC Indicates a general return code from the RVTERM service. It is set to zero when all VSAM and RLX/VSAM functions complete successfully.

Example of Coding:

```
/* Rexx */
...
CALL RVTERM 'ALL'
...
```

1.6.10 RVVERIFY - Run VERIFY on a VSAM file

`CALL RVVERIFY <ddname> [,REFRESH]` where

`<ddname>`	Is a required parameter, coded as either a REXX variable or literal string within quotes, whose value is the DDname of the file to which the VSAM dataset is allocated.
`REFRESH`	The optional REFRESH keyword requests that in addition to updating VSAM catalog statistics and high used RBA, VSAM should also update its in-storage control blocks to reflect new DASD extent information if the dataset was extended.

Usage:

The RVVERIFY function requests that VSAM perform catalog entry analysis and repair in case the last use of the dataset did not complete normal VSAM CLOSE processing. It performs the same functions as the IDCAMS VERIFY command. If the dataset is not OPEN when RVVERIFY is issued, RLX/VSAM issues an internal RVOPEN before requesting verification. After updating the catalog, RLX/VSAM issues an internal RVCLOSE. When the file is already opened, only the verify procedure is performed.

The following REXX Feedback Variables are set on completion of RVVERIFY:

RVRC	Indicates a general return code from the RVVERIFY service. It is set to zero when all VSAM and RLX/VSAM functions complete successfully.
RVMACRC	Is set to the return code passed back to RLX/VSAM by the VSAM VERIFY macro. Note that this is independent of the RVRC variable setting.
RVREASON	Is set to the reason code passed back to RLX/VSAM by the VSAM VERIFY macro via the RPL in case of VSAM VERIFY failure. When successful, RVREASON is set to 8 if catalog changes were required and to 0 if not.

1.7 Return codes

The following table enumerates the VSAM reason codes that may be returned in the RVREASON variable after an RLX/VSAM service is executed. Reason codes are grouped by the VSAM service (such as OPEN or CLOSE) which sets them. This section is provided for developer convenience only. It is not intended to supersede the IBM Messages and Codes manual nor any other VSAM publications.

OPEN reason codes:

Code	Condition
0	OPEN was successful
96	Unusable data set was opened for input
100	OPEN encountered an empty alternate index that is a part of an upgrade set
104	VTOC and catalog information are in conflict
108	Data and Index have been updated separately from each other
116	The data set was not properly closed and either OPEN's implicit verify was unsuccessful
118	The data set was not properly closed but OPEN's implicit verify was successfully executed
128	The data set was not properly allocated
132	Not enough storage to work areas or volume can't be mounted
136	Not enough virtual storage
140	The catalog indicates this data set has an invalid physical record size
144	An uncorrectable I/O error occurred while VSAM was reading or writing a catalog record
152	Authorization has failed
160	Invalid ACB options were specified in this context

CLOSE reason codes:

Code	Condition
0	CLOSE was successful
4	The data set is already closed

Record Management reason codes:

```
=======================================================================
|Code| Condition
|----+------------------------------------------------------------------
|  0 | CLOSE was successful
|  4 | VSAM is mounted another volume
|  8 | Duplicate Alternate Key exists
| 16 | The record is written into the new control area
| 28 | CI split indicator is detected
=======================================================================
```

Logical Errors:

```
=======================================================================
|Code| Condition
|----+------------------------------------------------------------------
|  4 | End of data set encountered
|  8 | Duplicate primary key, or alternate key with UNIQKEY option
| 12 | A key sequence error was detected using SEQ or SKP options
| 16 | Record not found, or the RBA is not found in the buffer pool
| 20 | Buffer is under exclusive control of another request
| 24 | Record resides on a volume that can't be mounted
| 32 | Specified RBA does not give the address of any data record in the
|    | data set
| 36 | Key is out of specified range for this data set
| 40 | Not enough virtual storage in your address space
| 68 | Type of processing does not match OPEN ACB options
| 72 | A Keyed request for access to an ESDS
| 76 | An Addressed access to RVPUT to a KSDS
| 80 | Request to RVERASE to an ESDS
| 88 | Request to RVGET without first establishing position, or
|    | illegal switch between forward and backward processing
| 92 | Request to update/delete the data record (RVPUT/RVERASE) without
|    | previous RVGET with 'UPD' RPL option
| 96 | Request to change key while making an update
|100 | Request to change the length of the record while making
|    | an addressed update
|108 | The record length is invalid
|112 | The key length is invalid (too large or equal to 0)
|116 | Request issued to function other than RVPUT to insert a record
|    | during initial load
=======================================================================
```

Chapter 2

Annotated RLX / VSAM

Coding Examples

This Chapter presents various RLX/VSAM coding examples which illustrate access to ESDS, KSDS and RRDS VSAM file types. The sample REXX EXECs illustrate record retrieval in sequential, direct and mixed modes as well as access by key and RBA (Relative Byte Address). The REXX source for all these examples is present in the RLXEXEC library.

Summary of Examples

2.1 VSAM ESDS Sequential Read
2.2 VSAM KSDS Direct record access and delete
2.3 VSAM ESDS Direct access
2.4 VSAM KSDS Record insert
2.5 VSAM KSDS Direct record retrieval and update in place
2.6 VSAM RRDS Direct record retrieval and deletion
2.7 VSAM RRDS Insert record
2.8 VSAM RRDS Direct retrieval and update in place
2.9 VSAM KSDS Direct positioning and sequential read
2.10 VSAM RRDS Direct positioning and sequential read
2.11 VSAM KSDS Sequential Search and Update
2.12 VSAM RRDS Sequential Search and Update

NOTE: *The RLXEXEC library contains additional RLX/VSAM coding examples whose names all start with the characters RXDVxxxx.*

(This page left intentionally blank)

The two EXECs illustrated in Figure 2.0 ('Allocated' and 'RvError') are *common routines* called by all the examples. The routine named Allocated is invoked as a function to allocate the VSAM dataset whose name it receives as an argument. The file name used in all cases is 'DDname' - the same name referenced on all the RLX/VSAM services. If successful, the Allocated routine returns a 'True' value of 1 but if unsuccessful returns a 'False' value of 0.

```
 Allocated:
 Arg DAname
 address tso "Alloc fi(DDname) da("DAname") shr reuse"
   say 'Allocate file for 'DAname', Rc='Rc
   If Rc \=Ø then Do
      say 'File Allocation Error!'
      return Ø
   End
Return 1
```

The second routine, RvError, is called for general error reporting in the event an RLX/VSAM function completes with a non-zero return code.

```
RvError :
 Arg RvFuncName
 say 'Error in the 'RvFuncname' function:'
 say 'RvReason = ' RvReason
 say 'RvMacRc  = ' RvMacRc
 say 'RvRc     = ' RvRc

 call RvClose 'DDname'
 address tso "free fi("DDname")"
Exit RvRc
```

Figure 2.0 Common routines Allocated and RvError.

2.1 VSAM ESDS Sequential Read

The following EXEC accesses a VSAM ESDS sequentially, one record at a time, until end-of-file is reached. In the figure, the numbers within comments in the right-hand margin (for example, /* 1 */) correspond to the numbered annotations which follow the figure.

```
/* REXX */

 VSAM_file_name = 'RAI.VSAM.ESDSDEMO.CLUSTER'                 /* 1  */

 If \Allocated(VSAM_file_name) then return 12                 /* 2  */

 call RvMsg 'NONE'                                            /* 3  */
 If RvRc \=Ø then call RvError 'RvMsg'                        /* 4  */

 call RvOpen 'DDname','(SEQ,IN)'                              /* 5  */
 If RvRc \=Ø then call RvError 'RvOpen'

 Do forever                                                   /* 6  */
    Call RvGet 'DDname',,'ADR,LOC'                            /* 7  */
    If RvRc \=Ø then call RvError 'RvGet'
    If RvEof = 'EOF' then do                                  /* 8  */
       say 'End of the DSN='VSAM_file_name' is encountered'
       leave
    end
    say RvRecord                                              /* 9  */
 End

 call RvClose 'DDname'                                        /* 1Ø */
 If RvRc \=Ø then call RvError 'RvClose'

 address tso "free fi("DDname")"                              /* 11 */
Return Ø
```

Figure 2.1 VSAM ESDS Direct access

1. Assign the name of the VSAM ESDS cluster to be processed to the REXX variable "VSAM_file_name".

2. Attempt to allocate the VSAM ESDS cluster to the file named DDname. If allocation fails (denoted by \Allocated), then terminate with return code 12.

3. Suppress messages from all RLX/VSAM functions.

4. Check REXX VSAM function completion by inspecting variable RvRc for a non-zero value.

5. Open the VSAM ESDS allocated to the file "DDname" with the following ACB options:

   "SEQ" denotes sequential access to the ESDS
   "IN" indicates records will be retrieved for read only

6. The loop executes until end-of-file.

7. The "RvGet" function retrieves the VSAM ESDS record from the file named "DDname" into the REXX variable named "RvRecord". Sequential retrieval is requested by the default RPL option 'SEQ'.

8. The REXX variable "RvEof" contains the value "EOF" when end-of-file is reached. Otherwise, it contains blanks.

9. The "RvGet" function returns the contents of the record just retrieved into the REXX variable "RvRecord".

10. Close the file.

11. Free the file whose DD name is 'DDname'.

2.2 VSAM KSDS Direct record access and delete

In the following EXEC (Figure 2.2) a VSAM KSDS is accessed directly via its record key. Once accessed, the record is deleted.

```
/* REXX */

 VSAM_file_name = 'RAI.VSAM.KSDSDEMO.CLUSTER'                   /* 1  */
 SearchKey = 'FISH003'                                          /* 2  */

 If \Allocated(VSAM_file_name) then return 12                   /* 3  */

 call RvMsg 'NONE'                                              /* 4  */
 If RvRC \=0 then call RvError 'RvMsg'                          /* 5  */

 call RvOpen 'DDname', '(KEY,DIR,IN,OUT)'                       /* 6  */
 If RvRc \=0 then call RvError 'RvOpen'

 SearchKey = Left(SearchKey,RvKeySz,' ')                        /* 7  */

 Call RvGet 'DDname',SearchKey,'DIR,UPD,LOC'                    /* 8  */
 If RvRc \=0 then
    call RvError 'RvGet'
 else do
    say 'Record = 'RvRecord                                     /* 9  */

    call RvErase 'DDname'                                       /* 10 */
    If RvRc \=0 then call RvError 'RvErase'
       else say 'Record deleted!'
 End

 call RvClose 'DDname'                                          /* 11 */
 If RvRc \=0 then call RvError 'RvClose'

 address tso "free fi("DDname")"                                /* 12 */
Return 0
```

Figure 2.2

1. Assign the name of the VSAM KSDS cluster to be processed to the REXX variable "VSAM_file_name".

2. Initialize "SearchKey", the REXX variable that contains the value of the key of the record to be updated.

3. Attempt to allocate the VSAM KSDS cluster to the file named DDname. If allocation fails (denoted by \Allocated), then terminate with return code 12.

4. Suppress messages from all RLX/VSAM functions.

5. Check REXX VSAM function completion by inspecting variable RvRc for a non-zero value.

6. Open the VSAM KSDS allocated to the file "DDname" with the following processing options:

"KEY"	The record will be retrieved by KEY value
"DIR"	Direct access will be used
"IN"	records will be retrieved
"OUT"	records can be rewritten

7. The "SearchKey" is padded with blanks on the right to the KSDS key length.

8. The "RvGet" function returns the record in the REXX variable "RvRecord". The 'DIR' RPL option requests direct retrieval using the value of the REXX variable "SearchKey" as a key value. The RPL option 'UPD' allows the retrieved record to be updated or deleted.

9. The REXX variable "RvRecord" contains the VSAM record just retrieved by the "RvGet" function.

10. The "RvErase" function deletes the record just retrieved by RvGet.

11. Close the file named "DDname".

12. Free the file name 'DDname'.

2.3 VSAM ESDS Direct access

In the following exec (Figure 2.3) a VSAM ESDS is accessed randomly via its relative byte address. This requires a method to compute a record's RBA. Since the the ESDS contains only fixed length records, the RBA can be computed from a relative record number (RRN).

```
/* REXX */
 VSAM_file_name = 'RAI.VSAM.ESDSDEMO.CLUSTER'                 /* 1  */
 SearchRRN  = 3                                                /* 2  */
 RecUpd     = 'My new updated record.'                         /* 3  */
 If \Allocated(VSAM_file_name) then return 12                  /* 4  */
 call RvMsg 'NONE'                                             /* 5  */
 If RvRC \=Ø then call RvError 'RvMsg'                         /* 6  */

 call RvOpen 'DDname', '(ADR,DIR,IN,OUT)'                      /* 7  */
 If RvRc \=Ø then call RvError 'RvOpen'

 SearchRBA = SearchRRN * RvMaxLr
 Call RvGet 'DDname',SearchRBA,'ADR,DIR,UPD,LOC'               /* 8  */
 If RvRc \=Ø then
    call RvError 'RvGet'
 else do
    say 'Record = 'RvRecord                                    /* 9  */
    RecUpd = Left(RecUpd,Length(RvRecord))                     /* 1Ø */
    call RvPut 'DDname', RecUpd,,'(LOC)'                       /* 11 */
    If RvRc \=Ø then call RvError 'RvPut'
       else say 'Record updated!'
 End

 call RvClose 'DDname'                                         /* 12 */
 If RvRc \=Ø then call RvError 'RvClose'

 address tso "free fi("DDname")"                               /* 13 */
Return Ø
```

Figure 2.3

1. Assign the name of the VSAM ESDS cluster to be processed to the REXX variable "VSAM_file_name".

2. Initialize "SearchRRN", the REXX variable that contains the RRN (Relative Record Number) of the record to be updated.

3. Initialize "RecUpd", the REXX variable that contains the new value for a record.

4. Attempt to allocate the VSAM ESDS cluster to the file named DDname. If allocation fails (denoted by \Allocated), then terminate with return code 12.

5. Suppress messages from all RLX/VSAM functions.

6. Check REXX VSAM function completion by inspecting variable RvRc for a non-zero value.

7. Open the VSAM ESDS allocated to the file "DDname" with the following ACB processing options:

"ADR"	The record will be retrieved via RBA
"DIR"	Direct access is requested
"IN,OUT"	File records can be read, updated and deleted

8. The "RvGet" function retrieves a record from the VSAM ESDS into the REXX variable "RvRecord". Direct retrieval is requested by the RPL option 'DIR'. The REXX variable "SearchRBA" contains the RBA while the RvGet function utilizes the following RPL (Request Parameter List) options:

   'UPD' Update is allowed
   'ADR' RBA is used as a search argument

9. The REXX variable "RvRecord" contains the VSAM record retrieved by a successful "RvGet" function.

10. "RecUpd" is padded with blanks to fill the entire record.

11. The "RvPut" function does an update in place of the record retrieved by the preceding "RvGet" function.

12. "RvClose" closes the file named 'DDname'.

13. The File allocated in item 4 is freed.

2.4 VSAM KSDS record insert

In this example (Figure 2.4) a record is inserted into a VSAM KSDS file.

```
/* REXX */
 VSAM_file_name = 'RAI.VSAM.KSDSDEMO.CLUSTER'                 /*  1 */
 SearchKey = 'FISH999'                                        /*  2 */
 RecIns    = 'My inserted record.'                            /*  3 */

 If \Allocated(VSAM_file_name) then return 12                 /*  4 */

 call RvMsg 'NONE'                                            /*  5 */
 If RvRC \=0 then call RvError 'RvMsg'                        /*  6 */

 call RvOpen 'DDname', '(KEY,DIR,OUT)'                        /*  7 */
 If RvRc \=0 then call RvError 'RvOpen'

 SearchKey = Left(SearchKey,RvKeySz,' ')                      /*  8 */

 RecIns = SearchKey || RecIns                                 /*  9 */

 call RvPut 'DDname', RecIns, SearchKey,'(DIR,LOC)'           /* 10 */
 If RvRc \=0 then call RvError 'RvPut'
    else say 'Record inserted!'

 call RvClose 'DDname'                                        /* 11 */
 If RvRc \=0 then call RvError 'RvClose'

 address tso "free fi("DDname")"                              /* 12 */
Return 0
```

Figure 2.4

1. Assign the name of the VSAM KSDS cluster to be processed to the REXX variable "VSAM_file_name".

2. Initialize "SearchKey", the REXX variable that contains the value of the key of the record to be inserted.

3. Initialize "RecIns", the REXX variable that contains the value of the new KSDS record. At this point the REXX variable RecIns does not contain the key-part of the record. That will be concatenated to RecIns in step 9.

4. Attempt to allocate the VSAM KSDS cluster to the file named DDname. If allocation fails (denoted by \Allocated), then terminate with return code 12.

5. Suppress messages from all RLX/VSAM functions.

6. Check REXX VSAM function completion by inspecting variable RvRc for a non-zero value.

7. Open the VSAM KSDS allocated to the file "DDname" with the following ACB processing options:

"KEY"	records will be retrieved by KEY
"DIR"	Direct access to a KSDS
"OUT"	new records can be written to the dataset but updating existing records is not allowed

8. The "SearchKey" is padded with blanks on the right to the length of the KSDS key field.

9. Concatenate the Key value ahead of the data portion of the record in REXX variable "RecIns".

10. The "RvPut" function writes the new record "RecIns" to the file named "DDname".

11. Close the file.

12. Free the file whose DD name is 'DDname'.

2.5 VSAM KSDS direct retrieval and update in place

In this example (Figure 2.5) a VSAM KSDS is accessed directly and updated in place.

```
/* REXX */
 VSAM_file_name = 'RAI.VSAM.KSDSDEMO.CLUSTER'                  /* 1  */
 SearchKey = 'FISH003'                                         /* 2  */
 RecUpd    = 'My new updated record.'                          /* 3  */

 If \Allocated(VSAM_file_name) then return 12                  /* 4  */

 call RvMsg 'NONE'                                             /* 5  */
 If RvRC \=0 then call RvError 'RvMsg'                         /* 6  */

 call RvOpen 'DDname', '(KEY,DIR,IN,OUT)'                      /* 7  */
 If RvRc \=0 then call RvError 'RvOpen'

 SearchKey = Left(SearchKey,RvKeySz,' ')                       /* 8  */

 Call RvGet 'DDname',SearchKey,'DIR,UPD,LOC'                   /* 9  */
 If RvRc \=0 then
    call RvError 'RvGet'
 else do
    say 'Record = 'RvRecord                                    /* 10 */
    RecUpd = SearchKey || RecUpd                               /* 11 */

    call RvPut 'DDname', RecUpd,,'(LOC)'                       /* 12 */
    If RvRc \=0 then call RvError 'RvPut'
       else say 'Record updated!'
 End

 call RvClose 'DDname'                                         /* 13 */
 If RvRc \=0 then call RvError 'RvClose'

 address tso "free fi("DDname")"                               /* 14 */
Return 0
```

Figure 2.5

1. Initialize "VSAM_file_name", the REXX variable that contains the name of the VSAM KSDS cluster to process.

2. Initialize "SearchKey", the REXX variable that contains the key value of the record to be updated.

3. Initialize "RecUpd", the REXX variable that contains the updated record value. At this point, RecUpd does not yet contain the key-portion of the record. This will be concatenated to RecUpd at step 12.

4. Attempt to allocate the VSAM KSDS cluster to file named DDname. If allocation fails (denoted by \Allocated), then terminate with return code 12.

5. Suppress messages from all RLX/VSAM functions.

6. Check REXX VSAM function completion by inspecting variable RvRc for a non-zero value.

7. Open the file with the following ACB options:

"KEY"	The record will be retrieved via KEY value
"DIR"	Direct access to the KSDS
"IN,OUT"	Records can be updated and deleted

8. REXX variable "SearchKey" is padded with blanks on the right to the length of the KSDS key field.

9. The "RvGet" function retrieves a record from the VSAM KSDS cluster into the REXX variable "RvRecord". The RPL option 'DIR' supports direct retrieval of records whose key value equals the values of the REXX variable "SearchKey". The record can be updated because the RPL option specifies 'UPD'.

10. The REXX variable "RvRecord" contains the VSAM record retrieved by the successful execution of the "RvGet" function.

11. "RecUpd" contains the "updated record" with the Key value at the beginning of the record.

12. The "RvPut" function writes a record from the "RecUpd" variable , overlaying the record retrieved by the RvGet function.

13. Close the file "DDname".

14. Free the file named 'DDname'.

2.6 VSAM RRDS direct record retrieval and deletion

In this example (Figure 2.6) a record in a VSAM RRDS whose relative record number is 3 is accessed directly and then deleted via the RvErase function. This effectively frees a slot in the RRDS.

```
/* REXX */
 VSAM_file_name = 'RAI.VSAM.RRDSDEMO.CLUSTER'                   /* 1  */
 SearchRRN = 3                                                  /* 2  */
 If \Allocated(VSAM_file_name) then return 12                   /* 3  */
 call RvMsg 'NONE'                                              /* 4  */
 If RvRC \=Ø then call RvError 'RvMsg' /* 5 */
 call RvOpen 'DDname', '(KEY,DIR,IN,OUT)'                       /* 6  */
 If RvRc \=Ø then call RvError 'RvOpen'
 Call RvGet 'DDname',SearchRRN,'DIR,UPD,LOC'                    /* 7  */
 If RvRc \=Ø then
    call RvError 'RvGet'
 else do
    say 'Record = 'RvRecord                                     /* 8  */
    call RvErase 'DDname'                                       /* 9  */
    If RvRc \=Ø then call RvError 'RvErase'
       else say 'Record deleted!'
 End
 call RvClose 'DDname'                                          /* 1Ø */
 If RvRc \=Ø then call RvError 'RvClose'
 address tso "free fi("DDname")"                                /* 11 */
Return Ø
```

Figure 2.6

1. Initialize "VSAM_file_name", the REXX variable that contains the name of the VSAM RRDS cluster to process.

2. Initialize the REXX variable "SearchRRN" to contain the value of the key of the record to be updated.

3. Attempt to allocate the VSAM RRDS to the file named DDname. If allocation fails (denoted by \Allocated), then terminate with return code 12.

4. Suppress messages from all RLX/VSAM functions.

5. Check REXX VSAM function completion by inspecting variable RvRc for a non-zero value.

6. Open the file with the following ACB options:

"KEY"	records will be retrieved via their RRNs
"DIR"	Direct access to a RRDS
"IN,OUT"	Records can be updated and deleted

7. The "RvGet" function retrieves a record from the VSAM RRDS cluster into the REXX variable "RvRecord". The RPL option 'DIR' supports direct retrieval of records whose key values are equal to the values of the REXX variable "SearchRRN". The record can be updated or deleted due to the UPD' RPL option.

8. The REXX variable "RvRecord" contains the VSAM record retrieved by the "RvGet" function.

9. The "RvErase" function deletes the record just retrieved by an "RvGet" issued with the 'UPD' RPL option (see 7).

10. Close the file.

11. Free the file named 'DDname'.

2.7 VSAM RRDS insert record

In this example (Figure 2.7) a record is inserted into a slot of a VSAM RRDS.

```
/* REXX */
 VSAM_file_name = 'RAI.VSAM.RRDSDEMO.CLUSTER'                 /*  1 */
 SearchRRN = 10                                              /*  2 */
 RecIns    = 'My inserted record.'                           /*  3 */
 If \Allocated(VSAM_file_name) then return 12                /*  4 */
 call RvMsg 'NONE'                                           /*  5 */
 If RvRC \=0 then call RvError 'RvMsg'                       /*  6 */
 call RvOpen 'DDname', '(KEY,DIR,OUT)'                       /*  7 */
 If RvRc \=0 then call RvError 'RvOpen'
 RecIns = Left(RecIns,RvMaxLr)                               /*  8 */
 call RvPut 'DDname', RecIns, SearchRRN,'(DIR,LOC)'          /*  9 */
 If RvRc \=0 then call RvError 'RvPut'
    else say 'Record inserted!'
 call RvClose 'DDname'                                       /* 10 */
 If RvRc \=0 then call RvError 'RvClose'
 Address tso "free fi("DDname")"                             /* 11 */
 Return 0
```

Figure 2.7

1. Initialize "VSAM_file_name", the REXX variable that contains the name of the VSAM RRDS cluster to process.

2. Initialize the REXX variable "SearchRRN" to contain the value of the key of the record to be inserted.

3. Initialize "RecIns", the REXX variable that contains the value of the new RRDS record.

4. Attempt to allocate the VSAM RRDS to the file named DDname. If allocation fails (denoted by \Allocated), then terminate with return code 12.

5. Suppress messages from all RLX/VSAM functions.

6. Check REXX VSAM function completion by inspecting variable RvRc for a non-zero value.

7. Open the file with the following ACB options:

"KEY"	records will be retrieved by their KEY values
"DIR"	Direct access to an RRDS
"OUT"	new records can be inserted but existing records cannot be updated

8. "RecIns" is padded with blanks to the length of the record in the RRDS. This length is defined by the REXX variable "RvMaxLr" and is initialized by the "RvOpen" function.

9. The "RvPut" function inserts the new record "RecIns" into the file allocated to "DDname".

10. Close the file.

11. Free the file named 'DDname'.

2.8 VSAM RRDS Direct retrieval and update in place

In this example (Figure 2.8) a record is retrieved from a VSAM RRDS and then updated in place.

```
/* REXX */
 VSAM_file_name = 'RAI.VSAM.RRDSDEMO.CLUSTER'                   /*  1 */
 SearchRRN  = 3                                                 /*  2 */
 RecUpd     = 'My new updated record.'                          /*  3 */
 If \Allocated(VSAM_file_name) then return 12                   /*  4 */
 call RvMsg 'NONE'                                              /*  5 */
 If RvRC \=0 then call RvError 'RvMsg'                          /*  6 */
 call RvOpen 'DDname', '(KEY,DIR,IN,OUT)'                       /*  7 */
 If RvRc \=0 then call RvError 'RvOpen'
 Call RvGet 'DDname',SearchRRN,'DIR,UPD,LOC'                    /*  8 */
 If RvRc \=0 then
    call RvError 'RvGet'
 else do
    say 'Record = 'RvRecord                                     /*  9 */
    RecUpd = Left(RecUpd,length(RvRecord))                      /* 10 */
    call RvPut 'DDname', RecUpd,,'(LOC)'                        /* 11 */
    If RvRc \=0 then call RvError 'RvPut'
       else say 'Record updated!'
 End
 call RvClose 'DDname'                                          /* 12 */
 If RvRc \=0 then call RvError 'RvClose'
 address tso "free fi("DDname")"                                /* 13 */
Return 0
```

Figure 2.8

1. Initialize "VSAM_file_name", the REXX variable that contains the name of the VSAM RRDS cluster to process.

2. Initialize the REXX variable "SearchRRN" to contain the value of the key of the record to be updated.

3. Initialize "RecUpd", the REXX variable that contains the value of the updated record.

4. Attempt to allocate the VSAM RRDS to the file named DDname. If allocation fails (denoted by \Allocated), then terminate with return code 12.

5. Suppress messages from all RLX/VSAM functions.

6. Check REXX VSAM function completion by inspecting variable RvRc for a non-zero value.

7. Open the file with the following ACB options:

"KEY"	records will be retrieved via their RRNs
"DIR"	Direct access to the file
"IN,OUT"	Records can updated and deleted

8. The "RvGet" function retrieves the record from the VSAM RRDS cluster into the REXX variable "RvRecord". Direct retrieval of records is indicated by the 'DIR' RPL option. The key value is equal to the value of the REXX variable "SearchRRN". Records can be changed due to the RPL option 'UPD'.

9. The REXX variable "RvRecord" contains the VSAM record retrieved by a successful RvGet" request.

10. "RecUpd" is padded with blanks to the cluster record length.

11. The"RvPut" function writes the record from the "RecUpd" variable, overlaying the last record retrieved.

12. Close the file.

13. Free the file named 'DDname'.

2.9 VSAM KSDS Direct positioning and sequential read

In this example (Figure 2.9) RLX/VSAM establishes an initial position in a VSAM file at the record key 'FISH003' and then processes records sequentially until end-of-file is reached.

```
/* REXX */
 VSAM_file_name = 'RAI.VSAM.KSDSDEMO.CLUSTER'                  /*  1 */
 SearchKey = 'FISH003'                                         /*  2 */
 If \Allocated(VSAM_file_name) then return 12                  /*  3 */
 call RvMsg 'NONE'                                             /*  4 */
 If RvRC \=0 then call RvError 'RvMsg'                         /*  5 */
 call RvOpen 'DDname', '(KEY,SEQ,IN)'                          /*  6 */
 If RvRc \=0 then call RvError 'RvOpen'
 SearchKey = Left(SearchKey,RvKeySz,' ')                       /*  7 */
 Call RvPoint 'DDname',SearchKey,'KEQ,LOC'                     /*  8 */
 If RvRc \=0 then call RvError 'RvPoint'

 Do forever
    Call RvGet 'DDname',,'LOC'                                 /*  9 */
    If RvRc \=0 then call RvError 'RvGet'
    If RvEof = 'EOF' then leave                                /* 10 */
    say 'Record = 'RvRecord                                    /* 11 */
 End

 call RvClose 'DDname'                                         /* 12 */
 If RvRc \=0 then call RvError 'RvGet'

 address tso "free fi("DDname")"                               /* 13 */
Return 0
```

Figure 2.9

1. Initialize "VSAM_file_name", the REXX variable that contains the name of the VSAM KSDS cluster to process.

2. Initialize the REXX variable "SearchKey" to contain the value of the key of the record to be updated.

3. Attempt to allocate the VSAM KSDS to the file named DDname. If allocation fails (denoted by \Allocated), then terminate with return code 12.

4. Suppress messages from all RLX/VSAM functions.

5. Check REXX VSAM function completion by inspecting variable RvRc for a non-zero value.

6. Open the file with the following ACB options:

"KEY"	records will be retrieved via their KEY
"SEQ"	sequential access to the records of a KSDS
"IN"	records will be retrieved on a READ only basis

7. The "SearchKey" is padded with blanks on the right to the length of the KSDS key. The REXX variable "RvKeySz" contains the file's key length after the RvOpen request completes successfully.

8. The "RvPoint" function establishes file position at a record identified by the key (the REXX variable "SearchKey"). Equal comparison is requested via the RPL option 'KEQ'. The alternative option, KGE, requests a greater or equal comparison.

9. The "RvGet" function retrieves the KSDS record into the REXX variable "RvRecord". The RPL option 'SEQ' requests sequential record retrieval while the default RPL option 'NUP' prevents records from being updated.

10. The REXX variable "RvEof" is set to "EOF" when end-of-file is reached. Otherwise, it contains blanks.

11. The REXX variable "RvRecord" contains the VSAM record retrieved by a successful RvGet" request.

12. Close the file.

13. Free the file named 'DDname'.

2.10 VSAM RRDS direct positioning and sequential read

This example (Figure 2.10) is similar to example 2.9, in that an RRDS is positioned to the designated relative record number and then processed sequentially until end-of-file is reached.

```
/* REXX */
 VSAM_file_name = 'RAI.VSAM.RRDSDEMO.CLUSTER'                   /*  1 */
 SearchRRN = 3                                                  /*  2 */
 If \Allocated(VSAM_file_name) then return 12                   /*  3 */
 call RvMsg 'NONE'                                              /*  4 */
 If RvRC \=Ø then call RvError 'RvMsg'                          /*  5 */
 call RvOpen 'DDname', '(KEY,SEQ,IN)'                           /*  6 */
 If RvRc \=Ø then call RvError 'RvOpen'
 Call RvPoint 'DDname',SearchRRN,'KEQ,LOC'                      /*  7 */
 If RvRc \=Ø then call RvError 'RvPoint'

 Do forever
    Call RvGet 'DDname',,'LOC'                                  /*  8 */
    If RvRc \=Ø then call RvError 'RvGet'
    If RvEof = 'EOF' then leave                                 /*  9 */
    say 'Record = 'RvRecord                                     /* 1Ø */
 End

 call RvClose 'DDname'                                          /* 11 */
 If RvRc \=Ø then call RvError 'RvGet'
 address tso "free fi("DDname")"                                /* 12 */
Return Ø
```

Figure 2.10

1. Initialize "VSAM_file_name", the REXX variable that contains the name of the VSAM RRDS cluster to process.

2. Initialize the REXX variable "SearchRRN" to contain the value of the RRN of the record to be updated.

3. Attempt to allocate the VSAM RRDS to the file named DDname. If allocation fails (denoted by \Allocated), then terminate with return code 12.

4. Suppress messages from all RLX/VSAM functions.

5. Check REXX VSAM function completion by inspecting variable RvRc for a non-zero value.

6. Open the file with the following ACB options:

"KEY"	records will be retrieved via RRN
"SEQ"	Sequential access to a RRDS
"IN"	records will be retrieved just for READ

7. The "RvPoint" function establishes the file position at a record identified by the RRN search argument (REXX variable "SearchRRN"). Equal comparison is requested via the RPL option 'KEQ'. The alternative option, 'KGE', requests a greater or equal comparison.

8. The "RvGet" function retrieves the KSDS record into the REXX variable "RvRecord". The RPL option 'SEQ' requests sequential record retrieval while the default RPL option 'NUP' prevents records from being updated.

9. The REXX variable "RvEof" is set to "EOF" when end-of-file is reached. Otherwise, it contains blanks.

10. The REXX variable "RvRecord" contains the VSAM record retrieved by a successful "RvGet" function.

11. Close the file.

12. Free the file named 'DDname'.

2.11 VSAM KSDS Sequential Search and Update

In this example (Figure 2.11) a KSDS is searched sequentially for a specific record to be updated.

```
/* REXX */
 VSAM_file_name = 'RAI.VSAM.KSDSDEMO.CLUSTER'                /*  1 */
 SearchKey = 'FISH003'                                       /*  2 */
 RecUpd    = 'My new updated record.'                        /*  3 */
 If \Allocated(VSAM_file_name) then return 12                /*  4 */
 call RvMsg 'NONE'                                           /*  5 */
 If RvRc \=0 then call RvError 'RvMsg'                       /*  6 */
 call RvOpen 'DDname','(KEY,SEQ,IN,OUT)'                     /*  7 */
 If RvRc \=0 then call RvError 'RvOpen'
 SearchKey = Left(SearchKey,RvKeySz,' ')                     /*  8 */

 Do forever                                                  /*  9 */
    Call RvGet 'DDname',,'UPD,LOC'                           /* 10 */
    If RvRc \=0 then call RvError 'RvGet'
    If RvEof = 'EOF' then do                                 /* 11 */
       say 'End of the DSN='VSAM_file_name' is encountered'
       leave
    end

    say RvRecord                                             /* 12 */
    If RvSKey = SearchKey then call UpdateRecord             /* 13 */
 End
 call RvClose 'DDname'                                       /* 14 */
 If RvRc \=0 then call RvError 'RvClose'
 address tso "free fi("DDname")"                             /* 15 */
Return 0                                                     /* 16 */

UpdateRecord :                                               /* 17 */
 RecUpd = SearchKey || RecUpd                                /* 18 */
 call RvPut 'DDname', RecUpd,,'(loc)'                        /* 19 */
 If RvRc \=0 then
    call RvError 'RvPut'
 else
    say 'Record updated, Rc='rc
return
```

Figure 2.11

1. Initialize "VSAM_file_name", the REXX variable that contains the name of the VSAM KSDS cluster to process.

2. Initialize the REXX variable "SearchKey" to contain the value of the key of the record to be updated.

3. Initialize the REXX variable "RecUpd" to contain the value of the new record.

4. Attempt to allocate the VSAM KSDS to the file named DDname. If allocation fails (denoted by \Allocated), then terminate with return code 12.

5. Suppress messages from all RLX/VSAM functions.

6. Check REXX VSAM function completion by inspecting variable RvRc for a non-zero value.

7. Open the file with the following ACB options:

"KEY"	records will be retrieved by KEY value
"SEQ"	Sequential access to a KSDS
"IN,OUT"	records can be read and updated

8. "SearchKey" is padded with blanks on the right to the KSDS key length.

9. The loop executes until end-of-file.

10. The "RvGet" function retrieves the KSDS record into the REXX variable "RvRecord". The RPL option 'SEQ' requests sequential record retrieval while the RPL option 'UPD' allows records to be updated.

11. The REXX variable "RvEof" is set to "EOF" when end-of-file is reached. Otherwise, it contains blanks.

12. The REXX variable "RvRecord" contains the VSAM record retrieved by a successful "RvGet" function.

13. If the value of the key of the currently retrieved record is equal to the REXX variable "SearchKey" then the record will be updated. When the "RvGet" function completes successfully, it returns the value of the key of the last retrieved record into the REXX variable "RvSKey".

14. Close the file.

15. Free the file named 'DDname'.

16. Normal completion of the exec.

17. Record update routine begins.

18. "RecUpd" contains the "new record" with the key value at the beginning.

19. The "RvPut" function writes the record from the variable "RecUpd". The record retrieved by the RvGet function with RPL option 'UPD' will be overlaid with the new value.

2.12 VSAM RRDS Sequential Search and Update

In this example (Figure 2.12) an RRDS is searched sequentially for a specific record to be updated.

```
/* REXX */
 VSAM_file_name = 'RAI.VSAM.RRDSDEMO.CLUSTER'                  /*  1 */
 SearchRRN = 3                                                 /*  2 */
 RecUpd    = 'My new updated record.'                          /*  3 */
 If \Allocated(VSAM_file_name) then return 12                  /*  4 */
 call RvMsg 'NONE'                                             /*  5 */
 If RvRc \=Ø then call RvError 'RvMsg'                         /*  6 */
 call RvOpen 'DDname','(KEY,SEQ,IN,OUT)'                       /*  7 */
 If RvRc \=Ø then call RvError 'RvOpen'

 Do forever                                                    /*  8 */
    Call RvGet 'DDname',,'UPD,LOC'                             /*  9 */
    If RvRc \=Ø then call RvError 'RvGet'
    If RvEof = 'EOF' then do                                   /* 1Ø */
       say 'End of the DSN='VSAM_file_name' is encountered'
       leave
    end
    say RvRecord                                               /* 11 */
    If RvSKey = SearchRRN then call UpdateRecord               /* 12 */
 End

 call RvClose 'DDname'                                         /* 13 */
 If RvRc \=Ø then call RvError 'RvClose'
 address tso "free fi("DDname")"                               /* 14 */
Return Ø                                                       /* 15 */

UpdateRecord :                                                 /* 16 */
 RecUpd = Left(RecUpd,length(RvRecord))                        /* 17 */

 call RvPut 'DDname', RecUpd,,'(loc)'                          /* 18 */
 If RvRc \=Ø then
    call RvError 'RvPut'
 else
    say 'Record updated, Rc='rc
return
```

Figure 2.12

1. Initialize "VSAM_file_name", the REXX variable that contains the name of the VSAM RRDS cluster to process.

2. Initialize the REXX variable "SearchRRN" to contain the value of the RRN of the record to be updated.

3. Initialize the REXX variable "RecUpd" to contain the value of the new record.

4. Attempt to allocate the VSAM RRDS to the file named DDname. If allocation fails (denoted by \Allocated), then terminate with return code 12.

5. Suppress messages from all RLX/VSAM functions.

6. Check REXX VSAM function completion by inspecting variable RvRc for a non-zero value.

7. Open the file with the following ACB options:

"KEY" -	records will be retrieved via their RRN
"SEQ"	Sequential access is requested
"IN,OUT"	Records can be updated and deleted

8. The loop executes until end-of-file.

9. The "RvGet" function retrieves the RRDS record into the REXX variable "RvRecord". The RPL option 'SEQ' requests sequential record retrieval while the RPL option 'UPD' allows records to be updated.

10. The REXX variable "RvEof" is set to "EOF" when end-of-file is reached. Otherwise, it contains blanks.

11. The REXX variable "RvRecord" contains the VSAM record retrieved by a successful "RvGet" function.

12. If the value of the key of the currently retrieved record is equal to the REXX variable "SearchRRN", then the record will be updated. When the "RvGet" function completes successfully, it returns the value of the key of the last retrieved record into the REXX variable "RvSKey".

13. Close the file.

14. Free the file named 'DDname'.

15. Normal completion of the exec.

16. Record update routine begins.

17. "RecUpd" contains the "new record" with the key value at the beginning.

18. The "RvPut" function writes the record from the variable "RecUpd". The record retrieved by the RvGet function with RPL option 'UPD' will be overlaid with the new contents of the record.

Chapter 3

ISPF Dialogs for RLX/VSAM

This chapter discusses the examples provided by RAI to demonstrate the use of RLX/ VSAM functions. These can serve as a tutorial for users who wish to explore the full range of RLX/ VSAM functions and their applicability. The RLX/VSAM dialogs let you exercise the RLX/VSAM REXX functions and learn their syntax. Working examples involving all VSAM file types are presented.

Section 3.1 describes the VSAM sample datasets and how they are loaded.

Section 3.2 discusses procedures to add, change and delete records within VSAM datasets. Examples are provided for VSAM ESDS, KSDS and RRDS datasets.

Section 3.3 presents an ISPF dialog you can use to simplify the definition and analysis of VSAM datasets. This is a fully functional application -- similar to ISPF option 3.2 -- which can be integrated into your ISPF development environment.

You can access the RLX/VSAM dialogs by following this procedure:

- Type-in `RLXUSER` to start the RLX user dialogs
- Select option 5 ('RLX for z/OS') to display the main menu
- Select option 5 from the RLX for z/OS main menu.
  The panel shown on Figure 3.1 will be displayed.

On this panel you can start REXX and VSAM status traces which display the processing flow of VSAM calls and provide feedback information. These options are provided as a convenience.

```
------------------------- RLX VSAM Sample Dialogs ----------------------
 Command ===>

    1  Define   - Define VSAM sample datasets, vol serial, DD name
    2  ESDS     - Conduct Entry Sequenced dataset sample dialog
    3  KSDS     - Conduct Key Sequenced dataset sample dialog
    4  RRDS     - Conduct Relative Record dataset sample dialog
    5  LDS      - Conduct Linear dataset sample dialog
    T  Tutorial - Description of this menu's options
    X  Exit     - Leave this dialog

    Execution Trace Options
       Rexx tracing  ===> N    (Y/N - start Rexx trace)
       VSAM status   ===> N    (Y/N - display status of VSAM operations)
```

Figure 3.1 RLX/VSAM Sample Dialogs

The options shown on the panel are described in the subsequent sections. Since the dialogs shown here are intended as a tutorial, two flags are offered under the heading `Execution Trace Options`:

`REXX tracing` lets you monitor the REXX instructions being executed.

`VSAM status` lets you see the REXX variables set by RLX/VSAM.

3.1 Define and load sample VSAM datasets

In order to exercise the RLX/VSAM samples you must define and load the VSAM datasets used by the dialog. Select Option 1 from the panel illustrated in Figure 3.1 to display the menu shown below in Figure 3.2.

```
---------------------- Define VSAM sample datasets ----------------------
 Command ===>

    1  Dsnames  - Review / Alter Vsam dataset names and VolSer
    2  ESDS     - Allocate and populate sample VSAM ESDS dataset
    3  KSDS     - Allocate and populate sample VSAM KSDS dataset
    4  RRDS     - Allocate and populate sample VSAM RRDS dataset
    5  LDS      - Allocate and populate sample VSAM LDS  dataset
    T  Tutorial - Description of this menu's options
    X  Exit     - Leave this menu

    Execution Options
       Rexx tracing  ===> N    (Y/N - Start Rexx trace)
       Vsam status   ===> N    (Y/N - Display status of VSAM operations)
```

Figure 3.2 Define VSAM sample datasets

Select option 1 to specify the VSAM datasets (fully qualified dataset names with no quotes), the volume on which they will be allocated, and the DDNAME used by the online dialog that allocates them. These datasets require 4 cylinders of 3390 or equivalent DASD. Select options 2 through 5 to allocate the sample datasets. The REXX and/or VSAM status displays will be produced if you specify Y in Execution Options.

The sample execs used to create the VSAM datasets are found in the RLXEXEC library, having member names:

RXDVDLE	Load VSAM ESDS dataset
RXDVDLK	Load VSAM KSDS dataset
RXDVDLR	Load VSAM RRDS dataset
RXDVDLL	Load VSAM LDS dataset

Options 2-5 are also useful as examples of loading various types of VSAM datasets which execute. You can use these REXX programs as prototypes for your own VSAM load programs.)

3.2 Using RLX / VSAM to manage VSAM datasets

Once the sample VSAM datasets have been defined and loaded, you may conduct the RLX/VSAM demonstration dialogdemonstration by selecting options 2 through 5 from the RLX/VSAM Main menu (shown on Figure 3.1). These dialogs use the VSAM datasets you defined and loaded in the previous section. For example, if you select option 2, the VSAM ESDS sample dialog panel -- shown in Figure 3.3. -- is displayed with the contents of the sample VSAM ESDS file created in Section 3.1.

```
ESDS File: 'USER.VSAMESDS'                                          ROW 1 OF 5
Command ===>

MAJOR CMDs: ADD - Add a new record at the end of the file
LINE  CMDs: C -Change,  D -Delete,  A or I - Add (insert) new record

S RBA         Lrecl   Data record
  00000000000 0000033 There was a young lady from Niger
  00000000033 0000032 Who smiled as she rode on a tiger
  00000000065 0000027 They returned from the ride
  00000000092 0000020 With the lady inside
  00000000112 0000036 And a smile on the face of the tiger

******************************* BOTTOM OF DATA ********************************


 RXDVE102I - This program will show you how to manage VSAM ESDS files using
 VSAM functions.  Use major and line commands to request desired options.
 Turn trace and status display on to see actual REXX statements being
 executed and REXX variables set by VSAM functions.
```

Figure 3.3 RLX/VSAM ESDS demonstration dialog

Use primary command ADD -- or single letter row commands **C-**Change, **D-**Delete, **A-**Add, **I-**Insert -- to process records in the VSAM ESDS dataset. Do not hesitate to alter these records since you can reload these files any time. Use this dialog as an example to help you develop your own dialog. Dialogs to manage other VSAM datasets are similar to this one.

The sample execs used to manage VSAM datasets are found in the RLXEXEC library, having member names:

RXDVESDS	Manage VSAM ESDS dataset
RXDVKSDS	Manage VSAM KSDS dataset
RXDVRRDS	Manage VSAM RRDS dataset
RXDVLDS	Manage VSAM LDS dataset

3.3 VSAM Cluster Definition Facility

You can access the VSAM cluster definition facility by executing the RXDVSAL exec.

RLX /SDK

RLX Software Development Kit

User Guide

Version 9.1

June 2010

Publication RDX-001-4

Relational Architects Intl

This Guide: (RAI Publication RDK-001-4)

This document applies to RLX/SDK Version 9 Release 1 (June 2010), and all subsequent releases, unless otherwise indicated in new editions or technical newsletters. Specifications contained herein are subject to change and will be reported in subsequent revisions or editions.

Purchase Orders for publications should be addressed to:

Documentation Coordinator
Relational Architects International
Riverview Historic Plaza
33 Newark Street
Hoboken NJ 07030

Tel: 201 420-0400
Fax: 201 420-4080
Email Sales@RELARC.COM

Reader comments regarding the product, its documentation, and suggested improvements are welcome. A reader comment form for this purpose is provided at the back of this publication.

Ed 11F10

RLX / SDK

Table of Contents

Section A

What's New in the RLX/SDK

Summary of Changes to RLX Version 9.1

Version 9 Release 1 of RLX for z/OS supports all functions available in prior releases of the product plus all fixes and enhacements provided in interim software releases.

- The RLX Product Family includes a new component named ***RDX*** which supports SQL and pureXML at the latest levels of DB2 for z/OS. RDX also provides a set of REXX extension services through its REXX Function Package.

- RDX is integrated with RLX, included in the RLX distribution libraries and installed together with RLX. In addition, RDX is available as a standalone product with its own distribution libraries and manual (RDX Guide and Reference: Publication RDX-001).

- RLX Version 9 now supports both ADDRESS RLX and ADDRESS RDX as host command environments. In addition, both RLX and RDX make use of same DB2 application plans. However, RDX establishes its own DB2 thread, independent from RLX, when both RLX and RDX are active in the same REXX exec. Thus, two discrete COMMIT / Unit-of-Recovery scopes will be concurrently active – a practice RAI strongly discourages.

- RAI recommends that RLX and RDX code be segregated in separate execs that use either ADDRESS RLX or ADDRESS RDX exclusively. ADDRESS RLX allows you to continue using your existing RLX applications without modification. However, when new DB2 facilities (such as pureXML support) are required, RAI recommends you develop a new REXX exec and use ADDRESS RDX.

Summary of Changes to RLX Version 8.1

RLX/SDK Version 8.1 contains many enhancements and quality improvements in the RLX Software Development Kit (RLX/SDK). The following summarize these new facilities:

- The SDKMGP function creates an internal map for the data structure associated with a PL/I INCLUDE. Maps generated by SDKMGP can in turn be used as input to the SDKMAP function that creates discrete REXX variables that correspond to fields with a PL/I data structure. The SDKMGP and SDKMAP functions can be used together to map records from a file into discrete REXX variables that correspond to the field structure defined by the PL/I INCLUDE.

- The SDKDIV function is a full implementation of MVS Data-In-Virtual facility. It lets you develop very fast I/O interfaces to VSAM Linear Datasets – natively from REXX.

- The SDKDSNU function is enhanced and can now be invoked in the TSO/ISPF foreground to execute DB2 utilities such as RUNSTATS and REORG. The SDKDSNU function requires that the RAI Server address space be installed.

- RLX is enhanced with System Authorization Facility support which allows you to manage access to RLX authorized facilities via RACF or other Resource Access Control products. For example, you can utilize the RLX Cross Memory Access functions without making them available to unauthorized users.

- The SDKSAF function provides native REXX access to the RAI System Authorization Facility.

Chapter 1

RLX/SDK

Concepts and Facilities

The RLX/SDK (Software Development Kit) is designed to increase the productivity of REXX programmers and make REXX suitable for a wider range of purposes. The evolving set of facilities that comprise the SDK are designed to complement and extend the standard facilities supplied with the TSO/E implementation of REXX. These additional capabilities make REXX a viable replacement for languages such as COBOL, PL/I, C -- and even Assembler language. They enable REXX to tackle complex, `industrial strength' applications. With the SDK, you can even develop system exits in REXX for such IBM components as z/OS, JES and RACF. The RLX/Software Development Kit functions may be categorized as follows:

1.1 Native REXX access to a variety of MVS supervisor services and macro instructions:

- Memory management functions such as GETMAIN, FREEMAIN and XSTORAGE. The latter service is an extension of the REXX STORAGE function which can operate in cross memory mode to move a block of memory between two address spaces.

- Program management functions -- such as LOAD and DELETE

- Resource Control Functions (such as ENQ, DEQ, GQSCAN and RACHECK) let your REXX applications authenticate and/or synchronize their access to serially reusable resources to allow them to be shared. The SDK's ENQSUM facility presents this information within the context of a meaningfully formatted menu driven dialog.

- Multi-tasking and Interprocess Communication through WAIT, POST and Cross Memory POST.

- Native REXX access to the Z/OS operator console and system log through such system macros as WTL, WTO, WTOR and DOM. In addition, the MVSCMD service enables your REXX execs to directly issue MVS and JES commands. The QEDIT function provides the means for your execs to manipulate command input buffers.

- TGET and TPUT services for TSO I/O, as well as low level 3270 Data stream functions -- all native to REXX.

- The means to issue *any* Supervisor call natively from REXX via the SVC function.

> ***NOTE:*** *The RLX authorized functions require that you install the Relational Architects Server address space. In addition, the RLX implementations of MVS Supervisor Services are only briefly described in this manual. They are more fully documented in the IBM publications which describe MVS Supervisor Services and Macro Instructions.*

1.2 REXX extension services

- Parsing, tokenizing and sorting services for both strings and arrays.

- Functions to convert between packed and floating point values and the character string representations of numeric values that REXX supports.

- Functions to translate data between ASCII and EBCDIC.

- An evolving set of REXX execs for such tasks as concatenating additional datasets to an existing file allocation.

- An evolving set of ISPF edit macros for tasks such as verifying long host commands or formatting your REXX source modules.

- Functions with which to Browse and Edit REXX arrays directly in memory. These highly efficient facilities use the ISPF Browse and Edit services *without* the need for dataset I/O.

- The SDKQREF function provides native REXX access to the reference database maintained by ChicagoSoft's Quick/Ref product.

1.3 Enhancements that make REXX a viable replacement for compiled and assembled languages

- Global Variable Services allow REXX modules to share selected variables and array data by placing them in a global pool. This enables developers to split large applications into separate modules which can share REXX variables and stems. Global variable services also make it much easier to develop generic, reusable components in REXX.

- Enhancements to the AcceleREXX compiler enable you to write system exit routines in REXX for such IBM components as MVS, JES and RACF. You can also develop exit routines in REXX for software products from other vendors.

1.4 Modify and extend REXX itself

- The Software Development Kit provides a set of interactive dialogs with which to write functions in REXX and then deploy them within REXX function packages. This enables you to develop frequently used routines *in REXX*, then combine them into function packages that afford the fastest access available in the REXX environment.

- Another set of SDK dialogs allow you to modify the *parameter modules* that customize REXX for environments like MVS, TSO, ISPF and NetView. These dialogs enable you to integrate your own functions, the RLX language extensions and functions supplied by other vendors into a cohesive REXX environment.

1.5 RLX / SDK Dialogs

The RLX/SDK dialogs let you exercise the RLX/SDK REXX functions and learn their syntax. These demonstration dialogs are accessed via Option 4 of the RLX for z/OS Main Menu.

Review Option 1-8 from the panel on Figure 1.1. Read the tutorials to obtain high level help in using the panels. Demonstration REXX programs are found in the RLXEXEC library -- all members have the prefix RXD*.

```
----------------------- RLX/SDK Demonstration Dialogs --------------------
 Command ===>

    1  Memory      - Memory Management Functions
    2  Resource    - Resource Control Functions
    3  GlobalVars  - Global Variable Services
    4  Operator    - MVS Operator console and log interaction
    5  HostCmd     - REXX host command definition, SDK functions
    6  TSO I/O     - TSO and 3270 Data stream functions
    7  Extensions  - Miscellaneous REXX extension functions
    8  MultiTask   - Multi-tasking and Interprocess Communication
    X  Exit        - Leave this Menu

 Enter END to exit
```

Figure 1.1 RLX/SDK Demonstration Dialogs

Chapter 2

Memory Management Functions

This section describes the virtual storage management functions supported by the RLX Software Development Kit. The GETMAIN, FREEMAIN, LOAD and DELETE functions correspond one for one with MVS supervisor services. Similarly, the VSMLIST and VSMREGN functions are native REXX implementations of the MVS System Macros of the same name. As such they are briefly described in this chapter and are more fully documented in the IBM publications for MVS Supervisor Services and Macro Instructions.

GETMAIN and FREEMAIN let you obtain and free virtual storage. You can obtain storage from a specific subpool, above or below the 16 MB line.

XSTORAGE is an extension of the REXX STORAGE function which can operate in cross memory mode. It allows you to move a block of memory between the caller's address space and some other address space. This function requires that you install the RAI Server address space.

2.1 GETMAIN - Obtain Virtual Storage

The GETMAIN function provides a native REXX interface from your EXECs to the MVS GETMAIN macro. The syntax of this function is as follows:

Syntax: `addr = GETMAIN(length,subpool,location,boundary,fill_char)`

`addr` Address of acquired storage if the GETMAIN service was successful

`length` Length of the area to acquire

`subpool` MVS subpool from which the getmained area should be obtained

`location` Location of area to be `getmained`. Possible values are:

- `BELOW` obtain area below address `X'ØØFFFFFF'` (24 bit addressable storage)
- `ANY` obtain storage anywhere
- `RES` obtain area in accordance with the residence mode of the called program : BELOW or ABOVE `X'ØØFFFFFF' addr`

`boundary` Boundary of getmain area. Possible values are:

- `DBLWD` getmained area must start on a double word boundary
- `PAGE` getmained area must start on a page boundary

`fill_char` Character to be used to initialize getmained area

`RC = Ø` `GETMAIN` successfully executed

`4` `GETMAIN` execution failed

`> 4` Function parameters error

Example

Obtain 32 bytes of virtual storage from storage subpool 15:

```
Rc = GETMAIN(32,15)
```

2.2 FREEMAIN - Free Virtual Storage

The FREEMAIN function provides a native REXX interface from your REXX program to the MVS FREEMAIN macro. The syntax of this function is

---

Syntax: `rc = FREEMAIN(addr,length,subpool)`

`addr` Address of the storage to be released which was obtained by a successful GETMAIN request.

`length` Length of the getmained area to be freed -- `Length` must be the same as the length referenced on the GETMAIN call.

`subpool` The MVS subpool number if you wish to free an entire subpool. Subpool 0 (zero) ***must not*** be specified

> ***NOTE:*** *The subpool parameter is mutually exclusive* with the `addr` *and* `length` *parameters.*

`RC = Ø` successful completion of the FREEMAIN macro

`> Ø` Error in parameters or bad `<addr>/<length>`

Example

Obtain 32 bytes of virtual storage in subpool 20 on a page boundary. Then free it.

```
addr@ = GETMAIN(32,2Ø,'PAGE')

Rc = FREEMAIN(addr@,32)
```

2.3 XSTORAGE - Virtual Storage Access and Alteration

This function is an extension of the STORAGE function provided by TSO/E REXX. You can obtain or change the contents of virtual storage in your own address space or go cross-memory to another address space. In order to use this function, the RAI Server Address space must be installed and the userID which invokes this function must be authorized to use XSTORAGE. Appendix R of the RLX Installation and Customization Guide describes how to define and grant access to the XSTORAGE function.

Syntax: `area = XSTORAGE(jobname,xmemaddr,length[,value])`

`area` When the value parameter is omitted, the XSTORAGE function copies the contents of the area located at address `xmemaddr` within the address space associated with the job named `jobname`. When the value parameter is specified, the XSTORAGE function replaces the contents of the area located at address `xmemaddr` with the value specified by the value parameter.

`jobname` The jobname of the address space in which XSTORAGE should execute. This parameter may be omitted if XSTORAGE is to execute in the issuer's address space.

`xmemaddr` Address of the virtual storage to be obtained or altered. The address may be specified as a hexadecimal address or as an *address expression*. See the description of address expressions below.

`length` The length of the block of virtual storage to be retrieved.

`value` An optional new value which should *overlay* the existing value at address `xmemaddr` for the length specified by `length`. The length of the value specified must be exactly equal to `length` and must not exceed 256 bytes.

WARNING: The XSTORAGE function with the value parameter must be used with EXTREME CAUTION. If used improperly, XSTORAGE can cause severe system damage. The use of XSTORAGE with the VALUE parameter is protected by the resource profile named RAI.SDK.XMUP.

REXX variables held by the XSTORAGE function:

RC	Call completion information	
	Ø	Function executed successfully
	4	Address xmemaddr is not allocated
	5	Address invalid, absent or syntactically incorrect
	6	Invalid address expression syntax
	11	Jobname parameter is Invalid or not specified
	12	Address parameter is invalid or not specified
	13	Address parameter contain non-hex character
	14	Length parameter was not specified
	15	Length parameter contains non-numeric character
	16	Zap value length not equal to the area length
	17	Specified job is not active
	18	Error when ESTAE issued within PMVDUMP
	19	RAI Server address space is not installed
	2Ø	Failed to getmain area in CSA
	21	Error during SRB scheduling
	22	Block is not allocated (from SRB or PMVDUMP)
	23	Invalid function (RFPSRB1)
	24	ZAP is not allowed in PSA areas
	25	ZAP length exceeds 256 characters
SRB1XMA@	Address of the memory block	
SRB1GTCB	TCB if in private pool	
SRB1RC	RC from SRB	
SRB1SP	Subpool number	
SRB1PKEY	Protect key	
SRB1MLOC	Memory block location, one of the following:	
	Ø1	CSA
	Ø2	SQA
	Ø3	PRIVATE
	Ø4	LSQA
	Ø5	CPOOLFIX
	Ø6	CPOOLPAG
	Ø7	CPOOLLCL
	88	PSA
	89	REAL NUCLEUS
	8A	R/W NUCLEUS
	8B	R/O NUCLEUS
	8C	EXT R/O NUCLEUS
SRB1SJN	Target jobname for cross memory storage	
SRB1ASCB	ASCB address of target address space	
SRB1ASID	ASID of target address space	

Example 1

Copy 64 bytes from the Communications Vector Table (CVT) and display it.

```
cvt = XSTORAGE(,'10%',64)
say c2x(cvt)
```

Example 2

Replace 8 bytes at address '02345000' within the address space whose jobname is LLA, with value 'ZAP00001' (which is also 8 characters in length).

```
xx = XSTORAGE('LLA','02345000',8,'ZAP00001')
```

2.3.1 Specifying Address Expressions

The `xmemaddr` parameter of the XSTORAGE function can be specified in an address expression notation similar to that supported by the TSO TEST command. The syntax of an address expression is:

```
addrexp = indaddr oper indaddr oper ...
indaddr = hexaddr [%|?]
hexaddr = a hexadecimal number of 1 to 8 digits
oper    = + | -
```

where:

`+ or -` are arithmetic operators used to add or subtract hexadecimal addresses.

`?` is the indirection addressing operator which references a 31-bit address

`%` is the indirection addressing operator which references a 24-bit address

An address expression is parsed left to right. Indirect addressing operators (if present) are evaluated *before* arithmetic operators. Indirect addressing operators can be repeated -- as in '10%?%...' -- while arithmetic operators must be interleaved with the addresses.

Example 1: Reference the address of the CVT which is stored at location hex '10':

```
'10%'
```

The result is the 24-bit address of the CVT

2.3.2 RXDXMEM exec - Display and Alter Virtual Storage

RXDXMEM is an ISPF dialog with which to browse and alter virtual storage via the XSTORAGE function. The RXDXMEM exec resides within the RLX library whose low level qualifier is RLXEXEC. The RXDXMEM panel resides within the RLXPLIB dataset. Enter the RXDXMEM command to display a panel like the following:

```
---------------------------- Cross Memory Access ----------------- Row 1 of 4
Command ===>

Specify parameters for Target Address Space
  Job name  . . . . . . .
  Address expression. . . 10%
  Length  . . . . . . . . 64
  Zap data  . . . . . . .

Return from interface
  Job Name  . . . . . . . RAI028       ASID  . . . . . 0049
  Address . . . . . . . . 00FD4F18     ASCB  . . . . . 00F9F580
  Subpool . . . . . . . . 0            TCB addr. . . . 00000000
  Protect key . . . . . . 0            SRB RC. . . . . 00
  Block location. . . . . 8A  / R/W NUCLEUS
  RC / Message. . . . . . 0   / XSTORAGE successfully executed

Address  Offset   +0       +4       +8       +C       EBCDIC
00FD4F18 00000000 00000218 00FECCC0 00FD4E94 00FD5500 *.......{..+m....*
00FD4F28 00000010 00000000 00FF7F94 00FF3BCE 00FE2AE4 *......"m.......U*
00FD4F38 00000020 00FE2914 015AEC48 811EACD0 00FEF620 *.....!..a..}..6.*
00FD4F48 00000030 00F40870 00FED128 0097225F 00FD5528 *.4....J..p.¬....*
```

Figure 2.1 RXDXMEM exec memory display

Enter the name of the job whose virtual storage you wish to access. Then specify the address of the virtual storage, its length and, optionally, an updating value.

2.4 DUMP - Dump Virtual Storage or Load module

The DUMP function lets you display either virtual storage or a load module. The display is in full screen mode within ISPF or in line mode within other REXX environments. The usage of the DUMP function is further described and illustrated in the sample exec named RXDRAM1 which resides within the RLXEXEC library. To use the DUMP function within ISPF, the panel named RXDDUMP must be available.

Syntax: `call DUMP(address,length,jobname,modname)`

`address` specifies the address of the virtual storage to be displayed. Specify an address expression as described in Section 2.3.1.

`length` specifies the length of the virtual storage to be displayed

`jobname` the name of the job if a cross memory dump is requested

`modname` the name of the load module to be displayed

Example 1

Dump the storage in the address space whose jobname is LLA, starting at address X'00023F90' for a length of 200 bytes.

```
Call DUMP('02345000',200,'LLA')
```

Example 2

Dump the load module named IRXISPRM. The standard MVS search order is used to locate the module.

```
Call DUMP('-','-','-','IRXISPRM')
```

2.5 LOAD function

The LOAD function provides a native REXX interface from your REXX program to the MVS LOAD macro.

Syntax: `Rc = LOAD(modname,ddname)`

`modname`	name of the load module to be loaded
`ddname`	specifies the name of a file which identifies a private library from which to load the module. If specified, DDname must be allocated. If no private library is specified, then the standard MVS search order is used to locate the module.

REXX variables returned by the LOAD Function:

`RC`	Call completion information `Ø` Function executed successfully `>Ø` Function failed to execute successfully
`SDKLOAD@`	Hexadecimal address of the load module in memory
`SDKLOAD#`	Decimal length of the load module
`SDKAMODE`	Addressing mode of the load module (either 24 or 31)
`SDKAPFCD`	The module's APF authorization code

Example

Load the module named 'IRXTSPRM' from the private library identified by file name ISPLLIB.

```
Call LOAD('IRXTSPRM','ISPLLIB')
```

2.6 DELETE - Load module in Virtual Storage

The DELETE function provides REXX with native access to the MVS DELETE SVC.

Syntax: `RC = DELETE(modname)`

modname — the name of the load module to be deleted from virtual storage. It is assumed that modname was loaded via the LOAD function

---

REXX variables returned by the DELETE Function:

RC — Call completion information

- Ø — Function executed successfully
- >Ø — Function execution failed

Example

Delete load module IRXTSPRM from virtual memory

```
RC = DELETE('IRXTSPRM')
```

2.7 VSMLIST - Obtain Virtual Storage Information

List the virtual storage subpools that are allocated, used, and/or free.

Syntax: `result = VSMLIST(spspec,space,tcb,loc,real)`

`result` A result area acquired and set by VSMLIST that contains subpool information in accordance with the specifications in the IBM documentation.

`spspec` VS subpool specification:

- `SQA` - map SQA subpools
- `CSA` - map CSA subpools
- `LSQA` - map LSQA subpools
- `PVT` - map of private area subpools
- `ALL` - map of all subpools
- `xxx` - map of a specific subpool xxx

Default: PVT

`space` Type of subpool information requested:

- `ALLOC` - Allocated blocks
- `FREE` - Free areas and allocated blocks
- `UNALLOC` - Unallocated blocks

Default: ALLOC

`tcb` Address of a specific TCB for which storage subpool information is requested. Applies to a subpool specification of PVT. If omitted, then report for all TCBs.

`loc` Location of subpools
- `ANY` - report on allocation above and below 16MB
- `BELOW` - report on allocation below the 16MB line only

`real` If the value REAL is coded, then block addresses are returned with their high order bit on, where appropriate, to indicate the storage resides above the 16MB line.

`rc = 0` VSMLIST was performed successfully
`= 4` Partial response. A bigger work area is required
`= 8` Information is incomplete
`= 12` Work area is less than 4K - too small

Example

Obtain a report for all CSA subpools that are allocated:

```
result = VSMLIST('CSA')
```

2.8 VSMREGN - Obtain information about a User Region

List the boundaries of the user region (via the VSMREGN macro).

Syntax: `result = VSMREGN()`

This function requires no parameters

`result` - a 16-byte area containing 4 full words in binary:

word1 - address of the user region below 16MB
word2 - length of the user region below 16MB
word3 - address of the user region above 16MB
word4 - length of the user region below 16MB

`RC` = 0 (always)

Example

```
result = VSMREGN()
below@ = Substr(result,Ø1,4)   /* address of region below 16MB */
below# = Substr(result,Ø5,4)   /* length  of region below 16MB */
above@ = Substr(result,Ø9,4)   /* address of region above 16MB */
above# = Substr(result,13,4)   /* length  of region above 16MB */
```

2.9 VSMUTIL - Information About Virtual Storage Utilization

The VSMUTIL function computes the total size of the allocated user region, both below and above the 16MB line. VSMUTIL returns the computed results in REXX variables.

Syntax: `Result = VSMUTIL()`

This function requires no parameters

REXX variables returned by the VSMUTIL function:

`RC`	Defines call completion `Ø` Function executed successfully `>Ø` Function execution failed
`$REGNB@`	Starting address of the user region below 16MB
`$REGNB#`	Size of the region below 16MB
`$REGNA@`	Starting address of the user region above 16MB
`$REGNA#`	Size of the region above 16MB
`$REGNBU`	Region used below 16MB
`$REGNAU`	Region used above 16MB

Example

The following exec calls VSMUTIL to obtain information about the virtual storage within the user region that is allocated both above and below the 16MB line. The example then computes and displays the virtual storage utilization as a percent.

```
numeric digits 10
arg debug

   call vsmutil

   $REGNBU#     = (x2d($regnbu) / x2d($regnb#) ) * 100
   $REGNAU#     = (x2d($regnau) / x2d($regna#) ) * 100

   say 'Function VSMUTIL executed with Rc='rc
   say ''

   call Show '$REGNB@' '/* Starting address of Region below 16MB */'
   call Show '$REGNB#' '/* Size of Region below 16MB             */'
   call Show '$REGNA@' '/* Starting address of Region above 16MB */'
   call Show '$REGNA#' '/* Size of Region above 16MB             */'
   say ''
   call Show '$REGNBU' '/* Region used below 16MB                */'
   call Show '$REGNAU' '/* Region used above 16MB                */'
   call Show '$REGNBU#' '/* Percent of Region used below 16MB     */'
   call Show '$REGNAU#' '/* Percent of Region used above 16MB     */'

Return

Show:
parse arg var text
   say left(var,8) '=' left(value(var),8) '  ' text
Return
```

Chapter 3

Resource Control Functions

This section describes the resource control functions supported by the RLX Software Development Kit. The SDK functions which correspond one for one to MVS supervisor services are briefly described in this chapter. They are more fully documented in various OS/390 publications.

The ENQ and DEQ functions allow independent processes to synchronize their access to serially reusable resources so these resources may be shared. The GQSCAN service and its enhanced version SDKGQS return information to the caller that describes these serially reusable resources. The RACHECK service lets your REXX execs check whether access to a protected resource should be permitted or denied ***before*** the resource is actually used. A protected resource is one defined to the security software (such as RACF, ACF2 or TOP Secret) installed at your site. The SDKSAF function provides access to the architected RAI - SAF interface.

ENQSUM is an ISPF dialog rather than a native REXX interface to an MVS system service. ENQSUM uses the GQSCAN function to obtain information about resource conflicts and displays those results on scrollable ISFP panels.

3.1 ENQ -- Obtain control of a resource

The ENQ function provides REXX execs with native access to the MVS ENQ service so your application can acquire control of a resource before using it. Execs that issue the ENQ function can optionally wait for a resource if it is not immediately available.

The parameter descriptions of the MVS ENQ macro also apply to the RLX ENQ function. Refer to the OS/390 - MVS Programming - Assembler Services manuals for additional details on the ENQ function.

Syntax:

```
rc = ENQ(qname,rname,control,scope,reqtype)
```

`qname`	specifies a 1-8 character Major name that identifies the first level of qualification.
`rname`	specifies the resource name (minor name) which may be from 1 to 255 characters in length
`control`	'E' - Exclusive, 'S' - Shared control
`scope`	'STEP' | 'SYSTEM' | 'SYSTEMS'
`reqtype`	'CHNG' | 'HAVE' | 'TEST' | 'USE' | 'NONE'

Return code values returned in REXX variable RC

0	Successful execution
40	No parameters specified
41	Incorrect QNAME
42	Incorrect RNAME
43	Error in CONTROL parameter (see ENQ macro)
44	Error in RET parameter (see ENQ macro)
45	Error in SCOPE parameter (see ENQ macro)

Example:

```
rc = ENQ('RAIQNAME','RAIRNAME','S','SYSTEMS','USE')
if rc = 0 then
   say 'ENQ was successful'
else
   say 'ENQ failed, rc='rc
```

3.2 DEQ -- Release control of a resource

The DEQ function provides REXX execs with native access to the MVS DEQ macro which releases resources previously acquired through ENQ function. The parameter descriptions of the MVS DEQ macro apply equally to the RLX DEQ function. Refer to the OS/390 - MVS Programming - Assembler Services manuals for additional details on the DEQ function.

Syntax: `rc = DEQ(qname,rname,scope,reqtype)`

`qname`	specifies a 1-8 character Major name that identifies the first level of qualification.
`rname`	specifies the resource name (minor name) which may be from 1 to 255 characters in length
`scope`	'STEP' | 'SYSTEM' | 'SYSTEMS'
`reqtype`	'HAVE' | 'NONE'

Return code values returned in REXX variable RC

0	Successful execution
40	No parameters specified
41	Incorrect QNAME
42	Incorrect RNAME
44	Error in RET parameter (see DEQ macro)
45	Error in SCOPE parameter (see DEQ macro)

Example:

```
rc = DEQ('RAIQNAME','RAIRNAME','SYSTEMS')
if rc = 0 then
   say 'DEQ was successful'
else
   say 'DEQ failed, rc='rc
```

3.3 GQSCAN -- Obtain resource control information

The GQSCAN function provides REXX execs with native access to the MVS GQSCAN macro. The parameter descriptions of the MVS GQSCAN macro apply equally to the RLX GQSCAN function. Refer to the OS/390 - MVS Programming - Assembler Services manuals for additional details on the GQSCAN function.

> ***NOTE:*** *This function has been supplanted by the SDKGQS function. It is presented here solely for compatibility with prior releases of the RLX / Software Development Kit.*

Syntax:

```
rc = GQSCAN(mode,address,length,scope,sysname,asid,reqlim,token)
```

or

```
rc = GQSCAN('QUIT',token)
```

`mode`	GENERIC | SPECIFIC | QUIT
`address`	Address of an answer area obtained through the GETMAIN unction
`length`	Length of the answer area obtained via the GETMAIN function
`qname`	Major name
`rname`	Minor name
`scope`	ALL | STEP | SYSTEM | SYSTEMS | LOCAL | GLOBAL
`sysname`	System name
`asid`	Address space ID
`reqlim`	Number of RIBE (resource owners) to be retrieved) for 'SPECIFIC' queries
`token`	Address of a fullword for the token to be passed to GQSCAN. The address must be in the character form returned by the GETMAIN function.

Return code values returned in the REXX variable RC

Ø	Successful execution
4	No resource satisfies the query
8	More data available if GQSCAN is reissued
4Ø	No parameters specified
41	Incorrect QNAME
42	Incorrect RNAME
45	Error in SCOPE parameter (see GQSCAN macro)
46	Error in MODE parameter (see GQSCAN macro)
47	Error in SYSNAME parameter (see GQSCAN macro)
48	Error in ASID parameter (see GQSCAN macro)
49	No QNAME and SYSNAME was specified
5Ø	Invalid area address
51	Invalid area length
52	Token is in error (see GQSCAN macro)
53	GRS SCAN was terminated

Example

Retrieve information describing all enqueued datasets

```
token@ = GETMAIN(4)            /* obtain full word for a token */
area@  = GETMAIN(1Ø24)         /* obtain an answer area        */

/* get information on all dataset enqueues */
rc = GQSCAN('GENERIC',area@,1Ø24,'SYSDSN',,'SYSTEM',,,,token@)
select
   when rc = Ø then say 'Area@ points to answer - query completed '
   when rc = 4 then say 'There were no resources satisfying query '
   when rc = 8 then say 'Area@ points to answer - more info available'
   otherwise
     say 'Error retrieving data from GRS'
end
```

Notes:

> A return code of 8 indicates that more data is available than can be mapped into the answer area. You can reissue GQSCAN with the same parameters to retrieve this data or issue `rc = GQSCAN('QUIT',token@)` to terminate the scan.

> You can use the STORAGE or MEMORY functions to map the answer area. See the structure maps of the MVS RIB and RIBE control blocks for details (mapping macro ISGRIB in SYS1.MACLIB).

3.4 SDKGQS -- Obtain resource control information (Enhanced)

The SDKGQS function provides REXX execs with native access to the MVS GQSCAN macro. The parameter descriptions of the MVS GQSCAN macro apply equally to the SDKGQS function. Refer to the OS/390 - MVS Programming - Assembler Services manuals for additional details on the GQSCAN function.

Syntax: `rc = SDKGQS(qname,rname,mode,scope,sysname,jobname)`

`qname` Major name (see ENQ, DEQ functions)

`rname` Minor name (see ENQ, DEQ macros)

`mode` GENERIC | SPECIFIC | QUIT

`scope` ALL | STEP | SYSTEM | SYSTEMS | LOCAL | GLOBAL

`sysname` System name

`jobname` Only resources owned by this jobname will be selected

`waitcnt` An integer which causes all ENQs with a wait count of equal or a greater value to be retrieved. Specify `waitcnt` as 1 to display all exclusive ENQ conflicts in the system causing execution waits.

REXX variables returned by the SDKGQS Function:

`RC` Defines call completion. Possible values of the RC variable are:

- `40` No parameters specified
- `41` Incorrect QNAME parameter
- `42` Incorrect RNAME parameter
- `45` Error in SCOPE parameter
- `47` Error in SYSNAME parameter
- `48` JOBNAME specified while SYSNAME was not.
  Both parameters must be specified or both must be omitted.
- `49` No QNAME and SYSNAME parameters were specified
- `50` Jobname is not active
- `51` Wait count is not an interger
- `52` Invalid token (program logic error -- report to RAI)

The SDKGQS function returns two groups of REXX stemmed variables. The first group, whose names start with the characters ‘RIB’, describe the resources being used. For each resource, a group of variables, whose names start with the characters ‘RIBE’, describe all users of the resource. This is depicted schematically as follows:

```
do i = 1 to rib.0           /* for every resource selected */
   display RIB              /* resource description        */
   do j = 1 to ribnribe.0   /* for every resource user     */
      display RIBE          /* resource users              */
   end
end
```

RIB variable (Resource description)

`RIB.Ø`	Number of resources for which information is returned
`RIBNTO.i`	Number of tasks owning the resource
`RIBNTWE.i`	Number of tasks waiting for exclusive control over the resource
`RIBNTWS.i`	Number of tasks waiting for shared control over the resource
`RIBNRIBE.i`	Number of RIBE (resource users) for this RIB (resource)
`RIBSCOPE.i`	Scope code
`RIBQNAME.i`	Resource queue name -- QNAME
`RIBRNAME.i`	Resource name -- RNAME

(RIBE variable (Resource users/requestors). Index *i* references a specific resource while index *j* references a specific requestor of the resource referenced by *i*.

`RIBEJBNM.i.j`	Job name of the Resource requestor
`RIBESYSN.i.j`	Resource requestor's system name
`RIBETCB.i.j`	Resource requestor's TCB address
`RIBEUCB.i.j`	Resource requestor's UCB address
`RIBEASID.i.j`	Resource requestor's ASID
`RIBERFLG.i.j`	Request flag
`RIBELFLG.i.j`	List flag
`RIBESFLG.i.j`	Status flag
`RIBESAID.i.j`	Service request ASID
`RIBEDEVN.i.j`	Device name

Example:

Retrieve and display information about the TSO userIDs enqueued (waiting) to edit member RLXTRACE of the library named RAI.RLX.RLXCNTL.

```
/* rexx */
dl.Ø            = Ø
qname           = 'spfedit'
dsn             = 'RAI.RLX.RLXCNTL'
member          = 'RLXTRACE'
rname           = left(dsn,44) || left(member,8)
mode            = 'S'
jobname         = ‘’
src             = sdkgqs(qname,rname,mode,scope,sysname,jobname)
if src         \= Ø then return
do I = 1 to RIB.Ø
   say ribqname.i ribrname.i
   do j = 1 to ribnribe.i
      say ribejbnm.i.j  /* display all jobs ENQed on resource */
   end
end

return
```

Notes:

> See macro ISGRIB in SYS1.MACLIB for the mapping of the MVS RIB and RIBE control blocks.

3.5 RACHECK -- Verify a User's authority to access a resource

RLX provides a native REXX interface to the MVS System Authorization Facility through the RACROUTE function. SAF, in turn, routes authorization requests to the security product (such as RACF, ACF2 or Top Secret) installed at your site. Refer to the RACF Macros and Interfaces manual for details on coding the RACROUT AUTH macro and the RACHECK requests you issue from REXX.

Syntax: `rc = RACHECK('parms')`

```
ENTITY(resource_name)
VOLSER(volsername)
  CLASS(class_name)
  ATTR( READ | UPDATE | CONTROL | ALTER )
  DSTYPE( N  |  V  |  M  |  T )
  LOG( ASID  | NOFAIL | NONE | NOSTAT )
  OLDVOL(volser_name)
  APPL(appl_name)
  ACCLVL(access_value)
  RACIND( YES | NO )
  GENERIC( YES | ASIS )
  FILESEQ(number)
  TAPELBL( STD | BLP | NL )
  STATUS( NONE | ERASE )

RACFROUTE macro parms
  REQSTOR(control_point_name)
  SUBSYS(subsystem_name)
  MSGRTRN( YES | NO )
  MSGSUPP( NO  | YES)
  RELEASE=( 1.6  |  1.7  |  1.8  |  1.8.1 )
  OWNER(userid)

Debugging parms
  LIST( YES  |  *NO )
  TRACE( YES,type | *NO )
```

Return code values returned in REXX variable RC

0	Access to a resource is authorized
>0	Resource access is not authorized

Example: Check whether the user executing this exec can update the SYS1.PARMLIB dataset:

```
call racheck 'ENTITY(SYS1.PARMLIB)',
             'VOLSER(RAMCAT) CLASS(DATASET) ATTR(UPDATE)'
if rc = 0 then
   say 'Access is allowed'
else
   say 'Access is not allowed'
```

3.6 ENQSUM dialog

The ENQSUM dialog displays various system enqueues and resource conflicts. The ENQSUM application is comprised of the REXX execs and ISPF panels described below:

RXDRC2 EXEC

This main exec displays the RXDRC2 panel on which input parameters are specified.

RXDENQS EXEC

This exec issues the GQSCAN macro based upon parameters specified on the RXDRC2 panel. System information returned in GQSCAN is formatted and mapped into an ISPF table. The ENQSUM dialog user can then select a specific resource and display information about the requestors of that resource. The RXDENQS dialog can be invoked as a stand-alone function with the following call:

```
CALL RXDENQS scope qname rname mode
```

where:

`scope` - STEP | SYSTEM | SYSTEMS | ALL

`qname` - name of the system queue, e.g. SYSDSN, IKJUSER, etc.

`rname` - name of the resource

`mode` - GENERIC | SPECIFIC, qualifies rname search

RXDENQS PANEL

A panel on which ENQ summary information is displayed.

RXDENQ PANEL

This panel displays detailed information about the requestors enqueued to a specific resource.

3.7 SDKSAF -- Architected System Authorization Facility

RLX provides a general purpose interface to System Authorization Facility (SAF). The RAI SAF implementation uses the class FACILITY to let you restrict access to any resource -- such as a REXX exec or command processor.

Syntax: `rc = SDKSAF(resource)`

`RC` Return code.
`Ø` = Resource access is allowed
`>Ø` = Resource access is not allowed

`resource` One of the ten resource ids: USR0, USR1, ..., USR9

There are ten user profiles defined in RAI SAF which correspond to the ten resource IDs you can specify in the SDKSAF function:

```
RAI.USRØ
RAI.USR1
...
RAI.USR2
```

Appendix R of the RLX Installation Guide describes how to associate a resource ID with a resource. You can define a RACF resource profile and then associate it with the user ID.

Example

Suppose you develop an exec named SECRET and wish to restrict its use to ID SYSPROG1. To do so, issue RACF commands like the following:

```
RDEFINE FACILITY RAI.USR1 UACC(NONE)    <- define profile RAI.USR1
PERMIT RAI.USR1 CLASS(FACILITY) ID(SYSPROG1) ACCESS(READ)
```

These commands first define USR1 as a resource ID (that will be associated with the SECRET exec). Next, permit only the UserID SYSPROG1 to access this resource. Lastly, invoke the SDKSAF function as follows at the beginning of the SECRET exec.

```
/* rexx */
if sdksaf('USR1') \= Ø then
   Return 16
...
```

With such an SDKSAF function reference in place, only user SYSPROG1 can continue execution. All other userIDs will receive a non-zero return code and exit immediately.

Chapter 4

Operator Console and Log Functions

This section describes and illustrates the WTL, WTO, WTOR, and DOM functions that provide REXX with native access to the MVS operator console and system log. The MVSCMD service enables REXX execs to directly issue MVS and JES commands while the QEDIT function provides a means for REXX applications to manipulate command input buffers. The SDKVTAM function allows you to issue VTAM commands from a REXX exec. The CALLRTM function lets you cancel a specific subtask within an address space or to terminate an entire address space in a manner similar to the MVS FORCE command.

The WTL, WTO, WTOR, DOM and QEDIT functions correspond to MVS system macros that are fully documented in MVS macro instruction publications.

Routing codes

The destination(s) to which the WTO and WTOR functions route messages are specified through routing codes. The first 16 routing codes are listed in the following table:

Route Code	Description	Route Code	Description
1	Master console action	9	System security
2	Master console information	10	System error/maintenance
3	Tape pool	11	Programmer information
4	Direct access pool	12	Emulators
5	Tape library	13	User defined
6	Disk library	15	User defined
7	Unit record pool	14	User defined
8	Teleprocessing control	16	User defined

Descriptor codes

The color, display intensity and scrollability of messages issued through these functions can be controlled through descriptor codes, the first 16 of which appear in the following table:

Desc Code	Description	Desc Code	Description
1	System failure	9	Operator request
2	Immediate action required	10	Dynamic status display
3	Eventual action required	11	Critical eventual action
4	System status	12	Reserved
5	Immediate command response	13	Reserved
6	Job status	15	Reserved
7	Application program	14	Reserved
8	Out-of-line message	16	Reserved

Routing and descriptor codes are each represented by a 16 bit value where each code (from 1 to 16) corresponds in a left to right sequence to one of the 16 bits. For example, to request routing codes 1, 2 and 3 for a message, specify the routing code value as '1110000000000000'b or 'E000'x. Use this same method to specify descriptor codes as well.

4.1 WTL -- Write to System Log

The WTL function enables REXX execs to write single-line messages to the system log. The parameter descriptions for the MVS WTL macro apply equally to the RLX WTL function. The OPTION=NOPREFIX is assumed on the WTL macro. WTL is an authorized facility. Appendix R of the RLX Installation and Customization Guide describes how to define and grant access to the WTL function.

Syntax: `RC = WTL(text)`

`text` Message text to be written to the system log

The WTL function returns the following REXX variables:

`RC` Defines call completion

- `0` Function executed successfully
- `>0` Function failed to execute successfully

`REASON` Further identifies error if RC not zero

Example

Write message to system log.

```
Rc = WTL('Start monitor' DATE() TIME())
```

4.2 WTO -- Write To System Console

The WTO function enables REXX execs to write messages to MVS console destination(s). The parameter descriptions for the MVS WTO macro apply equally to the RLX WTO function. WTO is an authorized facility. Appendix R of the RLX Installation and Customization Guide describes how to define and grant access to the WTO function.

Syntax: `Result = WTO(msg,{desc},{routcde})`

msg

A single or multi-line mssage be written upon a system console.

1 A single-line message is written with the following call:
RC=WTO(variable,desc,routcde)
where: **variable** is either a REXX variable or a literal string. Maximum length of a single message is 126 characters.
Example:
```
rc=WTO(‘Hello world!’)
```

2. A multi-line message is written with the following call:
RC=WTO(stem,desc-code,route-code)
where: **stem** is a name of REXX stem variable containing lines of the message. Format of each line is as follows:
t,message text where: **t** - is line type, :
- **C** - control line (maximum length 34 bytes)
- **D** - data line (maximum length 70 bytes)
- **L** - label line (maximum length 70 bytes)

Example:
```
msg.1 = 'C,Control line'
msg.2 = 'D,Data line 1'
msg.3 = 'D,Data line 2'
rc = WTO('msg.')
```

Note: This parameter is required.

desc

Message descriptor codes in hexadecimal format. This parameter corresponds to `DESC=(code-list)` parameter of WTO macro.
Default: `'0200'x` which corresponds to `DESC=(7)`
```
Meaning: '0200'x = '0000 0010 0000 0000'b
                    ---- --7- ---- ----
```
Bit 7 corresponds to (Application program message) is set on
Note: This parameter is optional.

routecde

Routine code(s) to be assigned to the message. This parameter corresponds to `ROUTCDE=(code-list)` parameter of WTO macro.
Default: `'4020'x` which corresponds to `ROUTCDE=(2,11)`
```
Meaning: '4020'x = '0100 0000 0010 0000'b
                    -2-- ---- --11 ----
```
Codes: 2=(Operator info), 11=(Programmer info)
Note: This parameter is optional.

The WTO function returns the following REXX variables:

RC Return code variable indicating call completion

```
00 = Message successfully issued
08 - access security error - profile RAI.SDK.OPER
     not authorized to the caller - see SDK014E
     message.
41 - invalid 'message' parameter
42 - 'desc' is incorectly specified
43 - 'routecde' is incorectly specified
```

Error return code from WTO macro call:

```
02 - Inconsistent parameters, see abend SD23
04 - incorrect message length
18 - Invalid WPL (ll field of message, etc.)
30 - Environmental error
```

WTOMSGID Message ID in hexadecimal format. You must not change this variable values in any way. The console message corresponding to this ID can subsequently be deleted from the system console via the DOM function.

Example

Write a message to the MVS system console.

```
rout = '3000'x  /* ROURCDE=(3,4) */
desc = '0100'x  /* DESC=(8) Out of line message */
msg  = 'WTO RLX/SDK MVS Operator Console Function’)
rc   =  WTO(msg,desc,rout)
Say "WTO function completed with RC="rc,
    "message ID="C2X(wtomsgid)
```

4.3 WTOR -- Write To the Operator with Reply

The WTOR function enables REXX execs to write prompting messages to the operator console, wait for a reply and return the operator's response in a REXX variable. The parameter descriptions for the MVS WTOR macro apply equally to the RLX WTOR function. WTOR is an authorized facility. Appendix R of the RLX Installation and Customization Guide describes how to define and grant access to the WTOR function.

Syntax: `Reply = WTOR(msg,routcde,timeout)`

`reply`	a user specified REXX variable into which WTOR returns the text of the operator's reply.
`msg`	a message to be written to the system console. It can be a REXX variable, literal string or REXX stem.
`routcde`	MVS routing code
`timeout`	Timeout interval in seconds. After this interval expires, the WTOR function unconditionally returns control to the caller.

The WTOR function returns the following REXX variables:

`RC` Defines call completion

- `0` Function executed successfully
- `>0` Function execution failed

When the WTOR function executes successfully, the exec which issues WTOR remains in a wait state until the operator responds in the form 'REPLY XX,reply text'. Here XX corresponds to the two digit message identifier assigned by MVS to the message. RLX returns the operator's reply in the REXX variable to the left of the function reference. In addition, RLX assigns the console message ID to the REXX variable WTOMSGID. The console message corresponding to this ID can subsequently be deleted from the system console via the DOM function.

Example

Write a message to the system console.

```
Rout   = '0020'x   /* Routcde = 11 - Programmer */
Msg = 'Reply Yes or No'
Reply =  WTOR(Msg,Rout)
If Rc = 0 Then
   say 'Console reply was' Reply
```

4.4 DOM -- Delete Operator Message

The DOM function enables REXX execs to delete messages routed to console destinations through the WTO and WTOR functions. The parameter descriptions for the MVS DOM macro apply equally to the RLX DOM service. DOM is an authorized facility. Appendix R of the RLX Installation and Customization Guide describes how to define and grant access to the WTL function.

Syntax: `Call DOM(wtomsgid)`

`wtomsgid` a message ID returned in the WTOMSGID variable after a successful WTO or WTOR request.

The DOM function returns the following REXX variables:

`RC` Defines call completion

- `Ø` Function executed successfully
- `8` Parameter error - WTOMSGID was not specified or was incorrectly specified

Example

Delete a previously issued WTO message

```
desc = 'Ø8ØØ'x   /* code=5 Immediate command response */
Reply =  WTO('IPL system 21:ØØ',Desc)
msgid = WTOMSGID  /* save message id */
...
Call DOM(msgid)
```

4.5 MVSCMD -- Issue MVS system command

The MVSCMD command provides a native REXX interface to SVC 34 - the MVS console command interface. MVSCMD is an authorized facility. Appendix R of the RLX Installation and Customization Guide describes how to define and grant access to the MVSCMD function.

Syntax: `Call MVSCMD(command)`

`command` any valid MVS or JES system command

The MVSCMD function returns the following REXX variables:

`RC` Defines call completion

- `0` Successful execution
- `8` Command is invalid or not specified

Example

Issue the MVS display time command 'D T'

```
Rc = MVSCMD('D T')
```

4.6 QEDIT -- Manipulate Command Input Buffers

The QEDIT function provides REXX execs with native access to the MVS QEDIT service through which the MVS system commands MODIFY and STOP are received. QEDIT is an authorized facility. Appendix R of the RLX Installation and Customization Guide describes how to define and grant access to QEDIT function.

Syntax: `command = QEDIT(ecb@)`

`command` an MVS console command received via an MVS MODIFY or STOP command. For modify commands such as: 'MODIFY jobname, modify_command' the command buffer contains 'MOD modify_command'. For stop commands such as: 'STOP jobname', the command buffer contains: 'STOP'.

`ecb@` Address of a termination ECB. This ECB can be posted by the QEDIT issuer to cause the service to return (terminate its wait) *before* an MVS STOP or MODIFY command is received.

The QEDIT function returns the following REXX variables:

`RC` Defines call completion

- `0` Command successfully executed
- `8` Parameter error or QEDIT execution error

Example

```
ecb@ = Getmain(4)     /* address of ecb */

/* attach exec SAMPLE1 and pass it an ecb address */
address ATTACH RCXATT 'sample1' ecb@
```

in REXX exec SAMPLE1**:**

```
...
arg ecb@
...
command = QEDIT(ecb@)   /* QEDIT will return when a MODIFY or STOP command
                           is issued or the termination ecb@ is posted */
return
```

4.7 SDKVTAM -- Issue VTAM commands and obtain responses

The SDKVTAM function provides REXX execs with the ability to issue Virtual Telecommunication Access Method (VTAM) commands and obtain command responses returned in REXX variables. The SDKVTAM function allows you to develop automated network management applications in REXX. SDKVTAM is an authorized facility. Appendix R of the RLX Installation and Customization Guide describes how to define and grant access to the SDKVTAM function.

Syntax: `rc= SDKVTAM(command,stem,applid,passwd)`

`command` A valid VTAM command (e.g. 'D NET,APPLS')

`stem` A REXX stem name to contain the command response

`applid` A VTAM APPL defined in SYS1.VTAMLST. Two definition types are allowed:

a) An explicit APPL name, which must be 8 characters long (e.g. 'RAIAPL00')

b) An APPL prefix (which must be 6 characters long) used as a base to form APPL names, e.g. 'RAIAPL'. In this case SDKVTAM forms names such as RAIAPL00, RAIAPL01, and so on which it tries to open to communicate with VTAM.

`passwd` A password specified on the VTAM APPL statement.
The default password is 'RAI'.

The SDKVTAM function returns the following REXX variables:

`RC` Defines call completion

0	Operation was successful
8	Initialization failed due to installation or password definition error
9	Stem name was too long (allow maximum 75 character stem name)
10	No VTAM command was supplied
11	Error in sending a command to VTAM
12	ACB name was not specified
13	Password is invalid
14	Receive of command response has failed
15	ACB already open
16	Either the ACB name was invalid or an incorrect password was supplied
17	ACB generate failed
19	ACB name is neither 6 nor 8 characters in length

Example of VTAM APPL node definition

The following illustrates how you can define VTAM APPL nodes for the SDKVTAM function in SYS1.VTAMLST: (These definitions allow up to five simultaneous SDKVTAM operators.)

```
RAIAPPL  VBUILD TYPE=APPL
RAIAPL   APPL  AUTH=(ACQ,PASS,SPO,VPACE),VPACING=7,PRTCT=RAI
RAIAPLØØ APPL  AUTH=(ACQ,PASS,SPO,VPACE),VPACING=7,PRTCT=RAI
RAIAPLØ1 APPL  AUTH=(ACQ,PASS,SPO,VPACE),VPACING=7,PRTCT=RAI
RAIAPLØ2 APPL  AUTH=(ACQ,PASS,SPO,VPACE),VPACING=7,PRTCT=RAI
RAIAPLØ3 APPL  AUTH=(ACQ,PASS,SPO,VPACE),VPACING=7,PRTCT=RAI
RAIAPLØ4 APPL  AUTH=(ACQ,PASS,SPO,VPACE),VPACING=7,PRTCT=RAI
```

Example

Issue a VTAM command and display the command response:

```
cmd  = 'D NET,MAJNODES'              /* VTAM command        */
stem = 'cmd.'                        /* Command output stem */
appl = 'RAIAPLØ1'                    /* VTAM appl name      */
pass = 'RAI'                         /* Password            */

call SDKVTAM cmd,'cmd.',appl,pass
if rc > Ø then
   say 'rxdVTAM - Error while executing SDKVTAM, rc='rc
else
   call sdkBRIF 'cmd.',,'Output from command='cmd
```

4.8 CALLRTM -- Issue an MVS CALLRTM macro from Rexx

The CALLRTM function is an authorized facility that allows you to cancel a specific subtask or an entire address space via the MVS CALLRTM macro. Since this function is even more powerful than the MVS CANCEL command, it should be used very judiciously. CALLRTM is an authorized facility. Appendix R of the RLX Installation and Customization Guide describes how to define and grant access to the CALLRTM function.

Syntax: `Rc = CALLRTM(type,asid,tcb,compcod,reason,step,term,dump,retry)`

`type`	ABTERM | MEMTERM - Type of termination request. ABTERM requests task termination while MEMTERM requests address space termination.
`asid`	ASID of the job (address space) where the abend will occur
`tcb`	Address of the task Control Block (TCB) within an address space to be terminated
`compcod`	Abend completion code to be set. The default is S222
`Reason`	Abend reason code to be set
`step`	YES | NO - cancel the jobstep TCB
`dump`	YES | NO - request a dump at entry to abend
`retry`	YES | NO - Allow or disallow retry exit in ESTAE processing

---

The CALLRTM function returns the following REXX variables:

`RC` Defines call completion

`Ø`	Operation was successful
`4`	The task has already been scheduled for termination by a previous ABTERM request.
`8`	An asynchronous unit of work has been scheduled to terminate the task
`24`	The ASID value is not valid
`28`	The TCB address or TTOKEN value is not valid
`44`	APF test failed

48 APF ON error

52 APF OFF error

56 No completion code

60 Not MVS/ESA when RETRY=NO was specified

64 Invalid request parameters

71-79 Error in parameters 1 through 7

88 Error in establishing an RAI environment

Example

This example illustrates a CALLRTM request to cancel a TCB at address ‘00FC0640’ in the address space whose ASID (in hexadecimal) is ‘00B4’. The task should be abended with system completion code S222 while the abend reason code is to be set to the hex value ‘00DC0001’. A DUMP should be taken and any ESTAE issued by the target subtask should be scheduled so it can optionally recover from the abend.

```
Rc = CALLRTM(‘ABTERM’,’00B4’,’00FC8640’,’222’,’00DC0001’,’YES’,’YES’,’YES’)
```

Chapter 5

Miscellaneous MVS Services

5.1 ASID -- MVS address space identifier functions

The ASID function returns the JOBNAME when you supply an associated address space identifier. Alternatively, ASID returns the address space identifier when you supply an associated JOBNAME.

Syntax: `result = ASID(jobname,asid)`

`jobname`	jobname to be located
`asid`	address space identifier corresponding to jobname

Function returns REXX variable RESULT with the value:

`null`	If neither `jobname` nor `asid` are specified or address space is not active
`1`	If both `jobname` and `asid` are specified and the address space is active
`Ø`	If both `jobname` and `asid` are specified but there is no such combination of jobname and asid
`asid`	If only `jobname` is specified and the corresponding `asid` is active
`jobname`	If only `asid` is specified and the corresponding `jobname` is active

Examples

```
/* find asid of job LLA */
lla_asid = ASID('LLA')

/* Confirm that the MVS master address space is active */
Rc = ASID('*MASTER*','ØØØ1')

if Rc = 1 then
   say '*MASTER* is active and its asid=ØØØ1'
else
   say 'Impossible'
```

5.2 SVC -- Issue an MVS SVC from REXX

The SVC function lets you issue an SVC from REXX without Assembler language programming.

Syntax: `Rc = SVC(SVC#,R1,RØ,R15)`

`SVC#` the SVC number(expressed in decimal) to be issued

`R1,RØ,R15` 4-byte hexadecimal values to be loaded into R1, R0 and R15 respectively. No conversion of any kind is performed on these values.

The SVC function returns the following REXX variables:

`RC` - is set to the value of R15 after the SVC executes
`SVC.RØ` - REXX variable containing the value of R0 after the SVC executes
`SVC.R1` - REXX variable containing the value of R1 after the SVC executes

Example: This example demonstrates a SVC function call to issue a LOAD macro (SVC 8)

```
modname = 'IRXISPRM'               /* name of module to be loaded */
rØ@     = Getmain(4)               /* acquire storage for the RØ value*/
r1@     = Getmain(4)               /* acquire storage for the R1 value */
name@   = Getmain(8,,'BELOW')      /* acquire storage for lmod name */
x = Storage(C2X(name@),8,modname)  /* move module name into storage*/
x = Storage(C2X(rØ@),4,name@)      /* save address of lmod in rØ */
x = Storage(C2X(r1@),4,'8ØØØØØØØ'x) /* set r1 as required by the SVC */
r15 = Svc(8,r1@,rØ@)               /* issue the SVC */
/*
   R15    = Ø - Load was successful
   SVC.RØ = Entry point of the load module
   SVC.R1 = Length of the load module in double words

   R15    > Ø - Load failed
   SVC.RØ = meaningless
   SVC.R1 = meaningless
*/
```

5.3 SDKNTS -- Invoke MVS Name/Token Services

The SDKNTS function lets you invoke MVS Name/Token Services (programs IEANTCR, IEANTDL, and IEANTRT). Name / Token services are available in MVS/ESA Version 4.2.2 and subsequent releases.

Syntax:

```
rc     = SDKNTS('CR',location,name,token)
token  = SDKNTS('RT',location,name)
rc     = SDKNTS('DL',location,name)
```

`'CR'` Create a new token
`'RT'` Retrieve a token
`'DL'` Delete Name/Token pair

`location`
`1 =` TASK level Name/Token
`4 =` SYSTEM level Name Token.
This parameter requires APF authorization.

`name` A 16-byte name to be used as a key to locate the associated user token.

`token` A 16-byte user token.

The SDKNTS function returns the following REXX variables:

`RC` is set to the value of R15 after the call to one of the Name/Token services (IEANTCR, IEANTDL, or IEANTRT).

Example: Establish a TCB based anchor to store control information.

```
/* REXX */
token        = getmain(32)         /* obtain 32 bytes for control area          */
tokenX       = c2x(token)          /* token in a hex format                      */
xx           = storage(tokenX,12,'Hello There!')  /* store text in the area     */
name         = 'MYAREA'            /* to be used as Name part of the Name/Token */
tcbbased     = 1                   /* name/token is task based (local)          */
rc           = sdkntc('CR',tcbbased,name,tokenX)
...
tokenX       = sdkntc('RT',tcbbased,name)  /* retrieve token using name         */
tokenX       = substr(tokenX,1,8)  /* addr in a first 8 bytes of 16-byte token  */
say storage(tokenX,12)             /* display text from the area pointed by tok */
...
tokenX       = sdkntc('DL',tcbbased,name)  /* delete name/token pair            */
Return
```

5.4 SDKJLKUP -- Active load module look-up

The SDKJLKUP function allows you to locate a load module and identify its attributes by searching the Job Pack Queue Area (JPQA), Link Pack Area (LPA) and the Nucleus. SDKJLKUP internally issues the MVS macros CSVQUERY and NUCLKUP. The search is conducted by specifying the following search arguments: load module name or an address.

Syntax:

```
rc   = SDKJLKUP('NAME',name)
rc   = SDKJLKUP('ADDR',address)
```

where

`'NAME'`	Indicates the second parameter is the name of a load module
`'ADDR'`	Indicates the second parameter is an address
`name`	Name of the load module to be found. Must be an 8-byte variable, left justified and padded with blanks.
`address`	An address at which a load module is located. If found, the address is somewhere within the load module. Address must specify an 8 byte variable which contains a hexadecimal address, right justified with leading zeros.

The SDKJLKUP function returns the following REXX variables:

`RC` is set to the value of R15 after the call to either the CSVQUERY or NUCLKUP macros. Return codes set by SDKJLKUP function itself are:

- `41` the first parameter is neither 'NAME' nor 'ADDR'
- `42` the second parameter is absent or is not 8 bytes in length

If the search is successful, the following REXX variables are set:

`$JPQNAME`	Name of load module (major name or alias name)
`$JPQMJNM`	Major load module name (vs. an alias name)
`$JPQLOAD`	Module load address
`$JPQEPA`	Module entry point
`$JPQLEN`	Load module length
`$JPQSP`	Subpool in which module was loaded
`$JPQATR1`	Attribute byte 1 (one byte variable, see Table 4.1)
`$JPQATR2`	Attribute byte 2 (one byte variable, see Table 4.1)
`$JPQATR3`	Attribute byte 3 (one byte variable, see Table 4.1)
`$JPQPID`	Name of MVS process responsible for load

Bit settings of Attribute Byte **1** have the following meanings when set:

Bit Ø	End-of-memory deletion
1	Loaded-to-global
2	Reentrant
3	Serially reusable
4	Not loadable only
5	Overlay format
6	Alias
7	Reserved

Bit settings of Attribute Byte **2** have the following meanings when set:

Bit Ø	Authorized library
1	Authorized program
2	AMODE ANY
3	AMODE 31
4-7	Reserved

Bit settings of Attribute Byte **3** have the following meanings when set:

Bit Ø	Resident above 16 megabytes
1	Job pack area resident
2	PLPA resident
3	MLPA resident
4	FLPA resident
5	CSA resident
6-7	Reserved

Figure 4.1 Bit Settings of Attribute Bytes

Example: A comprehensive example of the usage of SDKJLKUP function can be found in member SDK#JLU of the RLX library whose low level qualifier is RLXEXEC. The following example shows how SDKJLKUP can be used to locate the load module named ISPSTART:

```
/* REXX */
call sdkjlkup ‘NAME’,‘ISPSTART’
say 'Rc.........'Rc
say '$JPQNAME...'$JPQNAME
say '$JPQMJNM...'$JPQMJNM
say '$JPQLOAD...'$JPQLOAD
say '$JPQLEN....'$JPQLEN
say '$JPQSP.....'$JPQSP
say '$JPQATR1...'$JPQATR1
say '$JPQATR2...'$JPQATR2
say '$JPQATR3...'$JPQATR3
say '$JPQPID....'$JPQPID
Return
```

Chapter 6

TSO I/O and

3270 Data Stream Functions

6.1 TGET - TSO 3270 Terminal Input function

Syntax: `buffer = TGET(type,wait)`

`type:` Indicates the type of TGET macro. Possible values are:

- `ASIS` The contents of the input buffer are only minimally edited so the REXX programmer will have to analyze the 3270 data stream in order to distinguish between input data and 3270 data stream commands.
- `EDIT` The TGET issuer receives only application data. All 3270 data stream control characters are removed.

NOTE: *These options can be abbreviated to their respective first letters ('A' or 'E'). The Default is 'EDIT'.*

`wait:` Determines whether the REXX program waits for a response from the terminal ('WAIT' option) or returns immediately regardless of terminal input ('NOWAIT' option). These options can be abbreviated to their first letter ('W' or 'N'). The default is 'WAIT'.

`buffer:` A buffer containing terminal input.

Return codes

- `0` The operation was successful. The buffer contains terminal input.
- `4` NOWAIT was specified and there was no terminal input.
- `8` An attention interrupt from the terminal terminated the TGET function before input was received.
- `12` The input buffer was not large enough to accept the entire line entered at the terminal. Subsequent TGET macro instructions will obtain the rest of the input line.
- `16` Invalid parameters were passed to TGET.
- `20` The terminal was logged off and could not be reached.
- `28` Your input buffer was not large enough to accept the entire line entered at the terminal. Subsequent TGET macro instructions will obtain the rest of the input line. The data was received in ASIS mode.
- `32` Parameter errors were detected.

Example

```
/* terminal input will be in the REXX variable buffer */
   buffer = TGET('EDIT','WAIT')
```

6.2 TPUT - TSO 3270 Terminal Output function

Syntax:

```
Rc = TPUT(buffer,type)
```

`buffer:` A buffer containing data to be output to the terminal

`type:` Indicates the type of TPUT macro. Possible values are:

- `ASIS` The contents of the buffer will undergo only basic editing. The REXX programmer must build a 3270 data stream and embed data in it.

- `EDIT` Caller receives only application data. All 3270 data stream control characters are removed and discarded.

- `FULLSCR` Specifies that the application program built a full screen 3270 data stream which should not be edited by TPUT since special 3270 features are used.

These options can be abbreviated to their respective first letters ('A', 'E', 'F'). The default is 'EDIT'.

Return codes

- `0` TPUT completed successfully.
- `8` An attention interruption occurred while TPUT was processing. The message was not sent.
- `20` The terminal was logged off and could not be reached.
- `32` No storage is available.
- `40` Parameter errors were detected

Example

```
Rc = TPUT('Hello There')   /* display message 'Hello There' */
```

6.3 3270 Data Stream Functions

In order to use the 3270 Data Stream functions supported by RLX/SDK, you must first copy the exec named RXD3270 (a member of the RLXEXEC library) into your own REXX source exec.

You should also be familiar with the principles of 3270 datastream programming which are described in detail in the following IBM publications:

```
GA27-2739  3270 Information Display System - Introduction
GA23-0059  3270 Information Display System - Data Stream Programmer's Reference
```

6.3.1 $AID - Return 3270 AID value - e.g. ENTER, PF1, etc.

Syntax: `aid = $AID(aid_name)`

`aid_name` Name of the 3270 Attention Identification byte (AID).
Valid AIDs are: `PF1-PF24, PA1, PA2, PA3, CLEAR, and SYSREQ`

Example: `aid = $AID(PF1)   /* will generate aid character 'F1'x */`

6.3.2 $ATTR -- Generate field attribute byte

Syntax: `fattr = $ATTR(attr_list)`

`fattr:` A field attribute byte

`attr_list:` List of field attributes to be specified in the attribute byte returned by the $ATTR function.

Valid attributes are:

PROT	Protected field
HI	High intensity display
NUM	Simulate numeric key depressed
NON	Non-displayable field
MDT	Modify Data Tag (MDT) flag is on

Example: Generate a field attribute byte with the following characteristics: Protected and High display intensity:

fattr = $ATTR('PROT HI')

6.3.3 $BA2DA - Convert 3270 buffer address to decimal address

---

Syntax: `rowcol = $BA2DA(buffaddr)`

`rowcol` Concatenation of row value in decimal, ',' column value

`buffaddr` 3270 buffer address (2 bytes)

Example: Convert a 3270 buffer address (hex literal C2F8'x) to its row column equivalent (3,25)

```
rowcol = $BA2DA('C2F8'x)     /* rowcol variable contains '3,25' */
parse var rowcol row,col     /* row = 3 and col = 25 */
```

6.3.4 $CMD - Generate 3270 commands

Syntax: `cmdbyte = $CMD(3270cmd)`

`cmdbyte`	One byte containing a 3270 command
`3270cmd`	A valid 3270 command:

`W`	Write command
`EW`	Erase/Write command
`EWA`	Erase/Write alternate command
`RB`	Read Buffer command
`RM`	Read Modified command
`RMA`	Read Modified All command
`EAU`	Erase All Unprotected command
`WSF`	Write Structured Field

Example: Generate the 3270 Erase Write command

```
cmd = $CMD('EW')
```

6.3.5 $CLEAR - Clear the screen

Syntax: `rc = $CLEAR()`

`This command requires no parameters`

6.3.6 $COMP - Compress 3270 data stream using RA order

Syntax: `compstr = $COMP(stream)`

`compstr` Compressed 3270 data stream

`stream` Uncompressed 3270 data stream

Example: Compress 3270 data stream

```
compstr = $COMP(stream)
```

6.3.7 $DA2BA - Convert decimal (Row, Col) to 3270 buffer address

---

Syntax: `buffaddr = $DA2BA(row,col)`

`buffaddr` 2 byte 3270 buffer address

`row,col` Decimal row value , decimal column value

Example: Convert the decimal row and column value (3,25) to its corresponding 3270 buffer address. `$DA2BA` assigns the value 'C2F8'x to the REXX variable `buffaddr`.

```
buffaddr = $BA2BA(3,25)
```

6.3.8 $EUA - Generate Erase Unprotected to Address order

Syntax: `eua = $EUA(row,col)`

`eua`	Erase Unprotected to Address order
`row,col`	Decimal row and column

Example: `eua = $EUA(1,10)`

6.3.9 $IC - Generate Insert Cursor order

Syntax: `ic = $IC()`

`ic`	Insert Cursor order

Example: `ic = $IC()`

6.3.10 $PT - Generate Program Tab order

Syntax: `pt = $PT()`

`pt`	Program Tab order

Example: `pt = $PT()`

6.3.11 $RA - Generate Repeat to Address order

Syntax: `ra = $RA(row,col,char)`

`ra`	Repeat to Address order
`row`	Decimal row
`col`	Decimal column
`char`	Character to be repeated

Example: Generate a Repeat to Address order to blank out the screen (using the space character ' ') from the current buffer address to the buffer address that corresponds to row 20 and column 60.

```
ra = $RA(2Ø,6Ø,' ')
```

6.3.12 $SBA - Generate Start Buffer Address order

Syntax: `sba = $SBA(row,col)`

`sba`	Start Buffer Address Order
`row`	Decimal row
`col`	Decimal column

Example: Generate Start Buffer Address order from row 10 column 35:

```
sba = $SBA(1Ø,35)
```

6.3.13 $SF - Generate Start Field order

(PROT / NUM / HI / NON / MOD)

Syntax: `sf = $SF(attr_list)`

`sf`	Start Field order
`attr_list`	Field attribute list (see $ATTR function)

Example: Generate Start Field order for a numeric field:

```
sf = $SF('NUM')
```

6.3.14 $WCC - Generate Write Control Character

Syntax: `wcc = $WCC(attrlist)`

`wcc`	Write Command Control byte
`attrlist`	List of WCC attributes:

`ALARM`	Sound alarm
`RESTORE`	Restore keyboard is on
`RESET`	Reset MDT bit

Example: Generate a Write command and WCC byte to reset the keyboard and sound the alarm:

```
command = $CMD('W') || $WCC('RESET ALARM')
```

6.4 TPG -- TSO 3270 Terminal Query

Use the TPG function to transmit a command to the terminal and cause the device to respond immediately with input. The TPG function works with any terminal that supports the QUERY function.

Syntax: `rc = TPG(command, 'WAIT' | 'NOWAIT', 'NOHOLD' | 'HOLD')`

`command`	An immediate command (QUERY, WRITE SF, ...)
`WAIT`	Specifies that control is not returned to the program that issued TPG function until the output line is placed into a terminal output buffer. If no buffers are available, the issuing program is placed into a wait state until buffers become available, and the output line is placed into them.
`NOWAIT`	Specifies that control is returned to the program that issued TPG regardless of the availability of output buffers. RC indicates operation completion.
`NOHOLD`	Indicates that control is returned to the program that issued the TPG function as soon as the output line is placed in terminal output buffers.
`HOLD`	Specifies the program that issues the TPG function cannot continue its processing until the output line is written to the terminal or is deleted.

Return codes

`0`	TPG completed successfully.
`4`	NOWAIT was specified but no terminal buffer was available.
`8`	An attention interruption occurred while TPG was processing
`16`	Invalid parameter passed upon input.
`20`	The terminal was logged off and could not be reached.
`40`	Command requires a parameter that was not specified

Example

Issue the 3270 Read Buffer command using the TPG function. Then receive the contents of the current 3270 buffer via the TGET function. The buffer is formatted into 80-character lines and displayed with the SDKBRIF function.

```
/* Rexx */
command         = 'F2'x
rc              = TPG(command,'nowait','nohold')
if rc          \= Ø then do
   say 'TPG Rc='rc
   Return
end

buffer          = TGET('ASIS','W')
say 'TGET rc='rc 'Length='length(buffer)

j               = Ø
do i            = 1 to length(buffer) by 8Ø
   j            = j + 1
   ol.j         = substr(buffer,i,8Ø)
end
ol.Ø            = j
call sdkbrif 'ol.',,'Output from command='c2x(command),
                    'Buffer length='length(buffer)
Return
```

Chapter 7

REXX Read, Browse and Edit Functions

7.1 SDKBRIF - Browse REXX stemmed variables

The SDKBRIF service displays the contents of REXX stem variables. SDKBRIF makes use of the ISPF BRIF facility to display data in memory -- without the use of temporary datasets or dataset I/O. This affords high performance, coupled with the full range of function available with ISPF Browse.

Syntax:

```
Rc = SDKBRIF(stemname,maxlrecl,title,panel,format,DBCS,cmdproc)
```

where

`stemname` name of the REXX stem to be browsed. If not specified, the default stem name is 'SDKBRIF.'

`maxlrecl` Maximum logical record length of the data to be browsed. If not specified, the default maximum length is 32,760.

`title` A literal up to 54 characters long to appear in the title of the panel on which the data is displayed. The default title is the name of the REXX stem being browsed.

`panel` Name of the panel to be used by the ISPF BRIF service. The default panel is named ISRBPROBF.

`format` Name of the format to be used by the BRIF service

`DBCS` Indicates that Double Byte Characters are being displayed

`cmdproc` Name of the primary command processing routine to be used by the BRIF dialog. This routine must be the name of a valid load module. You may use AcceleREXX to compile REXX execs that will serve as cmdprocs.

Return Code

0 Processing was successful

8 Parameter error -- non numerical maxlrecl
or failed to load a cmdproc

Example:

Browse contents of a stem ‘s.’ using a maximum lrecl of 200.

```
Call SdkBrif 's.',200,'Example of SdkBrif display: Stem s.'
```

7.2 SDKEDIF - Edit REXX stemmed variables

The SDKEDIF service lets you edit the contents of REXX stem variables. SDKEDIF uses the ISPF EDIF facility to edit data directly in memory -- without the use of temporary datasets or dataset I/O. This affords high performance, coupled with the full range of function available with the ISPF Edit facility.

Syntax: `Rc = SDKEDIF(stemname,maxlrecl,title,profile,panel,macro,format,DBCS,cmdproc)`

where

`stemname` REXX stem name to be edited.
If not specified, the default stem name is 'SDKEDIF.'

`maxlrecl` Maximum logical record length of the data to be edited. If not specified, the value 255 is used.

`title` A literal up to 54 characters long to appear in the title of the panel. The default title is the name of the REXX stem being edited.

`profile` Name of the Edit profile to be used. The default edit profile name is EDIFPROF.

`panel` Name of the ISPF panel to be used by the EDIF service. The default panel name is ISPEDDE.

`macro` Name of the initial edit macro.

`format` Name of the format to be used by the EDIF service.

`DBCS` Indicates that Double Byte Characters are present in the data to be edited.

`cmdproc` Name of the primary command processing routine to be used by the EDIF dialog. This routine must be the name of a valid load module. You may use AcceleREXX to compile REXX execs that will serve as cmdprocs.

Return Code

0 EDIF was successful

4 No changes were made to the data present in the REXX stem

8 Parameter error: non numerical `maxlrecl` or failed to load a cmdproc

Example: Edit the contents of the REXX stem 's.' Specify a maximum lrecl of 200.

```
Call  SdkEdif 's.',200,'Example of SdkEdif display: Stem s.'
```

7.3 SDKREAD - Read into REXX stemmed variables

Read the records within a member of a partitioned dataset (PDS) into an array of REXX compound symbols that share a common stem. One or more members of a PDS may be accessed without having to repeatedly free and re-allocate the file.

Syntax: `Rc = SDKREAD(ddname,member,stemname)`

where

`ddname`	Name of the file to which the dataset is allocated
`member`	Optional member name if the allocated dataset is a PDS
`stemname`	Name of a REXX stem (e.g. 'name.') into which records will be read.

Return code

0	Successful execution, dataset records were loaded into REXX variables
8	RFA initialization failed
12	Failed to load the module RFP$TBLS
16	The file 'ddname' is not allocated
20	Failed to build a DCB for the dataset
24	Failed to open the file
28	No DDname parameter was passed
32	No member name was passed
36	No stem name was passed
44	The specified member name was not found in the PDS

Example: Read the member named ISR@PRIM of ISPPLIB PDS into the REXX stem named 'line.'

```
Rc = SDKREAD('ispplib','isr@prim','line.')
```

7.4 SDKWRITE -- Write from REXX stemmed variables

Write records into a sequential dataset or a member of a PDS from REXX stemmed variables.

Syntax: `Rc = SDKWRITE(ddname,member,stemname)`

`DDname`	Name of the file to which the output dataset is allocated
`member`	Optional member name if the allocated dataset is a PDS
`stemname`	Name of the REXX stem (e.g. 'name.') from which records will be written

---

Return code

0	Successfull execution
1	Record(s) were truncated
12	failed to load RFP$TBLS
16	DD name is not allocated
20	failed to build a DCB for the data set
24	failed to open the file
28	no DD statement passed
32	no member name passed as a parameter
44	STOW error
48	Failure to get DSN from JFCB
52	OBTAIN macro error

Example:

Write a PDS member named TESTLINE from the REXX stem 'line.'. The output file is allocated to the DDname 'workfile'.

```
line.0 = 3
line.1 = 'Line 1'
line.2 = 'Line 2'
line.3 = 'Line 3'
Rc = SDKWRITE('workfile','testline','line.')
```

7.5 SDKDDNAM -- Check if the specified DDname is allocated

Syntax: `Rc = SDKDDNAM(ddname)`

`ddname` Name of the file to which the dataset is allocated

Return code

0	DDNAME is not allocated
1	DDNAME is allocated

Example:

Check whether file ISPLLIB is allocated:

```
if SDKDDNAM('ISPLLIB') then
   say 'ISPLLIB is allocated'
else
   say 'ISPLLIB is not allocated'
```

7.6 BLDL -- REXX interface to the MVS BLDL macro

The BLDL function enables you to obtain PDS directory entry information directly from your REXX execs -- without assembler programming. You can search for members in a PDS allocated to a specific DDNAME or use the standard MVS search order. BLDL returns directory information into the REXX variable RESULT.

Syntax: `DIRENTRY = BLDL(member,ddname)`

`member` PDS member name - required

`ddname` Optional DDNAME

Return codes after call:

RC variable

0	Successful execution, directory information was returned in the RESULT variable
4	Member was not found
8	No member name was provided
12	Dataset name is not a PDS
16	Failed to open DDname

RESULT variable: If BLDL is successful, the RESULT variable is set to the character value of the following:

`TT R K Z C UD`

`TT` Relative track number for the beginning of the dataset

`R` The relative block record number on the track indicated by TTR

`K` indicates the concatenation number of the dataset.
For the first or only dataset this value is 0.

`Z` indicates where the system found the directory entry:

0	Private library
1	Link library
2	Job, task, or step library
3 - 255	job, task, or step library of parent task n, where n = Z-2

`C` Indicates the type: member or alias.

Bit:	**Meaning:**
0=0	Indicates a member name
0=1	Indicates an alias
1-2	Indicates the number of TTRN fields (max of 3) in user data field
3-7	indicates the total number of halfwords in the user data field

`UD` User data in the directory entry

Example:

Determine whether libraries allocated to the DDname ISPLLIB contain a load module named 'ISRISPRM'. If that load module is found, then RC = 0 and the RESULT variable is set to values in the `TT R Z C UD` format

```
CALL BLDL 'IRXISPRM','ISPLLIB'
```

7.7 STOW -- REXX interface to the MVS STOW macro

The STOW function provides a native REXX interface to the MVS STOW macro -- with full support of all STOW facilities. The STOW function allows you to ADD, CHANGE, UPDATE and DELETE directory entries in a PDS.

Syntax:

```
rc = STOW(function,ddname,member,type,newname,userfld)
```

`function`	Required parameter: ADD = add new directory entry DELETE = delete directory entry REPLACE = replace an existing directory entry CHANGE = change the member name of the directory entry
`ddname`	Required parameter: Optional DDname of the PDS where the member resides. If omitted, the tasklib, steplib or joblib will be used
`member`	Required parameter: The member name of the directory entry to be processed
`type`	Optional: Describes the directory entry type as either MEMBER or ALIAS. The default is MEMBER
`newname`	Optional parameter: A new name to be assigned to an alias name or member name
`userfld`	Optional: 0 - 62 bytes of user data to be placed in the directory entry

---

RC variable

0	STOW was successful
4-24	See return codes from the STOW macro instruction
28	Invalid function call parameters
32	RFA initialization failed
36	Requested DDname was not allocated
40	Failed to build DCB for DDname
44	Failed to OPEN DDname

Example:

Create an alias named 'ISR@PRM1' for the member 'ISRr@PRIM' located in the dataset allocated to the ISPLLIB DD statement. Place the user data 'TEST STOW' in the user field of the PDS directory for ISR@PRM1.

```
Rc  =  STOW('add','ispplib','isr@prim','alias','isr@prm1','TEST  STOW')
```

Chapter 8

Parsing, Tokenizing and Sorting

8.1 SDKSCAN - Tokenize a character string into REXX stemmed variables

The **SDKSCAN** function parses a string into its constituent tokens using a caller supplied set of token delimiters. SDKSCAN places the tokens into the REXX stemmed array specified by the caller.

Syntax: `Rc=SDKSCAN(string,{delims},{stemname},{Noblanks},{Literals},{Comments})`

where

`string` — A character string to be scanned into tokens. This parameter is required.

`delims` — The set of delimiters with which to scan and parse the string. If not specified, the following default delimiters are used: `x'010240',c':+-*|\()/"&=;<>'`

`stemname` — The name of the REXX stem into which SDKSCAN will place the scanned tokens. If not specified, the default stem name 'SDKSCAN.' is used. SDKSCAN places the number of scanned tokens into the Zero-th element of the array.

`Noblanks` — If specified, blank delimiters are discarded rather than being treated as separate tokens. Note that blanks within literal strings are always preserved. (The Noblanks parameter can be abbreviated as `N`.)

`Literals` — If specified, literal strings delineated by either single quotes ' or double quotes " will be processed as a single token. (The literals parameter can be abbreviated as `L`.)

`Comments` — If specified, REXX-style comments (`/* comments */`) are processed as one token. (This parameter can be abbreviated as `C`.)

Return Codes

0 Scan was successful

1 When Literal assembly is requested, this code indicates that an ending quote or double quote was not found for one of the literals.

2 When Comment assembly is requested, this code indicates that an end-of-comment delimiter '*/' was not found for one of the comments.

8 One or more input parameter(s) are in error.

Example:

```
Rc = SDKSCAN("keyword(k1) = 'literal 1'",,'S.','L','N')
```

In this example the string `"keyword(k1) = 'literal 1'"` is scanned into tokens and placed in the stem `'S.'`. The default set of delimiters is used. Literals are to be assembled into a single token and non-significant blanks should be deleted.

Result:
```
Rc   =  Ø
s.Ø  =  6
s.1  =  'keyword'
s.2  =  '('
s.3  =  'k1'
s.4  =  ')'
s.5  =  '='
s.6  =  'literal 1'
```

8.2 SDKSORT - Sort REXX stemmed variables

The SDKSORT function can be used to sort a series of blank delimited words within a REXX stemmed array in either ascending or descending sequence.

Syntax:

```
Format1: Rc = SDKSORT(stemname{,FldStart},{FldLength},{order})
Format2: Rc = SDKSORT(stemname{,FldDescr}{,FldDescr}...)
```

where

`stemname` The name of the REXX stem whose elements are to be sorted. The default stem name is SDKSORT.

Format 1 can be used when records contain only a single sort field. Format 1 preserves compatibility with the prior version of this function.

`FldStart` The starting position within the sort field. The default position is 1.

`FldLength` Length of the sort field. The default field length is that of the entire variable, to a maximum of 32,760.

`order` Direction of the sort, where A denotes ascending sort sequence (the default) and D specifies a descending collating sequence for the sort.

Format 2 must be used when records contain multiple sort fields. Format represents the new syntax of this function.

`FldDescr` Sorted field descriptor whose general format is as follows:

```
(fldstart,fldlength,CompType,order)
```

`CompType` Comparison data type. At present, only the character data type denoted by 'CH', is defined.

NOTES: ***(1)*** *The size of the REXX Stem to be sorted may be supplied in the zero-th element of the stem. If the zero-th element is not defined or has a non-numeric value, then the size of the stemmed array is determined by checking whether subsequent stem elements are defined (beginning from 1,2, etc.) The first undefined stem element is assumed to end the stemmed array.*

(2) *The comparison operation used by SDKSORT is character based. If, for example, you need to sort a stem containing 10 and 5, 5 follows 10 (in ascending order) because only one character (the common length) is compared.*

Return Codes

0	The sort was successful
4	The stem was not defined or is empty
8	The parameter(s) `fldstart` or `fldlength` are non-numerical or are outside the span of the character string. The sort is aborted.
12	The sort field is outside the record's range
42	`The FldStart` parameter is non-numeric
43	`The FldLength` parameter is non-numeric
44	`Order` is neither `'A'` nor `'D'`
45	Incomplete `FldDescr` parameter (Format 2)
46	No open parenthesis parameter `'('` found in `FldDescr` (format 2)
47	`FldStart` is not numeric (Format 2)
48	`FldLength` is non-numeric (Format 2)
49	Comma was not found but was expected in `FldDescr` (Format 2)
50	No closing parenthesis `')'` was found in `FldDescr` (Format 2)
51	`CompType 'CH'` was not found (Format 2)
52	`Stemname` is not defined or contains undefined element(s)
53	Getmain failed. Either the stem has too many elements or the combined size of all variable values exceeds the limit which can be acquired dynamically.

Example 1: Sort stem `'S.'` in ascending order using the entire contents of the REXX variables which comprise the stem 'S.'

```
Rc = SDKSORT('s.',,'A')
```

Result:	***Before sort***	***After sort***
	s.0 = 4	s.0 = 4
	s.1 = 8	s.1 = 1
	s.2 = 5	s.2 = 5
	s.3 = 9	s.3 = 8
	s.4 = 1	s.4 = 9

Example 2: Sort stem `'S.'` using Format 2 which designates 5 sort fields:

```
CALL SDKSORT  'w.','(1,3,ch,a)','(5,3,ch,d)','(9,3,ch,a)',,
                  '(13,3,ch,a)','(17,3,ch,a)'
```

8.3 SDKWSORT - Sort words in a string

Sort the blank-delimited words of a string.

Syntax: `Rc = SDKWSORT(string,order)`

where

`string` The string containing the words to be sorted.
This parameter is required.

`order` Sort sequence: A=ascending or D=descending

Return A sorted string of words delimited by a single blank. If no string is specified to SDKWSORT, an empty string is returned.

Example:

```
sorted  =  SdkWsort('zz 2ya a bba sssss bbb xxxx z aaa','A')
```

After the sort, the REXX variable `sorted` contains:

```
'a aaa bba bbb sssss xxxx z zz 2ya'
```

Chapter 9

Data Conversion and Mapping Functions

9.1 A2E - Convert ASCII to EBCDIC

The A2E function translates a string whose data is encoded in ASCII into an equivalent string encoded in the EBCDIC character set.

Syntax: `ebcdic = A2E(ascii)`

`ascii` An ASCII character string between 1 and 256 characters long to be converted into EBCDIC

`ebcdic` The resultant EBCDIC string converted from ASCII

Example: `result = A2E(ascii)`

9.2 D2P - Convert Decimal to Packed

The D2P function converts a character string representation of a numeric value (i.e., an edited decimal number) into an internal numeric value in packed decimal format.

Syntax: `result = D2P(decimal,resbytes)`

`decimal` Decimal number in EBCDIC, e.g. +1234.56

`resbytes` The number of bytes in the packed decimal result. This number may be between 1 and 8 bytes long.

Example: `packed = P2D(1234.56)`

After the function is executed, `packed = '123456C'x`

9.3 E2A - Convert EBCDIC to ASCII

Syntax: `ascii = E2A(ebcdic)`

`ebcdic` An EBCDIC character string (1 to 256 characters long) to be converted to ASCII code

`ascii` Resultant ASCII string converted from EBCDIC

Example: `result = A2E(ebcdic)`

9.4 P2D - Convert Packed to Decimal

The P2D function converts a packed decimal number into an edited decimal representation of that number.

Syntax: `rc = SDKMAP(function,address,map{,index1,index2,index3}`

Example: `decimal = P2D('123456C'x,2)`

After the function is executed, `decimal = 1234.56`

9.5 D2F - Convert a REXX decimal number to S/390 floating point

---

Syntax: `floatnum = D2F(decnum,format)`

`floatnum` S/390 floating point number in either `SINGLE` format (4 bytes) or `DOUBLE` format (8 bytes), depending upon the format parameter -- e.g:

Example 1: `'4122B841'X` is a SINGLE precision FLOATING POINT number corresponding to the decimal number 2.17

Example 1: `'41323D70A3D70A3D'X` is a DOUBLE precision FLOATING POINT number corresponding to the decimal number 3.14

`decnum` REXX decimal number, e.g. 3.14

`format` The format of the floating point number: either 'SINGLE' or 'DOUBLE'

Returned REXX variables:

Result: Contains the value of the floating point number (floatnum)

RC	contains a return code set to one of the following values:
0	conversion successful
4	significance or underflow condition
8	overflow condition
12	conversion error or null input
16	incorrect decnum: not specified or longer than 32 bytes.

Example:

```
float = D2F(2.65)
if rc <= 4 then
   say 'Conversion successful, hex value of float='C2H(float)
else
   say 'Conversion failed, RC='rc
```

9.6 F2D - Convert a S/390 floating point number to a REXX decimal number

---

Syntax: `decnum = F2D(floatnum,format)`

`decnum` result value as a REXX decimal number. e.g. 3.14

`floatnum` S/390 floating point number in either SINGLE format (4 bytes) or DOUBLE format (8 bytes) depending upon the format parameter.

Example 1: `'4122B841'X` is a SINGLE precision FLOATING POINT number corresponding to the decimal number 2.17.

Example 2: `'41323D70A3D70A3D'X` is a DOUBLE precision FLOATING POINT number corresponding to the decimal number 3.14

`format` The format of the floating point number:
either 'SINGLE' or 'DOUBLE'

Returned REXX variables:

Result: Contains the value of the REXX decimal number (decnum)

RC	contains a return code set to one of the following values:
0	conversion successful
4	significance or underflow error; conversion successful
8	overflow
12	conversion error or null input
16	incorrect floatnum (not specified)

Example:

```
decnum = F2D('412A6666'X)
if rc <= 4 then
   say 'Conversion successful, decimal number='decnum
else
   say 'Conversion failed, RC='rc
```

9.7 SDKMAP - Map REXX variables from / to data area

The SDKMAP function performs mapping between REXX variables and an area of memory. The FETCH function ***creates*** REXX variables from an area in memory in accordance with the specified map while the STORE function ***maps*** the values of REXX variables into a block of data in memory in accordance with the specified map.

Syntax: `rc = SDKMAP(function,address,map[,index1,index2,index3])`

`function` One of the following functions:

- FETCH Create REXX variables from the block of data in accordance with the specified map; or
- STORE Fetch values of REXX variables and create a block of data in accordance with the specified map

In both cases, all data conversions are performed in accordance with the data types defined by the map

`address` The address of the block of memory used as input (in the case of FETCH) or as an output (in the case of STORE)

`map` Name of the map used as input specifications for REXX variables

`index1,index2,`
`index3` Optional - up to three levels of indices to be used to create the names of REXX variables (e.g. ABC.1.3.4)

A map is a set of descriptors each of which has the following format:

```
MAPENTRY DC    AL4           LENGTH OF VARIABLE NAME
         DC    AL4           VARIABLE VALUE OFFSET
         DC    AL4           FIELD LENGTH
         DC    CL1           FIELD DATA TYPE
         DC    CL1           FLAG
         DC    ØC            Variable name (length is variable)
```

Field data types:

```
B   - a binary number (binary to decimal conversion)
C   - a character string - no conversion
H   - a string to be converted to hex digits
T   - Time in STCK format: mm/dd/yy-hh.mm.ss.hhhhhh
D   - Delta time as STCK: ØØ/ØØ/ØØ-hh.mm.ss.hhhhhh
F   - some special value (not implemented)
```

The last map entry starts with X'FF' .

Example:

```
/* rexx */
address tso
"alloc dd(MAPS) da('rlx.test.maps') shr reu"                                  (1)
"alloc dd(DATA) da('rlx.test.data') shr reu"                                  (2)
map  = sdkMG('MAPS','BLOCK1')        /* Map get function         */    (3)
rc   = LOAD('BLOCK1','DATA')         /* Load parameters          */    (4)
call sdkMAP 'STORE',sdkload@,map     /* Format map into rexx vars */    (5)
call Dumpvars                        /* display rexx variables   */    (6)
return

/*-------------------------------------------------------------------*/
/*  List all defined rexx variables                                  */
/*-------------------------------------------------------------------*/
Dumpvars :
trace 'O'
   vars = sdkVARS('REXXVARS')
   v.Ø = words(vars)
   do i = 1 to v.Ø
      var = word(vars,i)
      v.i = left(var,2Ø,'.') left(length(value(var)),5) value(var)
   end
   call sdkBRIF 'v.'
return
```

where

(1) Allocate a PDS containing one or more object module maps. Refer to description of the SDKMG function for details about generating object maps.

(2) Allocate a PDS containing load modules which will be used as the data area to be mapped

(3) Create a map from the object module (see the SDKMG function)

(4) Load a load module named BLOCK1 from the DDname DATA to be used as a data area. The address of the load module is returned in the REXX variable `sdkload@`.

(5) Create a series of REXX variables described by the map named `map` and populate them with the data extracted from the data area at the address `sdkload@` in accordance with the map. All data conversions from internal format to REXX character strings are performed automatically by the SDKMAP function.

(6) The Dumpvars REXX subroutine will obtain (via the SDKVARS function) the names of all active REXX variables and then display them in a scrollable format via the SDKBRIF function.

9.8 SDKMG - Create a map from an assembled object

The SDKMG function creates an internal map from the SYM records read from an assembled object module. The format of this internal map is descrbed in Section 9.5. The maps generated by the SDKMG function can be used as input to the SDKMAP function.

Syntax: `map = SDKMG(ddname,object)`

`ddname` The file to which a PDS containing object modules generated by assembler with the TEST parameter is allocated.

`object` The name of the object module generated by the assembler with the TEST parameter.

`map` A variable which will contain the map generated by the SDKMG function if execution is successful.

The sample JCL below illustrates how to create a map for the CVT (Communication Vector Table -- a major MVS control block).

```
//ASM      EXEC PGM=ASMA90,
//         PARM=(DECK,TERM,BATCH,OBJ,TEST)
//SYSLIB   DD  DSN=SYS1.MACLIB,DISP=SHR
//         DD  DSN=SYS1.AMODGEN,DISP=SHR
//SYSUT1   DD  DSN=&SYSUT1,UNIT=VIO,SPACE=(1700,(600,100))
//SYSUT2   DD  DSN=&SYSUT2,UNIT=VIO,SPACE=(1700,(300,50))
//SYSUT3   DD  DSN=&SYSUT3,UNIT=VIO,SPACE=(1700,(300,50))
//SYSLIN   DD  DUMMY
//SYSTERM  DD  SYSOUT=*,DCB=BLKSIZE=1089
//SYSPRINT DD  SYSOUT=*,DCB=BLKSIZE=1089
//SYSPUNCH DD  DSN=RLX.TEST.MAPS(CVT),DISP=SHR
//SYSIN    DD  *
CVT      CSECT
         CVT   PREFIX=YES,DSECT=YES,LIST=YES
         END
//
```

Example:

Create a set of REXX variables that map the contents of each field of the CVT control block. The CVT contains more than 300 fields so performing this conversion manually is a very long and tedious process.

```
/* rexx */
address tso
   "alloc dd(maps) da(paul.rcxobj) shr reu"
   map = sdkMG('maps','cvt')            /* create a map from object */
   cvt@ = c2x(storage(10,4))            /* address of CVT */
   call sdkMAP 'STORE',cvt@,map         /* create CVT variables */
   call Dumpvars                        /* display rexx variables */
return
```

9.9 SDKMGP - Create a map from a PL/I INCLUDE

The SDKMGP function creates an internal map for the data structure associated with a PL/I INCLUDE from the SYSADATA records generated by the PL/I compiler. The format of this internal map is described in Section 9.5. Maps generated by SDKMGP can in turn be used as input to the SDKMAP function.

Syntax: `map = SDKMGP(ddname,object)`

`ddname` The file to which an object module library is allocated. This PDS should contain object modules that were generated by an Assembly with the TEST parameter

`object` The name of the object module

`map` The name of a variable into which SDKMGP should return a map

---

The sample JCL in the figure below illustrates how to create a SYSADATA member for a PL/I INCLUDE, using the Enterprise PL/I Compiler for z/OS V3.3 (or later release).

```
//jobname  JOB
//PLIMAP   PROC MAP=
//*___________________________________________________________________
//* ENTERPRISE PL/I FOR Z/OS V3.3 OR LATER
//*___________________________________________________________________
//PLI    EXEC PGM=IBMZPLI,REGION=0M,PARM='XINFO(NODEF,NOXML,NOMSG,SYM)'
//STEPLIB  DD  DISP=SHR,DSN=PLI.IEL330.SIBMZCMP
//SYSLIB   DD  DISP=SHR,DSN=USER.PLI    *PL/I %INCLUDES
//SYSPRINT DD  SYSOUT=*
//SYSLIN   DD  DUMMY
//SYSUT1   DD  DSN=&SYSUT1,UNIT=SYSALLDA,DCB=BLKSIZE=1024,
//             SPACE=(1024,(200,50),,CONTIG,ROUND)
//SYSADATA DD  DSN=USER.MAPS(&MAP),DISP=SHR
//         PEND
//*
//MAP      EXEC PLIMAP,MAP=HOSTVAR
//PLI.SYSIN DD  *
 $MAPPER: PROC;
 %INCLUDE HOSTVAR;
 END $MAPPER;
/*
```

In the example above, the PL/I structure named HOSTVAR is copied via %INCLUDE into a dummy program named $MAPPER. The HOSTVAR INCLUDE resides within the copy library named USER.PLI. The PL/I compiler stores the SYSADATA output as a member of the PDS named USER.MAPS (whose attributes are LRECL=80 and RECFM=FB).

To create a SYSDATA member for your own PL/I INCLUDE structure, proceed as follows. First, replace the string ?pli.include.library’ in the example below with the name of your PL/I INCLUDE library. Next, replace the strings ?include1? and ?include2? with the name of the PL/I INCLUDE you wish to map.

```
//jobname  JOB
//PLIMAP   PROC MAP=
//*___________________________________________________________________
//* ENTERPRISE PL/I FOR Z/OS V3.3 OR LATER
//*___________________________________________________________________
//PLI    EXEC PGM=IBMZPLI,REGION=0M,PARM='XINFO(NODEF,NOXML,NOMSG,SYM)'
//STEPLIB  DD  DISP=SHR,DSN=PLI.IEL330.SIBMZCMP
//SYSLIB   DD  DISP=SHR,DSN=?pli.include.library?    *PL/I %INCLUDES
//SYSPRINT DD  SYSOUT=*
//SYSLIN   DD  DUMMY
//SYSUT1   DD  DSN=&SYSUT1,UNIT=SYSALLDA,DCB=BLKSIZE=1024,
//             SPACE=(1024,(200,50),,CONTIG,ROUND)
//SYSADATA DD  DSN=?pli.map.library(&MAP)?,DISP=SHR
//         PEND
//*
//MAP      EXEC PLIMAP,MAP=?include?
//PLI.SYSIN DD  *
 $MAPPER: PROC;
 %INCLUDE ?include?;
 END $MAPPER;
/*
```

Once a SYSADATA member exists for the PL/I INCLUDE, you can use the SDKMGP function to transform the SYSADATA into a map that the SDKMAP function can use to create discrete REXX variables that correspond to fields with a PL/I data structure.

Example:

The following example illustrates how to use the SDKMGP and SDKMAP functions to map records from a file into discrete REXX variables that correspond to the field structure defined by the PL/I INCLUDE.

> ***NOTE:*** the SDKMAP function converts the contents of non-character fields into character format and places them in REXX variables.

```
/* rexx */
Address TSO
 "ALLOC DD(MAPS) DA('"USER.MAPS"') SHR REU"  /* User PDS containing SYSADATA       */
 map      = SdkMGP('MAPS',’HOSTVAR’)         /* Create an internal map             */
 "ALLOC DD(INFILE) DA('"USER.FILE"') SHR     /* File whose records are to be mapped */
 Call SdkREAD 'INFILE',,'f.'                 /* Read file records into f. stem     */
 Do i     = 1 To f.0                         /* For every file record              */
    l     = Length(f.i)                      /* Obtain length of the record        */
    addr@ = C2X(Getmain(l))                  /* Obtain storage for the record      */
    x     = Storage(addr@,l,f.i)             /* Copy record into storage           */
    Call sdkMAP 'STORE',addr@,map            /* Create REXX vars for the record    */
End
Return
```

Chapter 10

Multi-Tasking and

Interprocess Communication

10.1 POST - Signal event completion

The POST function provides a native REXX interface to the MVS POST system service. POST provides a means to signal the completion of an event.

Syntax: `Rc = POST(ecb@,compcode)`

`ecb@` Specifies the binary address (4-byte fullword) of the ECB (event control block) to be POSTed. This address is returned by the GETMAIN function into a REXX variable. A WAIT service that references the address of this ecb must have been issued previously so that the ecb to be POSTed must be in a wait state.

`compcode` The completion code to be placed in the ECB. This code can subsequently be checked by the issuer of the WAIT function. The `compcode` parameter should be specified as a printable decimal number.

RC variable Is set to a return code from the MVS POST macro

Example: Post the `ecb` whose address is in the REXX variable `ecb@` with a completion code of 255.

```
Rc = POST(ecb@,255)
```

10.2 WAIT - Wait for event completion

The WAIT function provides a native REXX interface to the MVS WAIT system service. WAIT provides a means to halt the execution of a task and program until either a timer expires or an application defined event completes (as signaled by the POST system service).

Syntax:

```
Rc = WAIT('ECB',ecb@,longwait)
Rc = WAIT('ECBLIST',ecblist@,eventno,longwait)
Rc = WAIT('SEC',seconds)
```

`ECB / ECBLIST / SEC` denote the types of WAIT functions supported. One of these parameters must be coded exactly as shown:

`ecb@` Specifies the binary address (a 4-byte fullword) of the ECB (event control block) for which the WAIT is issued. The RLX/SDK's GETMAIN function returns this address into a REXX variable. Before issuing a WAIT, the 4 bytes of storage that comprise the ECB (at the storage address `ecb@`) must be cleared to binary zeros. This parameter is required when the WAIT type is `ECB`.

`ecblist@` The binary address of a list of ECBs built in accordance with the requirements of the MVS WAIT macro. The ECBLIST specifies a list of addresses. Each address is a 4-byte field aligned on a full-word boundary. The last ECB address must have its high order bit turned on to designate the end of the list. If the ECBLIST is not built and initialized correctly, the WAIT will fail. This list should only be specified with a WAIT type of `ECBLIST`.

`eventno` The number of events (i.e., number of POSTed ECBs) whose completion must be signaled in order to resume execution of the waiting task and program. This parameter is only valid when the WAIT type is `ECBLIST`. The default number of events which must complete is 1.

`longwait` Indicates whether the task can enter a long wait (such as for I/O to the terminal). Possible values for this parameter are YES and NO.

`seconds` The number of seconds to wait before control is returned to the issuing REXX program. The valid range for this parameter is between 0 and 86,400 seconds (24 hours).

RC variable

Contains a return code after WAIT macro

Example 1

Obtain an area of storage for an ECB and then issue the WAIT function.

```
ecb@ = GETMAIN(4)                       /* Obtain an area for the ECB */
Rc = Storage(C2X(ecb@),4,'00000000'x)   /* Clear the ECB to x'00'     */
Rc = WAIT('ECB',ecb@)                   /* Wait for event completion  */
```

The REXX application which issues the WAIT will continue when some other task issues the POST function. For example:

```
Rc = POST(ecb@)
```

NOTE: *The ecb address must be passed (by some means) to the task that will issue the POST.*

Example 2

Suspend for 30 seconds execution of the REXX application which issues the wait.

```
Rc = WAIT('SEC',30)
```

10.3 XPOST - Cross Memory Post

The XPOST function provides a means to signal the completion of an event to an address space *other than* the one in which the event took place. For example, a jobstep might complete in Address Space A and be reported to a REXX monitor application running in Address Space B. XPOST is an authorized facility. Appendix R of the RLX Installation and Customization Guide describes how to define and grant access to the XPOST function.

Syntax: `Rc = XPOST(jobname,ecb@,compcode)`

`jobname` The jobname of the remote address space to be POSTed when the event in the local address space completes.

`ecb@` Binary address of the ECB (event control block) in the remote address space to be POSTed. This address is returned by the GETMAIN function into a REXX variable. A call to the WAIT service that references the address of this ecb must have been issued previously in the remove address space.

`compcode` The completion code to be placed in the ECB. This code can subsequently be checked by the issuer of the WAIT function. The compcode parameter should be specified as a printable decimal number.

Return code:

Rc = 0	Xpost call was successful
8	Error in one or more parameters

Example: `Rc = XPOST('TSOUSER1',ecb@,12)`

Post an ecb with a completion code of 12. The ecb is in the address space whose jobname is TSOUSER1. The address of the ecb is in the REXX variable `ecb@`.

Chapter 11

Dataset Allocation Management Functions

11.1 SDKLA -- List Allocated Datasets

The SDKLA function allows you to obtain information about datasets which are currently allocated in your REXX environment. You can specify DDNAME or DATASET name as search criteria for allocation information. If no parameters are specified, then information about all allocated datasets are returned. The SDKLA function returns information in the following REXX variables:

```
SDKLA.0     - Number of datasets returned
SDKDDN.i    - Allocated DDNAME
SDKDSN.i    - Allocated Dataset name
SDKMEM.i    - Member name if a PDS with a specific member is allocated
SDKTYPE.i   - Dataset type: CATALOG, VSAM, PO, PS, DA
SDKALOC.i   - Dataset allocation: OLD, SHR, MOD
SDKDISP.i   - Dataset normal disposition: KEEP, DLET, CATL
SDKADSP.i   - Dataset abnormal completion disposition: KEEP, DLET, CATL
SDKOPEN.i   - Number of times the dataset was opened up to this moment
SDKVOL.i    - Dataset volume serial number
```

Syntax: `rc = SDKLA('TASK',tcb@)`

`TASK` Must be specified as shown. Indicates that all dataset names, if any, allocated to a tasklib will be returned. A tasklib is a DDNAME which is searched for executable load modules before STEPLIB.

`tcb@` The 8-byte hexadecimal address of a TCB (task control block) to be searched for a tasklib. If not specified, the current TCB is assumed.

`rc = SDKLA('DDNAME',ddname)`

`DDNAME` Must be specified as shown. This option indicates that all dataset names, if any, allocated to the specified DDNAME should be returned.

`ddname` The DDname to be used as a search criteria

`rc = SDKLA('DSNAME',dsnamepref)`

`DSNAME` Must be specified as shown. This option indicates that all dataset names which begin with `dsnamepref` will be returned.

`dsnamepref` A dataset name prefix to be used as a search criterion

RC variable

= 0 Function executed successfully
> 0 Error in function execution

Example: Display all dataset names allocated to the DDname ISPLLIB:

```
call SDKLA 'DDNAME','ISPLLIB'
call SDKBRIF 'DSNDSN.',5Ø,'DSNAMES Allocated to DDNAME ISPLLIB'
```

11.2 RAICONC - Concatenate datasets to an existing file

The RAICONC command lets you concatenate or de-concatenate datasets allocated to a specific DDname. The RAICONC command runs only in the TSO environment. Its invocation syntax is as follows:

```
+--------------------------------------------------------------------+
|  RAICONC parm1 parm2 ...                                           |
|    where parm1, parm2, ... are one of the following:'              |
| *       - indicates default value'                                 |
|                                                                    |
|    POS(*PRECAT    |  - concat / deconcat position '                |
|        POSTCAT    |'                                               |
|        REMOVE)'                                                    |
|    FILE(ddname)   |  - DD name to be used by this operation'       |
|      FI(ddname)   |'                                               |
|      DD(ddname)'                                                   |
|    DSN(dsnlist)   |  - List of dataset names.  These names may be  |
|     DA(dsnlist)   |    specified with or without quotes.           |
|     DATASET(dsnlist)'                                              |
|    TRACE(YES|*NO)    - Start REXX trace'                           |
|    VERBOSE(YES|*NO)  - Issue messages'                             |
|    HELP(YES|*NO)     - Display help for syntax of RAICONC          |
|                                                                    |
+--------------------------------------------------------------------+
```

11.3 RAILA - List file and dataset allocations

The RAILA command provides a list allocation facility that is more granular and readable than the output of the TSO LISTA command. A list of datasets allocated to a particular DD name can be requested. Alternatively, a global list of files and allocated datasets may be requested. The command syntax for the RAILA command is as follows:

```
+-------------------------------------------------------------------+
|   RAILA parm1 parm2 ...                                           |
|                                                                   |
|   where parm1, parm2, ... are one of the following:               |
|         DD(<ddname>)     - DDname to show                         |
|         FILE(<ddname>)                                            |
|         FI(<ddname>)                                              |
|         DSN(<dsn>)       - Dataset name to show                   |
|         DA(<dsn>)                                                 |
|         DATASET(<dsn>)                                            |
|         TRACE(YES|*NO)   - Start REXX trace                       |
|         LIST(SHORT*|FULL) - list all information about datasets  |
|                                                                   |
| *       - indicates default value'                                |
+-------------------------------------------------------------------+
```

Example:

Enter in Option 6 of TSO/ISPF: `RAILA DD(SYSEXEC)`.
Information similar to that shown below will be displayed:

```
********************************* Top of Data **********************************
-DDname- -VolSer- Allo Type OP Disp Adsp -Member- ----------------DataSet Name--
SYSEXEC  RA0003   SHR  PO   2  KEEP KEEP          RAI.SEM110.EXEC
  +02    RA0006   SHR  PO   2  KEEP KEEP          RAI.DEMO.EXEC
  +03    RA0008   SHR  PO   2  KEEP KEEP          RAI.PROD.EXEC
  +04    RA0007   SHR  PO   2  KEEP KEEP          RAI.RAI.EXEC
  +05    RA0003   SHR  PO   2  KEEP KEEP          RAI.RLX.VtRtMt.RLXEXEC
  +06    RA0006   SHR  PO   2  KEEP KEEP          RAI.RLX.VtRtMt.RCSEXEC
  +07    RA0008   SHR  PO   2  KEEP KEEP          RAI.DCA.DCAEXEC
  +08    RA0003   SHR  PO   2  KEEP KEEP          RAI.TTS.TTSEXEC
  +09    RA0006   SHR  PO   2  KEEP KEEP          QMF.DSQEXECE
  +10    RARES0   SHR  PO   2  KEEP KEEP          SYS1.ISP.VtRtMt.SISPEXEC
******************************** Bottom of Data ********************************
```

Chapter 12

REXX Edit Macros
Supplied with RLX / SDK

12.1 RLXBOX - Draw box within an Edited source file

RLXBOX is an ISPF edit macro which formats the comments you embed in your REXX execs within a *comments box*. To use it, first type RLXBOX on the ISPF/Edit command line. Next, position the cursor to the line and column where you wish to insert comments and press ENTER.

RLXBOX displays a panel on which you can enter comments, change the comment width and make other formatting adjustments. RLXBOX right-justifies the comments box to end in column 72. Lastly, RLXBOX aligns the text of the comments within the comment box in accordance with your specifications.

12.2 RLXF - REXX source module formatter

The RLXF macro formats your REXX execs. To use it while editing a REXX exec, type `!RLXF` on the EDIT command line. The exclamation point preceding the RLXF command is required since RLXF is distributed in compiled form. (You can optionally define RLXF to ISPF/Edit via the editor's DEFINE command. This lets you invoke RLXF without the preceding exclamation point).

The RLXF command displays a panel on which you can specify an indentation value to format your execs. The default is to indent 2 positions.

> ***NOTE:*** *RLXF does not save the exec being edited, so make sure you have created a backup copy of your exec if you wish to restore the original.*

12.3 RLXQUOTE - statement verification

It's easy to make mistakes with delimiters and continuation characters, particularly when coding long RLX statements and other long host commands. The RLXQUOTE edit macro is designed to help you verify the statements. First, RLXQUOTE checks that each line of a multi-line statement is properly delimited within either single or double quotes. Next, RLXQUOTE verifies that incomplete statements are properly continued with a comma onto another line. Lastly, RLXQUOTE ensures the statement is terminated with a semicolumn ';'. RLXQUOTE issues messages if it detects errors. This lets you correct the error and re-run RLXQUOTE until your statement is error free.

To verify your statements within ISPF edit, first key RLXQUOTE on the command line. Then position the cursor to the line to be verified and press ENTER.

12.4 RLXRUN - run the REXX exec you are editing

RLXRUN is a ISPF Edit macro that lets you run the exec you are currently editing. To run your exec, simply type `RLXRUN` on the edit command line. To pass parameters to the exec, type the parameter string following the RLXRUN command -- as in the following example: `RLXRUN parm1 parm2 parm3 ....`

12.5 RLXSRUN - run current REXX exec via the RLXS frontend

RLXSRUN is a ISPF Edit macro that invokes the exec you are currently editing through the RLXS command frontend. To run your exec as an RLX/TSO application (one invoked through the RLXS frontend) type `RLXSRUN` on the edit command line. To pass parameters to your RLX/TSO application, type the parameter string following the RLXSRUN command -- as in the following example: `RLXSRUN parm1 parm2 parm3 ....`

Chapter 13

REXX and RLX/SDK

Environment Control Functions

13.1 SDKINIT -- Initialize the SDK environment

Some SDK functions require a persistent environment for the duration of your RLX application. The SDKINIT function creates an MVS subtask and maintains it as long as your RLX application is active. The following call explicitly initializes the SDK environment:

Syntax: `RC = SDKINIT()`

However, the SDKINIT call is entirely optional since the RLX/Software Development Kit environment *implicitly* initializes itself when the first SDK function is called. In contrast, *explicit* termination of the SDK environment ***is*** required. See the description of the SDKTERM service for a more thorough discussion.

13.2 SDKRSVC -- Verify installation of the RAI User SVC

The RAI User SVC has been superceded by the RAI Server address space. The SDKRSVC service is maintained for compatibility with previous releases of RLX.

Syntax: `RC = SDKRSVC()`

The SDKRSVC function lets the caller know:

- whether the jobstep is APF authorized
- whether the RAI user SVC is installed -- and if so, what is its SVC number
- whether the RAI subsystem is defined

The SDKRSVC function returns the following REXX variables:

`RC` Defines call completion

- `0` Function executed successfully
- `>0` Severe error - report to RAI

`SDKSVC#` RAI User SVC number (if defined). Otherwise set to zero.

`SDKESR` Set to 0 if the SVC in SDKSVC# is a standard SVC. Set to 1 if the RAI SVC is an ESR (Extended Service Router) SVC 109)

`SDKAPF` Set to 1 if the jobstep is APF authorized (JSCBAUTH bit is on) Otherwise set to 0.

`SDKSSI` Contains the four character RAI Subsystem name (if defined). Otherwise set to blanks.

13.3 SDKTERM -- Terminate the SDK environment

Although the call to initialize the SDK environment is optional (call SDKINIT), termination of an active SDK environment is ***mandatory***. This is true whether the SDK environment was activated explicitly or implicitly.

Unless you invoke your REXX execs via the RLXS frontend, you ***must*** call the SDKTERM service -- or issue the analogous RLX TERM host command -- before your REXX application completes its processing. Otherwise, a System A03 abend will result because the SDK subtask is still active. Note that this abend occurs ***after*** your REXX exec completes.

The example in Figure 13.1 illustrates a call to the SDKTERM service. The example in Figure 13.2 illustrates an exec that issues an RLX TERM request in lieu of calling the SDKINIT service.

```
/* REXX */
call sdkinit                                                    (1)
.....
/*  some application processing */.....
.....
call sdkterm                                                    (2)
```

where

(1) The SDKINIT call explicitly initializes the SDK environment. Any RLX service would implicitly initialize the same persistent environment.

(2) The SDKTERM service cleans up the SDK environment and releases resources acquired on behalf of your exec.

Figure 13.1

```
/* REXX */
address rlx                                                     (1)

'rlx term'                                                      (2)
```

where

(1) ADDRESS RLX identifies the host command environment that receives and executes statements REXX does not recognize (known as host commands). The RLX host command environment provides fast ***branch entry*** for SQL requests, DB2 commands and ISPF dialog services.

(2) The RLX TERM service should always be called before your RLX execs complete their processing. RLX TERM cleans up the RLX environment and releases resources acquired on behalf of your exec. If your exec exits without issuing an RLX TERM request (or the analogous SDKTERM service), an A03 system abend will result.

Figure 13.2

13.4 SDKVARS - Return Values of System or REXX variables

The SDKVARS function returns a variety of environmental and system information in a series of REXX variables. The information returned is controlled by the parameters described below. If no information is available, then a null string is returned and the REXX variable RC is set to a non-zero value.

Syntax: `result = SDKVARS(parameter)`

`result` The value returned or the null string if no value is available

`parameter` May have one of the following values:

`CPUTIME`	CPU time used by the caller since the address space was created
`CPUID`	A 12-character CPU identifier
`MVSVER`	MVS version and FMID
`REXXVARS`	A list of REXX variable names defined in the caller's active exec
`DB2SYS`	A list of DB2 subsystem descriptors, each descriptor has the format

`(db2ssid char A|I procname version)`, where:

`db2ssid`	DB2 subsystem ID
`char`	the DB2 command recognition character
`A | I`	A=Active, I=Inactive
`procname`	the name of the DB2 master started task
`version`	DB2 version (only for active subsystems)

RC variable

= 0	Function executed successfully
> 0	Error in function execution

Example: Display all REXX variables defined in the active REXX exec:

```
vars = SDKVARS('REXXVARS')     /* retrieve all REXX vars into the variable vars */
do i = 1 to words(vars)
   var = word(vars,i)
   say var 'has value' value(var)
end
```

13.5 RAIENVIR - Display System Environment Information

RAIENVIR is a TSO command processor which displays a variety of system information. Typically this function will be used only for problem diagnosis.

Syntax: `RAIENVIR`

This function requires no parameters

Example:

From the TSO/ISPF Option 6 panel, enter the command: `RAIENVIR`.
Output similar to the following is displayed.

```
CONTROL PROGRAM FOLLOWS
SP6.Ø.6 HBB66Ø6
CONTROL PROGRAM VERID FOLLOWS

CONTROL PROGRAM RELEASE LEVEL FOLLOWS
Ø38
CPUID FIELD FOLLOWS
61Ø129573Ø9Ø
USER ID ->
RAIØ27
ACCESS CONTROL ENVIRONMENT ELEMENT DATA ->
ACEE:  y:           :RAIØ27  :RAIGRP  :::o :        G)-       RAILUØ26

ACEE USERID ->
RAIØ27
ACEE GROUP NAME ->
RAIGRP
ACEE TERMINAL ID ->
RAILUØ26
ACEE CONNECTED APPLICATION ->

ACEE SURROGATE USER ID ->

ASCB DATA FOR TSO ID FOLLOWS
    3ØØ.87
RAIENVIR LOADED FROM AN UNAUTHORIZED LIBRARY
***
```

13.6 RAIWHERE - Locate a load module, REXX exec, or CLIST

RAIWHERE is an ISPF dialog which searches the ISPLLIB, STEPLIB, LINKLIST, SYSPROC and SYSEXEC libraries for specified load modules, REXX execs and TSO CLISTs. By specifying an optional DDNAME, the search can be extended to any library.

Syntax: `RAIWHERE name ddname`

`name` The name of a load module, REXX exec or TSO CLIST to be located

`ddname` An optional DDNAME to search for any other PDS member

Example: Enter the following command from within ISPF: `RAIWHERE IRXISPRM`
Output similar to that displayed below will appear:

```
EDIT       Member IRXISPRM search                          Columns 00001 00072
Command ===>                                                  Scroll ===> CSR
****** ***************************** Top of Data ******************************
000001
000002 Standard search order: JobPackQ, Tasklib, Steplib
000003 Name      TT   R  K  Z  C User Data
000004 IRXISPRM 0005 1A 10 02 2E 00052000000000000C2E300052005
000005 Job, Task, or Steplib
000006
000007 DDNAME: STEPLIB
000008 No matches
000009
000010 DDNAME: ISPLLIB
000011 Name      TT   R  K  Z  C User Data
000012 IRXISPRM 0005 1A 10 00 2E 00052000000000000C2E300052005
000013 Private library
000014 RLX.VXXX.RLXLOAD
000015
000016 DDNAME: SYSEXEC
000017 No matches
000018
000019 DDNAME: SYSPROC
000020 No matches
****** **************************** Bottom of Data ****************************
```

Chapter 14

Miscellaneous Functions

14.1 SDKQREF -- Access to MVS/Quick-Ref

The SDKQREF function enables you to retrieve lines of information from the MVS/Quick-Ref database (a product of ChicagoSoft) and then load this data into an array of REXX stemmed variables that you specify. The SDKQREF function requires that Release 3.0 of MVS/Quick-Ref (or a subsequent release) be installed on your system.

> ***NOTE:*** *The MVS/Quick-Ref database should be allocated to the QWREFDD DDname before you request the SDKQREF service. You may also need to allocate the QWREFDDU DDname if you are going to access a Quick-Ref user database.*

Syntax: `Rc = SDKQREF(SearchPath,stemname,maxlines)`

where

`SearchPath` specifies the search criteria to retrieve information from the MVS/ Quick-Ref database. The format of the search string is described below.

`stemname` specifies the name of the REXX stem which will be loaded with the information retrieved from Quick-Ref that satisfies the search criteria. The SDKQREF service sets the zero-th element of the stemmed array to the number of lines of text that were retrieved and loaded into REXX compound variables that share the same stem.

`maxlines` specifies the maximum number of lines to be retrieved from the Quick-Ref database.

Return Codes

0	Search was successful
4	Maxlines exceeded and unread data remains in the database
8	Non-numerical maxlines parameter
28	MVS/Quick-Ref is not available on this system
32	Failed to obtain memory for output lines / or a searchpath was not specified

Specifying SearchPath

The MVS/Quick-Ref User Guide describes how to specify search paths in detail. The following are examples of search criteria.

Example:

```
topic_code'='topic_key
```

where

Valid topic_codes include:

M - MVS message
J - JCL job statement keywords
L - MVS utilities

Example: Retrieve the Quick-Ref database entry for the IBM message IEA995I. Load the retrieved lines of data into REXX compound variables which share the stem 'msg'

```
Call SdkQref 'M=IEA995I','msg.'
```

Chapter 15

REXX Environment Control

15.1 RFPSCM -- REXX Host Command Environment definition

This function lets you define your own REXX host command environment without assembly language programming. RFPSCM uses the REXX interface routine IRXSUBCM to add, change, update and query the REXX Host Command Environment table. The changes made via RPFSCM to the host command environment table remain in effect for the duration of your REXX application. *They do not permanently change any REXX parameter modules maintained in a load module library.*

Syntax:

```
Rc = RFPSCM(function,hostcmd,loadmodule,token)
```

`function` — Specifies the function to be performed. Valid values are:

- `ADD` — Adds a new entry to the REXX host command table
- `DELETE` — Deletes an entry from the REXX host command table
- `UPDATE` — Modifies an entry in the REXX host command table
- `QUERY` — Verifies that the referenced hostname is defined in the REXX host command table

`hostcmd` — Specifies the REXX host command -- e.g. TSO, ISPEXEC, RLX

`loadmodule` — Identifies the name of the host command environment replaceable routine (a load module) which will handle the host commands routed to the specified host command environment.

`token` — A character string up to 16-bytes long that will be passed to the replaceable routine associated with the host command environment.

RC variable

0	Function call successfully executed.
8	Requested function failed

Example 1

```
Rc = RFPSCM('QUERY','RLX')
if Rc = Ø then
   say ‘RLX host command is currently defined’
else
   say ‘RLX host command is not defined’
```

Example 2

Define to REXX a host command environment named RLX. The name of the replaceable routine to receive host commands is the load module named RLXH.

```
Rc = RFPSCM('ADD','RLX','RLXH','RLX command')
```

15.2 SDKRXF -- Dynamically Define REXX Functions

The SDKRXF function is used to maintain entries in function package load modules. The function package load module referenced on the call to SDKRXF must contain 1 or more empty entries which your REXX execs can alter dynamically. For example, the function package load modules shipped with RLX contain 20 EMPTY entries which your REXX execs can alter dynamically by calling the SDKRXF function. Note that any REXX functions you define via SDKRXF are valid *only* for the duration of the current REXX environment. Further, these changes are made in memory and do not permanently update the function package load modules maintained on disk.

Syntax: `result = sdkRXF(function,name,[loadmod],[ddname],[rfpname])`

`function` — Specifies the function to be performed. Valid values are:

- `ADD` — Add a new entry to the REXX function package. If the function is already defined then the function action module is reloaded
- `CHANGE` — Modifies an existing entry: `loadmod` and `ddname`. The old `loadmod` is deleted from storage.
- `QUERY` — The REXX function package is searched for the name specified in the call. If found, then a 32-byte entry is returned in the result variable. The format of the result entry is as follows:

NAME	Bytes	Description
`funcname`	8	REXX function name
`address`	4	Binary address of the `loadmod`'s entry point

- `DELETE` — The entry specified by the `name` parameter will be freed and the `loadmod` will be deleted from memory

`name` — The name to be assigned to a REXX function. This is the name by which a REXX exec calls the REXX function.

`loadmod` — The name of the load module that implements the function. The load module must reside in a private load library allocated to DDname or be accessible through the standard MVS search order.

`DDname` — Specifies a DDname of a private load library where `loadmod` resides.

`loadmodule` — Identifies the name of the host command environment replaceable routine (a load module) which will handle the host commands routed to the specified host command environment.

`rfpname` — The name of a REXX Function Package Directory load module which has been defined in a REXX parameter module. The Default is IRXFUSER.

Return code (Rc)

0 The operation was successful

4 The ADD action was issued but the entry already exists. No modifications took place.

8 For CHANGE, DELETE or QUERY actions, the entry specified by the name argument was not found. The operation failed.

12 An ADD operation failed – no additional entries in the REXX function package directory are available for use.

16 An ADD, CHANGE or DELETE action failed -- the REXX Function Package Directory resides in protected storage

20 Failed to open DDname for an ADD or CHANGE operation

24 The load module cannot be found in the library allocated to DDname

28 Failed to close DDname

32 The loadmod has unacceptable attributes -- must be AMODE(31) and be reentrant

36 SDKRXF parameter error

Example 1: Define a new function named USERF1 which is to be processed by the load module named USRFUNC1 and is to be loaded from the private library whose DDname is MYDD.

```
CALL sdkRXF 'ADD','USERF1','USRFUNC1','MYDD'
select
  when rc = 0 then say 'Function successfully defined'
  when rc = 4 then say 'Function already defined'
  otherwise
       say 'Failed to define function, RC='rc
end
```

Example 2: Obtain the entry from the REXX Function Package Directory corresponding to the USERF1 function:

```
result = sdkRXF('QUERY','USERF1')
if rc = 0 then do
   say 'Function.....'substr(result,1,8)
   say 'EP address...'substr(result,9,4)
   say 'EP name......'substr(result,17,8)
   say 'EP name......'substr(result,25,8)
end
else
   say 'Function not found'
```

Example 3: Delete the function USERF1 defined in 1.

```
CALL sdkRXF 'DELETE','USERF1'
if rc = 0 then
   say 'Function was successfully deleted'
else
   say    'Failed    to    delete    function,    RC='rc
```

Chapter 16

Maintaining REXX Function Packages and Parameter Modules

This chapter describes a set of dialogs supplied with the RLX/SDK that let you further customize and extend the REXX environment. You can enter the command RLXFPD to display the selection menu for these dialogs as illustrated in Figure 16.1.

Menu Options 1, 2 and 3 let you develop user functions written in REXX and then deploy them as part of REXX function packages. Options 4 and 5 let you modify the parameter modules that customize REXX for environments such as MVS, TSO, ISPF and NetView. These dialogs enable you to extend the various REXX environments by integrating RLX with your own functions, as well as with REXX extension products supplied by other vendors. Section 16.5 presents a complete example that illustrates how to write your own REXX functions (in REXX) and then deploy them as part of a REXX function package.

```
---------------- Manage REXX Function Packages and Parm Modules -------------
 Option ===>

    1  SourceFPD    - Read an existing FPD from a source library
    2  LoadFPD      - Read an existing FPD from a load library
    3  NewFPD       - Build a new FPD
    4  SourceRPM    - Read an existing REXX Parm Module from a source library
    5  LoadRPM      - Read an existing REXX Parm Module from a load library
    T  Tutorial     - Description of this menu's options
    X  Exit         - Leave this dialog
```

Figure 16.1 Manage REXX Function Packages and Parameter Modules

16.1 Creating and Maintaining REXX Function Packages

IBM's TSO/REXX Reference describes how you can group frequently used external functions and subroutines into function packages -- i.e. groups of routines packaged together for high performance access. Routines link-edited within function packages exhibit the fastest access because when REXX encounters a function reference or subroutine call, it first searches the directories of the function packages defined to it. If REXX fails to find the function, the search continues for a load module by that name. Lastly, REXX will search for an exec or TSO CLIST that resides in a library allocated to either SYSPROC or SYSEXEC.

The AcceleREXX compiler enables you to write these frequently used routines in REXX and then compile them into object modules. Once compiled, you can include them in REXX function packages using the dialogs described in this section. The requisite development and deployment steps include the following:

- Develop the REXX function
- Compile the function
  (using AcceleREXX, the IBM REXX compiler or some other REXX compiler)
- Define the function as an entry in a REXX function package directory
- Define the REXX function package in one or more REXX parameter modules
- Allocate the load library which contains the REXX Function Package and REXX parameter modules. (Within a TSO, MVS, or NetView environment allocate the library to the STEPLIB ddname. Alternatively, within an TSO/ISPF environment you can allocate the load library to either STEPLIB or ISPLLIB.)

Options 1, 2 and 3 from the selection menu shown in Figure 16.1 allow you to modify an existing REXX Function Package or create a new one. The dialog selected by option 1 supports REXX function packages maintained in source form. Option 2 lets you review and revise a REXX Function Package directory that exists in load module format. Option 3 selects a dialog that creates a new Function Package and its associated directory.

16.2 Read and maintain an existing source Function Package Directory

Selecting option 1 from the menu in Figure 16.1 produces the panel shown in Figure 16.2. You must specify either an *ISPF library* or *Other partitioned dataset* which contains the source of the REXX Function Package Directory you wish to process. If you do not specify a member name, the member selection list will be presented as shown in Figure 16.3.

```
------------- Change source REXX Function Package Directory ---------------------
Command ===>

ISPF Library
   Project ===> RLX
   Group   ===> VtRtMt
   Type    ===> RLXCNTL
   Member  ===>                (Blank or pattern for member selection list)

Other partitioned dataset:
   Data set name  ===>
```

Figure 16.2   Manage REXX Function Package Directories and Parameter Modules

```
SourceFPD: RLX.VtRtMt.RLXCNTL ----------------------------------  ROW 1 OF 39
Command ===>                                                  Scroll ===> HALF

  Name     Action   Lib VV.MM  Created     Changed      Size  Init   Mod   ID
  RLXISPRM Selected 1  32 Ø6 99/Ø4/29 99/Ø4/11 1Ø:51    13Ø   14Ø     Ø RAIA
  NETVIEW           1  32 Ø5 99/Ø4/29 99/12/Ø9 19:14     57    59     4 RAI5
  PMVSRBLT          1  Ø1 Ø2 99/Ø4/3Ø 99/Ø5/Ø1 14:4Ø     24    24     Ø RAI4
  RCSBIND           1  Ø1 Ø4 99/Ø1/14 99/Ø2/Ø1 15:Ø8     44    43     Ø RAI4
```

Figure 16.3   Member selection list dialog

You may only select a single member from the list. If more than one member is selected, the dialog uses the first member you select. Once you select a source member, press PF3 to exit. The dialog reads the selected Function Package source module and builds an ISPF table which contains the REXX function definitions. The resulting display appears in Figure 16.4.

```
----------------------------- Conduct FPD dialog ---------------  ROW 1 OF 70
Command ===>                                                Scroll ===> HALF

ROW Commands: Insert, Delete, Repeat, Change, Undelete
Primary cmds: ADD, END

REXX FPD Header
   Identifier  ===> RFPFLOC    (FPD identifier - any name)
   Hdr size    ===> 24         (FPD header size - fixed by design)
   No of slots ===> 70         (Number of REXX functions defined)
   Reserved    ===> 0          (A fullword of zeros - reserved)
   Entry size  ===> 32         (FPD entry size  - fixed by design)

S Status   FuncName EpName   DDname Static
           RVOPEN   RVOPEN           Y
           RVCLOSE  RVCLOSE          Y
           RVGET    RVGET            Y
           RVPUT    RVPUT            Y
           RVPOINT  RVPOINT          Y
           RVERASE  RVERASE          Y
           RVENDREQ RVENDREQ         Y
           RVVERIFY RVVERIFY         Y
           RVMSG    RVMSG            Y
           RVTERM   RVTERM           Y
```

Figure 16.4 Function Package Directory Modification Dialog

The dialog supports a variety of row commands to manipulate the scrollable list of REXX functions. The descriptions of the column headings associated with the scrollable portion of the display illustrated in Figure 16.4 are as follows:

S denotes the Row command field into which you can enter the following row command letters: I for Insert, D for Delete, R for Repeat, C for Change and U for Undelete.

Status denotes the status of the row as either Deleted, Updated or Inserted.

FuncName specifies the name of the REXX function. This name is used to invoke the function from a REXX exec.

EpName specifies the name of the program entry point that handles this function call. If the entry point is link-edited within the *same* load module that contains the Function Package Directory, the call will be resolved as an assembler BALR instruction. If the function is link-edited within a *separate* load module, the REXX interpreter will fetch the load module, then BALR to it.

DDname DDNAME from which to load **EpName.** If a DDNAME is not specified, REXX will locate the function through the standard MVS search order.

.

Static This flag indicates whether the **EpName** defined above is to be loaded (denoted by the value **N**) or is to be link-edited together with the REXX Function Package Directory.

After you modify the REXX Function Package Directory, press PF3 to display the panel shown in Figure 16.5.

Specify the object and load module libraries for which prompts appear on this panel. If you use static functions, provide the name of the object module library. These modules will be link-edited with the Function Package Directory to form the Function Package Load Module.

Once you press the ENTER key, the dialog tailors the JCL with which to assemble and link-edit the REXX Function Package Load Module. Submit this job and check the completion codes for each of its steps -- all of them should be 0.

```
----------------------------- Tailor New REXX FPD -----------------------------
Command ===>

Create jobstream to assemble a new REXX FPD
   Enter Y / N  ===> Y    (Y = Create JCL, N = Terminate dialog)

Name assigned to REXX FPD load module
   Enter name   ===> RFPFLOC

Specify Object Library containing functions to be included in Function Package
   Enter dsname ===> 'USER.OBJECT'

Specify Load Module Library to contain new REXX FPD
   Enter dsname ===> 'USER.LOAD'
```

Figure 16.5 Create source of a new REXX Function Package Directory

16.3 Read and maintain an existing Function Package Directory load module

```
----------- Change object of REXX Function Package Directory ------------------
Command ===>

ISPF Library
   Project ===> RLX
   Group   ===> VtRtMt
   Type    ===> RLXLOAD
   Member  ===>                (Blank or pattern for member selection list)

Other partitioned or sequential data set:
   Data set name  ===>

Load from STEPLIB
   Module name    ===>

Library for RFP source module:
   Data set name  ===> 'USER.ASM'
```

Figure 16.6 Change REXX FPD from load module

The dialog invoked via Option 2 of the selection menu illustrated in Figure 16.1 displays the panel shown in Figure 16.6. It lets you specify the name of a private library from which to fetch the REXX Function Package Load Module. Alternatively, you can modify a REXX Function Package loaded through the standard MVS search order. To do so, specify the module's name in the `Module name` field that follows the `'Load from STEPLIB'` prompt.

Next, specify the name of the output library into which the source of the Function Package directory should be tailored. The prompt labeled `'Library for RFP source module'` identifies this source module library. The remainder of the dialog with which to edit, assemble and link-edit a REXX Function Package Directory is the same as that described for a REXX Function Package in source format.

16.4 Define a new REXX Function Package Directory

```
----------------------------- Create new REXX FPD ---------------------------
 Command ===>
 FPD008I - Enter name of a New REXX function package directory
 ISPF Library
    Project ===> RLX
    Group   ===> VtRtMt
    Type    ===> RLXCNTL
    Member  ===>                (Blank or pattern for member selection list)

 Other partitioned dataset:
    Data set name  ===> 'USER.ASM(USERFPD)'
```

Figure 16.7 Create new REXX FPD

Option 3 of the selection menu illustrated in Figure 16.1 enables you to define a *new* REXX Function Package and its associated directory.

The panel illustrated in Figure 16.7 allows you to specify the name of a partitioned dataset and member into which the source module for a new REXX Function Package Directory will be written. Be sure to enter the name of a member that does not already exist within that library. The remainder of the dialog with which to edit, assemble and link-edit a REXX Function Package Directory is the same as that described for a REXX function package in source format.

16.5 Maintaining REXX Parameter Modules

This section describes how to modify the parameter modules that customize the REXX language processor for environments such as MVS, TSO, ISPF and NetView. Options 4 and 5 from the menu illustrated in Figure 16.1 provide a pair of dialogs to modify an existing REXX Parameter Module. The REXX parameter modules, in turn, define the REXX Function Packages and host command environments associated with a REXX language processor environment. These dialogs allow you to modify only the REXX Host Command table and the names of the REXX function packages defined in a REXX parameter module. In order to change any other parameters in a REXX Parameter module, you must modify the ISPF skeletons used by the RLX/SDK dialogs to tailor the source code for the REXX parameter module. These file tailoring skeletons can be found in the RLXSLIB dataset. Their member names are RXDFPD$M, RXDFPD$T, RXDFPD$I and RXDFPD$N.

16.6 Maintaining parameter modules in source form

```
-------------- Process REXX Parameter Module from source input --------------
Command ===>

Library containing source of REXX Parameter Module
   PDS name     ===> 'RLX.VtRtMt.RLXCNTL'

REXX Parameter Module Environment
   Environment  ===> I   (M=MVS; T=TSO; I=ISPF; N=NetView)

Load library for REXX Parameter Module
   PDS name     ===> 'USER.LOAD'
```

Figure 16.8 Modify REXX Parm module from source input

If you have source code for a REXX Parameter module you want to modify, select option 4 from the menu shown of the Figure 16.1. The panel shown in Figure 16.8 will be displayed.

Relational Architects distributes six REXX parameter modules in source format as well as load module format. These modules have been modified to include definitions for the host command environments and function packages associated with RLX. They include:

RLXISPRM	for the RLXS frontend within an ISPF environment
RLXSPPRM	for RLX within a DB2 stored procedure addr space
IRXPARMS	for the MVS environment
IRXTSPRM	for the TSO environment
IRXISPRM	for the TSO/ISPF environment
DSIRXPRM	for the NetView environment

The REXX parameter modules distributed in source form reside in the RLXCNTL library while the REXX parameter load modules reside in the RLXLOAD dataset. After you enter all required data on the panel shown in the Figure 16.8, press the ENTER key. The dialog will display the panel shown in Figure 16.9.

With this scrollable panel, you can modify the definitions for REXX function packages and REXX Host Command Environments. When you finish editing these definitions, press the PF3 key. The dialog will tailor a REXX Parameter source module like the one illustrated in Figure 16.10.

You have the opportunity to review and edit the tailored assembly language source module. When you finish, press PF3 to exit. The dialog will then tailor a job to assemble and link-edit the REXX Parameter module.

```
------------------------ Modify REXX Parameter Module ----------  ROW 1 OF 15
Command ===>                                                  Scroll ===> HALF

ROW Commands: Insert, Delete, Repeat, Change, Undelete
Primary cmds: ADD, END, CANCEL

                        Function Package Directories
   USER                           LOCAL                        SYSTEM
   RFPFLOC                        IRXFLOC                      IRXEFMVS
   IRXFUSER                                                    IRXEFPCK



                       REXX Host Command Environments
S Status   Name     Processor
           MVS      IRXSTAM
           TSO      IRXSTAM
           LINK     IRXSTAM
           ATTACH   IRXSTAM
           ISPEXEC  IRXSTAM
           ISREDIT  IRXSTAM
           RLX      RMVH
           RXSQL    RMVH
           CONSOLE  IRXSTAM
```

Figure 16.9 Modify REXX Parm module dialog

```
EDIT ---- RLX.VtRtMt.RLXFILE(IRXISPRM) - 01.00 ------------- COLUMNS 001 072
COMMAND ===>                                                  SCROLL ===> PAGE

000001          TITLE ' IRXISPRM - REXX ISPF PARM MODULE'
000002 *-----------------------------------------------------------------------
000003 *
000004 *   (C) COPYRIGHT RELATIONAL ARCHITECTS INTL.
000005 *       ALL RIGHTS RESERVED
000006 *
000007 *   REXX PARAMETER MODULE FOR THE ISPF ENVIRONMENT. THIS NON-EXECUTABLE
000008 *   LOAD MODULE IS USED BY IRXINIT TO INITIALIZE A LANGUAGE PROCESSOR
000009 *   ENVIRONMENT.  REFER TO SC28-1883 FOR DETAILS.
000010 *
000011 *-----------------------------------------------------------------------
000012 IRXISPRM CSECT
000013 ID       DC    CL8'IRXPARMS'      IDENTIFIES THE PARAMETER BLOCK
000014 VERSION  DC    CL4'0200'          IDENTIFIES THE VERSION OF PARMBLOCK
000015 LANGUAGE DC    CL3'ENU'           LANGUAGE: US ENGLISH IN MIXED CASE
000016          DC    CL1' '             RESERVED
000017 MODNAME@ DC    A(MODNAMET)        ADDR REXX REPLACEABLE MODULE NAME TBL
000018 SUBCOMT@ DC    A(SUBCOMTB)        ADDR HOST COMMAND ENVIRONMENT TABLE
000019 PACKTB@  DC    A(FUNCPTBL)        ADDR FUNCTION PACKAGE TABLE
```

Figure 16.10 Tailored REXX Parm module

16.7 Read and maintain an existing REXX parameter load module

```
---------------- Process REXX Parameter Module from load module ---------------
Command ===>

Specify REXX Environment
   Environment  ===> I   (M=MVS; T=TSO; I=ISPF; N=NetView)

Load REXX Parameter Module from a private library
   PDS name     ===> 'RLX.VtRtMt.RLXLOAD'

Use the standard MVS search order to Load the REXX parameter module
   From STEPLIB ===> N   (Y=load from STEPLIB)
```

Figure 16.11 Tailored REXX Parm module from load module

Option 5 from the REXX Function Package and Parameter Module menu (illustrated in Figure 16.1) invokes a dialog to modify a REXX parameter module in load module format. This dialog displays the panel shown in Figure 16.11.

You must indicate the type of environment for which the REXX Parameter module is intended, along with the name of a load library from which to fetch the REXX parameter module. If you want to load the REXX parameter module from a specific load module library, name that library at the prompt `'Load REXX Parameter Module from a private library'`. If you simply want to modify the REXX parameter module loaded through the standard MVS search order, enter 'Y' into the 'from STEPLIB' field. When you request that the module be loaded from STEPLIB, the dialog ignores the name of any private load module library you specify.

The remainder of the dialog to edit a REXX parameter load module is the same as modifying a REXX parameter module in source format. See the dialog described in Section 16.6.

16.8 Tutorial for creating a REXX function package

This tutorial section details the steps needed to create a REXX function package from three REXX sample execs named RXDFPDTT, RXDFPDT1 and RXDFPDT2. These three source procedures (which reside in the RLXEXEC library) are provided to let you exercise this tutorial. The functions themselves are very simple. They are intended solely to demonstrate REXX function package directory management using the RLX/SDK dialogs. They do not illustrate coding for the kinds of routines you might compile and link-edit within a REXX function package. The discussion assumes the use of the following datasets:

USER.ASM the dataset that contains the source of the REXX Function Package Directory and the REXX Parameter module

USER.OBJECT The library into which will be written the object code produced by compiling REXX execs RXDFPDT1 and RXDFPDT2.

USER.LOAD The library into which the REXX function package and REXX parameter modules will be link-edited.

The sequence of steps to create and activate a REXX function package are as follows:

(1) Use the dialog described in Chapter 3 of the AcceleREXX Compile Reference to compile the REXX execs named RXDFPDT1 and RXDFPDT2. You should have already installed the AcceleREXX compiler and initialized the user datasets referenced by the compile process. This discussion assumes that the compiled REXX object code will be written into the library named USER.OBJECT.

(2) From the main menu illustrated in Figure 16.1, select option 3 to create a new REXX Function Package and Directory. Specify the name of the source dataset into which the new Function Package Directory module (which we shall call USERFPD) will be written. Follow the procedure described in Section 16.4 entitled 'Define a new REXX Function Package Directory'.

(3) When the panel 'Conduct FPD Dialog' is displayed, enter the information as illustrated in Figure 16.12.

Press PF3 to exit. Confirm that you wish to create a REXX Function Package Directory. The dialog will tailor the JCL to assemble and link-edit the REXX Function package load module. Submit this job and check the completion codes for each of its steps -- they must all be 0. Once the job completes successfully, the load library (named 'USER.LOAD' in this example) should contain the load module named USERFPD whose directory defines two REXX functions -- named FPDT1 and FPDT2, respectively.

```
------------------------------ Conduct FPD dialog ----------------  ROW 1 OF 2
Command ===>                                                  Scroll ===> HALF

ROW Commands: Insert, Delete, Repeat, Change, Undelete
Primary cmds: ADD, END

REXX FPD Header
   Identifier  ===> USERFPD    (FPD identifier - any name)
   Hdr size    ===> 0          (FPD header size - fixed by design)
   No of slots ===> 2          (Number of REXX functions defined)
   Reserved    ===> 0          (A fullword of zeros - reserved)
   Entry size  ===> 32         (FPD entry size  - fixed by design)

S Status   FuncName EpName   DDname Static
  Inserted FPDT1    RXDFPDT1          Y
  Inserted FPDT2    RXDFPDT2          Y
```

Figure 16.12 Demonstration REXX functions

(4) Next the newly created REXX function package must be defined within the REXX Parameter module used in the TSO/ISPF environment. For this purpose, Select option 5 from the menu illustrated in Figure 16.1 and enter information as shown in Figure 16.13.

```
---------------- Process REXX Parameter Module from load module --------------
Command ===>

REXX Parameter Module Environment
   Environment  ===> I   (M=MVS; T=TSO; I=ISPF; N=NetView)

Load library for REXX Parameter Module
   PDS name     ===> 'USER.LOAD'

Load existing REXX parameter module from STEPLIB
    From STEPLIB ===> Y   (Y=load from STEPLIB)
```

Figure 16.13 Edit REXX Parm module for the TSO/ISPF environment

(5) Modify the REXX parameter module to add a definition for the newly created user function package. Make the changes as illustrated in Figure 16.14 to define the user function package named USERFPD.

Press PF3 to exit. Review the tailored REXX Parameter module, then press PF3. The dialog will tailor and display the JCL to assemble and link-edit the REXX Parameter module. Check the SYSLMOD DD statement in the LKEDIT step. Make sure it specifies the name of your load module library (USER.LOAD in the example in this section). Once you press the ENTER key, the dialog will tailor the JCL to assemble and link-edit the REXX Parameter Module. Submit this jobstream and check the completion codes for each of its steps -- they must all be 0.

```
------------------------ Modify REXX Parameter Module -----------  ROW 1 OF 6
Command ===>                                                 Scroll ===> HALF

ROW Commands: Insert, Delete, Repeat, Change, Undelete
Primary cmds: ADD, END, CANCEL

                      Function Package Directories
   USER                          LOCAL                        SYSTEM
   IRXFUSER                      IRXFLOC                      IRXEFMVS
   USERFPD                                                    IRXEFPCK



                     REXX Host Command Environments
S Status   Name     Processor
           MVS      IRXSTAM
           TSO      IRXSTAM
           LINK     IRXSTAM
           ATTACH   IRXSTAM
           ISPEXEC  IRXSTAM
           ISREDIT  IRXSTAM
```

Figure 16.14 Modify REXX Parm module

(6) At this point, load modules for both the REXX Function Package and REXX Parameter module should exist as members of the USER.LOAD library, which should appear as the first library in the list of datasets concatenated to the DDname ISPLLIB. Moreover, the ISPLLIB file must be allocated before you enter ISPF. You can use the RAICONC command to concatenate USER.LOAD to the ISPLLIB DDname by entering the following command at the TSO READY prompt:

```
RAICONC DD(ISPLLIB) DA('USER.LOAD')
```

(7) Lastly, start ISPF and select Option 6 from the ISPF Primary Option Menu. From Option 6 enter the command `RXDFPDTT`. You should see the display illustrated in Figure 16.15.

If the resulting command output appears as in Figure 16.15, you have successfully defined and installed the sample REXX function package and updated the REXX parameter module associated with the TSO/ISPF environment.

```
-------------------------  TSO COMMAND PROCESSOR  ---------------------------
ENTER TSO COMMAND, CLIST, OR REXX EXEC BELOW:

===> rxdfpdtt

RXDFPDTT - REXX FPD DEMO main function
RXDFPDTT - Received parameters:
RXDFPDTT - About to invoke RXDFPDT1 as a function FPDT1
RXDRFPT1 - REXX FPD demo function
RXDRFPT1 - Received parameters: p1=FIRST p2=SECOND p3=THIRD
RXDRFPT1 - USERID= RAI4 will return 4
RXDFPDTT - About to invoke RXDFPDT2 as a function FPDT2
RXDRFPT2 - REXX FPD demo function
RXDRFPT2 - Received parameters: p1=FIRST p2=SECOND p3=THIRD
RXDRFPT2 - USERID= RAI4 will return 4
***
```

Figure 16.15 Output produced by the RXDFPDTT command

Chapter 17

Writing System Exit Routines in REXX

This chapter describes and illustrates how you can develop system exits for such IBM components as MVS, JES and RACF, as well as for software products from other vendors. You can write your system exit routines in REXX and then compile them into executable load modules with AcceleREXX.

(This page left blank intentionally)

17.1 System exits and the Parameters they Receive

Before you attempt to implement a system exit, you should review the exit specification provided in the appropriate reference manual. This discussion assumes that any parameter list passed to an exit adheres to OS linkage conventions: Register 1 points to a list of addresses and each address in turn points to a parameter. Consider, for example, the following exit specification.

```
Register 1 points to a parameter list which contains four 31 bit addresses:
p1, p2, p3, p4.  Each address in turn points to one of the parameters
diagrammed below:

P1 -----> TSO USER ID. Eight characters
P2 -----> FUNCTION REQUEST:  One character which can have the one of the following
values:
          B'10000001' RESERVED
          B'01000000' ADD SECTION
          B'00100000' EDIT
          B'00010000' MOD
          B'00001000' DELETE
          B'00000100' RENAME
          B'00000010' COPY
P3 -----> SECTION ENTRY OF REQUEST. 34 characters
P4 -----> DATASET NAME. 44 characters

Upon return R15 must be set to 0 if the operation is to be allowed
and to 15 otherwise.
```

Figure 17.1 Example of system exit parameters specification

In order for a REXX procedure to function as a system exit, the parameters it receives **must** be passed in an OS compliant parameter list format -- as described in this section and illustrated in Figure 17.1. The number of parameters (addresses in the parmlist) may vary from 1 to 20.

Typically, the exit will obtain the parameters passed to it and will perform some function based upon the parameters it receives. Figure 17.2 illustrates an actual system exit written in REXX.

> ***CAUTION: The user should take into consideration the performance consequences of using REXX. If an exit is to be used very heavily, it probably should be written in assembly language.***

```
IF VALUE('$RCX$') = 1 THEN                                  (1)
   SAY 'Compiled exec'
ELSE
   SAY 'Not compiled exec'
ARG parms                                                   (2)
p1@ = SUBSTR(parms,1,8)
p2@ = SUBSTR(parms,9,8)
p3@ = SUBSTR(parms,17,8)
p4@ = SUBSTR(parms,25,8)
SOP_tsoid = STORAGE(p1@,8)                                  (3)
SOP_func  = STORAGE(p2@,1)
SOP_sect  = STORAGE(p3@,34)
SOP_dsn   = STORAGE(p4@,44)
SELECT
   WHEN SOP_func = '40'x THEN func = 'ADD'
   WHEN SOP_func = '20'x THEN func = 'EDIT'
   WHEN SOP_func = '10'x THEN func = 'MOD'
   WHEN SOP_func = '08'x THEN func = 'DELETE'
   WHEN SOP_func = '04'x THEN func = 'RENAME'
   WHEN SOP_func = '02'x THEN func = 'COPY'
   OTHERWISE;                 func = 'INVALID'
END
SAY 'TSO id  ='SOP_tsoid                                    (4)
SAY 'Function='func 'Hex value='C2X(SOP_func)
SAY 'Section ='SOP_sect
SAY 'Dataset ='SOP_dsn

/*  Exit specific code */
IF SUBSTR(SOP_tsoid,1,3) = 'SYS' THEN                       (5)
   success = 1
ELSE
   success = 0

/*  Exit return code */
IF success THEN                                             (6)
   RETURN 0
ELSE
   RETURN 15
```

Figure 17.2 Example of system exit written in REXX

(1) The Special variable RCX is set to 1 (true) if the exec is compiled, and is undefined if the exec is not compiled.

(2) The REXX arg instruction obtains all four parameters addresses as a single 32 byte character string. The four parameter addresses are contiguous within the REXX variable parms -- without delimiters or intervening blanks. The REXX substr function is used to extract each discrete parameter address as an 8 character string.

(3) Using the parameter addresses obtained in (2), the actual values pointed to by the parameters are obtained via the REXX STORAGE function. The Parameter values are in accordance with the exit specification which appears in Figure 17.1.

(4) These SAY instructions display the parameter values simply for information purposes.

(5) Logic specific to your exit implementation can make the decision whether to allow or disallow the operation.

(6) Return to the caller with a value in Register 15. It can be set to 0 (via `RETURN 0`) to allow the operation, or to 15 (via `RETURN 15`) to disallow it.

You can use this example as a guide to implementing your own system exits in REXX. Note that the number of parameters in this schema is limited to 20. Also, the addresses passed to your exit point to the *actual* parameters supplied by the caller -- not a copy.

17.2 Compiling system exit routines written in REXX

In order to compile a REXX exec to be used as a system exit, you must specify the number of discrete parameters the exit routine will receive. To do so, specify the `PARMNO(nn)` parameter via the AcceleREXX input stream defined by the `RCXPARMS` DD statement. The value **nn** specifies the number of discrete parameters (in the range of 1-20) to be passed to the REXX exit routine.

17.3 Testing system exit routines written in REXX

Once you write and compile your REXX exit, you need to test it for correctness. You can optionally write a compiled or assembly language driver that simulates the parameter list expected by the exit and then invokes the compiled REXX load module. The assembler routine in Figure 17.3 illustrates how you might invoke and test the REXX exit routine presented in Figure 17.2, once it is compiled.

```
         TITLE 'RLXMKS1 - TEST SYSTEM EXIT WRITTEN IN REXX'
*-----------------------------------------------------------------------------
*
*        FUNCTION
*        ========
*        THIS MODULE SIMULATES THE INVOCATION OF AN EXIT WRITTEN IN REXX.
*        AT COMPILE TIME, THE USER SPECIFIES PARMNO PARAMETER TO INDICATE
*        THE NUMBER OF PARAMETERS TO BE PASSED TO THE REXX ROUTINE.
*        THE ACCELEREXX RUN-TIME STUB WILL NOT ATTEMPT TO DETERMINE HOW
*        MANY PARAMETERS ARE BEING PASSED.  INSTEAD, THE STUB WILL ASSUME THAT
*        R1 POINTS TO A PARMLIST CONSISTING OF PARMNO OF PARAMETERS.
*        THE STUB WILL TRANSLATE THE LIST OF ADDRESSES TO PRINTABLE HEX AND
*        PASS THESE TRANSLATED HEX ADDRESSES TO THE REXX EXIT ROUTINE.
*
*-----------------------------------------------------------------------------
RLXMKS1  CSECT
         LR    R7,R14              SAVE R14 FOR RETURN
         LR    R12,R15             R12 IS OUR BASE REGISTER
         USING RLXMKS1,R12         ESTABLISH BASE
         LOAD  EP=RLXMKS2          LOAD REXX SYSTEM EXIT
         LR    R15,R0              R15 = ADDRESS OF REXX SYSTEM EXIT
*
         LA    R1,PARMLIST         R1 POINTS TO PARAMETER LIST
         BALR  R14,R15             EXECUTE USER EXIT
         BR    R7                  RETURN TO THE CALLER
*
PARMLIST DC    A(P1)               POINTER TO 1-ST PARM
         DC    A(P2)               POINTER TO 2-ND PARM
         DC    A(P3)               POINTER TO 3-RD PARM
         DC    A(P4)               POINTER TO 4-TH PARM
P1       DC    CL8'USERID1'        USER ID
P2       DC    XL1'40'             ADD SECTION
P3       DC    CL34'SECTION_ENTRY_REQUEST_________'
P4       DC    CL44'SYS1.PROCLIB(IEASYS00)1234567890ABCDEF'
         END   RLXMKS1
```

Figure 17.3 Example of an assembler program to test system exit

Chapter 18

REXX Global Variable Services

The Global Variable Services provided by the RLX/Software Development Kit help developers build sophisticated and complex REXX applications using an Object Oriented Methodology. Ordinarily, REXX lets you invoke a subroutine and pass it parameters. However, the subroutine or function cannot return more than one scalar result variable to its caller. In addition, external functions and subroutines cannot access the variable pools of their callers.

The Global Variable Services provided by the RLX/SDK let you place selected REXX variables in a global pool where they can be accessed by REXX execs as well as external functions and subroutines. You can split large REXX execs into multiple subroutines and functions which share REXX variables and stems. Sample execs RXDGV and RXDGV1, found in the RLXEXEC library, illustrate the usage of Global Variables.

The following functions let you manipulate the Global Variable Pool:

GVGET	Get REXX variable(s) from the Global Variable Pool
GVPUT	Place REXX variable(s) into the Global Variable Pool
GVDEL	Delete REXX variable(s) from the Global Variable Pool

Syntax: `rc = FUNC(<var_list>)`

`FUNC`	global variable function: either GVGET, GVPUT or GVDEL
`<var_list>`	(<var> | <var.> | $POOL)
`<var>`	any valid REXX variable name, can be a specific simple variable or compound variable such as `simple_variable` or `stem.3`.
<var.>	any valid REXX variable stem All compound variables which share this stem will be used
`$POOL`	coded exactly as shown. The entire REXX program variable / Global variable pool will be used

---

NOTE: *You can repeat variables in a list as many times as REXX allows (up to 27 parameters). e.g.* `RC = GVGET ('var1', 'var2', 'stem1.', 'var3')`

Example 1

Save the value of the REXX variable VAR1 in the Global Variable Pool.

```
rc = GVPUT('var1')
```

Retrieve the value of the variable named VAR1 from the Global Variable Pool and place it in the variable pool of the currently active REXX exec

```
rc = GVGET('var1')
```

Example 2

Fetch all variables that share the stem `Long_variable`. from the REXX pool and save them into the Global Variable Pool.

```
Long_variable.1 = 'a1'
Long_variable.2 = 'a2'
Long_variable.last = 'a3'

rc = GVPUT('Long_variable.')
```

Example 3

Save all variables from the REXX program variable pool into the Global Variable Pool. Global variable services creates a special variable named $POOL which contains the names of all variables in the pool.

```
rc = GVPUT('$POOL')
```

Consider the following two REXX EXECs:

```
EXECA:          a=1; b=2; c=3; rc=GVPUT('$POOL'); call EXECB; exit

EXECB:          x = GVGET($POOL)
                Do while $POOL \= ''
                   parse var $POOL varname $POOL
                   say varname'='VALUE(varname)
                End
```

In the preceding example, EXECB will display `a=1, b=2; c=3; rc=00`

Chapter 19

REXX Cross Memory

Client/Server Interface

The Cross Memory Client/Server Interface for REXX allows you to communicate via Cross Memory facilities ***between*** address spaces residing within the same OS/390 system. An interface program named SDKCS is a member of the RLX/SDK portfolio of functions. You can invoke SDKCS as a SERVER or as a CLIENT. When SDKCS is invoked as a SERVER and runs in batch, it can receive commands from CLIENT(s), execute them, and return (send) responses to client commands. The various functions of SDKCS require that the RLX/SDK authorized facilities be installed.

19.1 SDKCS function

The SDKCS function implements inter-address space communication using Cross Memory POST and Cross Memory Data transfer. SDKCS may be invoked in two modes: SERVER and CLIENT.

19.1.1 Server invocation

The following illustrates the command syntax of the REXX call to SDKCS in order to operate as a SERVER:

Syntax: `rc = SDKCS('SERVER', ServerName, AppName [,Ecb_address)`

where

`SERVER`	This keyword directs SDKCS to create a server
`ServerName`	A name (1-8 characters) to be assigned to this server
`AppName`	A name (1-8 characters) of a REXX exec which will process commands sent by clients
`Ecb_address`	An EBCDIC 8-byte hexadecimal address of a fullword in the caller's address space which the SDKCS function sets to the address of server's ECB. The caller of the SDKCS service may post this ECB with a Post code of 1 to direct the Server to terminate.

Example 1

Start a server named USERSERV. Direct the server to invoke the REXX exec named SERVEXEC to process commands:

```
rc = SDKCS('SERVER','USERSERV','SERVEXEC')
```

19.1.2 Client invocation

The following illustrates how a CLIENT exec can send a command to a SERVER address space and obtain a response:

Syntax: `response = SDKCS('CLIENT', ServerName,command)`

where

`response`	contains the response from the command executed by the server.
`CLIENT`	This keyword identifies the invoker as a CLIENT
`ServerName`	specifies the name of the server which is to execute the client command. If the designated server is not active, the command is rejected. The server name should be 1-8 characters long.
`command`	the command string to be passed to the server for execution. Your exec may use any REXX facilities to build the command string.

The following commands are fixed by the architecture of the SDKCS function:

`#STOP` causes the server to terminate further processing -- without sending a response to the CLIENT.

`#SET` governs logging and tracing within the Server address space. #SET assigns a value to the client server status flag (variable `csstflag`). Permissible values for the #SET command are 'T', 'L' and ' ' as described next under #STATUS.

`#STATUS` directs SDKCS to return to the issuer a set of packed variables which describe the status of the server. These variables are:

`csstflag` A trace flag which can have the following values:

' ' no trace

'T' a REXX TRACE 'R' is issued by the server

'L' a log message will be issued by the server for each command it receives and each response it returns.

Example 2

Send a TSO LISTALC command to the server named USERSERV which was started in Example 1.

```
response = SDKCS('CLIENT','USERSERV','LISTALC')
```

The Server named USERSERV receives the command. SDKCS is responsible for calling the REXX exec named SERVEXEC (the exec defined when the server was started) and passing it the LISTA command string.
NOTE It is the user's responsibility to develop the REXX exec (SERVEXEC in this example) which will recognize and process the LISTALC command.

19.2 REXX Exec to process client's commands

The user must develop the REXX server execs which receive and execute commands sent from client address spaces. Server execs can optionally return responses to clients. Example 3 illustrates such a server exec.

Example 3

In this example, responses from TSO commands are trapped in MSG.i stem variables. The exec packs the response lines together, delimiting the lines with an EOL character. SERVEXEC returns the command response via the REXX RETURN statement. The SDKCS function is responsible for delivering this response to the client by placing it in the client's `response` variable.

```
/* REXX */
arg command
   response = ''      /* response to a command */
   EOL = '01'x        /* End Of Line character */
   select
     when command = 'LISTALC' then
          call tso_command
     when command = 'HELLO'   then
          response = 'Response to HELLO command'
     otherwise
          nop
   end
return response

tso_command :
   xx = outtrap('msg.')
   address TSO command
   xx = outtrap('off')
   do i = 1 to msg.0
      response = response msg.i || EOL
   end
return
```

19.3 SDKCSD Demonstration Dialog

The REXX exec named SDKCSD is an example of a TSO/ISPF dialog which uses the SDKCS interface. The SDKCSD dialog allows you to start and stop a Server, obtain and display server status and send commands to the server to be executed. Enter the command 'TSO SDKCSD' to start the dialog and display the panel shown in Figure 19.1

```
--------------------------- C/S Server Control ----------------------------
 Option ===>

stat  - server's status (default)       set   - set trace flag to T or L
    start - start server (submit job)       run   - send command to server
    stop  - stop server

Server Specification
    Server name     ===> TTS
    Server App      ===> SDKCSAPP

Dataset containing JCL for Server Job (used with start command)
    Dataset name    ===> 'RLX.VtRtMt.RLXCNTL(SDKCS)

Server Status
    Server jobname  ===> RAISERV
    Processed cmds  ===> 13
    Elapsed time    ===> 259
    CPU time        ===> 0.67
    Trace flag      ===> L
```

Figure 19.1 SDKCSD Dialog Control Panel

Chapter 20

Invoking REXX execs from Compiled and Assembled Code

The RLX/SDK enables compiled code (programs written in COBOL, PL1, C and other compiled and assembled languages) to call any REXX exec through the RFPH2R (High level language to REXX) interface routine. COB2REXX is a synonym or alias for the RFPH2R service. As such, the discussion in this chapter applies equally to both RFPH2R and COB2REXX. The calling program may execute in the TSO foreground, natively in batch (EXEC PGM=program_name) in batch under the TSO Terminal Monitor Program IKJEFT01.

The RFPH2R and COB2REXX interface routines provide the following services:

- Convert compiled code parameters to REXX EFPL parameters
- Locate the called REXX exec in the set of datasets concatenated to the DDname SYSEXEC.
- Read the REXX exec into memory and prepare it for the REXX interpreter
- Call the REXX interpreter to execute the REXX procedure
- Obtain the return value from the REXX exec (if any) and pass it back to the calling program written in a compiled or assembled language.

RFPH2R and COB2REXX both use another standard component of the RLX/SDK, the Program Cache Facility (PCF) to optimize REXX load performance. PCF ensures that load and exec preparation are performed only once, the first time the exec is invoked. Subsequent calls are executed from memory. This caching mechanism *significantly* improves the performance of REXX execs that are called repeatedly.

20.1 Invoking RFPH2R or COB2REXX

You can invoke a REXX exec from a compiled or assembled program with a call such as

```
CALL RFPH2R   exec-name, parm-length, parms, result-length, result
CALL COB2REXX exec-name, parm-length, parms, result-length, result
```

where:

`exec-name` a 8 byyte character variable containing the name of the REXX procedure to be executed. If exec-name is less than 8 bytes it must be padded with blanks.

Example: `77  EXEC-NAME    PIC X(8)  VALUE ‘REXXEXEC’.`

`parm-length` An full word binary variable that contains the length of the parameter string to be passed to the REXX exec.

Example: `77 PARM-LENGTH   PIC S9(9) COMP VALUE 20.`

`parm` contains the actual parameters to be passed to the REXX exec. The length of this parameter string should correspond to the length specified in the in the parm-length parameter.

Example: `77 PARMS  PIC X(20) VALUE ‘ALLOC DD(DD1) DA(DSN1) SHR’.`

`result-length` A full word binary variable that contains the maximum length of any result value that the invoked exec may specify on a REXX RETURN or EXIT statement. On return from RFPH2R, this maximum length is replaced by the ***actual*** length of the result. If the actual length of the result is *greater* than the maximum length of the caller’s result area, then the result is *truncated*.

Example: `77 RESULT-LENGTH   PIC S9(9) COMP VALUE 10.`

`Result` An area for any result returned by the called REXX exec.

Example: `77 RESULT   PIC X(10).`

20.2 RFPH2R and COB2REXX calls to SDKTERM

An application that invokes REXX execs through RFPH2R or COB2REXX should terminate the RLX/SDK environment before the end of the main procedure or COBOL run unit. This is accomplished by calling RFPH2R or COB2REXX with the `exec-name` set to 'SDKTERM' as illustrated below. If this call is not issued, a system A03 abend may result.

Example:

```
77  SDKTERM  PIC X(Ø8) VALUE 'SDKTERM'.

CALL 'RFPH2R' USING SDKTERM.
```

or

```
CALL 'COB2REXX' USING SDKTERM.
```

20.3 Executing TSO commands in MVS Batch

The ADDRESS RLX facility of the RLX/Software Development Kit enables your REXX execs to issue TSO commands in a native MVS Batch environment. Use ADDRESS RLX instead of ADDRESS TSO to execute TSO commands when TSO is ***not*** active. .

Example:

```
/* rexx */
address RLX 'ALLOCATE FI(DDNAME1) DA(DATASET.NAME) SHR'
```

NOTE: *ADDRESS TSO is only valid if the JCL EXEC statement specifies the TSO Terminal Monitor Program IKJEFT01.*

20.4 RFPH2R sample invocation

This section illustrates how a COBOL program uses the RFPH2R interface routine to call a REXX exec. In this example the COBOL program named RFPH2RC calls a REXX exec named RFPH2RE.. Before the COBOL program RFPH2R issues a GOBACK, it invokes the exec named SDKTERM in order to terminate the RLX/SDK environment.

Calling COBOL program

```
 ID DIVISION.
 PROGRAM-ID.    RFPH2RC.
 AUTHOR.        Paul Verba.
 INSTALLATION.  RELATIONAL ARCHITECTS INTL.
 DATE-WRITTEN.  DECEMBER, 1999.
 DATE-COMPILED.
*
*  This COBOL program illustrates calling a REXX exec and passing
*  it parameters.  The called EXEC also may pass back a response
*  in the RESULT field.
*
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER. IBM-3090.
 OBJECT-COMPUTER. IBM-3090.

 DATA DIVISION.
 WORKING-STORAGE SECTION.

 77  REXX-EXEC           PIC X(08) VALUE 'RFPH2RE '.
 77  REXX-TERM           PIC X(08) VALUE 'SDKTERM '.
 77  BUFFER-LENGTH       PIC S9(9) COMP VALUE 6.
 77  BUFFER              PIC X(44) VALUE 'LONG'.
 77  RESULT-LENGTH       PIC S9(9) COMP VALUE 1024.
 77  RESULT              PIC X(1024).


 PROCEDURE DIVISION.
     DISPLAY 'RFPH2RC - Processing started'.
*
*    Call RFPH2R to illustrate receipt of a large message of 1020 bytes
*
     MOVE 4         TO BUFFER-LENGTH.
     MOVE 'LONG'    TO BUFFER.
     MOVE 1024      TO RESULT-LENGTH.
     PERFORM INVOKE-RFPH2R.
*
*    Execute command in the ADDRESS RLX host command environment
*
     MOVE 41        TO BUFFER-LENGTH.
     MOVE 'ALLOC DD(DD1) DA(''SYS1.PROCLIB'') SHR REU ' TO BUFFER.
     MOVE 1024      TO RESULT-LENGTH.
     PERFORM INVOKE-RFPH2R.
*
*    Terminate RFPM task
*
     CALL 'RFPH2R' USING REXX-TERM.
     DISPLAY 'RFPH2RC - Processing completed'.
     GOBACK.

 INVOKE-RFPH2R.
     CALL 'RFPH2R' USING
                  REXX-EXEC
                  BUFFER-LENGTH BUFFER
                  RESULT-LENGTH RESULT.
     DISPLAY 'RFPH2RC - Buffer  length ...' BUFFER-LENGTH.
     DISPLAY '        - Buffer . . . . .' BUFFER.
     DISPLAY '        - Result length. .' RESULT-LENGTH.
     DISPLAY '        - Result . . . . .' RESULT.
```

REXX exec RFPH2RE

```
/* rexx */
Arg args
 Say ' RFPH2RE - Input arguments =' args

 If args           = 'LONG' Then Do    /* Request for a large data blk*/
    Say ' RFPH2RE - Return 1020 bytes of data'
    Return Copies('-*',510)
    End

 /* Execute a TSO command and pass the return code back to the caller
 */

 say ' RFPH2RE - Executing the following command via ADDRESS RLX'
 say ' >>>' args
 Address RLX args
 say ' RFPH2RE - Return code =' rc
Return rc
```

JCL to execute the RFPH2RC program

This section illustrates the JCL used to run the sample compiled and assembled programs that use the RFPH2R interface routine to call a REXX exec. The source of the RFPH2RE exec in the RLXEXEC. This JCL is in the RFPH2RJ member of the RLXCNTL library. The source for the sample calling routines of the other calling routines is in.

The following programs invoke the sample REXX exec via the RFPH2R interface. The source for each of these programs is a member of the RLXCNTL library.

RFPH2RA - Sample Assembler program
RFPH2RC - Sample COBOL program
RFPH2RP - Sample PL/I program

Finally, the source for the RFPH2RE exec invoked via the RFPH2R interface is in the RFPH2RE member of the RLXEXEC library.

```
//jobname  JOB
//*RUNASM  EXEC PGM=RFPH2RA,REGION=4M            <- Sample Assembler
//*RUNPLI  EXEC PGM=RFPH2RP,REGION=4M            <- Sample PL/I
//RUNCOB   EXEC PGM=RFPH2RC,REGION=4M            <- Sample COBOL
//STEPLIB  DD  DSN=RLX.VtRtMt.RLXLOAD,DISP=SHR
//SYSEXEC  DD  DSN=RLX.VtRtMt.RLXEXEC,DISP=SHR
//SYSTSPRT DD  SYSOUT=*
//SYSPRINT DD  SYSOUT=*
//SYSOUT   DD  SYSOUT=*
//SYSTSIN  DD  DUMMY
//
```

Chapter 21

DB2 Related Functions of the SDK

21.1 Running DSNUTILB from REXX execs (SDKDSNU)

The SDKDSNU function allows your REXX execs to invoke DB2 utilities while running in batch as well as in the TSO foreground. SDKDSNU is an authorized facility. Appendix R of the RLX Installation and Customization Guide describes how to define and grant access to the SDKDSNU function.

Syntax: `RC = SDKDSNU(parms)`

`parms` Parameters passed to the DSNUTILB program

Function returns REXX variable:

`RC` Defines whether DSNUTILB was successfully invoked. RC does not reflect whether DSNUTILB executed successfully.

- `Ø` DSNUTILB was invoked successfully. Examine SYSTSPRINT to determine the results of the utility's execution.
- `>Ø` Failure to ATTACH DSNUTILB program. Error codes are the same as those returned by the MVS ATTACHX macro.

In order for SDKDSNU to work properly, the following installation prerequisites must be met:

1. The RAI Server address space must be installed as described in the RLX Installation Guide.
2. A DSNLOAD file must be allocated. The DB2 system load library allocated to the DSNLOAD DDname must be APF authorized.
3. The TSO user or jobname must have READ access to the profile named RAI.SDK.DSNU within CLASS(FACILITY) as documented in Appendix R of the RLX Installation Guide.

21.2 The SDK#DSNU Exec

Member SDK#DSNU in the RLXEXEC library illustrates how a REXX exec can invoke a DB2 utility (such as RUNSTATS) via the SDKDSNU function.

```
/* REXX */
Address TSO

   call msg 'off'
   "free dd(sysut1 sysin sysprint utprint)"
   call msg 'on'

   call Create_sysin

   "alloc dd(sysut1)",                                                   (2)
         "cylinders space(1,1) unit(vio)"
   "alloc dd(sysprint) cylinders space(1,1) unit(vio)",
         "blksize(1330) lrecl(133) reu"
   "alloc dd(utprint) dummy shr reu"

   "alloc dd(DSNLOAD)",                                                  (3)
         "da('DSN.SDSNLOAD') shr reu"

   call SDKDSNU "DSN1,U001,"                                             (4)

   drop f.
   call sdkread 'sysprint',,'f.'                                         (5)
   call sdkbrif 'f.',,'SYSPRINT after Util='word(s.1,1) 'Rc='Rc          (6)

Return

Create_sysin :                                                           (1)
   s.1 = 'RUNSTATS TABLESPACE RAIODE.SEMSRSLI'
   s.2 = '         REPORT YES'
   s.3 = '         UPDATE ALL'
   s.0 = 3
   call sdkedif 's.',,'Modify RUNSTATS control cards'
   "alloc dd(sysin) cylinders space(1,1) unit(vio) recfm(F B)",
         "lrecl(80) blksize(3120)"
   "execio * diskw sysin (stem s. finis"
Return
```

where

(1) Create a REXX stem ‘s.’. Allow the user to edit it and write a temporary dataset to the file allocated to SYSIN

(2) Allocate all the datasets necessary to execute the DB2 utility (in this case RUNSTATS)

(3) Allocate the APF authorized DB2 Load library

(4) Invoke SDKDSNU to execute the DB2 RUNSTATS utility

(5) Copy SYSPRINT (the RUNSTATS output) into the REXX stem ‘f.’.

(6) Browse the RUNSTATS output within the stemmed array named ‘f.’.

Chapter 22

Data In Virtual Interface

The MVS DIV macro establishes a window in your address space and enables your program to reference or update data from a Data In Virtual object without actually using I/O instructions. The SDKDIV implementation of DIV supports only VSAM linear datasets. Dataspaces and hiperspaces are not supported at this time.

This chapter discusses the SDKDIV function and briefly describes its parameters. This chapter is not a tutorial for DIV programming. For details on Data In Virtual concepts and programming refer to the following IBM publications:

```
OS/390 Assembler Programming Guide
OS/390 Assembler Programming Reference
```

22.1 Overview of the Data In Virtual Interface

SDKDIV is a full implementation of the MVS Data-In-Virtual facility. SDKDIV is based upon the DIV macro. All DIV macro parameters that relate to VSAM linear datasets are supported. No support for dataspaces or hiperspaces is available at this time.

SDKDIV syntax differs from that of the DIV macro in some instances because the implementation is tailored for the convenience of REXX developers. All SDKDIV parameters are positional. They are different for every function. The following provides an overview of the DIV functions and the parameters they use.

```
SDKDIV 'IDENTIFY',<'idvar'>,<type>,<ddname>,<stoken>,<checking>,
                       <ttoken>
```

The IDENTIFY function selects the data-in-virtual object (i.e. linear data set) you want to process. When you specify IDENTIFY, you must also specify the name of a REXX variable. This variable is assigned an 8 byte value that is referenced on all subsequent SDKDIV calls. A type parameter is also required. A dataset object must also specify TYPE=DA and DDNAME. To bypass data-in-virtual validity checking, code NO for <checking>. To assign ownership of the <idvar> to another task, code a value for <ttoken>.

```
SDKDIV 'ACCESS',<idvar>,<mode>,<locview>
```

The ACCESS function requests permission to access a data-in-virtual object. When you specify ACCESS, you must also specify <idvar> and <mode>. You may optionally specify <locview>. <Idvar> specifies the token which identifies the object you want to access. The system does not accept a read request if there is already an updater. Nor does the system accept an update request if there is any other user currently accessing the object.

```
SDKDIV 'MAP',<idvar>,<area>,<offset>,<span>,<stoken>,<retain>,
                  <pfcount>
```

The MAP function establishes addressability to the object in a specified range of virtual storage, called the virtual window. When you specify MAP, you must also specify <idvar> and <area>. You may optionally specify <offset>, <span>, <stoken>, <retain>, and <pfcount>. <Area> is a REXX variable that contains the hex address of an area obtained via the GETMAIN function. When you specify <type> as DA, you can issue more than one MAP with different <stoken>s.

```
SDKDIV 'RESET',<idvar>,<offset>,<span>,<release>
```

The RESET function releases changes made in the window since the last SAVE operation. When you specify RESET, you must also specify <idvar>. You may optionally specify <offset>, <span>, and <release>. If the window corresponds to blocks of the object, the current contents of the object replace the data that has changed in the window when the program next references the window. RESET does not change the object. Do not specify RESET for a storage range that contains disabled reference (DREF) storage.

```
SDKDIV 'SAVE',<idvar>,<offset>,<span>,<stoken>,<listvar>
```

The SAVE function writes changed pages from the window to the corresponding blocks in the object. When you specify SAVE, you must also specify <idvar>. You may optionally specify <offset>, <span>, and <stoken>. The system writes changed pages from the window to the blocks specified by <offset> and <span>.

Do not specify SAVE for a storage range that contains disabled reference (DREF) storage. Optionally, SAVE accepts a user list that the application specifies through the <listvar> parameter. The user list contains information returned by the SAVELIST service. If you specify a user list as input for SAVE, you cannot specify <offset> and <span> and the system saves only those pages specified in the user list.

```
SDKDIV 'SAVELIST',<idvar>,<'listvar'>,<listsize>
```

The SAVELIST function returns into the REXX variable named <'listvar'> the addresses of the first and last changed pages in each range of changed pages within the window.

```
SDKDIV 'UNMAP',<idvar>,<area>,<retain>,<stoken>
```

The UNMAP function terminates a virtual window by removing the correspondence between virtual pages in the window and blocks in the object. After UNMAP is complete, the contents of the pages depend on the value you specify for RETAIN. The virtual pages in the former window either retain the current view of the object or appear as if they had just been obtained. When you specify UNMAP, you must also specify <idvar> and <area>. You may also specify <retain> and <stoken>. UNMAP has no effect on the object itself and does not save data from the virtual window. If you want to save the data in the window, invoke SAVE before you invoke UNMAP.

If you issued multiple MAPs with different <stoken>s, use <stoken> on UNMAP. If you do not specify <stoken>, the system scans the mapped ranges and unmaps the first range that matches the virtual area. Issuing UNACCESS or UNIDENTIFY automatically unmaps all mapped ranges.

```
SDKDIV 'UNACCESS',<idvar>
```

The UNACCESS function relinquishes your permission to read from, or write to, a data-in-virtual object. When you specify UNACCESS, you must also specify <idvar> which provides the address of the unique name that was returned by the IDENTIFY service. When you invoke UNACCESS, any outstanding windows for the specified <idvar> are automatically unmapped with an implied <retain> NO.

```
SDKDIV 'UNIDENTIFY',<idvar>
```

The UNIDENTIFY function ends the use of a data-in-virtual object under a previously assigned ID. When you specify UNIDENTIFY, you must also specify <idvar> which provides the address of the unique name that was returned by the IDENTIFY service. If the object is still accessed or mapped under the specified ID, the system will automatically unaccess and unmap it with an implied <retain> NO.

22.2 SDKDIV -- Data In Virtual function

This section presents the syntax of the SDKDIV function along with an overview of variables returned by the SDKDIV function.

Syntax:

```
call SDKDIV 'INIT'
call SDKDIV 'TERM'
call SDKDIV ‘IDENTIFY’,’DA’,ddname,stoken,checking,ttoken
call SDKDIV ‘UNIDENTIFY’,divdid
call SDKDIV ‘ACCESS’,divdid,mode,locview
call SDKDIV ‘UNACCESS’,divdid
call SDKDIV ‘MAP’,divdid,area,offset,span,stoken,retain,pfcount
call SDKDIV ‘UNMAP’,divdid,area,retain,stoken
call SDKDIV ‘RESET’,divdid,offset,span,release
call SDKDIV ‘SAVE’,divdid,offset,span,stoken,listaddr,listsize
call SDKDIV ‘SAVELIST’,divdid,listaddr,listsize
```

The arguments to the SDKDIV function are as follows:

`area` The address of a data area to hold one or more pages of the object. You can obtain this area via an RLX/SDK GETMAIN function like so:

```
area = c2x(getmain(4Ø96,3,'any','page')).
```

This statement obtains 3 4K pages to be used by the MAP service.

`checking` Direct data-in-virtual to:
- `YES` perform data validity checking
- `NO` do not perform data validity checking

`ddname` Identifies the name of the file to which the VSAM linear data set is allocated. DDname is used with the IDENTIFY service.

`divdid` A unique 8-byte field associated with the object.

`mode` Access mode:
- `READ` allows only read for the object.
- `UPDATE` allows read and update for the object.

`listaddr` Specifies the address of a list in which each entry contains two fullwords which contain the virtual range to be saved. There must be between 3 and 255 entries in this list. `listaddr` can be obtained via a GETMAIN statement such as

```
listaddr = c2x(GETMAIN((4+4)*5,Ø,’ANY’)).
```

This call obtains a pointer to a list that contains 5 entries.

`listsize` Specifies a size of the list pointed to by the `listaddr` parameter. The REXX assignment statement sets the size of the list to 5 entries as in listsize = 5 `locview` Two options are allowed:

- `MAP` directs the system to establish a local copy of the object in the caller's address space.
- `NONE` Specifies the system is not to create a local copy of the object.

`offset` Specifies the beginning of a continuous range of blocks in a Data In Virtual object. `Offset` is used with `span` to define a continuous range of blocks in an object.

`pfcount` Specifies the additional pages the system is to read into central storage on a page fault: 0 - 255.

`release` Specifies whether pages in virtual range are to be released:

- `YES` release ALL pages in a virtual range;
- `NO` release only changed pages in a release range.

`retain` Determines what data appears in the window (area):

- `YES` -- when used with the MAP service, the data in the virtual range stays the same. The system considers all pages in the range changed.
- `NO` -- when used with the MAP service, data from the object replaces pages in the range. When used with the UNMAP service, the data in the virtual range is initialized with nulls.

`span` Specifies the number of 4K blocks that are to be processed by the MAP, RESET and SAVE services.

`stoken` This parameter must be coded as blanks since it is not supported.

`ttoken` This parameter must be coded as blanks since it is not supported.

The SDKDIV function returns the following REXX variables:

RC — Defines call completion

- Ø — Function executed successfully

SDKDIV Return code

- 41 — Invalid DIVDID parameter
- 42 — Invalid area parameter
- 43 — Invalid checking parameter
- 44 — Invalid DDNAME parameter
- 45 — Invalid LISTADDR parameter
- 46 — Invalid LISTSIZE parameter
- 47 — Invalid LOCVIEW parameter
- 48 — Invalid MODE parameter
- 49 — Invalid OFFSET parameter
- 5Ø — Invalid RETAIN parameter
- 51 — Invalid SPAN parameter
- 52 — Invalid STOKEN parameter
- 53 — Invalid TTOKEN parameter
- 54 — Invalid TYPE parameter
- 55 — Invalid PFCOUNT parameter
- 56 — Invalid RELEASE parameter

DIV Macro Return codes

Refer to: '`OS/39Ø Assembler Programming Reference`' for a list of the valid Return Codes and Reason Codes after executing the DIV macro.

DIVDID — A unique 8-byte field associated with the object returned by the IDENTIFY service.

DIVRC — decimal return code from the DIV service

DIVRRC — Hexadecimal reason code returned by the DIV service

DIVSIZE — Size of VSAM Linear data set in 4K blocks (returned by the IDENTIFY service)

DIVDDN — DDNAME of the DIV VSAM linear data objects

Example

An example which utilizes all SDKDIV functions can be found in the SDK#DIV member of the RLXEXEC library.

Chapter 23

Date and Time conversion functions

This chapter presents three Data and Time conversion routines. The SDKSTCK function obtains a timestamp in STCK instruction format and converts it to a printable format. SDKSTCK can also compute the difference between two STCK timestamps. The CONVTOD and STCKCONV functions are REXX implementations of MVS macros of the same name. CONVTOD and STCKCONV support various conversions between timestamps in STCK format and time-of-day format.

For details on the usage of this functions refer to the following IBM publications:

```
OS/390 Assembler Programming Guide
OS/390 Assembler Programming Reference
```

23.1 SDKSTCK – Convert Store Clock format to printable format

Syntax:

```
stck   = SDKSTCK('STCK')
stck   = SDKSTCK('P2STCK',pstck)
pstck  = SDKSTCK('STCK2P',stck)
stck   = SDKSTCK('DELTA',stck1,stck2)
```

Functions

`STCK`	Obtain a current timestamp by issuing the STCK instruction
`P2STCK`	Convert a printable timestamp to an internal STCK format
`STCK2P`	Convert an internal STCK timestamp to a printable format
`DELTA`	Subtract stck1 from stck2 and return the result in internal format

Parameter

`stck` An 8-byte timestamp in STCK format

`pstck` A printable timestamp in the format: `MM/DD/YYYY-HH:MM:SS.thmiju`

`MM`	the month of the year
`DD`	the day of the month
`YYYY`	the year
`HH`	hours, based on a 24-hour clock
`MM`	minutes
`SS`	seconds
`t`	tenths of a second
`h`	hundredths of a second
`m`	milliseconds
`i`	ten-thousandths of a second
`j`	hundred-thousandths of a second
`u`	microseconds

The SDKSTCK function returns the following REXX variables:

`RC` Defines call completion

`Ø`	Function executed successfully
`>Ø`	Function failed to execute successfully

Example 1

```
stck = SDKSTCK('STCK')
```

After this call, the REXX variable `stck` contains an 8-byte timestamp in internal STCK format.

Example 2

```
pstck = SDKSTCK('STCK2P',stck)
```

After this call, the REXX variable `pstck` contains a timestamp in `MM/DD/YYYY-HH:MM:SS.thmiju` format.

Example 3

```
stck = SDKSTCK('P2STCK',pstck)
```

After this call, the REXX variable `stck` contains a timestamp in internal STCK format. The `pstck` variable contains a printable STCK value in `MM/DD/YYYY-HH:MM:SS.thmiju` format.

23.2 CONVTOD – Convert Time-of-Day format to Store Clock format

The CONVTOD function accepts time and date values in several different formats and converts them to time-of-day (TOD) format. The input time and date formats are compatible with those returned by the STCKCONV function.

Syntax: `stck = CONVTOD(timetype,time,datetype,date,offset)`

where

`stck` Is the printable form of the internal STCK format.
STCK is a 16 character value such as `'AF1D238C7949F000'`.

`timetype` Specifies the format of the input time value

`DEC` Unsigned packed decimal digits representing a time value in the form `HHMMSSthmiju0000` where:

HH	hours, based on a 24-hour clock
MM	minutes
SS	seconds
t	tenths of a second
h	hundredths of a second
m	milliseconds
i	ten-thousandths of a second
j	hundred-thousandths of a second
u	microseconds

BIN Unsigned number representing a time value in 0.01 of seconds

MIC An unsigned 64-bit binary number representing a time value in microseconds. Bit 51 represents 1 microsecond. This represents the *internal* STCK format.

time Time in the format specified by the timetype parameter

datetype datetype - Specifies the format of the input date value:

YYDDD	ØCYYDDD
YYYYDDD	ØYYYYDDD
DDMMYYYY	DDMMYYYY
MMDDYYYY	MMDDYYYY
YYYYMMDD	YYYYMMDD

where

ØC	the century - 00 represents 19YY, 01 represents 20YY
YY	the last two digits of the year
YYYY	the year
DDD	the day of the year (Julian date)
DD	the day of the month
MM	the month of the year

date Date in the format specified by the datetype parameter

offset Optional, number in HHMM format to be added to the input time.

The CONVTOD function returns the following REXX variables:

RC Defines call completion

Ø	Function executed successfully
41	Invalid timetype parameter
42	Invalid time parameter
43	Invalid datetype parameter
44	Invalid date parameter
45	Invalid offset parameter

Example

```
/* Rexx */
timetype        = 'DEC'
time            = '1156051365430000'
datetype        = 'YYYYMMDD'
date            = '19970815'
tod             = convtod(timetype,time,datetype,date)
say 'TOD='tod 'Rc='rc
```

After this call, the REXX variable `tod` is set to: `'AF1D238C7949F000'`

23.3 STCKCONV - Convert Store Clock format to Time-of-day format

The STCKCONV function converts an input time-of-day (TOD) clock value to a time-of-day and date value. STCKCONV then returns these converted values to the caller in the format they request. The time and date formats supported by STCKCONV are compatible with the format returned by the TIME macro. The TIME macro returns the contents of the TOD clock as a time-of-day and date value. The time-of-day and date formats supported by STCKCONV are compatible with the formats accepted by the CONVTOD function.

Syntax: `TOD = STCKCONV(stck,timetype,datetype)`

where

`TOD` 32-byte character string value which depends upon the `timetype` and `datetype` parameters.

`stsk` Is the printable form of the internal STCK format. STCK is a 16 character value such as `'AF1D238C7949F000'`.

`timetype` Specifies the format of the input time value

`DEC` Unsigned packed decimal digits representing a time value in the form `HHMMSSthmiju0000`, where:

- `HH` hours, based on a 24-hour clock
- `MM` minutes
- `SS` seconds
- `t` tenths of a second
- `h` hundredths of a second
- `m` milliseconds
- `I` ten-thousandths of a second
- `j` hundred-thousandths of a second
- `u` microseconds

`BIN`	Unsigned number representing a time value in 0.01 of a second
`MIC`	Unsigned 64-bit binary number representing a time value in microseconds. Bit 51 represents 1 microsecond. This represents the internal STCK format.

`Datetype` Specifies the format of the input date value:

```
YYDDD     ØCYYDDDF
YYYYDDD   ØYYYYDDD
DDMMYYYY  DDMMYYYY
```

where

`ØC`	the century - 00 represents 19YY, 01 represents 20YY
`YY`	the last two digits of the year
`YYYY`	the year
`DDD`	the day of the year (Julian date)
`DD`	the day of the month
`MM`	the month of the year

The STCKCONV function returns the following REXX variables:

`RC` Defines call completion

`Ø`	Function executed successfully
`41`	Invalid stck parameter
`42`	Invalid timetype parameter
`43`	Invalid datetype parameter

Example

```
/* Rexx */
stck           = 'AF1D6Ø36995A56ØØ'
timetype       = 'DEC'
datetype       = 'YYYYMMDD'
tod            = stckconv(stck,timetype,datetype)
say 'TOD='tod 'Rc='rc
Return
```

This exec displays a TOD value as: `"16272965315700001997081500000000"`

AcceleREXX

Interpretative REXX Compiler

User

Guide

Version 2.2

June 2010

Publication RCX-001-6

Relational Architects Intl

This Guide: (RAI Publication RCX-001-6)

This document applies to AcceleREXX Version 2.2 (June 2010) and RLX for z/OS Version 9 Release 1 (June 2010), and all subsequent releases, unless otherwise indicated in new editions or technical newsletters. Specifications contained herein are subject to change and will be reported in subsequent revisions or editions.

Purchase Orders for publications should be addressed to:

Documentation Coordinator
Relational Architects Intl.
Riverview Historic Plaza
33 Newark Street
Hoboken NJ 07030 USA

Tel: 201 420-0400
Fax: 201 420-4080
Email Sales@relarc.com

Reader comments regarding the product, its documentation, and suggested improvements are welcome. A reader comment form for this purpose is provided at the back of this publication.

Ed 21F10

AcceleREXX

Table of Contents

Chapter 1

AcceleREXX

Concepts and Facilities

AcceleREXX reads REXX execs and produces object modules that can be link-edited using the standard Linkage Editor or Binder to create executable load modules. AcceleREXX supports the complete REXX language (including INTERPRET and TRACE instructions), as well as the standard IBM REXX language extensions, host commands and function packages described in the TSO/E REXX Reference.

Compiled REXX programs can have broader capabilities than interpreted ones. One example is support of the ADDRESS LINK and ADDRESS ATTACH host commands. Compiled REXX execs can be invoked by an ADDRESS LINK or ADDRESS ATTACH command, while interpretive REXX execs cannot.

1.1 The Compilation Process

The process of compiling a REXX exec into an executable module consists of the following steps:

> The REXX compiler options are read and validated.

> A REXX exec allocated to the RCXIN DD statement is read into memory.

> The syntax of the REXX exec is verified and comment lines are removed.

> The exec is condensed and, optionally, encrypted for confidentiality.

> A listing and symbol cross reference is optionally printed to the file defined by the RCXPRINT DD statement.

> An object module consisting of executable code, REXX source statements and references to external subroutines is built in the format accepted by the Linkage Editor or Binder and written to the RCXOBJ DD dataset. This dataset may be a temporary or permanent file.

> Finally, the object module produced in the preceding step is passed to the Linkage Editor or Binder to produce an executable module with attributes RMODE=31 and AMODE=ANY. Optionally, it can be made reenterable.

1.2 Locating a Compiled Program

As with any load module, AcceleREXX compiled programs are placed in a load library and can be invoked via the MVS LINK macro instruction. The standard search order for load modules is:

1. The job pack area
2. The task library
3. The step library
4. The link pack area (LPA)

Compiled programs can be placed in ISPLLIB when running under TSO/ISPF since it is used as a tasklib.

1.3 Invoking a Compiled Program

Compiled REXX programs can be invoked three ways:

As commands: This invocation type is used with REXX execs executing as the main routine or when invoked via ADDRESS LINK or ADDRESS ATTACH.

As functions: In this invocation type, a REXX exec ***must*** return a value.

As subroutines: In this invocation type, the REXX exec may return a character string of variable length or a numeric return code. The caller can inspect the REXX special variable RESULT for the value returned by the subroutine.

When the compiled REXX exec is invoked as a subroutine or function, it is assumed that upon termination, control will be returned to a caller that is also a REXX exec (compiled or interpretive). In the case of command invocation, it is assumed control will be returned to the Operating System (which may in turn return control to an exec).

1.3.1 Invoking a Compiled REXX Program in a non-TSO address space

If a compiled program does not use TSO/E facilities, it can be invoked in a non-TSO address space with the JCL shown in Figure 1.1. The compiled exec can be invoked directly, like any load module. There is no need for a frontend program like IRXJCL.

> ***NOTE:*** *Henceforth, we use RCXTMAIN as the sample exec which is provided as one of the AcceleREXX IVP's.*

```
//RAIJOB  JOB                                          (1)
//STEP1   exec PGM=RCXTMAIN,PARM='run-time parameters' (2)
//STEPLIB  DD  DSN=SOME.USER.LOAD,DISP=SHR             (3)
//SYSTSPRT DD  SYSOUT=*                                (4)
//SYSTSIN  DD  *                                       (5)
SOME INPUT IF NEEDED
/*
```

Figure 1.1 JCL to invoke a compiled exec in a non-TSO address space

In Figure 1.1,

(1) Specifies a valid job card

(2) PGM specifies the name of a compiled exec -- in this case RCXTMAIN. The PARM operand specifies parameters passed to RCXTMAIN which the REXX ARG instruction can parse.

(3) Specifies the library in which the compiled RCXTMAIN load module resides.

(4) The SYSTSPRT DD statement specifies a destination for the REXX SAY instruction

(5) The SYSTSIN DD statement, if required, provides input to be processed by the REXX PULL instruction.

1.3.2 Invoking a Compiled REXX Program within TSO

In a TSO address space, a compiled exec can be invoked using the TSO CALL command at the READY prompt, as shown below:

```
READY
CALL '&RLXHLQ.RLXLOAD(RCXTMAIN)' 'P1 P2'
```

Alternatively, if RCXTMAIN is in a library eligible for search by the MVS LOAD macro, it can be invoked as follows:

```
READY
RCXTMAIN P1 P2
```

1.3.3 Invoking a Compiled REXX Program in a TSO/ISPF environment

If TSO/ISPF is active, a compiled REXX exec can be invoked using the ISPF SELECT service -- e.g.

```
ISPEXEC SELECT CMD(RCXTMAIN run-time parameters')
```

Alternatively, either of the methods described in Section 1.3.2 above can be used to invoke a compiled REXX application that will make use of ISPF dialog services and access ISPF dialog variables.

1.3.4 Invoking a Compiled REXX Program as an ISPF EDIT macro

AcceleREXX can be used to compile ISPF EDIT macros written in REXX. The RCXF exec distributed with AcceleREXX is an example of such a compiled Edit macro. This macro (which handles exec formatting and indentation) can be invoked during Edit by entering `!RCXF (IN 2` on the command line. (e.g. `COMMAND ===> !RCXF (IN 2`). Execution of this macro causes the statements at each nesting level of IF-THEN-ELSE statements to be indented two spaces.

> ***NOTE:*** *The exclamation point prefix (!) is used to distinguish compiled program macros from macros whose code will be interpreted.*

1.3.5 Invoking a Compiled REXX Program from a High Level Language

Assembler and high level languages such as COBOL and PL/1 can directly call compiled REXX programs the same way they call any external routine. A compiled REXX program uses standard OS linkage conventions and is a well-behaved application. The compiled REXX application appears to the caller as an Assembly language routine.

On entry to a compiled REXX program, registers must be set as detailed below. Upon return, all of the caller's registers are restored -- except R15 which will contain a return code.

R0	Undefined
R1	The address of a standard OS parameter list: `R1 ===> parm_ptr@ ===> LL,command_buffer` where LL is the length of a command_buffer ***NOTE:*** *A parameter must always be passed even if its length is zero.*
R2 through R12	Undefined
R13	Address of register save area
R14	Caller's return address
R15	Compiled REXX program entry point

1.4 REXX Compiler options

AcceleREXX can be controlled by the run-time parameters discussed in this section. AcceleREXX is implemented as the RCXC command. Before invoking it you must allocate several data sets:

Alternatively, Chapter 3 describes the AcceleREXX ISPF dialog with which you can select one or more source REXX execs for compilation and link edit.

RCXIN — This file contains the source exec. It can be a sequential file or a PDS with an explicitly specified member name. File attributes must be RECFM = FB or VB and LRECL = between 80 and 255.

RCXOBJ — This file will contain an object module generated by AcceleREXX. This file must be sequential or a PDS with an explicitly specified member name. You can allocate it as a temporary output file which will be passed to the Linkage Editor or Binder.

RCXPRINT — Defines an optional output file for REXX program and Cross Reference listings.

RCXPARMS — Defines an optional input file containing parameters that govern AcceleREXX. Generally, parameters are passed to the compiler in the PARM field of the JCL exec statement. The PARM field is limited to 100 characters while input supplied through the RCXPARMS file is *not* constrained by this limitation.

All AcceleREXX parameters have the fixed form KEYWORD(VALUE). A description of these compile options follows in alphabetical order:

BOUNDS(left,right)

This parameter indicates the range of columns within an exec source line which contain REXX statements for processing. This parameter was introduced to exclude sequence numbers sometimes found in columns 73-80 or 1-8. **left** designates a beginning column number while **right** designates an ending column number between which to read REXX statements. For example, to exclude sequence numbers in columns 73-80, code `BOUNDS(1,72)`.

If this parameter is not coded, the entire exec line is selected for processing.

COMPREC(lrecl|0)

This parameter designates the maximum logical record length for a compressed exec. The compiler will remove comments and redundant spaces from an exec. It will also combine multiple source lines, separating them with a semicolon (;). If you use this option, the performance of your exec will improve slightly. The default value for this parameter is 0 (zero) -- REXX source lines will not be combined.

CONDENSE(YES|NO)

CONDENSE(YES) directs AcceleREXX to remove comments and unnecessary spaces from the REXX exec. If you intend to use the REXX TRACE instruction in a compiled exec, you may wish to retain the comments to assist with debugging. In that case, choose CONDENSE(NO).

CSECT(name)

This parameter must always be specified; **name** designates the control section which will be generated by the compiler.

ENCODE(YES|NO)

This option directs AcceleREXX whether to encode (YES) or not to encode (NO) an exec's statements. Use YES to insure that the code within a compiled exec remains confidential. There is some overhead associated with encoding, so use this option judiciously.

INITHCE(hostname|TSO)

This parameter defines the Initial Host Command Environment assigned to a compiled exec when it is invoked. The default for **hostname** is TSO. To invoke a compiled exec in a non-TSO address space you must use a **hostname** of MVS or some other TSO-independent host command environment.

LIST(YES|NO)

Specify `YES' if you wish to obtain a source listing for an exec.

PAGESZ(pageno)

Specify the number of printed lines per page for the program listing and the Symbol Cross Reference. The default is 55 lines per page.

SLINK(rtn1,rtn2,...)

This option specifies a list of external subroutines which will be link-edited together with the compiled exec. These external subroutines can be other REXX execs, either compiled or interpretive. This option enables substantial performance improvement if you use modular program design and have external subroutines.

> ***NOTE:*** *If you intend to invoke a compiled REXX exec using ADDRESS LINK or ADDRESS ATTACH, you should not statically link the exec with its caller. This is a limitation of these host commands, not of AcceleREXX.*

SPECIAL(Y|N)

The SPECIAL compile option lets you specify whether or not the compiled exec will issue ISPF dialog service requests or execute in the NetView environment. Coding Y signals AcceleREXX to conduct the special initialization processing required by these two environments while N (the default) suppresses this special initialization processing.

> ***NOTE:*** *REXX execs which will merely execute within ISPF but do not issue ISPF dialog services* ***need not*** *be compiled with the SPECIAL option.*

XREF(YES|NO)

Specify `YES' if you wish to obtain a Symbol Cross Reference.

Chapter 2

AcceleREXX Precompiler Directives

This chapter is intentionally omitted.

When released, this chapter will describe a set of AcceleREXX precompiler directives that are planned for a future release of AcceleREXX. These directives -- to be specified as REXX comments -- will enable developers to do such things as conditionally include or exclude sections of REXX code and format their execs to clarify their structure and functionality.

Chapter 3

The RLX for z/OS Menu

and

AcceleREXX Compiler Dialogs

3.1 The Main Menu

The RLX for z/OS dialog and Main Menu is invoked from Option 6 of the ISPF Primary Menu by entering the command RCX. This command requires no parameters. It is assumed that RLX installed successfully and all required ISPF libraries are allocated.

After the RCX command is entered, the RLX for z/OS Main Menu (shown in Figure 3.1) is displayed.

```
---------------------------  RLX for z/OS Main Menu  --------------------------
 Option ===>

    1  Environment  - Setup and maintain RLX environment
    2  Compile      - Compile discrete REXX programs
    3  MergeCompile - Merge and compile multiple REXX programs
    4  SDK          - REXX Software Development Kit
    5  VSAM         - REXX to VSAM Interface
    T  Tutorial     - Description of this menu's options
    X  Exit         - Leave this dialog

 Enter END command to Exit
```

Figure 3.1 RLX for z/OS Main Menu

You should select Option 1 -- Environment -- at least once to define a job card and default dataset names to be used by subsequent compile and demonstration dialogs. (To verify the meaning/purpose of the selections available on this -- or subsequent -- panels, use the Tutorial.)

3.2 The RLX for z/OS Setup Dialog

```
--------------------------- RLX for z/OS Setup Menu --------------------------
 Command ===>

    1  Job / Parm  - Specify JOB statement(s) and JCL parameters
    2  Dsnames     - Specify High Level Qualifiers for dataset names
    3  Product     - Review / Adjust names of product libraries
    4  User        - Review / Adjust names of user libraries
    5  Allocate    - Allocate AcceleREXX user libraries
    6  Color       - Define screen color attributes
    T  Tutorial    - Description of this menu's options
    X  Exit        - Leave Setup Menu
```

Figure 3.2 The RLX for z/OS Setup Menu

When you select Option 1, the panel depicted in Figure 3.2 is displayed.

Options 1 through 6 of the RLX for z/OS Setup Menu apply to all users. As such, they are discussed in this Chapter. Each user must select Options 1 through 6 on their first invocation of RLX for z/OS to specify a job card (and other JCL information) and to identify the names of the RLX for z/OS system and user datasets. Thereafter, choose the Setup Menu options to change `User libraries' or other options.

3.2.1 Specify JOB and JCL Parameters (Option 1 on the Main Menu)

The panel on Figure 3.3 prompts you to specify valid JOB and JCL parameters for the jobs tailored by the RLX for z/OS dialogs.

```
-------------------- RAI Product Installation Parameters --------------------
 Command ===>
 RLX204 - Specify RLX Job and Jobparm values and press ENTER

 RAI Background Job/Jobparm statement(s)
    ===> //RAI1RLX JOB (ACCOUNT),CLASS=A,MSGCLASS=H
    ===>
    ===>
    ===>

 Specify UNIT names for new disk datasets (E.G. - SYSDA, DISK, VIO)
    Temp UNIT name     ===> SYSDA         (for temporary datasets e.g. VIO)
    Perm UNIT name     ===> SYSDA         (for permanent datasets e.g. SYSDA)
    VSAM volume        ===> VOL001        (on which to allocate VSAM datasets)

 Specify name of the IBM ASSEMBLER program
    System Assembler   ===> ASMA90        (IEV90 or ASMA90)
```

Figure 3.3 JCL Parameters

3.2.2 Specify dataset names

Option 2 of the RLX for z/OS setup menu lets you specify High Level Qualifiers (HLQ) for RLX for z/OS product datasets and user datasets. The dialog will generate all dataset names in REXX variables using HLQ and a fixed last qualifier. You may find it convenient to use this dialog to change the names of all your RLX for z/OS datasets. Figure 3.4 illustrates the dataset specification panel.

You may need to alter the names of the RLX for z/OS product libraries when a new RLX for z/OS release is installed. Change the HLQ of user libraries as needed.

```
----------------  Specify Names of Product and User Libraries  ----------------
 Command ===>


 This dialog allows you to specify the user libraries required by
 various RLX for z/OS components.  You may enter High Level Qualifiers
 to be used to generate names for the user libraries.

 Specify High Level qualifiers of RLX for z/OS product libraries
    Project  ===> RLX
    Library  ===> Tvrm

 Specify High Level qualifiers of your REXX Compiler user libraries
    Project  ===> user
    Library  ===> library
```

Figure 3.4 Specify Dataset names

3.2.3 Specify product dataset names

Option 3 lets you alter the dataset names generated by the Option 2 dialog. The low level qualifier of the RLX libraries is fixed by the dialog architecture and cannot be changed. Figure 3.5 illustrates the RLX for z/OS product library display.

```
------------------------------ Product Libraries ------------------------------
Command ===>

Only the Project and Library qualifiers of the dataset names may be
changed.  The type is fixed by the architecture.

Product Load Library (Contains REXX compiler code)
   Project ===> RLX
   Library ===> Tvrm
   Type    ===> RLXLOAD

Product No Call Library (Contains REXX compiler subroutines)
   Project ===> RLX
   Library ===> Tvrm
   Type    ===> RLXNCAL

Product Exec Library (Contains Sample REXX programs)
   Project ===> RLX
   Library ===> Tvrm
   Type    ===> RLXEXEC
```

Figure 3.5 Specify RLX for z/OS product libraries

3.2.4 Specify user dataset names

Option 4 of the RLX for z/OS setup menu lets you further alter your own user libraries to be used in the AcceleREXX compile dialogs and by demonstration programs. Figure 3.6 illustrates the user dataset specification panel.

```
------------------------------- User Libraries ----------------------------
Command ===>

User OBJECT Library (Output of REXX Compiler)
   Project ===> RLX
   Library ===> Tvrm
   Type    ===> RCXOBJ

User Load Library (Output from Linkage Editor -- executable load modules)
   Project ===> RLX
   Library ===> Tvrm
   Type    ===> RCXLOAD

User File Tailoring Library
   Project ===> RLX
   Library ===> Tvrm
   Type    ===> RLXFILE
```

Figure 3.6 Specify RLX for z/OS user libraries

The libraries specified on this panel hold the following contents, respectively:

The user OBJECT library contains an object module -- a REXX exec compiled by AcceleREXX. Use it to incrementally compile several REXX programs and then link-edit them into one executable load module.

The user LOAD library contains executable compiled REXX programs which have been link-edited. To execute these programs, the LOAD library must be allocated to ISPLLIB before ISPF is initialized.

The user file tailoring library contains JCL tailored by a REXX Compile dialog and other objects tailored by an RLX/SDK demonstration dialog.

3.2.5 Allocate user datasets

Option 5 of the RLX for z/OS setup menu lets you allocate the user datasets you specified in Option 4. If these datasets have already been allocated, you will receive a warning message. Figure 3.7 illustrates the dataset allocation panel.

```
-------------------------- Allocate User Libraries  -------------------------
Command ===>


The following user libraries were specified in the dialog and will be allocated
by your request.

   User OBJ  library       ===> RLX.Tvrm.RCXOBJ
   User LOAD library       ===> RLX.Tvrm.RCXLOAD
   User File Tailoring lib ===> RLX.Tvrm.RLXFILE

User library allocation
   Allocate ===> N       (Y - allocate libraries; N - do not allocate)
   Volume   ===> VOL001  (DASD volume for user libraries)
```

Figure 3.7 Allocate RLX for z/OS user libraries

3.2.6 Specify Color Preferences

Option 6 of the RLX for z/OS setup menu lets you specify color attributes for all RLX for z/OS panels. You can change color attributes anytime. Figure 3.8 illustrates the color attribute specification panel.

```
-------------------------- Screen Color Attributes --------------------------
Command ===>

Specify Color Attributes             W - White
   Input  - Normal  ===> B           R - Red
   Input  - High    ===> G           B - Blue
   Output - Normal  ===> Y           G - Green
   Output - High    ===> R           P - Pink
   Text   - Normal  ===> T           Y - Yellow
   Text   - High    ===> W           T - Turquoise
```

Figure 3.8 Color Preferences

3.3 The AcceleREXX Discrete Compile Dialog

The AcceleREXX Compile dialog lets you select one or more source REXX procedures for compilation in a single pass.

Option 2 of the RLX for z/OS Main Menu (shown in Figure 3.1) lets you compile a number of discrete REXX programs.

Figure 3.9 illustrates the AcceleREXX Prescreen for compiling discrete REXX programs. On it, you identify the REXX source module you wish to compile. Alternatively, you can specify an entire library or a concatenated *set* of libraries from which one or more REXX source modules can be selected. Each selected source module will produce one load module.

Either the `ISPF library' or `Other Data set name' can be specified with the `Other Data set name' taking precedence.

```
---------------------------- AcceleREXX Compiler -------------------------
Command ===>
RCXØ24 - Select EXEC(s) to be compiled and linked into discrete load modules
ISPF library:
   Project ===> RLX
   Group   ===> Tvrm     ===>          ===>           ===>
   Type    ===> RLXEXEC
   Member  ===> RNV*           (Blank or pattern for member selection list)

Other partitioned or sequential data set:
   Data set name  ===>
```

Figure 3.9 AcceleREXX Compile Panel 1

You may specify a pattern for the member name in accordance with ISPF conventions to limit the display to a list of members ***which meet your selection criteria***. This pattern can be specified in either the Member name field (as illustrated above by RNV*) or by specifying an `Other Dataset Name' in the form DATA.SET.NAME(PATTERN*).

Figure 3.10 illustrates the resulting member list from which one or more REXX source modules may be selected for processing.

```
REXX Compile: RLX.Tvrm.RLXEXEC --------------------------------- ROW 1 OF 6
Command ===>                                                Scroll ===> HALF

  Name     Action  Lib VV.MM  Created     Changed      Size  Init    Mod   ID
s RNVF              1  01 79 03/08/31 03/09/22 15:55    217   135      0 RLX4
s RNVFHELP          1  01 03 03/08/31 03/09/22 15:39     32    35      0 RLX4
s RNVFHLP1          1  01 10 03/09/01 03/09/22 15:42     33    31      0 RLX4
  RNVFPAN           1  01 17 03/08/31 03/09/22 15:40     33    35      0 RLX4
  RNVFPANS          1  01 08 03/09/01 03/09/22 15:41     33    31      0 RLX4
  RNVL              1  01 01 03/08/31 03/09/22 15:43    100   100      0 RLX4
```

Figure 3.10 AcceleREXX Member Selection List

You can key a code into the row selection field to the left of the member name. The `S' selection code identifies a REXX source module you wish to compile with AcceleREXX. Use the 'S' row command to select REXX programs and 'U' to unselect REXX programs.

Once you have selected all members from the currently displayed panel, press ENTER or one of the scroll PF keys. The dialog will redisplay the member list with the literal `` `Selected' `` appearing to the right of the selected member name(s).

You can continue to scroll, select and unselect members for as long as necessary. When you've completed your selections, press the END PF key or key-in the END primary command. The dialog will advance you to the panel illustrated in Figure 3.11.

```
REXX Compile: RLX.Tvrm.RLXEXEC --------------------------------- ROW 1 OF 6
Command ===>                                                Scroll ===> HALF

  Name     Action  Lib VV.MM  Created     Changed      Size  Init    Mod   ID
  RNVF     Selected 1  01 79 03/08/31 03/09/22 15:55    217   135      0 RLX4
  RNVFHELP Selected 1  01 03 03/08/31 03/09/22 15:39     32    35      0 RLX4
  RNVFHLP1 Selected 1  01 10 03/09/01 03/09/22 15:42     33    31      0 RLX4
  RNVFPAN           1  01 17 03/08/31 03/09/22 15:40     33    35      0 RLX4
  RNVFPANS          1  01 08 03/09/01 03/09/22 15:41     33    31      0 RLX4
  RNVL              1  01 01 03/08/31 03/09/22 15:43    100   100      0 RLX4
```

Figure 3.11 AcceleREXX Member Selection List

3.3.1 AcceleREXX Compile / Link-Edit Specifications

Figure 3.12 illustrates the panel on which you can specify various AcceleREXX compile options as well as attributes of the link-edited load module.

The set of prompts grouped under the heading `Compile Options` were discussed in detail in Section 1.4 of this manual and are not repeated here.

```
-------------  Discrete REXX Compile / Link Edit Specifications  -------------
 Command ===>
 RCXØ21 - Specify AcceleREXX compile and Link Edit options and press ENTER
 Specify target Load Library for the REXX load module(s)
    Node 1   ===> RLX
    Node 2   ===> Tvrm
    Node 3   ===> RCXLOAD        (may be blank)

 REXX Compile Options            (Blanks accept defaults)
    SPECIAL  ===> N              (Y/N - uses ISPF services)
    BOUNDS   ===>    -           (within which REXX source occurs - default ALL)
    INITHCE  ===> TSO            (Initial REXX Host Command Environment)
    CONDENSE ===>                (Y/N - Remove comments and blanks)
    ENCODE   ===>                (Y/N - Encrypt REXX source text)
    LRECL    ===>                (Size of logical records in condensed EXEC)
    LIST     ===>                (Y/N - Produce REXX EXEC listing)
    XREF     ===>                (Y/N - Produce symbol cross reference listing)
    PAGESZ   ===>                (Depth of print page in lines)
```

Figure 3.12 Compile / Link-Edit Options

The ***Target load library*** prompts lets you specify the load module library into which the load module(s) should be link-edited. The default is the AcceleREXX User Load library specified through Option 5 of the AcceleREXX Setup Menu.

Review and revise these Compile and Link-Edit options as appropriate and press ENTER.

3.4 The AcceleREXX Merge Compile Dialog

The AcceleREXX Merge Compile dialog lets you select one or more source REXX procedures for compilation into a composite load module in a single pass. In addition, you can select *previously compiled* REXX object modules to be link-edited in the current load module.

Options 3 of the RLX for z/OS Main Menu (shown in Figure 3.1) lets merge several REXX programs into a composite REXX program using the AcceleREXX precompiler. The Merge Compile dialog uses similar panels to those displayed by the Discrete Compile dialog.

Figure 3.13 illustrates the panel on which you can specify various AcceleREXX compile options as well as attributes of the link-edited load module.

The set of prompts grouped under the headings `Target Load Library and AcceleREXX Compile Options` were discussed in detail in Section 3.3.1 and Section 1.4 respectively of this manual and are not repeated here.

```
------------  Composite REXX Compile / Link Edit Specifications  ------------
 Command ===>
 RCX021 - Specify AcceleREXX compile and Link Edit options and press ENTER
 Composite Specifications       (for the link edited REXX load module)
    Name     ===> RNVDSEL       (of the composite load module)
    Entry Pt ===> RNVDSEL       (First AcceleREXX'd routine to receive control)
    Special  ===> N             (Y/N - uses ISPF services)
    Select   ===>               (Y/N - link edit previously compiled EXECs)
    Parm Num ===> 0             (Number of parameters: For system exits only)

 Specify target Load Library for the Compiled REXX load module
    Node 1   ===> RLX
    Node 2   ===> Tvrm
    Node 3   ===> RCXLOAD       (may be blank)

 AcceleREXX Compile Options     (Blanks accept AcceleREXX defaults)
    BOUNDS   ===>    -          (within which REXX source occurs - default ALL)
    INITHCE  ===> TSO           (Initial REXX Host Command Environment)
    CONDENSE ===>               (Y/N - Remove comments and blanks)
    ENCODE   ===>               (Y/N - Encrypt REXX source text)
    LRECL    ===>               (Size of logical records in condensed EXEC)
    LIST     ===>               (Y/N - Produce REXX EXEC listing)
    XREF     ===>               (Y/N - Produce symbol cross reference listing)
    PAGESZ   ===>               (Depth of print page in lines)
```

Figure 3.13 Merge Compile / Link Edit options

The ***Name*** field lets you designate the name of the link-edited composite load module. The default is simply the first (or only) EXEC selected for compilation.

The ***Entry Pt*** prompt identifies the name of the entry point EXEC -- i.e., the first EXEC within the link-edited composite to receive control when the compiled application is invoked. Once again, the default is simply the first or only module selected.

The ***Special*** prompt asks you to specify whether the compiled EXEC(s) will require special processing. This is the case if the compiled EXEC issues ISPF dialog services or executes within the NetView environment.

> ***NOTE:*** *If the compiled EXEC will simply execute within a TSO/ISPF environment but does not request ISPF services, then special processing is* ***not*** *required.*

The ***Select*** prompt lets you select additional, previously compiled REXX EXEC(s) (that exist in Object module form) for inclusion in the link-edited composite load module. This facility is described and illustrated in the Section 3.4.1.

The ***Parm Num*** prompt specifies the number of discrete parameters a compiled REXX exec will receive. This parameter is applicable ***only*** to compiled execs that will run as system exits.

3.4.1 Linkage Editing Previously Compiled Object Modules

If you specified `Y' in response to the Select field prompt, you will have an opportunity to select additional, previously compiled REXX EXEC(s) for inclusion in the link-edited load module. These additional modules exist in object module form.

Figure 3.14 illustrates the panel that lets you specify a library (or a concatenated set of libraries) from which one or more compiled REXX object modules can be selected. Once again you have the opportunity to specify a single module name, a generic pattern, or the entire contents of a library (or set of libraries) as illustrated in Figure 3.14.

Key the `S' selection code into the row selection field to the left of each object module you wish to include in the composite load module (as illustrated in Figure 3.15). The resulting panel confirms your selections -- as illustrated in Figure 3.16.

```
---------------------------- AcceleREXX Link Edit -------------------------
Command ===>
RCXØ22 - Select previously compiled REXX EXECs for Link Edit in RCXSAMP
Object or NCAL library:
   Project ===> RLX
   Group   ===> Tvrm    ===>           ===>           ===>
   Type    ===> RCXOBJ
   Member  ===> R*             (Blank or pattern for member selection list)

Other partitioned Object or NCAL library
   Data set name  ===>
```

Figure 3.14 Object Library Specification

```
REXX Link Edit: RLX.Tvrm.RCXOBJ ------------------------------  ROW 1 OF 10
Command ===>                                                  Scroll ===> HALF

  Name     Action  Lib VV.MM  Created     Changed      Size  Init    Mod   ID
s RMVDERR          1
s RMVDGV           1
s RMVDPDS          1
  RMVDSEL          1
  RNVF             1
  RNVFHELP         1
  RNVFHLP1         1
  RNVFPAN          1
  RNVFPANS         1
  RNVL             1
```

Figure 3.15 Object Module Selection

```
REXX Link Edit: RCX.Tvrm.RCXOBJ ------------------------------  ROW 1 OF 10
Command ===>                                                  Scroll ===> HALF

  Name     Action  Lib VV.MM  Created     Changed      Size  Init    Mod   ID
s RMVDERR  Selected 1
s RMVDGV   Selected 1
s RMVDPDS  Selected 1
  RMVDSEL          1
  RNVF             1
  RNVFHELP         1
  RNVFHLP1         1
  RNVFPAN          1
  RNVFPANS         1
  RNVL             1
```

Figure 3.16 Object Module Selection

3.5 Tailor and Submit the Compile / Link-Edit Jobstream

Once you have completed your source and object module selections for either the discrete or merge compile dialogs, press the END PF key.

The display will lock while AcceleREXX tailors the compile link-edit jobstream. You can review and revise the generated JCL and submit the job. When it completes successfully, the compiled REXX load module is available for use.

Be sure the target load module library is allocated as described in Section 1.2 before you attempt to invoke the compiled REXX application. In addition, since the load module is accessible through the standard MVS search order, it will be invoked ***before*** the interpreted source REXX EXEC.

3.6 REXX Formatter - The RCXF Exec

You should consider adopting certain standards in writing REXX code to enhance the maintainability and comprehensibility of your REXX applications. AcceleREXX provides an EDIT macro named RCXF which can be used to format your execs in a uniform way. To make use of the RCXF macro, Edit your exec in ISPF EDIT and on the command line type !rcxf as shown in Figure 3.6.1 below.

```
EDIT       RAIXN.DEMO.EXEC(RCXTEST) - 01.04                Columns 00001 00072
Command ===> !rcxf                                            Scroll ===> CSR
****** ***************************** Top of Data ******************************
000001 /* REXX */
000002 trace 'O'
000003 address RLX
000004 string1        = 'AcceleREXX rocks the z/OS REXX world!' /* Literal */
000005 do i =1 to words(string1)
000006    if word(string1,i) = Caps('rexx') then do /* check sought */
000007       say 'OK, found REXX in word' i /* found */
000008       if i > 5 then /* check how many words */
000009          say 'Looked for a long time' /* long search */
000010       else
000011          say 'Found quite quickly'    /* short search */
000012       leave
000013    end
000014    else
000015       say 'Word' i 'is not it'        /* substring not found */
000016 end
000017 returN 8
000018
000019 Caps: procedure         /* subroutine to fold to upper case */
000020  parse upper arg args   /* make upper */
000021 return args            /* and return to caller */
****** *************************** Bottom of Data ***************************
```

Figure 3.6.1 Unformatted REXX Exec named RCXTEST

After you press the ENTER key, the panel shown in Figure 3.6.2 is displayed:

```
---------------------------- REXX Source Formatter ----------------------------
 Command ===>

 Exec being formatted
   Exec name  . . . . . . . . RCXTEST
   Line count . . . . . . . . 21

 Format options
   Indent left spaces . . . . 3  (Shift IF/SELECT/DO logic groups)
   Left margin  . . . . . . . 2  (Create left margin)
   Right margin . . . . . . . 72 (Create left margin)
   Right justify comments . . Y  (Y/N)

 Capitalize keywords
   Enable capitalization  . . Y  (Y/N)
   Variable to lowercase  . . Y  (Y/N)
   Labels to uppercase  . . . Y  (Y/N)
   Functions to cap 1st . . . Y  (Y/N)
   Keyword to uppercase . . . Y  (Y/N)

 Cross Reference Options (XREF)
   Create XREF  . . . . . . . N (Y/N)
   Dataset name for XREF  . . 'RAIØ27.RCXF.XREF'

 Diagnostics
   Debug  . . . . . . . . . . N (Y/N)
   Debug lines  . . . . . . . 2ØØØ
   REXX TRACE type  . . . . . O (O/R/C/I/L)

 Notes
 o Check EXEC before saving - it may not be formatted in accordance
   with your expectations.
 o Always create a backup of your exec and thoroughly test the formatted exec.
```

Figure 3.6.2 Specify RCXF parameters

Specify the parameters required by the RCXF formatter and press the ENTER key. The formatted exec shown in Figure 3.6.3 will be displayed:

```
EDIT       RAIXN.DEMO.EXEC(RCXTEST) - 01.05                Columns 00001 00072
Command ===>                                                  Scroll ===> CSR
****** ***************************** Top of Data ******************************
/* REXX */
 TRACE 'O'
 ADDRESS rlx
 string1     = 'AcceleREXX rocks the z/OS REXX world!'  /* Literal */
 DO i =1 TO WORDS(string1)
    IF WORD(string1,i) = Caps('rexx') THEN DO         /* check sought */
       SAY 'OK, found REXX in word' i                        /* found */
       IF i > 5 THEN                          /* check how many words */
         SAY 'Looked for a long time'                  /* long search */
         ELSE
           SAY 'Found quite quickly'                  /* short search */
           LEAVE
    END
    ELSE
      SAY 'Word' i 'is not it'                 /* substring not found */

   END
 RETURN 8

Caps: PROCEDURE                  /* subroutine to fold to upper case */
 PARSE UPPER ARG args                                  /* make upper */
 RETURN args                                /* and return to caller */
****** **************************** Bottom of Data ****************************
```

Figure 3.6.3 The RCXTEST exec as formatted by the RCXF macro

The RCXF macro formatted the RCXTEST exec as follows:

- A left margin was established at column 2
- Every IF/SELECT/DO construct was indented 3 blank positions to the right
- The right margin was fixed at column 72
- Comments were right justified at the right margin
- Capitalization was enabled and the following formatting was subsequently applied:
    - The names of all REXX variables were converted to lower case letters
    - All labels and non built-in functions have their first letter capitalized
    - All REXX keywords were converted to upper case
- No Cross Reference was generated
- No diagnostics are in effect

Chapter 4

AcceleREXX Batch Procedures

This chapter discusses a set of batch procedures you can use instead of -- or in addition to -- the AcceleREXX compile dialogs described in Chapter 3. The three AcceleREXX procedures supplied in the RLXCNTL library include the following:

RCXP Invoke the AcceleREXX precompiler to combine several REXX execs into a single source module

RCXC Compile a single (or combined) REXX source module using AcceleREXX

RCXLKED Link-edit one or more previously compiled REXX execs into a single, composite load module

You can copy these RLXCNTL members into one of the catalogued procedure libraries defined at your installation or place these procedures instream within your JCL. In the latter case you will need to terminate the procedure with a JCL '// PEND' statement.

Before using these procedures, we recommend that you review and edit them to specify the names of the RLX system datasets (of which AcceleREXX is a part) defined at installation time. This way, there will be fewer parameters to specify when you invoke these procedures.

4.1 RCXP - Invoke the AcceleREXX precompiler

The RCXP procedure illustrated in Figure 4.1 invokes the AcceleREXX precompiler. This procedure combines multiple REXX execs into a single REXX source module which can then be compiled and link-edited via the RCXC and RCXLKED procedures, respectively.

The numbers in the figure's right margin correspond to the numbered, annotating paragraphs below. These paragraphs describe the functions of the DDnames in the RCXP procedure.

(1) STEPLIB — Identifies the RLXLOAD library which contains the RCXP program

(2) RCXIN — Specifies the name of the library which contains the REXX exec(s) to be precompiled

(3) RCXOUT — Identifies the dataset to which the preprocessed and merged source exec will be written.

(4) SYSTSPRT — AcceleREXX precompiler messages are written to this file

(5) RCXPARMS — The parameters that direct the precompiler are supplied through the file defined with the RCXPARMS DD statement. AcceleREXX precompiler parameters include the following:

`NAME(MAINEXEC)`	Name of the merged exec produced by the precompiler
`LIST(YES/NO)`	Governs whether a listing of the merged exec will be produced
`XREF(YES/NO)`	Governs whether a combined module cross reference will be produced
`PAGESZ(55)`	specifies the maximum size of the printed page of a program listing
`SLINK(EXEC1,EXEC2,.)`	lists all the execs that should be merged into a single REXX source exec

```
//*-----------------------------------------------------------------------
//*  (C) COPYRIGHT RELATIONAL ARCHITECTS INTL.
//*  LICENSED MATERIAL - PROGRAM PROPERTY RELATIONAL ARCHITECTS INTL
//*-----------------------------------------------------------------------
//RCXP    PROC RLXLOAD='RLX.Tvrm.RLXLOAD', ** RLX SYSTEM LOADLIB
//             RCXIN='USER.EXEC'            ** SOURCE EXEC LIBRARY
//RCXP    EXEC PGM=RCXP,REGION=2M,COND=(4,LT)
//STEPLIB  DD  DSN=&RLXLOAD,DISP=SHR                             (1)
//RCXIN    DD  DSN=&RCXIN,DISP=SHR                               (2)
//RCXOUT   DD  DSN=&RSOURCE,UNIT=SYSDA,SPACE=(CYL,(10,1)),       (3)
//             DCB=(LRECL=255,RECFM=VB,BLKSIZE=1299),
//             DISP=(NEW,PASS)            /CONCATENATED REXX SOURCE
//SYSTSPRT DD  SYSOUT=*,DCB=(LRECL=121,BLKSIZE=1210,RECFM=FB)    (4)
//SYSTSIN  DD  DUMMY
//RCXPARMS DD  DUMMY                                             (5)
```

Figure 4.1 RCXP catalogued procedure

Figure 4.2 illustrates how to invoke the RCXP catalogued procedure to produce a merged, composite REXX source module. The numbers in the figure's right margin correspond to the numbered, annotating paragraphs that follow:

```
//JOB     JOB  (PARMS)
//STEP1   EXEC RCXP
//RCXPARMS DD  *                    (1)
NAME(EXAMPLE)                       (2)
SLINK(EXEC1,EXEC2,EXEC3)            (3)
/*
```

Figure 4.2 Sample JCL to invoke the RCXP catalogued procedure

(1) AcceleREXX precompiler parameters are supplied through the RCXPARMS DD statement.

(2) The merged exec produced by the precompiler will be named EXAMPLE

(3) The three source execs to be combined are named EXEC1, EXEC2 and EXEC3

4.2 RCXC - Invoke the AcceleREXX compiler

The RCXC procedure illustrated in Figure 4.3 invokes AcceleREXX to compile your REXX exec.

```
//*-------------------------------------------------------------------
//*  (C) COPYRIGHT RELATIONAL ARCHIRECTS INTL.
//*  LICENSED MATERIAL - PROGRAM PROPERTY RELATIONAL ARCHITECTS INTL
//*-------------------------------------------------------------------
//*  ACCELEREXX - COMPILE SINGLE REXX EXEC
//*-------------------------------------------------------------------
//RCXC    PROC RLXLOAD='RLX.Tvrm.RLXLOAD', ** RLX SYSTEM LOADLIB
//             RCXIN='USER.EXEC',           ** SOURCE EXEC LIBRARY
//             RCXOBJ='USER.OBJECT',        ** USER OBJECT MODULE
//             EXEC=TEMPEXEC,               ** EXEC TO COMPILE
//RCXC    EXEC PGM=RCXC,COND=(4,LT),REGION=2M
//STEPLIB  DD  DSN=&RLXLOAD,DISP=SHR
//RCXIN    DD  DSN=&RCXIN,DISP=SHR
//RCXOBJ   DD  DSN=&RCXOBJ(&EXEC),DISP=SHR
//RCXPRINT DD  SYSOUT=*,DCB=(LRECL=121,BLKSIZE=1210,RECFM=FBA)
//SYSTSPRT DD  SYSOUT=*,DCB=(LRECL=121,BLKSIZE=1210,RECFM=FB)
//SYSPRINT DD  SYSOUT=*
//SYSTSIN  DD  DUMMY
```

Figure 4.3 RCXC catalogued procedure

The RCXC procedure uses the following files:

STEPLIB	identifies the RLXLOAD library which contains the RCXC program
RCXIN	identifies the library which contains the REXX exec to be compiled
RCXOBJ	defines the dataset to which the compiled REXX exec will be written in object code format
RCXPRINT	defines the AcceleREXX listing file
SYSTSPRT	specifies the listing file for RCXC compiler messages
RCXPARMS	defines the dataset which supplies AcceleREXX compile parameters. The AcceleREXX compile parameters are described in detail in Chapter 1.

Figure 4.4 illustrates how to invoke the RCXC catalogued procedure to compile a REXX exec. The numbers in the figure's right margin correspond to the numbered, annotating paragraphs that follow:

```
//JOB     JOB  (PARMS)
//STEP1   EXEC RCXC
//RCXPARMS  DD *                                  (1)
CSECT(EXAMPLE)                                    (2)
INITHCE(RLX)                                      (3)
```

Figure 4.4 Sample JCL to invoke RCXC cataloged procedure

(1) AcceleREXX compiler parameters are supplied through the RCXPARMS DD statement.

(2) The name of the object module produced by the compiler will be named EXAMPLE

(3) The initial REXX host command environment when the compiled REXX procedure begins execution will be RLX. This is specified through the INITHCE parameter.

4.3 RCXLKED - Link-edit the compiled REXX Application

The RCXLKED procedure illustrated in Figure 4.5 invokes the linkage editor to produce an executable load module from one or more compiled REXX object modules.

The RCXLKED procedure consists of these three steps:

- The step named LMDDTS produces a date and time stamp that the linkage editor will include in the compiled REXX load module.
- The step named RCSGRCP generates the control cards that direct the processing of the linkage editor.
- Lastly, the step named LKED actually invokes the linkage editor.

```
//*-----------------------------------------------------------------------
//*  (C) COPYRIGHT RELATIONAL ARCHITECTS INTL.
//*  LICENSED MATERIAL - PROGRAM PROPERTY RELATIONAL ARCHITECTS INTL
//*-----------------------------------------------------------------------
//* ACCELEREXX - LINKAGE EDIT ONE OR SEVERAL COMPILED REXX EXEC(S)
//*-----------------------------------------------------------------------
//RCXLKED PROC RLXLOAD='RLX.Tvrm.RLXLOAD', ** RLX SYSTEM LOADLIB
//             RLXNCAL='RLX.Tvrm.RLXNCAL', ** RLX SYSTEM NCAL LIB
//             RCXOBJ='USER.RCXOBJ',         ** USER OBJECT MODULE
//             RCXLOAD='USER.LOAD',          ** USER LOAD LIBRARY
//             CSECT=MAINEXEC,               ** NAME OF ENTRY POINT
//             LMOD=LMODNAME                 ** NAME OF LOAD MODULE
//*-----------------------------------------------------------------------
//*        PRODUCE DATE AND TIME STAMP TO BE STORED
//*        IN THE LOAD MODULE'S PDS DIRECTORY ENTRY
//*-----------------------------------------------------------------------
//LMDDTS   EXEC PGM=LMDDTS,REGION=128K,COND=(4,LT)
//STEPLIB  DD  DSN=&RLXLOAD,DISP=SHR
//SYSLIN   DD  DSN=&LMDDTS,UNIT=SYSDA,SPACE=(10,(200,50)),
//             DISP=(NEW,PASS)
//*-----------------------------------------------------------------------
//*        CREATE INPUT FOR LINKAGE EDITOR
//*-----------------------------------------------------------------------
//RCSGRCP EXEC PGM=RCSGLKI2,PARM='&CSECT,&LMOD',
//             REGION=2M,COND=(4,LT)
//STEPLIB  DD  DSN=&RLXLOAD,DISP=SHR
//SYSOUT   DD  DSN=&LKEDIN,UNIT=SYSDA,SPACE=(TRK,(1,1)),
//             DISP=(NEW,PASS),DCB=BLKSIZE=80
//SYSTSPRT DD  SYSOUT=*
//*-----------------------------------------------------------------------
//*       LINKAGE EDIT ONE OR SEVERAL COMPILED REXX EXEC(S)
//*-----------------------------------------------------------------------
//LKED    EXEC PGM=IEWL,REGION=2M,COND=(4,LT),
//        PARM='LIST,XREF,REUS,RENT,REFR,SIZE=(512000,256000)'
//RLXSYS   DD  DSN=&RLXNCAL,DISP=SHR
//SYSLIB   DD  DSN=&RCXOBJ,DISP=SHR
//SYSLMOD  DD  DSN=&RCXLOAD,DISP=SHR
//SYSUT1   DD  DSN=&SYSUT1,UNIT=SYSDA,SPACE=(1024,(50,20))
//SYSPRINT DD  SYSOUT=*,DCB=(RECFM=FB,LRECL=121,BLKSIZE=1210)
//SYSLIN   DD  DSN=&LMDDTS,DISP=(OLD,DELETE)   /* DATE AND TIME STAMP
//         DD  DSN=&LKEDIN,DISP=(OLD,DELETE)   /* CONTROL CARDS
```

Figure 4.5 RCXLKED catalogued procedure

Figure 4.6 illustrates how to invoke the RCXLKED catalogued procedure to link-edit several compiled REXX procedures into a single executable load module. The numbers in the figure's right margin correspond to the numbered, annotating paragraphs that follow.

```
//JOB3    JOB  (PARMS)
//STEP1   EXEC RCXLKED,                              (1)
//             CSECT=EXAMPLE,                        (2)
/              LMOD=EXAMPLE                          (3)
```

Figure 4.6 Sample JCL to invoke RCXLKED catalogued procedure

(1) RCXLKED is invoked as a catalogued procedure. RCXLKED can also be included instream with the JCL.

(2) The procedure's CSECT parameter specifies the name of the main (or only) routine that should receive control when the compiled REXX application begins to execute.

(3) The procedure's LMOD parameter specifies the name to be assigned to the link-edited load module.

4.4 Illustrative AcceleREXX Compile / Link-Edit Jobstream

Figure 4.7 provides a complete example of an AcceleREXX jobstream that compiles and link-edits the REXX exec named EXAMPLE. This JCL produces a compiled REXX load module named EXAMPLE that will execute in lieu of the interpretive REXX exec named EXAMPLE.

```
//RLXCOMPL  JOB (...)
//STEP1   EXEC RCXC                                     (1)
//RCXPARMS DD  *                                        (2)
CSECT(EXAMPLE)
//STEP2   EXEC RCXLKED,                                 (3)
//             CSECT=EXAMPLE,                           (4)
//             LMOD=EXAMPLE                             (5)
```

Figure 4.7   AcceleREXX compile / link-edit job

(1) The AcceleREXX compiler is invoked through the catalogued procedure named RCXC.

(2) The CSECT keyword parameter of the RCXPARMS DD statement specifies the name of the compiled REXX object module to be produced by AcceleREXX.

(3) AcceleREXX link-edit processing is invoked through the catalogued procedure named RCXLKED.

(4) The CSECT parameter of the RCXLKED procedure specifies the name of the compiled module that should receive control when the load module begins to execute.

(5) The procedure's LMOD parameter specifies the name to be assigned to the link-edited load module.

4.5 Relink a Load Module Created by AcceleREXX with a new version of RCXAPI (Job RCXJLNK)

If the AcceleREXX stub module named RCXAPI should be changed for any reason, you may need to relink the load modules produced by the AcceleREXX compiler (RCXC). The following JCL can be used to replace the RCXC stub. This job is supplied in the RCXJLNK member of the RLXCNTL library.

```
//jobcard  JOB
//*
// SET RLXLIB=&HLQ.RLXLOAD
// SET USRLIB=&HLQ.RLXLOAD
//*
//*------------------------------------------------------------------
//* Replace RCXAPI with an updated version
//*
//* Note: Edit this member with CAPS ON
//*
//* 1.  Specify a valid jobcard
//* 2.  Set RLXLIB= to the name of the RLXLOAD library
//* 3.  Set USRLIB= to the name of your user load library
//* 4.  Replace ?usrmod? with the name of the user load module to be
//*     created by AcceleREXX.
//* 5.  Submit this JCL and expect CC 0004
//*
//*------------------------------------------------------------------
//RELINK   EXEC PGM=IEWL,REGION=0M,
//         PARM='LIST,XREF,REUS,RENT,SIZE=(512000,256000)'
//REPLACE  DD  DISP=SHR,DSN=&RLXLIB
//SYSLIB   DD  DISP=SHR,DSN=&USRLIB
//SYSLMOD  DD  DISP=SHR,DSN=&USRLIB
//SYSPRINT DD  SYSOUT=*
//SYSLIN   DD  *
  REPLACE RCXAPI
  INCLUDE SYSLIB(?usrmod?)
  INCLUDE REPLACE(RCXAPI)
  ENTRY   ?usrmod?
  MODE    AMODE(31),RMODE(ANY)
  NAME    ?usrmod?(R)
/*
```

Figure 4.5   Job RCXJLNK

Follow the instructions embedded in the JCL to prepare the job for execution. Use the RCXJLNK member when instructed by RAI technical support to replace the RCXAPI CSECT within your AcceleREXX'd load modules with the updated version of RCXAPI.