IBM Compiler and Library for REXX on System z

User's Guide and Reference

Version 1 Release 4

SH19-8160-06

Note

Before using this information and the product it supports, be sure to read the general information under Appendix G, "Notices," on page 277.

Seventh Edition, August 2013

This edition applies to version 1 release 4 of IBM Compiler for REXX on System z (product number 5695-013) and the IBM Library for REXX on System z (product number 5695-014), and to all subsequent releases and modifications until otherwise indicated in new editions.

This edition replaces SH19-8160-04.

Contents

About This Book

This book is intended to help you compile and run programs written in the Restructured EXtended eXecutor (REXX) language. It describes how to use the:

- System z® (referred to as the Compiler)
- IBM Library for REXX on System z (referred to as the Library)
- IBM Library for REXX in REXX/VSE (also referred to as the Library)

It also describes how the Alternate Library can be used by software developers and users of z/OS® or z/VM® who do not have the IBM Library for REXX on System z.

In addition to this, it describes the REXX Stream I/O function package and its usage for z/OS TSO/E.

It is assumed that you are familiar with the REXX language and with the operating system under which you compile or run your programs:

- z/OS or OS/390® with Time Sharing Option Extensions (TSO/E)
- CMS on Virtual Machine/Extended Architecture (VM/XA), Virtual Machine/Enterprise System Architecture (VM/ESA), or z/VM
- Virtual Storage Extended/Enterprise System Architecture (VSE/ESA) with REXX/VSE

Some of the information applies to all systems: z/OS, z/VM, and VSE/ESA. Information that applies to only one system is indicated in the text.

How to Read the Syntax Notation

The notation used to define the command syntax in this book is as follows:

- A symbol (word) in boldface, such as **CEXEC**, denotes a keyword.
- Words in italics, such as *options-list*, denote variables or collections of variables.
- The brackets [and] delimit optional parts of the commands.
- The logical **OR** character | separates choices within brackets.

The notation of syntax diagrams is described in Chapter 14, "How to Read the Syntax Diagrams," on page 129.

How This Book Is Organized

This book is organized as follows:

- Part 1, "Programming Reference Information," on page 1 provides an overview and describes how to invoke the Compiler. It also lists the compiler options and control directives, and explains the differences between the language processed by the Compiler and the language processed by the interpreters.
- Part 2, "Customizing the Compiler and Library," on page 115 contains information for the system programmer about customizing the Compiler and the Library.
- Part 3, "Stream I/O for TSO/E REXX," on page 127 describes the REXX Stream I/O function package and its usage for z/OS TSO/E.
- Part 4, "Messages," on page 151 contains messages and their explanations.

- The **appendixes** contain reference information, such as cataloged procedures.

How to Send Your Comments

Your feedback is important in helping to provide the most accurate and high-quality information. If you have any comments about this book or any other REXX documentation:

- Visit our home page at: `http://www.ibm.com/software/awdtools/rexx/` There you can access the Internet Online Form where you can enter comments and send them.
- z/OS or z/VM customers can also use the IBM Service Center to raise a PMR against:

  ```
  RETAIN queue: RASS,148
  ```

  Specify the program ID of your IBM Compiler and Library for REXX on System z installation:

  ```
  569501303 (Compiler under z/OS)
  569501403 (Library under z/OS)
  569501304 (Compiler under z/VM)
  569501404 (Library under z/VM)
  ```

What's New in Release 4

Here you find an overview of the latest enhancements, changes, and highlights provided in Release 4.

Release 4 introduces new naming conventions:

- IBM Compiler and Library for SAA REXX/370 is now IBM Compiler and Library for REXX on System z
- MVS™ and MVS/ESA are now z/OS
- VM is now z/VM

The IBM Compiler for REXX on System z Release 4 and the IBM Library for REXX on System z Release 4 are enhanced to meet your continued requirement for REXX support on z/VM and z/OS. They:

- Include all service since Release 3
- Provide more efficient system management
- Support improved error checking
- Provide more information during debugging
- Improve error checking during compilation
- Allow increased flexibility

IBM Compiler for REXX on System z

For z/VM and z/OS environments:

- Error checking of built-in functions is improved.
- The Compiler now issues messages even if there is no message repository installed under z/VM and z/OS.
- A new date conversion function is provided that allows you to specify the input format, the input date, and the output format.
- Debugging capabilities are improved by writing the source code file identifier into the compiled code.
- Host BIF plausibility checks are provided.
- Date conversion and separation characters are supported.
- The input file ID is written into the CEXEC.
- REXX utilities have been enhanced, such as REXXF for both products.
- New compiler directives provide improved systems management and debugging:
  - %STUB to include a named stub already at compilation time into the object output file. It simplifies link-editing of object modules for z/OS.
  - %SYSTIME to retrieve the compilation time.
  - %SYSDATE to retrieve the compilation date.
  - %TESTHALT to place testhalt hooks at specific places.
- You can generate link-edited modules supporting nearly all of the previous stub conventions with only one stub named MULTI.
- The following compiler options have been added or enhanced:

- SLINE(A) Automatic SLINE includes the source program depending on the compiler option TRACE and ALTERNATE, and/or the SOURCELINE built-in function (BIF) that are used
- NO/OPTIMIZE bypasses the compiler optimization step
- FORMAT(C) formats the error messages and cross references with column numbers.
- LIBLEVEL(*n*) restricts the usage of REXX language constructs to a specific library level
- XREF has been enhanced to provide more details in the compiler listing, such as information about exposed and dropped variables, variables without assignment, and optimization stoppers.

For z/VM environments only:

- Hardcoded messages are provided when the message repository is not available.
- The new compiler option OLDDATE(CIOP) sets the file date of the CEXEC, IEXEC, OBJECT, PRINT output file to the file date of the source. It simplifies maintenance, because it allows you to set the date/time stamp of various output files to the date/time stamp of the source file.
- Sequence numbers are supported.

For OS/390 and z/OS environments only:

- The new compiler option DDNAMES allows you to redefine the standard ddnames by reading a control file.
- The interface to ISPF services is simplified.
- The portability with Stream I/O for TSO/E REXX function package is improved.

IBM Library for REXX on System z

For OS/390 and z/OS environments only:

- Interfaces to operating system functions are simplified with a multi-purpose STUB.
- Sample code for interfacing with other programming languages such as C, Cobol, PL/I, or Assembler is available.
- Stubs are enhanced to make the LANG (CREX) parameter obsolete.

Part 1. Programming Reference Information

Chapter 1. Overview

This chapter provides an overview of the features and functions of the:

- IBM Compiler for REXX on System z
- IBM Library for REXX on System z
- Alternate Library

The Compiler translates REXX source programs into compiled programs. The Library contains routines that are called by compiled programs at runtime. The Alternate Library contains a language processor that transforms the compiled programs and runs them with the interpreter. It can be used by z/OS and z/VM users who do not have the IBM Library for REXX on System z to run compiled programs.

The Compiler and Library run on z/OS systems with TSO/E, and under CMS on VM/XA, VM/ESA, and z/VM systems. The IBM Library for REXX in REXX/VSE runs under VSE/ESA.

You may prefer to leave some programs uncompiled. This would be a good choice for simple programs that are not used frequently. An example is a program that renames all the files in a library in accordance with a new naming convention, and then never needs to be run again.

Background information about compilers

Instructions written in any high-level language, such as REXX, must be prepared for execution. The two types of programs that can perform this task are:

- An *interpreter*, which parses and executes an instruction before it parses and executes the next instruction.
- A *compiler*, which translates all the instructions of a program into a machine code program. It can keep the machine code program for later execution. It does not execute the program.

The *input* to a compiler is the source program that you write.

The *output* from a compiler is the *compiled program* and the *listing*.

The *process* of translating a source program into a compiled program is known as *compilation*.

The Level of REXX Supported by the Compiler

The Compiler supports REXX language level 3.48 on z/OS in TSO/E Version 2 Release 4, CMS in VM/ESA releases earlier than Release 2.1, and on VSE/ESA in REXX/VSE Version 1 Release 1. On CMS in VM/ESA Release 2.1 and subsequent releases, the language level supported is 4.02.

Most of your existing REXX programs should compile without error and should give the same runtime results without modification.

Most of the language features that are new in VM/ESA Release 2 and TSO/E Version 2 Release 4 are available when running compiled programs, even when

they are not accepted by the interpreters. See Chapter 8, "Language Differences between the Compiler and the Interpreters," on page 91 for details.

Using the Compiler in Program Development

One effective way of using the Compiler to develop REXX programs is the following:

1. Compile the program with the TRACE and NOTESTHALT compiler options and without the %TESTHALT control directive. The SLINE or SLINE(AUTO) compiler option is required. This step performs comprehensive error checking and produces an output that can be traced.
2. Debug the program using the output of the previous step.
3. Compile the program with the NOTRACE compiler option and, if required, the TESTHALT compiler option and %TESTHALT control directive.

Background information about error checking

The compiler scans an entire program for such errors as incorrect instructions and variable names, even in parts of a program that are not used when the program is run. By contrast, the interpreter stops as soon as it detects an error. It does not detect syntax errors in parts of a program that are not used during a particular invocation.

The compiler, however, cannot detect errors that arise at runtime. Consider this assignment:

```
averagescore = totalscore/numberofgames
```

This is valid during compilation, but could give an error at runtime. For example, if the variable `numberofgames` is assigned the value zero, an arithmetic error occurs.

Forms and Uses of Output

The Compiler can produce output in the following forms:

- **Compiled EXECs:** These behave exactly like interpreted REXX programs. They are invoked the same way by the system's EXEC handler, and the search sequence is the same. The easiest way of replacing interpreted programs with compiled programs is by producing compiled EXECs. Users need not know whether the REXX programs they use are compiled EXECs or interpretable programs. Compiled EXECs can be sent to VSE/ESA to be run there. In this book, compiled EXECs are often referred to as CEXEC output.
- **Object modules under z/OS or TEXT files under z/VM:** These must be transformed into executable form (load modules) before they can be used. Load modules and MODULE files are invoked the same way as load modules derived from other compilers, and the same search sequence applies. However, the search sequence is different from that of interpreted REXX programs and compiled EXECs. These load modules can be used as commands and as parts of REXX function packages. Object modules or MODULE files can be sent to VSE/ESA to build phases.
- **IEXEC output:** This output contains the expanded source of the REXX program being compiled. Expanded means that the main program and all the parts included at compilation time by means of the %INCLUDE directive are contained in the IEXEC output. Only the text within the specified margins is contained in the IEXEC output. Note, however, that the default setting of MARGINS includes the entire text in the input records.

You can produce all forms of output in one compilation. Compiled EXECs and object modules contain the compiled code for the program:

- **Generate load modules from object modules:** Under z/OS, object modules can be used to generate load modules. You need to link-edit the object modules with stubs before you can run them or before you can link them with other programs. See "Object Modules (z/OS)" on page 72 and Appendix A, "Interface for Object Modules (z/OS)," on page 205 for more information.
- **Generate load modules from TEXT files:** Under z/VM, a TEXT file can be processed into a MODULE file. The MODULE file can be invoked like any other z/VM module. See "TEXT Files (z/VM)" on page 75 and Appendix B, "Interface for TEXT Files (z/VM)," on page 221 for more information.
- **Build phases from object modules:** Under VSE/ESA, object modules can be used to build phases. You need to combine the object modules with the appropriate stub, before you can use them. See "Object Modules (VSE/ESA)" on page 77 and Appendix C, "Interface for Object Modules (VSE/ESA)," on page 225 for more information.
- **Link TEXT files to Assembler programs:** A TEXT file can be linked to an Assembler program. See "TEXT Files (z/VM)" on page 75 for more information.

Porting and Running Compiled REXX Programs

A REXX program compiled under z/OS can run under z/VM. Similarly, a REXX program compiled under z/VM can run under z/OS. A REXX program compiled under z/OS or z/VM can run under VSE/ESA if REXX/VSE is installed.

Note:

1. Machine mode 370 is no longer supported.
2. You must only recompile programs that were compiled with the z/VM REXX Compiler or with the IBM Compiler for SAA REXX/370 Release 2 if they contain the CONDENSE compiler option. Otherwise you need not recompile existing programs to run with the latest release level of REXX.
3. See also Chapter 7, "Converting CEXEC Output between Operating Systems," on page 85 for more information.

If you compiled your program under z/OS using:

- The CEXEC option, and want to run it under:
  - z/OS, see "CEXEC" on page 20
  - MVS OpenEdition, see "Converting from z/OS to MVS OpenEdition" on page 85
  - z/VM , see "Converting from z/OS to z/VM" on page 85
  - VSE/ESA, see "Converting from z/OS to VSE/ESA" on page 86
- The OBJECT option, and want to run it under:
  - z/OS, see "OBJECT" on page 30
  - z/VM, transfer the OBJECT output to z/VM and generate a module; see "TEXT Files (z/VM)" on page 75
  - VSE/ESA, transfer the OBJECT output to VSE/ESA and generate a phase; see "Object Modules (VSE/ESA)" on page 77

If you compiled your program under z/VM using:

- The CEXEC option, and want to run it under:
  - z/OS, see "Converting from z/VM to z/OS" on page 86

  - z/VM, see "CEXEC" on page 20
  - VSE/ESA, see "Converting from z/VM to VSE/ESA" on page 87
- The OBJECT option, and want to run it under:
  - z/OS, transfer the OBJECT output to z/OS and generate an object module; see "Object Modules (z/OS)" on page 72
  - z/VM, see "OBJECT" on page 30
  - VSE/ESA, transfer the OBJECT output to VSE/ESA and generate a phase; see "Object Modules (VSE/ESA)" on page 77

Calling and Linking REXX Programs

Compiled REXX programs can interface with other programs in the same ways as interpreted REXX programs. For details, refer to *TSO/E REXX/MVS: Reference*, *IBM VSE/ESA REXX/VSE: Reference*, or to the corresponding z/VM documentation.

Running above 16 Megabytes in Virtual Storage

Under z/OS systems and under z/VM systems running in XA mode, the Compiler, the Library, and the compiled REXX programs can run above 16 megabytes in virtual storage. Under VSE/ESA, the compiled REXX programs can run above 16 megabytes in virtual storage. This requires no user action. Data used during a compilation or by a running program can reside above 16 megabytes in virtual storage.

SAA Compliance

The Systems Application Architecture® (SAA) definitions of software interfaces, conventions, and protocols provide a framework for designing and developing applications that are consistent within and across several operating systems.

The SAA REXX interface is supported by the interpreters under TSO/E, CMS, and VSE/ESA, and can be used in any of these environments. Users whose programs run under TSO/E, CMS, or VSE/ESA can use the language extensions provided by these interpreters. If you plan to run your programs in other environments, however, some restrictions may apply. For details of the restrictions, consult the *Systems Application Architecture Common Programming Interface REXX Level 2 Reference*.

To help you to write programs for use in all SAA environments, the Compiler can optionally check for SAA compliance. With this option in effect, a warning message is issued for each non-SAA item found in a program.

Choosing the National Language

The Compiler and Library provide optional support for languages other than American English. The language you select is used for:
- Messages
- Some of the constant text in the compiler listing, such as the page headings
- Help panels
- Compiler invocation panels under z/OS

For information on selecting a national language:
- Under z/OS, see the descriptions of:
  - The SETLANG function in the *TSO/E REXX/MVS: Reference*

  - The PLANGUAGE and SLANGUAGE operands of the PROFILE command in the *TSO/E Command Reference*
- Under z/VM, see the description of the SET LANGUAGE command in the command reference for your system.
- Under VSE/ESA, only English is supported when running the IBM Library for REXX in REXX/VSE.

Alternate Library Overview

The Alternate Library enables users who do not have the Library installed to run compiled REXX programs. It contains a language processor that transforms the compiled programs and runs them with the interpreter, which is part of TSO/E and CMS.

Software developers can distribute the Alternate Library, free of charge, with their compiled REXX programs. If their customer:

- Has the Library installed, the programs run as compiled REXX programs
- Installs the Alternate Library, the programs are interpreted

Distributing the compiled REXX program, without the source, offers the following advantages:

- Maintenance of the program is simplified, because the code cannot be modified inadvertently.
- Compiled programs can be shipped in load module format and used to create function packages, even for users who do not have the Library.

Note:

1. With the Alternate Library, the performance of compiled REXX programs is similar to that of interpreted programs. The performance advantages of compiled REXX are available only when the Library is installed.
2. To work with the Alternate Library, you must set the ALTERNATE and SLINE compiler options.

Stream I/O for TSO/E REXX Function Package

This function package is a collection of I/O functions that follow the stream I/O concept. It extends and enhances the I/O capabilities of REXX for TSO/E, and shields the complexity of z/OS data set I/O to some degree. Further, the use of stream I/O functions provides for easier coding syntax and leads to better portability of REXX programs among different operating system platforms. The stream I/O concept is introduced in Chapter 16, “Understanding the Stream I/O Concept,” on page 133.

This function package can be used with TSO/E REXX on z/OS, OS/390, and MVS systems that provide the MVS Name/Token Services, which are required to hook the function package into an existing TSO/E REXX installation. It is a loadable file that contains multiple object files bound together. Before its functions can be accessed and executed, the function package must be properly integrated into TSO/E REXX. For more information refer to Part 3, “Stream I/O for TSO/E REXX,” on page 127.

Note: It is assumed that you are familiar with the REXX language, the TSO/E environment, and the logical organization of data sets in the z/OS environment.

Alias Definitions and Member Names under z/OS

The following table provides an overview of alias definitions and member names. It also identifies the corresponding data sets.

Table 1. Alias and Member Names for Use with the Compiler

Alias	Member	Data Set
Procedures		
REXXC	FANCMC	*prefix*.SFANPRC
REXXCG	FANCMCG	*prefix*.SFANPRC
REXXCL	FANCMCL	*prefix*.SFANPRC
REXXCLG	FANCMCLG	*prefix*.SFANPRC
REXXOEC	FANCMOEC	*prefix*.SFANPRC
Commands		
REXXC	FANC	*prefix*.SFANCMD
REXXF	FANCMF	*prefix*.SFANCMD
REXXV	FANV	*prefix*.SFANCMD

Table 2. Alias and Member Names for Use with the Library

Alias	Member	Data Set
Procedure		
REXXL	EAGL	*prefix*.SEAGPRC
Commands		
REXXF	EAGCMF	*prefix*.SEAGCMD
REXXL	EAGCML	*prefix*.SEAGCMD
REXXQ	EAGQRLIB	*prefix*.SEAGCMD
REXXV	EAGV	*prefix*.SEAGCMD

Chapter 2. Invoking the Compiler

This chapter describes in detail the various ways of invoking the IBM Compiler for REXX on System z under z/OS and under z/VM.

To use the Compiler, you supply:
- A source program.
- Compiler options. These control aspects of the Compiler's processing.

Depending on the options used, the Compiler produces the following types of output:
- The compiled program, which can be a compiled EXEC, an object module for z/OS or VSE/ESA, or a TEXT file for z/VM
- The compiler listing, which may include a source listing, messages, and a cross-reference listing
- Messages on the terminal
- IEXEC output, which can be interpreted

If you compile a program that was previously only interpreted, you may find that, at runtime, its behavior is not identical. This is because there are some differences between the language that is processed by the Compiler and by the interpreters. For more information refer to Chapter 8, "Language Differences between the Compiler and the Interpreters," on page 91.

Invoking the Compiler under z/OS

z/OS users can invoke the Compiler by using:
- REXXC—see "Invoking the Compiler with the REXXC (FANC) EXEC"
- ISPF compiler invocation panels—see "Invoking the Compiler with ISPF Panels" on page 11
- JCL statements—see "Invoking the Compiler with JCL Statements" on page 13
- Cataloged procedures—see "Invoking the Compiler with Cataloged Procedures" on page 13
- The 'REXXCOMP' command—see "Invoking the Compiler with the 'REXXCOMP' Command" on page 13

Invoking the Compiler with the REXXC (FANC) EXEC

You can invoke the Compiler in a TSO/E environment by using the compiler invocation EXEC: REXXC (FANC). (See also "Alias Definitions and Member Names under z/OS" on page 8.) REXXC is supplied with the Compiler to compile REXX source programs. It must run in a TSO/E address space.

To start the EXEC, enter the REXXC command in the following format:

REXXC *source* [*options-list*]

where:

source Specifies the data set containing the REXX source program.

REXXC allocates the specified or default output data sets if they do not already exist. It uses defaults for data set attributes and allocation values that are described in "Customizing the REXXC EXEC" on page 117. For

information about how the names of the default data sets are derived, see "Derived Default Data Set Names."

REXXC checks the data set organization for each output. It ends with an error rather than overwriting a partitioned data set with a sequential data set of the same name, and vice versa.

options-list
Any of the compiler options that are described in "Compiler Options" on page 19. They can be specified in any order.

You can use the following options to explicitly specify where the Compiler output is to be stored:

- "BASE" on page 19
- "CEXEC" on page 20
- "DUMP" on page 25
- "IEXEC" on page 27
- "OBJECT" on page 30
- "PRINT" on page 34

Derived Default Data Set Names

If you do not specify data set names, REXXC derives default names for output data sets. The following tables show the default data set names that may be created by the REXXC command.

Table 3 shows the defaults that are derived from the specified source (or the **BASE** option's value, if specified). The source program was either a member of a partitioned data set or a sequential data set.

Table 3. Defaults that Are Derived from the Specified Source or the BASE option

Option	Partitioned Data Set pref.cccc.qual(member)	Sequential Data Set pref.cccc.qual
CEXEC	upref.cccc.CEXEC(member)	upref.cccc.qual.CEXEC
IEXEC	upref.cccc.IEXEC(member)	upref.cccc.qual.IEXEC
OBJECT	upref.cccc.OBJ(member)	upref.cccc.qual.OBJ
PRINT	upref.cccc.member.LIST	upref.cccc.qual.LIST
DUMP	upref.cccc.member.DUMP	upref.cccc.qual.DUMP

The default name for the *load-data-set-name* parameter of the OBJECT option is derived from the name of the data set that contains the output from the OBJECT option. This can be either a member of a partitioned data set or a sequential data set:

Partitioned Data Set *pref.cccc.qual***(member)**
upref.cccc.LOAD(*csect*)

Sequential Data Set *pref.cccc.qual*
upref.cccc.qual.LOAD(*csect*)

where:

pref
Represents the prefix.

cccc
Represents one or several data-set name qualifiers.

qual
Represents the last level qualifier.

csect
Represents the name the Compiler puts in the ESD from the OBJECT output. See Chapter 6, "Using Object Modules and TEXT Files," on page 71 for more information on *csect*.

upref
Represents the user's default prefix (as set by the PROFILE PREFIX command). It is used for the output data sets.

An Example

For example, you may have stored an interpretable REXX program named **SAMPLE** in the data set *pref*.REXX.EXEC, which is allocated to the ddname **SYSPROC**.

You can generate a compiled REXX EXEC by allocating the data set *pref*.REXX.CEXEC to the ddname **SYSEXEC** and entering the following command:

```
rexxc rexx.exec(sample) cexec(rexx.cexec(sample)) print(*)
```

In this command, **print(*)** is an option that writes the listing to ddname **SYSTERM**. Installation defaults are used for options that you do not specify.

You can run a compiled program or an interpreted EXEC, by entering its name as a command. However, your compiled program must be in the search sequence (see *TSO/E REXX/MVS Reference* for information on search sequence). For example, by entering: **sample**

Invoking the Compiler with ISPF Panels

Under ISPF, you can invoke the Compiler from the Foreground REXX Compilation panel or the Batch REXX Compilation panel. The panels, Figure 1 on page 12 and Figure 2 on page 12, are similar to those for other high-level language compilers.

Because the ISPF panels use the REXXC EXEC to invoke the Compiler, you can specify the enhanced options as well as all other Compiler options.

To use the **Foreground REXX Compilation** panel:

1. Select **FOREGROUND** on the ISPF/PDF Primary Option Menu.
2. Select REXX Compiler.
3. Enter the appropriate data set names with the extensions as described in the online help and the compiler options listed in "Compiler Options" on page 19.

```
---------------------  FOREGROUND REXX COMPILATION  ----------------------
 COMMAND ===>

 ISPF LIBRARY:
    PROJECT ===> TEST
    GROUP   ===> LIB1      ===> LIB2      ===> LIB3      ===>
    TYPE    ===> REXX
    MEMBER  ===>               (Blank or pattern for member selection list)

 OTHER PARTITIONED OR SEQUENTIAL DATA SET:
    DATA SET NAME  ===>

 LIST ID ===>

 COMPILER OPTIONS:
         ===>
         ===>

 INCLUDE DATA SETS:
         ===>
         ===>
         ===>
```

Figure 1. Foreground REXX Compilation Panel (Panel ID: FANFP14)

Note: This panel may have been customized by your system administrator.

To use the **Batch REXX Compilation** panel:

1. Select **BATCH** on the ISPF/PDF Primary Option Menu.
2. Select REXX Compiler.
3. Enter the appropriate data set names with the extensions as described in the online help and the compiler options listed in "Compiler Options" on page 19.

```
------------------------  BATCH REXX COMPILATION  ------------------------
 COMMAND ===>

 ISPF LIBRARY:
    PROJECT ===> TEST
    GROUP   ===> LIB1      ===> LIB2      ===> LIB3      ===>
    TYPE    ===> REXX
    MEMBER  ===>                  (Blank or pattern for member selection list)

 OTHER PARTITIONED OR SEQUENTIAL DATA SET:
    DATA SET NAME  ===>

 LIST ID      ===>                 (Blank for hardcopy listing)
 SYSOUT CLASS ===> *               (If hardcopy requested)

 COMPILER OPTIONS:
         ===>
         ===>

 INCLUDE DATA SETS:
         ===>
         ===>
         ===>
```

Figure 2. Batch REXX Compilation Panel (Panel ID: FANJP14)

Note: This panel may have been customized by your system administrator.

The source program you specify must be stored in an ISPF library, a partitioned data set, or a sequential data set. If you do not specify a member name of a library or partitioned data set, a list is displayed from which you can select the member to be compiled.

The default output data set names are the same as those described for the REXXC EXEC (see "Derived Default Data Set Names" on page 10) with the following additions:

- If the **PRINT** option is not specified, the compiler listing is named *upref.mmm*.LIST, where *upref* is the user's default data set prefix and *mmm* is the specified list identifier (**LIST ID**) or the member name of the source program.
- The first group is used for the default output data set names if the source comes from an ISPF library and more than one group is specified. Figure 1 on page 12 and Figure 2 on page 12 show examples of a first group ISPF library name **TEST.LIB1.REXX**.

In contrast to the compilation panels for other languages, not only the compiler options but all REXXC command options can be specified. For example, you can explicitly specify data set names for compiler output, thus overriding the defaults.

Online help is available for the invocation panels.

Invoking the Compiler with JCL Statements

You can invoke the Compiler from a z/OS batch environment by writing and running your own JCL statements or by running the supplied cataloged procedures as described in "Invoking the Compiler with Cataloged Procedures."

The JCL statements that you need are:

- A JOB statement that identifies the start of the job.
- An EXEC statement (`PGM=REXXCOMP`) that identifies the Compiler and the compiler options. Additionally, a **JOBLIB** or **STEPLIB** data definition (DD) statement may be necessary, so that the system can locate the **REXXCOMP** program.
- DD statements that identify both the input and the output data sets that the Compiler requires. These are described in "Standard Data Sets Provided for the Compiler" on page 14.
- A delimiter statement that separates data in the input stream from the JCL statements that follow the data.
- Job entry subsystem (JES) control statements that provide information to the JES.

Invoking the Compiler with Cataloged Procedures

You can compile a REXX program in a z/OS batch environment by using a cataloged procedure that is invoked by an EXEC statement in your job. The main advantage of using cataloged procedures is that they can include most of the JCL statements that you would otherwise have to write yourself. This is useful for sets of JCL statements that you use regularly.

These cataloged procedures are listed in Appendix D, "The z/OS Cataloged Procedures Supplied by IBM," on page 231.

Note: Your system administrator may have customized the cataloged procedures on your system.

Invoking the Compiler with the 'REXXCOMP' Command

You can also invoke the Compiler in the foreground using **ADDRESS LINKMVS 'REXXCOMP'**. In this case, ensure that an input data set is allocated under SYSIN. If there is no data set, TSO displays the prompt mode. To exit the prompt mode, specify **/***.

Standard Data Sets Provided for the Compiler

The Compiler requires some standard input and output data sets. The number of data sets depends on the compiler options specified. You must define these data sets in DD statements with the ddnames shown in Table 4. The SYSIN DD statement is always required. DD statements corresponding to %INCLUDE directives are also required. Their data control block (DCB) requirements correspond to those of SYSIN in the following table.

Note:

1. Under SYSIN a data set name must be defined.
2. All data sets of the SYSIN concatenation are compiled in one step. These data sets can be sequential (PS), a PDS with member specification, or both.
3. A PDS without member specification is not supported.
4. To perform the compile step with ddnames that are selected by the user instead of using the standard names, the Compiler must read the renaming table from a data set defined by the standard name FANDDN. For more information refer to "DDNAMES" on page 23.

Table 4. Data Sets Required by the Compiler (z/OS)

DDNAME	Record Format RECFM	Record Size LRECL	Contents	Required for Option
FANDDN	F, FB	≤32 760	Input to the Compiler	
	V, VB	≤32 756		
SYSCEXEC	F, FB	≤32 760 and ≥20	Compiled EXEC	CEXEC
	V, VB	≤32 756 and ≥24		
SYSDUMP	FA, FBA	121	Formatted dumps	DUMP
	VA, VBA	125		
SYSIEXEC (refer to "IEXEC" on page 27 for more details.)	F, FB	≤32 760	Expanded source program	IEXEC
	V, VB	≤32 756		
SYSIN	F, FB	≤32 760	Input to the Compiler	
	V, VB	≤32 756		
SYSPRINT	FA, FBA	121	Listing, including messages	PRINT
	VA, VBA	125		
SYSPUNCH	F, FB	80	Object module	OBJECT
SYSLIB	F, FB	≤32 760	Input to the Compiler	
	V, VB	≤32 756		
SYSTERM	F, FB	80 (Recommended)	Errors, error messages, message summary	TERMINAL or for messages of severity T
	FA, FBA	81 (Recommended)		
	V, VB	84 (Recommended)		
	VA, VBA	85 (Recommended)		

Invoking the Compiler under z/VM

z/VM users can invoke the Compiler by using:
- REXXD—see "Invoking the Compiler with REXXD"
- REXXC—see "Invoking the Compiler with the REXXC EXEC" on page 17
- "Batch Jobs" on page 18

Invoking the Compiler with REXXD

A sample compiler invocation dialog, REXXD, is supplied with the Compiler to compile REXX source programs. From this panel, you can invoke the Compiler and perform associated tasks, such as inspecting the listing and editing the source program. The main advantage of using an interactive dialog is that you do not have to remember any commands or options: you are prompted for all the necessary information.

Note: The sample dialog may have been customized by your system administrator. Ask your system administrator what command you should enter to start this dialog if you do not succeed in using REXXD.

Start the dialog as follows:

REXXD [*source-file-identifier*]

where:

source-file-identifier
: Is the file identifier of the source program. If you omit the file identifier, the program last processed with REXXD is used again. You need not fully specify the source file identifier. If you specify only the file name, all accessed disks are searched for a REXX program that has this file name and one of the supported file types (listed in variable $.0ptypes in the file REXXDX XEDIT; see "Customizing the Compiler Invocation Shells" on page 119). Alternatively, the file type could be prefixed according to the rule specified in REXXDX in variable $.0ssft. The selected file identifier appears in the main panel of the dialog. You can change it there if you wish.

An Example

Enter the following command, for example, to invoke the dialog:

rexxd test exec a1

The following panel appears:

```
                                IBM Compiler for REXX on System z, Release 4
Specify a program.                    Licensed Materials -  Property of IBM
Then select an action.              5695-013 (C) Copyright IBM Corp. 1989, 2003
                                                          All rights reserved.
 Program  . . . .  TEST EXEC A1           Output disk: _

Action . . . . . _              Source active              Compiled
                   1  Compile TEST EXEC A1          into TEST CEXEC A1
                   2  Switch (rename) source and compiled exec

                   3  Run active  (source)  program with argument string
                   4  Edit source program
                   5  Inspect compiler listing
                   6  Print source program
                   7  Print compiler listing

                   8  Specify compiler options

Argument string:  ____________________________________________________________


Command ===> ________________________________________________________________
Enter  F1=Help  F2=Filelist  F3=Exit
                                                                  F12=Cancel
```

Figure 3. Main Panel of the Sample Compiler Invocation Dialog

Use the various functions of the dialog as you need them:

- In the field **Program**, type or change the identifier of the program you want to work with.
- In the field **Output disk**, you can specify the disk on which the Compiler output is to be stored.
- To select an action, type its number in the selection field and press the Enter key.
- You can use the default compiler options to begin with.
- Whenever you need further guidance, press the Help key (F1) for online help.

When you start using the Compiler regularly, set up suitable values in the **REXX Compiler Options Specifications** panel, shown in Figure 4 on page 17, and save them for future use. The compiler options are explained in the online help and in "Compiler Options" on page 19.

Setting the Compiler Options

When you select the "Specify compiler options" action you get the following panels that prompt you for the compiler options:

```
                REXX Compiler Options Specifications  1 of 2

Specify which output files you want and their File-IDs    More: +
                              File  identifiers
     Program name             TEST      EXEC      A1
   Y  Compiler listing (Y/N/P) =         LISTING   =
   Y  Compiled EXEC    (Y/N)   =         C*        =
   N  TEXT file        (Y/N)   =         TEXT      =
   N  IEXEC file       (Y/N)   =         I*        =

Specify compiler messages to be issued
   I  FLAG   Minimum severity of messages to be shown (I/W/E/S/T/N)
   N  TERM   Display messages at the terminal (Y/N)
   N  SAA    SAA-compliance checking (Y/N)
   *  LL     LIBLEVEL (*/2/3/4/5/6)

Specify contents of compiler listing
   Y  SOURCE Include source listing (Y/N)
   N  XREF   Include cross-reference listing (Y/S/N)
   N  FORMAT Format with column numbers (Y/N)
   55 LC     Number of lines per page (10-99 or, for no page headings, 0 or N)



Command ===> ____________________________________________________________
Enter  F1=Help  F2=Filelist  F3=Exit  F4=Save  F5=Refresh  F6=Reset F8=Fwd
                                                            F12=Cancel
```

Figure 4. Options Specification Panel (1 of 2)

```
                REXX Compiler Options Specifications  2 of 2

Specify additional compiler options                      More: -

Additional options
  N  SL     Support SOURCELINE built-in function (Y/A/N)
  N  TH     Support HI immediate command (Y/N)
  S  NOC    Error level to suppress compilation (*/W/E/S/T)
  N  COND   Condense compiled program (Y/N)
  N  DL     Include ESD and RLD in TEXT output (Y/N)
  N  ALT    Compiled program supports Alternate Library (Y/N)
  N  TR     Compiled program can be traced (Y/N)
  N    OLDD Apply OLDDATE to Cexec/Print/Object/Iexec (Y/N/(C|P|O|I))
  1    *    MARGINS  Left and right source margins

Special compiler diagnostics
  N    DUMP Produce diagnostic output (0-2047, Y, or N)
  Y    OPT  Optimize compiled program (Y/N)


Command ===> ____________________________________________________________
Enter  F1=Help  F2=Filelist  F3=Exit  F4=Save  F5=Refresh  F6=Reset F7=Bkwd
                                                            F12=Cancel
```

Figure 5. Options Specification Panel (2 of 2)

The current default options are displayed. You can type and optionally save new values in any of the fields. The compiler invocation dialog will use the saved options the next time it is invoked.

Invoking the Compiler with the REXXC EXEC

The compiler invocation EXEC, REXXC, operates in line mode; using it can be quicker than the dialog. For any options that you do not specify, the EXEC uses defaults defined when the Compiler was installed. You may prefer this method if you are an experienced z/VM user.

A sample compiler invocation EXEC, REXXC, is supplied with the Compiler to compile REXX source programs.

Note: Ask your system administrator what command you should enter to start this EXEC if you do not succeed in using the IBM-supplied EXEC.

Enter the command to start the EXEC in the following format:

REXXC *source-file-identifier* **[(***options-list***)]**

where:

source-file-identifier
: Is the file identifier of the source program. You need not fully specify the source file identifier. If the file type is not specified, EXEC is used. If you do not specify the file mode, it defaults according to the CMS search order.

options-list
: Is a list of compiler options to be used, separated by blanks. For details of the options that can be specified, see "Compiler Options" on page 19. The defined defaults are used for any options that you do not specify. See "Setting Up Installation Defaults for the Compiler Options" on page 120 for details.

Batch Jobs

The Compiler can run in a batch machine with the z/VM Batch Facility or with the IBM licensed program VM Batch Facility (Program Number 5664-364). To run the compiler invocation EXEC in batch, use your standard procedure for submitting batch jobs.

Chapter 3. Compiler Options and Control Directives

This chapter describes the compiler options, including the enhanced options for REXXC, and the control directives that are available.

While the Compiler options are specified when the Compiler is invoked, the control directives are defined within your program as part of the REXX code.

Compiler Options

This section describes the functions and syntax of the compiler options, along with their abbreviations and defaults supplied by IBM.

Make sure you separate the options by blanks. The last specification of an option takes precedence.

The compiler options are described in alphabetical order.

ALTERNATE

The ALTERNATE option specifies that at runtime the Alternate Library may be used.

ALTERNATE
Creates a compiled program of CEXEC or OBJECT type that can run both with the Alternate Library and the Library.

The SLINE compiler option must also be specified as described in "SLINE" on page 35.

If the DLINK option is specified, the program can take advantage of directly linked programs only when running with the Library. For programs that run with the Alternate Library, DLINK has no effect; the standard REXX search order is used. See "Creating REXX Programs for Use with the Alternate Library (z/OS, z/VM)" on page 112 for more information.

NOALTERNATE
Creates a compiled program of CEXEC or OBJECT type that will run using the Library. The program cannot run with the Alternate Library.

Abbreviations:
ALT, NOALT

IBM default:
NOALTERNATE

BASE

The BASE option can be used only when invoking the Compiler with the REXXC EXEC under z/OS (see "Invoking the Compiler with the REXXC (FANC) EXEC" on page 9) or when invoking REXXC indirectly using the ISPF panels (see "Invoking the Compiler with ISPF Panels" on page 11).

It can be used to specify the base for constructing the default output data set names for CEXEC, DUMP, IEXEC, OBJECT, and PRINT output.

BASE(*data-set-name***[(***member***)])**
The data set name and member name are used to construct the default data set names for compiler output.

If the BASE option is not specified, the output data set names are created as explained in "Derived Default Data Set Names" on page 10.

CEXEC

The CEXEC option specifies whether the Compiler is to produce a compiled EXEC. See also "OBJECT" on page 30 for an alternative form of compiled output.

CEXEC
Under z/OS, this option produces a compiled EXEC in the data set allocated to the ddname SYSCEXEC.

CEXEC[(*data-set-name***)]**
Can be used only when invoking the Compiler with the REXXC EXEC under z/OS (see "Invoking the Compiler with the REXXC (FANC) EXEC" on page 9). Generates a compiled EXEC.

This option is extended so that you can specify the name of the data set in which the compiled EXEC is to be stored. A default data set name is used if you do not specify *data-set-name*.

CEXEC[(*file-identifier***)]**
Under z/VM, this option produces a compiled EXEC. You need not fully specify the file identifier. The default file name is the name of the source file. The default file type is the letter C concatenated with the source file type. The default file mode is the file mode of the source file, provided you currently have read/write access to that minidisk; otherwise, file mode A1 is used.

NOCEXEC
Does not produce a compiled EXEC.

Abbreviations:
CE, NOCE

IBM default:
CEXEC

You can use compiled EXECs for:
- Programs to be used in command environments
- XEDIT macros
- PDF edit macros
- GDDM® macros
- Pipe filters
- Any other program that is not required to be in the form of a TEXT file or object module

Background information about compiled EXECs

You can replace your existing source EXECs with compiled EXECs. The search order for compiled and interpretable EXECs is the same, and they can be invoked in the same way. This makes it possible to ensure that there is no difference, from a user's point of view, between invoking a compiled EXEC and invoking the interpreter for the source program.

To achieve this aim:

- **Under z/OS,** using the explicit method of invoking EXECs, the TSO/E EXEC command specifies the location of the REXX EXEC.

  Using the implicit method of invoking EXECs, the interpretable EXEC is invoked as a command using the member name of the interpretable EXEC. For the system to give control to the compiled EXEC, the EXEC must have the same member name and must come earlier in the search order than the interpretable EXEC. For more information, see "Organizing Compiled and Interpretable EXECs under z/OS" on page 45, *TSO/E REXX/MVS Reference*, and *TSO/E Command Reference*.
- **Under z/VM,** the compiled EXEC must be given the same file type, such as EXEC or XEDIT, that the source program would have for interpretation. The source file must, therefore, be renamed, removed, or moved further down the search order. The sample compiler-invocation dialog, REXXD, handles this requirement. See "Invoking the Compiler with REXXD" on page 15 for a description of this dialog.

  A compiled EXEC behaves the same as an interpretable EXEC, the:
  - EXECLOAD command makes the EXEC resident
  - DCSSGEN utility loads the EXEC in a discontiguous saved segment (DCSS)
  - EXEC can be loaded and started through the CMS EXEC handler

  Under VSE/ESA, the compiled EXEC must be stored in a sublibrary with member type PROC. To ensure that the compiled REXX program is found before the interpretable one, use the **LIBDEF** statement as described in "Organizing Compiled and Interpretable EXECs under VSE/ESA" on page 46. See "Converting from z/OS to VSE/ESA" on page 86 or "Converting from z/VM to VSE/ESA" on page 87 for details.

The compiler writes information about the source file and the compilation to the compiled EXEC. The information includes the name of the source file (in z/OS, the data set name of the first data set in the SYSIN concatenation; in z/VM, the file ID), and the date and time of the compilation. The first 160 bytes of the compiled program are reserved for this information. You can use a text editor to browse or view the information.

Note: If you open a REXX compiled output file in edit mode, you must not update or save it.

COMPILE

The COMPILE option specifies whether the Compiler is to produce compiled code after all error checking has been performed. (The CEXEC and OBJECT options determine which files are created.)

COMPILE
: Generates compiled code, unless:
  - NOTRACE is in effect and a severe or terminating error is detected
  - TRACE is in effect and a terminating error is detected

NOCOMPILE
: Unconditionally suppresses the generation of compiled code after all error checking.

NOCOMPILE(W)
: Suppresses the generation of compiled code if a warning, error, severe error, or terminating error is detected.

NOCOMPILE(E)
Suppresses the generation of compiled code if an error, severe error, or terminating error is detected.

NOCOMPILE(S)
Suppresses the generation of compiled code if a severe error or terminating error is detected.

Abbreviations:
C, NOC

IBM default:
NOCOMPILE(S)

Note:

1. If you specify COMPILE with TRACE in effect, you receive output even if severe errors are diagnosed. If you specify COMPILE with NOTRACE in effect, you receive the same output as with **NOC(S)**.
2. You should only run a compiled REXX EXEC if it does not contain any errors. Otherwise unpredictable results may occur.

CONDENSE

The CONDENSE option specifies whether the generated output is to be condensed to take up less space. The saving in space can be up to 66%. The condensed program is uncondensed in storage prior to execution.

Note: The DLINK option and the CONDENSE option are mutually exclusive.

CONDENSE
Condenses the output generated by the CEXEC or the OBJECT compiler option, or both.

NOCONDENSE
Does not condense the output generated by the CEXEC or the OBJECT compiler option.

Abbreviations:
COND, NOCOND

IBM default:
NOCONDENSE

Background information about condensed programs

The size of a compiled REXX program often exceeds the size of the source program. You can use the CONDENSE compiler option to significantly reduce the size of both CEXEC type output and OBJECT type output. The time taken to load the condensed program is shorter. However, the execution time is longer because the program must be uncondensed before it is run.

It is recommended that you:

- Use the CONDENSE compiler option for programs that are not started frequently, such as active programs on a server or programs that are run only once a day and do not stop the execution.
- Do not use the CONDENSE compiler option for programs that are run frequently because of the time required to unpack the program each time it is run.

This option reduces the amount of:

- Disk space required by compiled REXX programs
- Virtual storage required by preloaded compiled REXX programs
- I/O activity required to load compiled REXX programs

When a condensed compiled REXX program is invoked, the program is automatically uncondensed. A condensed compiled REXX program requires more storage while it is running:

- During the uncondense operation, an additional 128KB (KB equals 1024 bytes) of storage are required.
- While a condensed compiled REXX program is running, both the condensed and the uncondensed copy exist in storage.
- Additional CPU time is required to uncondense the compiled REXX program. Apart from that, the performance characteristics of a condensed program equal the performance characteristics of an uncondensed program.

Note: The CONDENSE option can also be used to make a program unreadable if the source lines were included in the compiled program using the SLINE option.

DDNAMES

Under z/OS the DDNAMES option allows you to perform the compilation step with alternate DDNAMES that are selected by the user instead of using the standard names (described in Table 4 on page 14). The Compiler must read the renaming table from the data set defined by the standard name FANDDN.

If a DDNAME, such as SYSIN, is occupied by another program, you can use FANDDN to define an alternate *ddname* to replace the standard Compiler DDNAME.

DDNAMES
The data set defined by FANDDN is read and the specified Compiler DDNAMES are replaced.

DDNAMES(*ddname***)**
The data set defined by DDNAME (*ddname*) is read and the specified Compiler DDNAMES are replaced.

NODDNAMES
The data set defined by FANDDN is not read, and the specified DDNAMES are not replaced. This is the default option.

Abbreviations:
DD, DD(*ddname*), NODD

IBM default:
NODDNAMES

The alternate DDNAMES data set may be PO or PS organized with:

```
RECFM = F | FB
```

and

```
LRECL ≥ 17
```

The records must conform to the following rules:

- Comment records start with an asterisk (*) in the leftmost column.
- Blank records are treated as comment records.

- Renaming records specify the standard DDNAME in column 1 to 8 and the user defined DDNAME in column 10 to 17, both left justified.

  If the LRECL is greater than 17, there must be a blank character in column 18, and the remaining columns are ignored.
- Input is not case sensitive, DDNAMES are translated to uppercase.

Here is an example of alternate DDNAME definitions:

```
----+----1----+----2----+----3----+----4----+----5----+----6----+----7----+----8
*Alternate DDname definitions:

SYSIN    MYIN     : input  <- REXX source code
SYSCEXEC MYCEXEC  : output -> compiled EXEC (CEXEC  format)
SYSDUMP  MYDUMP   : output -> REXX compiler dump
SYSPRINT MYPRINT  : output -> REXX compiler listing
SYSPUNCH MYPUNCH  : output -> compiled EXEC (OBJECT format)
SYSTERM  MYTERM   : output -> REXX compiler messages
SYSIEXEC MYIEXEC  : output -> expanded source
SYSLIB   MYLIB    : input  <- REXX INCLUDE members
```

Note:

1. The alternate DDNAME must not be the same as a standard compiler DDNAME. Otherwise you might overwrite existing input data sets.
2. You cannot rename the standard DDNAME FANDDN, however, you can use the DDNAMES(*ddname*) option instead.
3. The alternate *ddname* applies only to the compilation step where it is defined.

DLINK

The DLINK option specifies whether the OBJECT output is to contain references to external routines and functions. External references are generated in the form of weak external references, requiring explicit inclusion of referenced programs when linking or loading.

Note:

1. The name can have a maximum length of 8 characters.
2. The DLINK option and the CONDENSE option are mutually exclusive.
3. The DLINK option and the TRACE option are mutually exclusive.
4. The DLINK option has no effect for programs that run with the Alternate Library.

DLINK

Generates weak external references in the OBJECT output for:

- Subroutines preceded by a CALL statement.
- External function calls.

  If the name is defined within quotes, blanks are not allowed. The name should be written in uppercase, because the underlying subsystem might not support mixed case.

NODLINK

Does not generate weak external references in the OBJECT output.

Abbreviations:

DL, NODL

IBM default:

NODLINK

Background information about directly linked external programs

When external functions and subroutines are linked directly to the REXX program, the REXX search order is bypassed, and the linked program is invoked directly. The advantages are:

- Better performance, as no search for the program is needed
- No possibility of accidentally accessing a program with the same name located earlier in the search order
- Improved packaging, because a program and its external subroutines can be linked into one load module

External functions and subroutines linked directly to a REXX program can be:

- Compiled REXX programs of type OBJECT.
  - In z/OS they must be linked with the EFPL or MULTI stub; see Appendix A, "Interface for Object Modules (z/OS)," on page 205.
  - In VSE/ESA they must be combined with the EFPL stub; see Appendix C, "Interface for Object Modules (VSE/ESA)," on page 225.
- Programs that are written in any programming language that conforms to the following linkage conventions:
  - Under z/OS and VSE/ESA, a directly linked program is invoked with an EFPL. It must conform to the linkage conventions for external functions and subroutines, as described in *TSO/E REXX/MVS: Reference* and in *IBM VSE/ESA REXX/VSE: Reference*.
  - Under z/VM, SVC linkage conventions are used, and register 13 must not be changed by the program. When applicable, the directly linked program is invoked in AMODE 31, and arguments are not copied below 16MB (MB equals 1 048 576 bytes) in virtual storage. The call type is X'05', a 6-word extended PLIST is passed to the invoked program. See Appendix B, "Interface for TEXT Files (z/VM)," on page 221 for details.

DUMP

Note: The DUMP option is not designed for program debugging. Use this option only if you suspect an error in the Compiler and if an IBM support representative asks for interphase dumps.

The DUMP option provides diagnostic information for use by IBM support personnel. If this option is specified, formatted dumps of the Compiler's control blocks and intermediate texts are taken after selected phases. Under z/OS, the dump is written to the SYSDUMP data set. Under z/VM, the dump file is sent to the virtual printer.

DUMP(*n*)
: Produces the interphase dumps specified by the value of *n*, where *n* is a number in the range 0 through 2047. The meaning of this parameter is fully described in the *IBM Compiler and Library for REXX on System z: Diagnosis Guide*.

DUMP
: Produces all interphase dumps.

DUMP[([*data-set-name*][,*n*])]
: Can be used only when invoking the Compiler with the REXXC EXEC under z/OS (see "Invoking the Compiler with the REXXC (FANC) EXEC" on page 9). Produces formatted dumps.

This option is extended so that you can specify the name of the data set in which the formatted dumps are to be stored. A default data set name is used if you do not specify *data-set-name*. All possible dumps are produced if you do not specify *n*.

NODUMP
Does not produce dumps.

Abbreviations:
DU, NODU

IBM default:
NODUMP

FLAG

The FLAG option specifies the minimum severity of errors for which messages are to be issued. (The PRINT and TERMINAL options specify where the messages appear.)

FLAG Is equivalent to FLAG(I).

FLAG(I)
Issues all messages, including informational messages.

FLAG(W)
Issues messages only for warnings, errors, severe errors, and terminating errors.

FLAG(E)
Issues messages only for errors, severe errors, and terminating errors.

FLAG(S)
Issues messages only for severe errors and terminating errors.

FLAG(T)
Issues messages only for terminating errors.

NOFLAG
Is equivalent to FLAG(T).

Abbreviations:
F, NOF

IBM default:
FLAG(I)

FORMAT

The FORMAT compiler option specifies that, in addition to the line numbers, the column numbers are to be included in the list of error messages and the cross-reference listing.

FORMAT
Is equivalent to FORMAT(C).

FORMAT(C)
Formats the error messages and cross reference with column numbers.

NOFORMAT
Does not format the error messages and cross reference with column numbers.

Abbreviations:
FO, NOFO

IBM default:
NOFORMAT

IEXEC

The IEXEC option generates an expanded output that contains the REXX source program and all members included by means of the %INCLUDE control directive. The IEXEC output is an interpretable REXX program.

The IEXEC output can contain fixed-length or variable-length records. Fixed-length records are written only if:

- All input files (REXX source and included files) have fixed-length records of identical record length.
- All %INCLUDE directives are defined either on separate lines or at the very end of a line to avoid a split of the line.
- Either all files contain sequence numbers or none of the files contains sequence numbers.
- Under z/OS, the output data set is explicitly defined with RECFM=F or FB.

In all other cases, variable-length records are written.

The Compiler does not write sequence numbers to the IEXEC output. This is because the sequence numbers from any %INCLUDE file might not be compatible with the sequence numbers from the main REXX source program and lead to error messages issued by many text editors. However, the LRECL values provided by the Compiler as default values provide 8 bytes for any renumbering.

If variable-length records are written to the IEXEC output, the records that originated from fixed-record-length files contain the trailing blanks they had in the originating file. This is necessary to ensure that the SOURCELINE built-in function, if called, gives the same results when the compiled program is run and when the IEXEC output is interpreted.

If you edit an IEXEC output of variable record length with a text editor like, for example, XEDIT under CMS, you may inadvertently remove the trailing blanks.

IEXEC Under z/OS, this option produces IEXEC output and stores it in the data set allocated to the ddname SYSIEXEC.

IEXEC[(*data-set-name***)]**
Can be used only when invoking the Compiler with the REXXC EXEC under z/OS (see "Invoking the Compiler with the REXXC (FANC) EXEC" on page 9). Generates IEXEC output.

This option is extended so that you can specify the name of the data set in which the IEXEC output is to be stored. A default data set name is used if you do not specify *data-set-name*.

IEXEC[(*file-identifier***)]**
Under z/VM, this option produces IEXEC output. You need not fully specify the file identifier. The default file name is the name of the source file. The default file type is the letter I concatenated with the source file type. The default file mode is the file mode of the source file, provided you currently have read/write access to that minidisk; otherwise, file mode A1 is used.

NOIEXEC
Does not produce IEXEC output.

Abbreviations:
I, NOI

IBM default:
NOIEXEC

Background information about calculating record lengths in z/OS

This box describes the record lengths supported by the Compiler. If you allocate a file for IEXEC output and assign an LRECL value to it, the value must conform to the description given in this box. The default values used by the Compiler are described at the end of the box.

For fixed-record lengths, LRECL must be set to one of the following:

- Without sequence numbers

  ```
  right_margin - left_margin + 1
  ```

- With sequence numbers

  ```
  right_margin - left_margin + 1 + 8
  ```

The MARGINS values apply to the records remaining after the Compiler has removed the sequence numbers. If you have set MARGINS to the default value **MARGINS(1 *)**, LRECL is equal to the record length of the record length of the source files.

For variable-length records, LRECL must be greater than, or equal to, one of the following:

- If none of the files contain sequence numbers

  ```
  right_margin - left_margin + 5
  ```

- If any of the files contain sequence numbers

  ```
  right_margin - left_margin + 5 + 8
  ```

If you specified * for right_margin, the value of right_margin in the last two expressions must be set to the length of the longest input record.

If no LRECL, RECFM, and BLKSIZE (z/OS) parameters have been assigned to the IEXEC output file, the Compiler supplies the following default values:

```
RECFM = V (CMS file) or VB (z/OS)
LRECL = max. value of (right_margin - left_margin + 5 + x)
        where x=8 if the record contains sequence numbers or
              x=0 if the record does not contain sequence numbers
BLKSIZE = 10 * LRECL
```

If you compile fixed-length records and want to have a fixed-length IEXEC file, create a file that assigns values to the RECFM, LRECL, and BLKSIZE parameters before calling the Compiler.

LIBLEVEL

The LIBLEVEL option specifies the version of the Library (minimum Library level) required to run the compiled program.

LIBLEVEL(*n*)
The level of the Library required to run the compiled program, where *n* is the minimum Library level number as shown in Table 5 on page 29. The Compiler checks that the language features used in the program are compatible with the Library level specified. If a feature is found that requires a higher Library level, this is flagged in the source listing.

LIBLEVEL(*)
Specifies that all levels of the Library are supported.

Abbreviations:
LL(*n*)

IBM default:
LL(*)

The following table shows the language features supported by the different Library levels.

Table 5. Library levels

Library Level	Library Name	New or Changed Features
2	Runtime Library Release 1 (TSO)	• CALL ON ERROR|FAILURE|HALT NAME built-in function • Addressing tails of compound variables with 1 or 2 components • Assignments
3	Runtime Library Release 2	• Arithmetic operations, for example, addition, multiplication • Binary strings including B2X and X2B built-in functions • Variable reference list (variable name enclosed in parentheses) in DROP and EXPOSE • Alternate Library via PTF
4	Runtime Library Release 3	• STREAM, LINES, LINEIN, LINEOUT, CHARS, CHARIN, and CHAROUT built-in functions • CALL|SIGNAL OFF NOTREADY • CALL|SIGNAL ON NOTREADY • TRACE statement and TRACE built-in function • INTERPRET statement
5	Runtime Library Release 3	• Date conversion
6	Runtime Library Release 3 and Release 4	• Date separation character

Note:

1. LIBLEVEL 0 and 1 are no longer supported.
2. The library level has not been changed for Release 4.
3. For more information refer to “TRACE” on page 37.

LINECOUNT

The LINECOUNT option specifies the maximum number of lines to be included on each page of the compiler listing. This number includes the header lines and any blank lines. You can specify that there are to be no page breaks within the source and cross-reference listings; this is useful if you intend to display the listing at a terminal, because there are no page headers to scroll through. However, if you print such a listing, your output continues from one page to the next without a break.

LINECOUNT(*n*)
Puts *n* lines on each page of the compiler listing, where *n* is a number in the range 10 through 99.

LINECOUNT(0)
Creates continuous output in the compiler listing.

Abbreviation:
LC

IBM default:
LINECOUNT(55)

MARGINS

The MARGINS option specifies the left and right margins of the REXX program. Only the text contained within the specified margins is compiled. The compiler listing, however, always contains the complete input records.

If the SLINE option is specified, the OBJECT or CEXEC output contains only the text within the specified margins. Similarly, if the IEXEC option is specified, the IEXEC output contains only the text within the specified margins.

If the first record of the source file contains only decimal digits in the first 8 bytes (RECFM=V|VB) or in the last 8 bytes (RECFM=F|FB), then the file is assumed to contain sequence numbers. In this case, the sequence numbers are removed and the specified margin values are applied to the remaining part of the record. Only the text contained within the specified margins is compiled.

Each file included by means of the %INCLUDE control directive is checked for sequence numbers. Therefore, a REXX source file can include files with different record formats and files with or without sequence numbers.

MARGINS(*left* [*right*])

left Specifies the first column of the source file containing valid REXX code. Valid values for *left* are:
- Under z/OS: from 1 to 32 760
- Under z/VM: from 1 to 65 535

right Specifies the last column of the source file containing valid REXX code. Valid values for *right* are:
- * (asterisk), the default, to indicate the last column of the input record
- Under z/OS: from *left* to 32 760
- Under z/VM: from *left* to 65 535

Abbreviation:
M

IBM default:
MARGINS(1 *)

OBJECT

Under z/OS, the OBJECT option specifies whether the Compiler is to produce an object module.

Under z/VM, the OBJECT option specifies whether the Compiler is to produce a TEXT file.

OBJECT
Under z/OS, this option produces an object module in the data set allocated to the ddname SYSPUNCH.

OBJECT[(*obj-data-set-name***)** |
(*[obj-data-set-name],stub[,load-data-set-name]***)]**

Generates an object module and, optionally, a load module.

Note: OBJECT can only be used when invoking the Compiler with the REXXC EXEC under z/OS (see "Invoking the Compiler with the REXXC (FANC) EXEC" on page 9).

obj-data-set-name
You can specify the name of the data set in which the object output is to be stored. A default data set name is used if you do not specify *obj-data-set-name*.

stub
You can specify a stub, which can be a member name, the name of a partitioned data set including a member name, or a predefined stub name. (Refer to "Stubs" on page 211 for a list of *stubnames* and member names.) If a stub is specified, a load module is created when the Compiler creates an OBJECT output.

Note: As the stubs are part of the Library, this form of invocation is available only if the Library is installed.

load-data-set-name
You can specify the name of the data set that is to contain the load module. If the member name is omitted, a default member name is assumed. The default data set name is used if you do not specify *load-data-set-name*.

OBJECT[(*file-identifier***)]**
Under z/VM, this option produces a TEXT file that has the file identifier you specify. The file identifier need not be fully specified. The default file name is the file name of the source file. The default file type is TEXT. The default file mode is the file mode of the source file, provided you currently have read/write access to that minidisk; otherwise, file mode A1 is used.

NOOBJECT
Does not produce an object module or a TEXT file.

Abbreviations:
OBJ, NOOBJ

IBM default:
NOOBJECT

Background information about using OBJECT output under z/OS

Under z/OS, object modules can be used to create load modules. The load modules can be used as commands and parts of REXX function packages.

Load modules are invoked in the same way as output from other high-level language Compilers:
- From z/OS JCL statements
- From the TSO/E command line

- As a host command
- As part of a function package from within a REXX program

See Chapter 6, "Using Object Modules and TEXT Files," on page 71 for information about function packages, and Appendix A, "Interface for Object Modules (z/OS)," on page 205 for more information.

For ISPF restrictions, see *ISPF/PDF Guide and Reference*.

For more information see Chapter 6, "Using Object Modules and TEXT Files," on page 71 and Appendix C, "Interface for Object Modules (VSE/ESA)," on page 225.

Background information about using OBJECT output under z/VM

Under z/VM, the Compiler can produce a TEXT file. A TEXT file can be processed into a MODULE file, which can then be started like a CMS command. A TEXT file can also be linked to an Assembler program. A MODULE file can also be used to create a function package from a REXX program.

Note:

1. MODULE files come after EXEC files in the CMS search order.
2. Although these TEXT files can be linked with other compiled programs, they must receive standard SVC PLISTs as input, unlike other high-level language programs. See Appendix B, "Interface for TEXT Files (z/VM)," on page 221 for details.
3. If your program is in the form of a MODULE file and it calls another module, the called module may overlay your program in storage. This occurs, for example, when both modules are loaded at the default start address. You can avoid this by specifying a start address when loading TEXT files or by using the NUCXLOAD command or the RLDSAVE option of the LOAD command.
4. For ISPF restrictions, see *ISPF/PDF Guide for VM*.

For more information on OBJECT output, see Chapter 6, "Using Object Modules and TEXT Files," on page 71 and Appendix B, "Interface for TEXT Files (z/VM)," on page 221.

Background information about using OBJECT output under VSE/ESA

Under VSE/ESA, the output from the OBJECT option can be used to create a phase. The output must be generated either on z/OS or on z/VM, then transferred to VSE/ESA. When it is on VSE/ESA, phases can be built. The phases can be invoked as programs from JCL, or as parts of REXX function packages.

For more information see Chapter 6, "Using Object Modules and TEXT Files," on page 71 and Appendix C, "Interface for Object Modules (VSE/ESA)," on page 225.

OLDDATE

The OLDDATE option controls the creation date and time of the files that are generated by the Compiler under z/VM. It can be set to the date and time, when the source file was last changed.

OLDDATE

Applies to all output files.

OLDDATE(*nnnn*)

Selects the types of output files: (***nnnn***) can contain a selection list of generated files. The following rules apply for this selection list:

- Separator blanks are not allowed.
- The list may be empty or a concatenation of any of the following characters that select the corresponding output file:

  C CEXEC (the compiled EXEC)

  P PRINT (the compiler LISTING)

  I IEXEC (the expanded EXEC)

  O OBJECT (the TEXT file)

Note:

1. You can specify the letters CPIO in any order.
2. If print output is sent to a virtual printer, the OLDDATE option has no effect.

Here are examples for the OLDDATE(*nnnn*) option:

OLDDATE(C)
OLDDATE option applies to CEXEC file only.

OLDDATE(IC)
OLDDATE option applies to CEXEC and IEXEC file only.

OLDDATE()
OLDDATE() option applies to all output files, this is identical to the option OLDDATE and to the option OLDDATE(CPIO).

NOOLDDATE
Generates files with the current date and time. This option is used by default.

Abbreviations:
OLDD/NOOLDD

IBM default:
NOOLDDATE

OPTIMIZE

The OPTIMIZE option specifies whether the object code is to be optimized to reduce the amount of CPU time it requires at runtime.

OPTIMIZE
The compiled output is optimized.

NOOPTIMIZE
The compiled output is not optimized.

Abbreviations:
OPT/NOOPT

IBM default:
OPTIMIZE

Note:

1. This option can also be coded as OPTIMISE/NOOPTIMISE to support British spelling.
2. Use this option only to verify a defect encountered, then report the problem to your IBM representative.

PRINT

The PRINT option specifies whether a compiler listing is to be created and, if so, where it is to be printed or stored.

The listing shows the compiler options used and, depending on which other compiler options are in effect, the source program, messages, and cross-reference listing. See also Chapter 5, "Understanding the Compiler Listing," on page 51.

PRINT
Under z/OS, this option creates a compiler listing in the data set allocated to the ddname SYSPRINT.

Under z/VM, this option creates a compiler listing and sends it to the virtual printer.

PRINT[(*data-set-name*|*|****)]**
Can be used only when invoking the Compiler with the REXXC EXEC under z/OS (see "Invoking the Compiler with the REXXC (FANC) EXEC" on page 9).

This option is extended so that you can specify the name of the data set where the compiler output listing is to be stored. If you specify an asterisk, the listing is written to the terminal. A default data set name is used if you do not specify *data-set-name* or * (asterisk). If you specify ** (two asterisks), any preallocation for **SYSPRINT** is used.

PRINT([*file-identifier***])**
Under z/VM, this option creates a compiler listing file that has the file identifier you specify, or a default file identifier. You need not fully specify the file identifier. The default file name is the file name of the source file. The default file type is LISTING. The default file mode is the file mode of the source file, provided you currently have read/write access to that minidisk; otherwise, file mode A1 is used.

NOPRINT
Does not create a compiler listing.

Abbreviations:
PR, NOPR

IBM default:

- Under z/OS: PRINT
- Under z/VM: PRINT()

SAA

The SAA option specifies whether the Compiler is to check the source program for REXX language elements that are not part of level 4.02 of the SAA REXX interface. When this option is in effect and the FLAG option is set to I or W, a warning message is issued for each non-SAA item found.

Note: The Compiler does not detect the following:

- A non-SAA item if it is contained in an instruction that is not fully analyzed until runtime. For example, `DATE('C')` is flagged as a non-SAA item. However, `INTERPRET "SAY DATE('C')"` is not flagged because the contents of the character string after INTERPRET are evaluated at runtime.
- Wrong arguments in stream I/O built-in functions or a wrong number of arguments in stream I/O built-in functions.

- DBCS symbols are not flagged if a program is compiled with `Options 'ETMODE'` in effect.

SAA Checks for SAA compliance.

NOSAA
Does not check for SAA compliance.

Abbreviations:
None

IBM default:
NOSAA

SLINE

The SLINE option specifies whether the Compiler is to include the source program in the compiled output and, consequently, support the SOURCELINE built-in function at runtime. If you require support for Alternate Libraries or full tracing, you should also set this option. If the MARGINS option is specified, the compiled output contains only the text between the specified margins.

This option also determines whether the source code appears in traceback messages, which are issued for runtime errors. If you specify SLINE, users can see the source code. Also, the compiled program is larger. See also "SOURCELINE Built-In Function" on page 94.

SLINE
Includes the source program in the compiled code.

SLINE(AUTO)
Includes the source program in the compiled code only if one or more of the following are met:
- The SOURCELINE built-in function is found in the program.
- The TRACE compiler option is set.
- The ALTERNATE compiler option is set.

NOSLINE
Does not include the source program in the compiled code.

Abbreviations:
SL, SL(A), NOSL

IBM default:
NOSLINE

SOURCE

The SOURCE option specifies whether the compiler listing is to include a source listing. If you specify NOSOURCE, only erroneous source lines are included in the listing with the corresponding messages. See also "Source Listing" on page 52.

SOURCE
Produces a source listing.

NOSOURCE
Does not produce a source listing.

Abbreviations:
S, NOS

IBM default:
SOURCE

TERMINAL

The TERMINAL option specifies whether messages and the message summary are to be displayed at the terminal (z/VM) or to be written to the data set allocated to the ddname SYSTERM (z/OS), in addition to being included in the compiler listing. The messages depend on the setting of the FLAG option. Use the TERMINAL option when you expect only a small number of errors.

Note:

1. Under z/OS, if SYSPRINT and SYSTERM are allocated to the same destination, messages that would otherwise be issued to both SYSPRINT and SYSTERM are issued only once.
2. Terminating errors are always displayed.

TERMINAL
Displays messages at the terminal. A message displayed at the terminal is always preceded by the source line that contains the error. If no messages are issued, the message summary is not displayed.

NOTERMINAL
Does not display messages at the terminal.

Abbreviations:
TERM, NOTERM

IBM default:
NOTERMINAL

TESTHALT

The TESTHALT compiler option specifies whether the compiled program is to contain code that supports the halt condition. One way to set the halt condition is, for example, the HI (Halt Interpretation) immediate command.

Specify the TESTHALT option to halt the program without consequently affecting the operation of any other programs. This is especially useful when you want to halt an edit macro that is looping, without terminating the whole editing session, as the HE command would do in z/OS, or as the HX command would do in z/VM.

To specify TESTHALT hooks in the program independently of the TESTHALT compiler option, use the %TESTHALT compiler directive:

- For further information, see "%TESTHALT" on page 43.
- For performance considerations, see "TESTHALT Option" on page 109.
- Also see "Halt Condition" on page 91.

TESTHALT
Generates code that supports the HI command.

NOTESTHALT
Does not generate code that supports the HI command.

Abbreviations:
TH, NOTH

IBM default:
NOTESTHALT

TRACE

The TRACE option specifies that the compiled program can be traced. The performance of a program compiled with the TRACE option is not as good as that of the same program compiled with the NOTRACE option. However, a program compiled with the TRACE option usually has a better performance than the same program when it is interpreted.

TRACE

Creates a compiled program of CEXEC or OBJECT type that can be traced. The TRACE instruction and the TRACE built-in function are supported, except for the trace setting SCAN. The initial trace setting is NORMAL, as with the interpreter.

Note:

1. You must also specify the SLINE compiler option described in "SLINE" on page 35.
2. To ensure that all source statements can be run by the Library, the Compiler performs a pseudo compile of the source to determine the required Library level and subsequently writes this value to the compiler listing and object.
3. However, if you define language constructs in interactive TRACE mode or in the character strings in INTERPRET clauses, which are not supported by the installed Library, these definitions are flagged at runtime.

NOTRACE

Creates a compiled program of CEXEC or OBJECT type that cannot be traced. The compiled program behaves the same as interpreted programs that run with TRACE set to OFF. At runtime, all valid options in the TRACE instructions and TRACE built-in functions are set to OFF.

Note: If the program is compiled with the ALTERNATE option and run with the Alternate Library, it can be traced like a normal interpreted program.

Abbreviations:

TR, NOTR

IBM default:

NOTRACE

XREF

The XREF option specifies whether the compiler listing is to include a cross-reference listing for all:

- Variables:
  - Dropped (d) variables
  - Exposed (e) variables
  - Variables that are not initialized (SIMPV+++). These variables do not have an assignment.
- Labels
- Constants
- Built-in functions
- External routines
- Source lines that contain recognized commands and ADDRESS clauses

- Lines that contain erroneous clauses may or may not appear in the command list
- Optimizing stoppers from top to bottom. For more information refer to "Optimization Stoppers" on page 107. If you are using the TESTHALT compiler option or %TESTHALT compiler directive, refer to "Halt Condition" on page 91.

  Note: As this information increases the size of the compiler listing considerably, you can specify XREF(SHORT) to suppress this information.

The cross-reference listing indicates the numbers of the lines on which the above items are referenced. It is useful for debugging and program maintenance. See also "Cross-Reference Listing" on page 55.

XREF Produces a cross-reference listing.

XREF(SHORT)
Produces a cross-reference listing that does not contain the following:
- Constants
- Commands
- Optimizing stoppers

NOXREF
Does not produce a cross-reference listing.

Abbreviations:
X, X(S), NOX

IBM default:
NOXREF

Control Directives

This section describes the functions and syntax of the compiler control directives in alphabetic order.

A control directive always starts with **/*%** and ends with ***/**.

%COPYRIGHT

The %COPYRIGHT control directive inserts a notice (for example a copyright notice) in the form of a visible text string in the CEXEC, OBJECT output, and core image of the compiled program. The text string starts after the header part.

The %COPYRIGHT control directive is contained in a comment; it is recognized as a control directive only by the Compiler (it is treated as a normal comment by the interpreter):

```
/*%COPYRIGHT (c) copyright MY company 2003*/
```

The %COPYRIGHT control directive is recognized as such only if it immediately follows a `/*` comment delimiter. The word %COPYRIGHT can be in mixed case.

The notice can be broken into several %COPYRIGHT control directives. The text following %COPYRIGHT, starting with the first nonblank character and up to the end of the comment, is called a copyright part and is used to build the copyright notice. The final copyright notice is the concatenation of all copyright parts defined in the program.

This is an example of a REXX program that contains %COPYRIGHT control directives:

```
/*%COPYRIGHT This is an example of a copyright */
Say 'Hello'
/*%COPYRIGHT notice. */
```

The following string is taken as the copyright notice:

```
This is an example of a copyright notice.
```

Note: Blank characters immediately following %COPYRIGHT are ignored. Blank characters at the end of a copyright part, preceding the */ delimiter, are taken as part of the copyright notice.

A copyright part can contain comments. The text in these comments is taken as such and used as part of the copyright notice, even if the comment contained in a copyright part begins with a directive. For example:

```
/*%COPYRIGHT Example of a copyright notice containing a /*%COPYRIGHT comment*/.*/
```

The resulting copyright notice is:

```
Example of a copyright notice containing a /*%COPYRIGHT comment*/
```

%INCLUDE

The %INCLUDE control directive inserts, at compilation time, REXX code contained in z/OS data sets or in CMS files into the REXX source program.

The %INCLUDE control directive is contained in a comment; it is recognized as a control directive only by the Compiler (it is treated as a normal comment by the interpreter):

```
/*%INCLUDE file1 */
```

For a %INCLUDE directive to be recognized as such, the following must be true:

- The directive immediately follows a /* comment delimiter.
- The directive is not part of another %INCLUDE directive or of a %COPYRIGHT directive.
- The name of the file to be included starts with the first nonblank character following /*%INCLUDE and must not contain any blank characters.
- The z/OS data-set identifiers member and ddname and the CMS file identifiers filename and ddname are restricted to 8 characters in length.

Note:

1. The word %INCLUDE can be in mixed case.
2. You can only specify nested comments following the file name.
3. Files that are included by means of %INCLUDE directives can contain %INCLUDE directives.

This is an example of how %INCLUDE directives can be specified:

```
/*%INCLUDE file1 */
Say 'Hello 1'
/*%INCLUDE file2 */ Say 'Hello 2'
```

The contents of file1 will be inserted before Say 'Hello 1'. The last line in the example is split into two parts, forming two lines.

```
1. /*%INCLUDE file2 */
2.                     Say 'Hello 2'
```

The contents of file2 will be inserted between the first part and the second part, immediately following the */ delimiter. In the compiler listing and IEXEC output, the first line is truncated. The second part of the line is not reformatted. However, the space previously occupied by the %INCLUDE directive and any statements preceding it, is replaced by blanks. If the IEXEC option has been specified, the IEXEC output will have, in this case, variable length format (see "IEXEC" on page 27).

Note:

1. At the end of the first part of a split line, a line end is implied.
2. The built-in function SOURCELINE() returns the line number of the final line in the expanded program, or 0 if the program was compiled with the NOSLINE option.

The naming convention for included files is as follows:

- Under z/OS:
  - `/*%INCLUDE member */`

    Search for member:
    1. In the concatenation with ddname SYSLIB, if it is allocated
    2. In the same partitioned data set as the source, if the source is in a partitioned data set
  - `/*%INCLUDE ddname(member) */`

    Search for member in the concatenation with ddname.
- Under z/VM:
  - `/*%INCLUDE filename */`

    Search for a file with file name filename and file type COPY on all accessed disks. If it does not exist, search for a file with file name filename and file type REXXINCL on all accessed disks. If it also does not exist, search for a file with file name filename and file type EXEC on all accessed disks.

    If more than one file is found for a specific file type, the one on the minidisk which comes earlier in the search order is included.
  - `/*%INCLUDE ddname(filename) */`
    1. `FILEDEF ddname DISK fn ft [fm]`

       can be used to specify a collection of files.

       Note: ft must be COPY, REXXINCL, or EXEC, otherwise the file will not be found.

       Search for a file with file name fn and file type COPY within the specified collection. If it does not exist, search for a file with file name fn and file type REXXINCL within the specified collection. If it also does not exist, search for a file with file name fn and file type EXEC within the specified collection.

       If more than one file is found for a specific file type, the one on the minidisk which comes earlier in the search order is included.
    2. `CREATE NAMEDEF fm ddname ( FILEMODE`

       or

       `CREATE NAMEDEF dirid ddname`

       followed by:

       `ACCESS dirid fm`

can be used to identify a specific minidisk. Search for a file with file name `filename`, file type `COPY`, and file mode `fm`. If it does not exist, search for a file with file name `filename`, file type `REXXINCL`, and file mode `fm`. If it also does not exist, search for a file with file name `filename`, file type `EXEC`, and file mode `fm`.

If a file is found for a specific file type, it is included.

3. Members of **MACLIB**s can be included. If ddname is SYSLIB, all **MACLIB**s established with the command `GLOBAL MACLIB` are searched until a member with name `filename` is found and included.

   If ddname is not SYSLIB, search within the **MACLIB** with name `ddname` for a member with name `filename` and include it.

The names of the data sets or files that have been included are contained in the compiler listing.

%PAGE

The %PAGE listing control directive causes an unconditional skip to a new page in the source listing.

The %PAGE listing control directive is contained in a comment; it is recognized as a control directive only by the Compiler (it is treated as a normal comment by the interpreter):

```
/*%PAGE  */
```

The %PAGE listing control directive is recognized as such only if it immediately follows a /* comment delimiter and these characters are the first nonblank characters on the line. The word %PAGE can be in mixed case. The rest of the line can contain any other characters. It is good practice to close the comment on the same line.

A line that contains the %PAGE listing control directive is printed as the last line on the current page of the listing; the next line in the source program starts a new page. If the compiler option LINECOUNT(0) is specified, however, %PAGE has no effect.

%STUB

The %STUB prelink control directive simplifies link-editing under z/OS. It generates object modules that include preselected STUB code. The z/OS stub code is inserted into the object output.

To define the %STUB prelink control directive you must specify the stub name *stubname* in the source:

```
/*%STUB stubname*/
```

Refer to "Stubs" on page 211 for a list of *stubnames* that are supplied with the Library to provide interfaces with the various types of parameter-passing conventions.

Note:

1. %STUB code can be compiled under z/OS and z/VM.
2. The OBJECT option must be in effect for the STUB code to be included.
3. CEXEC output is not affected.
4. An example is shown in "Improving Packaging and Performance" on page 73.

5. Object modules generated with STUB code terminate abnormally when they are run under z/VM.
6. Stubs are required when running REXX link-edited under z/OS.
7. See also “REXXL (z/OS)” on page 74.
8. See also “Stubs” on page 211.

%SYSDATE

The %SYSDATE control directive inserts, at compilation time, code to create the variable **SYSDATE**, which contains the compilation date.

Because %SYSDATE is contained in a comment only the Compiler recognizes it as a control directive. %SYSDATE must immediately follow a /* comment delimiter.

```
/*%SYSDATE */
/*%SYSDATE(option) */
```

The word %SYSDATE can also be in lowercase or mixed case.

The comment containing %SYSDATE must not be contained in a clause:

```
say /*%sysdate */ 'hello'
```

Instead, enclose the comment in semicolons (;) or put it on a new line:

```
say 'hello'
/*%sysdate */
```

The option for %SYSDATE is one of the formats of the REXX DATE built-in function, namely B, D, E, M, N, O, S, U, or W. C and J are not supported.

The variable SYSDATE is not set if running with the alternate library or if compiled with option TRACE. In the latter case, or if executing under the interpreter, the contents of the variable **SYSDATE** are set to the character string "SYSDATE" if no SIGNAL ON NOVALUE has been executed. If a SIGNAL ON NOVALUE has been executed, the NOVALUE condition is raised during execution. The code generated by the Compiler does not raise the NOVALUE condition if compiled with NOTRACE.

The following example raises a NOVALUE condition if interpreted or compiled with TRACE:

```
/*%sysdate */
say 'compilation date=' sysdate
```

To avoid a NOVALUE condition, change the previous example as follows:

```
sysdate = ''
/*%sysdate */
if (sysdate <> '') then say 'compilation date=' sysdate
```

%SYSTIME

The %SYSTIME control directive inserts, at compilation time, code to create the variable **SYSTIME**, which contains the compilation time.

Because %SYSTIME is contained in a comment only the Compiler recognizes it as a control directive. %SYSTIME must immediately follows a /* comment delimiter.

```
/*%SYSTIME */
/*%SYSTIME(option) */
```

The word %SYSTIME can also be in lowercase or mixed case.

The comment containing %SYSTIME must not be contained in a clause:

```
say /*%systime */ 'hello'
```

Instead, enclose the comment in semicolons (;) or put it on a new line:

```
say 'hello'
/*%systime */
```

The option for %SYSTIME is one of the formats of the REXX TIME built-in function, namely C, H, L, M, N, or S. E and R are not supported.

The variable SYSTIME is not set if running with the alternate library or if compiled with option TRACE. In the latter case, or if executing under the interpreter, the contents of the variable **SYSTIME** are set to the character string "SYSTIME" if no SIGNAL ON NOVALUE has been executed. If a SIGNAL ON NOVALUE has been executed, the NOVALUE condition is raised during execution. The code generated by the Compiler does not raise the NOVALUE condition if compiled with NOTRACE.

The following example raises a NOVALUE condition if interpreted or compiled with TRACE:

```
/*%systime */
say 'compilation time=' systime
```

To avoid a NOVALUE condition, change the previous example as follows:

```
systime = ''
/*%systime */
if (systime <> '') then say 'compilation time=' systime
```

%TESTHALT

The %TESTHALT control directive inserts, at compilation time, code to support the HALT condition. It enables you to halt a program at specific statements during program execution.

The %TESTHALT control directive is contained in a comment; it is recognized as a control directive only by the Compiler (it is treated as a normal comment by the interpreter):

```
/*%TESTHALT  */
```

The %TESTHALT control directive is recognized as such only if it immediately follows a /* comment delimiter. The word %TESTHALT can be in mixed case.

The generated code for the TESTHALT hook is placed at the beginning of the clause containing the %TESTHALT compiler directive. In the following example, the TESTHALT hook is generated before the SAY keyword.

```
say 'hello'  /*%testhalt */
```

If you want the TESTHALT hook to be generated after the SAY clause, use a semicolon (;) to end the clause, or put the compiler directive on a new line:

```
say 'hello';  /*%testhalt */
```

```
say 'hello'
/*%testhalt */
```

The %TESTHALT control directive provides better control over the TESTHALT hooks than the TESTHALT compiler option. It can be used either together with the TESTHALT compiler option to provide additional hooks, or without. In the latter case, only the hooks specified by the control directive are generated. Using the %TESTHALT control directive without the TESTHALT compiler option improves the runtime performance of the REXX program. This is because each TESTHALT hook is an overhead in the compiled program and the Compiler optimizes the program less if it contains TESTHALT hooks.

Chapter 4. Runtime Considerations

This chapter contains suggestions for organizing your libraries and other information for improving the running of compiled programs. (Under z/VM, see the online help for information on how to run a program from the REXXD compiler-invocation dialog.)

Note: To run compiled REXX programs, either the IBM Library for REXX on System z or the Alternate Library must be installed on z/VM or z/OS. REXX/VSE must be installed on VSE/ESA.

Organizing Compiled and Interpretable EXECs under z/OS

Because REXX programs can either be interpreted or run compiled, you might inadvertently run the source program with the interpreter when you intend to run the compiled program.

You can avoid such situations by following the procedure described below. For the purposes of this procedure, assume that your REXX source programs are stored in the production library *pref.cccc*.EXEC, which is in your search order.

1. Compile the programs and store them in the data set *pref.cccc*.CEXEC. For example, to compile a REXX program named **ROULETTE** you could enter the following REXXC command:

   rexxc '*pref.cccc***.exec(roulette)' cexec('***pref.cccc***.cexec(roulette)')**
2. Save the source programs in the data set *pref.cccc*.SEXEC. In this example, the program **ROULETTE** is saved in *pref.cccc*.SEXEC(roulette).
3. Copy the compiled EXECs by means of the REXXF command from the *pref.cccc*.CEXEC data set to the *pref.cccc*.EXEC data set. (See "REXXF (FANCMF) under z/OS" on page 87.) You now run the compiled EXECs that are in this data set, because it is in the search order. However, if you want to run an interpretable REXX EXEC, copy it from the *pref.cccc*.SEXEC data set to the *pref.cccc*.EXEC data set.

The advantages of this organization include the following:

- Users can browse the source code of EXECs in the source library.
- Users can store copies of the source code of EXECs in their private EXEC libraries for tracing or execution.
- Source EXECs can be maintained in the source library. When the modifications are completed and tested, the EXECs can be compiled and stored in the production library.
- Because the data sets containing source programs and compiled EXECs have the same data set attributes, users can easily move and replace source programs and compiled EXECs.

For other ways to switch between interpreted and compiled REXX programs, see "Background information about compiled EXECs" on page 20.

Organizing Compiled and Interpretable EXECs under z/VM

Because REXX programs can either be interpreted or run compiled, you might inadvertently interpret the source program when you intend to run the compiled program. The following examples show how this could occur:

- You have a compiled EXEC called **ROULETTE**. It is stored on a library disk, which is accessed as your L-disk. You enter **roulette** to invoke the compiled EXEC. But if the source program is on your A-disk and also has a file type of EXEC, you invoke the interpreter instead.
- You have access to a compiled REXX program called **ROULETTE MODULE**. You enter **roulette** to invoke the module. However, EXEC files precede MODULE files in the CMS search order. So if you still have access to the source program and its file type is EXEC, you invoke the interpreter instead.

You can avoid such situations by changing the file type of the source file after compilation. The following table shows a suggested naming convention.

Type of File	Recommended File Type
Source file after compilation	SEXEC, SXEDIT, and so on, as applicable
Compiled EXEC immediately after compilation, when the source file type may be EXEC.	CEXEC
Compiled EXEC ready for execution	EXEC or other required file type, such as XEDIT

Note: You can also make source files unavailable by removing them from any disks accessed by the program's users.

If you are using the compiler-invocation dialog, REXXD, use the Switch (rename) action to rename the files appropriately. Otherwise, use the CMS RENAME command, as required.

Organizing Compiled and Interpretable EXECs under VSE/ESA

Because REXX programs can either be interpreted or run compiled, you might inadvertently run the source program with the interpreter when you intend to run the compiled program.

You can avoid such situations by following the procedure described below.

- Keep the source for all REXX programs in a library called **REXXLIB.EXEC**. Each member has a member type of **PROC**.
- Once an EXEC is ready to be compiled, send it to either z/VM or z/OS and compile it.
- After the compilation, send it back to VSE/ESA, and catalog the output in a library called **REXXLIB.CEXEC**. The member name is the same as that of the original source, and the member type is **PROC**. See "Converting from z/OS to VSE/ESA" on page 86 and "Converting from z/VM to VSE/ESA" on page 87 for more information.
- Use the following **LIBDEF** statement when running REXX programs:

```
LIBDEF PROC,SEARCH=(REXXLIB.CEXEC,REXXLIB.EXEC)
```

This ensures that the compiled REXX program, if it exists, is found before the interpreted REXX program. If there is no compiled REXX program, the interpreted program is found.

The advantages of this organization include the following:

- The source code of REXX EXECs is maintained in a central sublibrary, and can always be retrieved.
- If a member with the same name is deleted in the **REXXLIB.CEXEC** sublibrary, a subsequent invocation will invoke the interpreted program.

Use of the Alternate Library (z/OS, z/VM)

The Alternate Library is necessary for:

- Customers who want to run compiled REXX programs, but do not have the Library installed
- Software developers who want to make their programs available to users who do not have the Library installed

Users of the Library do not need the Alternate Library. The Library provides more functions and better performance than the Alternate Library. Software developers must test their applications with the Library and with the Alternate Library.

By enabling their programs to run with both the Library and the Alternate Library, software developers give their customers the following possibilities:

- Use the Alternate Library provided with the application, if they have no library installed.
- Use the IBM Library for REXX on System z, if it is installed.

Use the SLINE and ALT options to enable a compiled program to run also with the Alternate Library. Use the CONDENSE option to hide the source lines.

Other Runtime Considerations

- **Activation of the Alternate Library**
  - Under z/OS, the Alternate Library is activated in different ways depending on its intended use:
    - Software developers use the Alternate Library from the ddname STEPLIB. This is because they need to have both the Library and the Alternate Library installed. To lower storage consumption, the Library must reside in the link pack area (LPA) instead of residing in every address space in the system. To test their programs with the Alternate Library, software developers use the ddname STEPLIB to override the Library.
    - Customers use the Alternate Library from the LINKLIST. This is because the LINKLIST is searched after the LPA. Customers should always use the Library, if it is available. By placing the Alternate Library in the LINKLIST, they will never override the Library in the LPA.

    Table 6 summarizes the possible library locations.

Table 6. Library and Alternate Library Locations (z/OS)

Library name	**Library location: SW developer**	**Library location: Customer**
IBM Library for REXX on System z	LPA	LPA

Table 6. Library and Alternate Library Locations (z/OS) (continued)

Library name	Library location: SW developer	Library location: Customer
Alternate Library	STEPLIB	LINKLIST

  - Under z/VM, the Alternate Library must always be loaded from disk to avoid conflicts with the Library.
    - Software developers activate the Alternate Library like this:
      1. Copy **EAGALPRC MODULE**, the library loader of the Alternate Library, to a disk that is ahead of the disk containing the library loader of the library (**EAGRTPRC MODULE**) in the system search order. Name this copy **EAGRTPRC MODULE**.
      2. Copy **EAGALUME TXTAMENG**, the message repository of the Alternate Library, to a disk that is ahead of the disk containing the message repository of the library (**EAGUME TXTAMENG**) in the system search order. Name this copy **EAGUME TXTAMENG**. If in your installation **EAGUME TXTAMENG** has been renamed to **EAGUME TEXT**, then name your copy **EAGUME TEXT**, as well.
      3. To ensure that the library loader from this disk is being used, you can either IPL your virtual machine, or issue the command **NUCXDROP EAGRTPRC**.
    - Customers who do not have the Library installed do not need to do anything to use the Alternate Library. The Alternate Library is available after it has been installed.

- **Batch mode:** Unless your program issues host commands that must be executed in the foreground or is designed to be run interactively, you can run it in batch mode. Use your standard procedure for submitting batch jobs.
- **Error handling:** If an instruction has an error, the Library might not raise the same error that the interpreter would raise.

  If the length of a variable's value is greater than 16MB, the results are unpredictable.
- **Interfaces with interpreted programs:** There are no restrictions on the mutual invocation of compiled programs and interpreted programs: a compiled program can call an interpreted program, and an interpreted program can call a compiled program. When a program is invoked, z/OS, z/VM, or VSE/ESA starts the correct language processor—either the interpreter or the Library.
- **Loading the Library under z/VM:** Depending on the system setup, the z/VM Library can be loaded in two different ways:
  1. The Library and the message repository are always available and do not need to be explicitly loaded, if they are installed as logical segments. See "Defining the Library as a Logical Segment" on page 121 for more information.
  2. The Library is loaded into virtual storage the first time a compiled REXX program is run and remains loaded after the program ends. The Library is loaded in the following way:
     a. The library loader (EAGRTPRC MODULE), which is itself loaded from disk, receives control and runs in the transient program area.
     b. The library loader loads the message repository.
     c. The library loader loads the Library from a DCSS unless one of the following conditions applies:
        - No DCSS exists.

- With Release 5 of CMS, the DCSS overlaps the storage of the virtual machine. With subsequent releases, the storage where the segment resides is in use. Storage can be reserved with the SEGMENT RESERVE command in CMS.
- The library loader has been customized so that it does not look for the Library in a DCSS.

If any of these conditions apply, the Library is loaded from disk.

d. The library loader makes the Library a nucleus extension and names it EAGRTPRC.

Note:

a. With systems before VM/ESA Release 1.1, the Library is made a nucleus extension of length 0. This ensures that a NUCXDROP EAGRTPRC or NUCXDROP * command issued from a compiled REXX program does not free the storage into which the Library is loaded. If a NUCXDROP command is issued, a new copy of the Library is loaded the next time a compiled REXX program is run; the storage occupied by the previous copy is not regained.

b. With VM/ESA Release 1.1 or a subsequent release, the Library is loaded by issuing a NUCXLOAD command with the PERM option, so that a NUCXDROP * command will not release the Library. Storage can be regained by issuing a NUCXDROP EAGRTPRC command. This command must not be issued while a compiled REXX program is running, otherwise unpredictable results may occur.

c. A **NUCXDROP EAGRTPRC** command must be issued before purging the segment that contains the Library, otherwise an **ABEND** will occur.

- **Runtime messages:** In certain cases, the Library gives more information about the error than is provided by the interpreter's error messages. In these cases, a secondary message then follows the main message. For example, if your program **BRCL EXEC** calls, on line 115, the LASTPOS built-in function with a negative value for the *start* argument, the following messages are displayed:
  - `EAGREX4000E Error 40 running compiled BRCL EXEC, line 115: Incorrect call to routine`
  - `EAGREX4003I Argument not positive`

  For explanations of the runtime messages, see Chapter 20, "Runtime Messages," on page 181.

  Note: Secondary messages are for your information only. They are not accessible through the ERRORTEXT function and do not affect the setting of the special variable RC.
- **SETVAR:** Starting with Release 2 of the IBM Compiler and Library, the VALUE built-in function provides the same support as **RXSETVAR**.

  Note:

  1. For compatibility with earlier releases, **RXSETVAR** and **SETVAR** are still supported by Release 4.
  2. It is, however, recommended that you now use the **VALUE** built-in function, because it provides a better performance.
- **Some common errors:** This section lists some common errors that can occur at runtime.

  Under z/OS:

- **Library not found:** If the Library is not in the LPA, in the LINKLIST concatenation, or defined in the STEPLIB DD statement, the following failure occurs:

```
CSV003I REQUESTED MODULE EAGRTPRC NOT FOUND
CSV003I REQUESTED MODULE EAGRTXLD NOT FOUND
CSV003I REQUESTED MODULE EAGRTXVH NOT FOUND
+IRX0158E The runtime processor EAGRTPRC could not be found.
```

Note: Make sure that the data sets contained in the Library or Alternate Library have not been dropped from the LPA or STEPLIB concatenation. This can occur when using APF authorized data sets.

Under z/VM:

- **Module A Overlaid by Module B:** If your program is in the form of a module, module A, and it calls another module, module B, module B might overlay your program in storage. This occurs if, for example, both modules are loaded at the default starting address. The failure occurs when module B tries to return control to your program.

  To determine whether an overlay caused the failure, recompile the program, creating a compiled EXEC, and recreate the circumstances in which the failure occurred. If the problem disappears, the failure was almost certainly caused by a module overlay. In this case, either continue to run the program as a compiled EXEC or explicitly specify a different starting address when loading your module. If the problem persists, the failure has a different cause, and you should contact your system support personnel.
- **Return Code -3:** If you get a return code of -3 when you invoke your program, it usually means that the program was not found. However, it can alternatively mean that the Library was not found. So, if you get this return code when the program is available, make the Library available—either in a DCSS or on disk.
- **SVC depth:** A maximum supervisor call (SVC) nesting depth of 200 is supported by CMS. The CMS EXEC processor invokes the Library by means of an SVC. The invocation of a compiled REXX program of CEXEC type requires one SVC more than the invocation of an interpreted REXX program. The maximum SVC nesting depth is reached earlier, for example, in recursive programs.

- **Testing the Halt Condition:** Testing for the halt condition is supported only for programs that are compiled with the TESTHALT Compiler option or use the %TESTHALT directive. See “Halt Condition” on page 91 for details.
- **Tracing compiled programs:** Tracing of compiled programs is supported only for programs that are compiled with the TRACE Compiler option. See “TRACE Instruction and TRACE Built-In Function” on page 96 for details.

Chapter 5. Understanding the Compiler Listing

The Compiler produces a listing for each compilation unless the NOPRINT option was specified. You can print the listing or store it in a z/OS data set or in a CMS file; see the description of the PRINT option at "PRINT" on page 34 for details.

The compiler listing consists of the following items:

- The compilation summary
- The source listing, if the SOURCE option was specified
- Any messages that were produced and that were not suppressed by the FLAG option
- A cross-reference listing, if the XREF option was specified
- The compilation statistics

At the end of this chapter you find an example of a complete compiler listing.

Compilation Summary

The information at the beginning of a compiler listing shows the outcome of the compilation, and the options in effect for the compilation.

The text `Compiled with OPTIONS 'ETMODE'` follows the last compiler option if the program was compiled with ETMODE in effect.

An example of a compilation summary is shown here:

```
1===> Compilation Summary                        ROULETTE EXEC     A1
 IBM Compiler for REXX on System z 4.0  LVL -NONE--    Time: 15:40:14      Date: 2003-05-20                  Page:    1


 Compilation successful

 Compiler Options

      NOALTERNATE
        CEXEC     (ROULETTE CEXEC    A1)                    RECFM=F,LRECL=1024
      NOCOMPILE   (S)
      NOCONDENSE
      NODLINK
      NODUMP
        FLAG      (I)
        FORMAT    (C)
      NOIEXEC
        LIBLEVEL  (*)
        LINECOUNT (55)
        MARGINS   (1 *)
        OBJECT    (ROULETTE TEXT     A1)                    RECFM=F,LRECL=80
      NOOLDDATE
        OPTIMIZE
        PRINT     (ROULETTE LISTING  A1)                    RECFM=V,LRECL=121
      NOSAA
        SLINE     (A)
        SOURCE
        SYSIN     (ROULETTE EXEC     A1)                    RECFM=V,LRECL=99
      NOTERMINAL
      NOTESTHALT
      NOTRACE
        XREF
      Minimum Library Level required: 3

      SLINE(AUTO) in effect, no source lines included
```

Figure 6. Extract of Compiler Listing Showing the Compilation Summary as Printed under z/VM

Source Listing

Figure 7 on page 54 shows an extract from a source listing. You can control the page breaks in this listing by using the %PAGE listing control directive, as described at "Compiler Control Directives" on page 91. Each line of the listing contains the following information:

If The nesting level of IF instructions

Do The nesting level of DO instructions

Sel The nesting level of SELECT instructions

For example, a `2` in the `If` column indicates that the instruction on that line is part of an IF instruction that is nested within another IF instruction.

Line The line number in the expanded source program. Source lines that are longer than the space available in a listing line are split and continued on subsequent lines of the listing. The space available depends on whether sequence numbers, %INCLUDE files, or both, have been found.

C Continuation (`C`) or splitting (`S`) of a line.

- **`C`** Continuation line indicator. Indicates that the source line is longer than the space available and continues on this line.
- **`S`** Split line indicator. The source line has been split as a result of text following the closing `*/` characters terminating a %INCLUDE directive. The `S` is printed to the first split line that follows the included records.

----+----1----+----2----+

Columns of the source ranging from 1 to the number of columns available. If margins are specified, the characters > and < indicate which part of the source has been compiled. The character > is placed one column to the left of the left margin, if this is >1 and fits on the line. The character < is placed one column to the right of the right margin, if it fits on the line. For example, if you specified `MARGINS (5 12)`, the margins indicator shows:

```
--->+----1--<-+----2----+-
```

Sequence

Contains the sequence numbers taken from the records from the main source file and any included files. Sequence numbers are expected in the last eight character positions of the record for fixed-length records and in the first eight character positions for variable-length records. If the source files do not contain sequence numbers, there is no `Sequence` column, but the space is used by the REXX source.

The following examples show the sequence number at the beginning (first example) and at the end (second example) of a record:

```
1===> Source Listing                          VARTEST  EXEC     A1
 IBM Compiler for REXX on System z 4.0  LVL -NONE--    Time: 14:07:17       Date: 2003-05-20                    Page:    2
  If  Do Sel  Line C ----+----1----+----2----+----3----+----4----+----5----+----6----+----7----+----8----+----9- Sequence

                1   /* REXX VARTEST */                                                                          00000000
                2   Exit rc                                                                                     00000002
```

```
1===> Source Listing                          FIXTEST  EXEC     A1
 IBM Compiler for REXX on System z 4.0  LVL -NONE--    Time: 14:07:17       Date: 2003-05-20                    Page:    2
  If  Do Sel  Line C ----+----1----+----2----+----3----+----4----+----5----+----6----+----7----+----8 Sequence Incl  Recd

                1   /* REXX FIXTEST             */                                                  00100000          1
                2   rc=4                                                                            00100100          2
                3   /*%Include include      */                                                     00200000          3
                4   /* REXX INCLUDE COPY A - I am hopefully included          */                    00000001    1     1
                5                                                                                   00300000          4
                6   Exit rc
```

Incl Identifies the file, main or included, from which the line was taken.

If the column contains a blank, the print line is taken from the main REXX source file whose file ID is printed in the first header line of the listing.

A number in this column refers to a %INCLUDE file in the list of included files that is printed in the compilation statistics sublisting. (See Figure 10 on page 59.) This number is a reference number, which does not indicate nesting of included files. The nesting of included files can be derived from the contents of the `Recd` column.

If the source files do not have any included files, there is no `Incl` column, but the space is used by the REXX source.

Recd Number of REXX lines within the main or the included file. The numbering begins with 1 for each file, so that nested files can be recognized by a break in the line number sequence.

If the source files do not have any included files, there is no `Recd` column.

```
1===> Source Listing                              ROUTELETTE EXEC     A1
 IBM Compiler for REXX on System z 4.0  LVL -NONE--    Time: 15:40:14       Date: 2003-05-20                    Page:    2
  If  Do Sel  Line C ----+----1----+----2----+----3----+----4----+----5----+----6----+----7----+----8----+---- Incl  Recd

                1   /* REXX ***************************************************************                          1
                2   *                  Roulette Implementation in REXX                                              2
                3   * This program can be used instead of the wheel usually employed in                            3
                4   * casinos.                                                                                     4
                5   * Press enter to proceed to the game's next step.                                             5
                6   * After the display of a number you can stop play by entering "end".                          6
                7   *                                                                                            7
                8   ********************************************************************/                         8
                9   Call set_color                     /* initialize c.i with color of i */                      9
               10   rr.=0                              /* initialize statistics          */                     10
               11   Say '** Welcome to Roulette **'    /* welcome the user               */                     11
               12   Do Forever                         /* repeat till end requested      */                     12
      1        13     Say                              /* an empty separator line        */                     13
      1        14     Say 'Faites vos jeux'            /* ask players to make their bets */                     14
      1        15     Call pause('W')                  /* wait for input to proceed      */                     15
      1        16     Say 'Rien ne va plus'            /* stop them                      */                     16
      1        17     Call pause('W')                  /* wait for input to proceed      */                     17
      1        18     r=Random(0,36)                   /* get random number from 0 to 36 */                     18
      1        19     rr.r=rr.r+1;                     /* maintain statistics            */                     19
      1        20     If r=0 Then                      /* zero                           */                     20
    1 1        21       Say ' 0 ZERO'                  /* good for the casino            */                     21
      1        22     Else Do                          /* any other number (1 to 36)     */                     22
    1 2        23       If r//2=0 Then                 /* even number                    */                     23
    2 2        24         pi='pair';                   /*   in French                    */                     24
    1 2        25       Else                           /* odd number                     */                     25
    2 2        26         pi='impair';                 /*   in French                    */                     26
    1 2        27       If r<=18  Then                 /* lower half                     */                     27
    2 2        28         mp='manque';                 /*   in French                    */                     28
    1 2        29       Else                           /* upper half                     */                     29
    2 2        30         mp='passe'                   /*   in French                    */                     30
    1 2        31       Say Right(r,2) Left(pi,6) c.r mp  /* show where the ball stopped and the number's att        31
                  C ributes */
    1 1        32       End                            /* and the number's attributes    */                     32
      1        33     If pause('E')='END' Then         /* check if termination request   */                     33
    1 1        34       Leave                          /* If so, end the loop            */                     34
               35     End                              /* end of one game, ready for next*/                     35
               36   Say '     ** Merci et au revoir **' /* thanks and good bye            */                     36
               37   Exit                               /* Exit the program               */                     37
               38                                                                                               38
               39   /*%Include setcolor*/                                                                       39
               40   set_color:          /* Set up c.i to contain the color of each number */                  1     1
               41     c.='noir '                       /* set all of them to black       */                  1     2
               42     rouge='1 3 5 7 9 12 14 16 18 19 21 23 25 27 30 32 34 36'                              1     3
               43     Do While rouge¬=''               /* process list of red numbers    */                  1     4
      1        44       Parse Var rouge t rouge        /* pick the first in the list     */                  1     5
      1        45       c.t='rouge'                    /* set its color to red           */                  1     6
               46       End                                                                                   1     7
               47     Return
```

Figure 7. Extract of Source Listing as Printed under z/VM

Messages

Compiler messages are preceded by the erroneous source line. However, if the error does not occur in the REXX program, for example if there is an incorrect option or an error opening the output file, the error messages precede the first source line. If you request a source listing, the messages are interspersed in the listing, as shown in Figure 8 on page 55. Otherwise, only the erroneous source lines and their corresponding messages are included in the listing.

Notice that there is a vertical bar between the source line and the message line. This marker is placed at or near the part of the instruction in the printed source line, continuation line, or split line that caused the message. One error may cause more than one message.

The result of an expression following an INTERPRET instruction is not analyzed by the Compiler. If it contains errors, they are detected only when the INTERPRET instruction is executed.

```
1===> Source Listing            SYS03140.T154049.RA000.RXTLISTS.SYSINPDS.H01(ROULETTE)
 IBM Compiler for REXX on System z 4.0  LVL -NONE--    Time: 15:40:51       Date: 2003-05-20                  Page:    2
  If  Do Sel  Line C ----+----1----+----2----+----3----+----4----+----5----+----6----+----7----+----8 Sequence Incl  Recd

                 1   /* REXX ***************************************************************00010000                  1
                 2   *                  Roulette Implementation in REXX                     00020000                  2
                 3   * This program can be used instead of the wheel usually employed in     00030000                  3
                 4   * casinos.                                                             00040000                  4
                 5   * Press enter to proceed to the game's next step.                      00050000                  5
                 6   * After the display of a number you can stop play by entering "end".    00060000                  6
                 7   *                                                                     00070000                  7
                 8   ********************************************************************/00080000                  8
                 9   Call set_color                     /* initialize c.i with color of i */00090000                  9
                10   rr.=0                              /* initialize statistics          */00100000                 10
                11   Say '** Welcome to Roulette **'    /* welcome the user               */00110000                 11
                12   Do Forever                         /* repeat till end requested      */00120000                 12
       1        13     Say                             /* an empty separator line        */00130000                 13
       1        14     Say 'Faites vos jeux'           /* ask players to make their bets */00140000                 14
       1        15     Call pause('W')                 /* wait for input to proceed      */00150000                 15
       1        16     Say 'Rien ne va plus'           /* stop them                      */00160000                 16
       1        17     Call pause('W')                 /* wait for input to proceed      */00170000                 17
       1        18     r=Random(0,36)                  /* get random number from 0 to 36 */00180000                 18
       1        19     rr,r=rr.r+1;                    /* maintain statistics            */00190000                 19
                         |
 +++FANPAR0566S Unexpected "," in expression
       1        20     If r=0 Then                     /* zero                           */00200000                 20
   1   1        21       Say ' 0 ZERO'                 /* good for the casino            */00210000                 21
       1        22     Else Do                         /* any other number (1 to 36)     */00220000                 22
   1   2        23       If r//2=0 Then                /* even number                    */00230000                 23
   2   2        24         pi='pair';                  /*   in French                    */00240000                 24
   1   2        25       Else                          /* odd number                     */00250000                 25
   2   2        26         pi='impair';                /*   in French                    */00260000                 26
   1   2        27       If r<=18  Then                /* lower half                     */00270000                 27
   2   2        28         mp='manque';                /*   in French                    */00280000                 28
   1   2        29       Else                          /* upper half                     */00290000                 29
   2   2        30         mp='passe'                  /*   in French                    */00300000                 30
   1   2        31       Say Right(r,2) Left(pi,6) c.r mp  /* show where the ball stopped  */00310000                 31
   1   1        32       End                           /* and the number's attributes    */00320000                 32
       1        33     If pause('E')=      Then        /* check if termination request   */00330000                 33
                            |
 +++FANPAR0561S Right operand missing
   1   1        34       Leave                         /* If so, end the loop            */00340000                 34
                35     End                             /* end of one game, ready for next*/00350000                 35
                36   Say '     ** Merci et au revoir **' /* thanks and good bye           */00360000                 36
                37   Exit                              /* Exit the program               */00370000                 37
                38                                                                          00380000                 38
                39   set_color:                                                           00390000                 39
                40   /*%Include setcolor*/                                                 00400000                 40
                41   set_color:          /* Set up c.i to contain the color of each number */         00010000    1     1
                       |
 +++FANPAR0071W Duplicate label: Only first occurrence on line 39 used
                42     c.='noir '                      /* set all of them to black       */         00020000    1     2
                43     rouge='1 3 5 7 9 12 14 16 18 19 21 23 25 27 30 32 34 36'                    00030000    1     3
                44     Do While rouge¬=''              /* process list of red numbers    */         00040000    1     4
       1        45       Parse Var rouge t rouge       /* pick the first in the list     */         00050000    1     5
1===> Source Listing            SYS03140.T154049.RA000.RXTLISTS.SYSINPDS.H01(ROULETTE)
 IBM Compiler for REXX on System z 4.0  LVL -NONE--    Time: 15:40:51       Date: 2003-05-20                  Page:    3
  If  Do Sel  Line C ----+----1----+----2----+----3----+----4----+----5----+----6----+----7--<-+----8 Sequence Incl  Recd

       1        46       c.t='rouge'                   /* set its color to red           */         00060000    1     6
                47       End                                                                       00070000    1     7
                48     Return
```

Figure 8. Extract of Source Listing with Messages as Printed under z/OS

Cross-Reference Listing

For each item used in a program except for host commands, the cross-reference listing shows:

- The attribute of the item. Because REXX does not require you to declare the type of data to be stored in a variable, the attributes do not indicate formal data types.
- The numbers of the lines on which it is referenced in the program.

Note: If you do not want to list constants, commands, and optimizing stoppers, specify the XREF(S) compiler option as described in "XREF" on page 37.

Each entry in the cross-reference listing contains the following information:

Item The text of the item. Symbols are shown in uppercase, except for DBCS

characters. Literal strings are shown enclosed in single quotes. If the text is longer than 30 characters, the rest of the text is continued on subsequent lines of the listing.

Attribute
The attribute of the item, according to the classification of tokens defined in REXX. The meanings of the values in this column are:

BIN STR
A binary string

BUILT-IN
A built-in function

COMP VAR
A compound variable

CONST SYM
A constant symbol

DBCS RTN
A function for manipulating DBCS strings

EXT BIF
A stream I/O built-in function

EXT RTN
An external routine

HEX STR
A hexadecimal string

LABEL A label definition

LABEL +++
A multiple-label definition or a reference to an undefined label

LIT STR
A literal string

NUMBER A number

SIMP VAR
A simple variable

SIMPV+++
A variable that is not initialized. It does not have an assignment.

STEM A stem

SYSTM RTN
A function supplied by IBM that is specific to a system, such as DIAG under z/VM, SYSVAR under z/OS, or ASSIGN under VSE.

Line Reference
The number of each line on which the item is referenced. The meanings of the characters provided in parentheses in the listing are described in the following. You can review line references for:

- Labels:

  (d) Indicates a valid label definition

  (u) Indicates a reference to an undefined label

  (m) Indicates a duplicate label definition

  (c) The label is referred to in a CALL clause

(C) The label is referred to in a CALL ON clause

(s) The label is referred to in a SIGNAL clause

(S) The label is referred to in a SIGNAL ON clause

(f) The label is referred to as a function call.

- Variables:

  (s) Sets the variable named in the ITEM column

  (d) Indicates that the variable was dropped

  (e) Indicates that the variable was exposed

  (SIMPV+++)
  Indicates that the variable is not initialized. It does not have an assignment.

Figure 9 on page 58 shows the cross-reference listing for the **ROULETTE EXEC** in Figure 8 on page 55.

```
1===> Cross Reference Listing                    ROULETTE EXEC     A1
 IBM Compiler for REXX on System z 4.0  LVL -NONE--    Time: 15:40:14       Date: 2003-05-20                    Page:    3
 Item                          Attribute Line References


 ----- Labels, Built-in Functions, External Routines -----

 LEFT                          BUILT-IN  31:20(f)
 PAUSE                         EXT RTN   15:8(c)  17:8(c)  33:6(f)
 RANDOM                        BUILT-IN  18:5(f)
 RIGHT                         BUILT-IN  31:9(f)
 SET_COLOR                     LABEL      9:6(c)  40:1(d)

 ----- Constants -----

 ''                            LIT STR   43:19
 '     ** Merci et au revoir ** LIT STR   36:5
 '
 ' 0 ZERO'                     LIT STR   21:9
 '** Welcome to Roulette **'   LIT STR   11:5
 'impair'                      LIT STR   26:10
 'manque'                      LIT STR   28:10
 'noir '                       LIT STR   41:6
 'pair'                        LIT STR   24:10
 'passe'                       LIT STR   30:10
 'rouge'                       LIT STR   45:9
 'E'                           LIT STR   33:12
 'END'                         LIT STR   33:17
 'Faites vos jeux'             LIT STR   14:7
 'Rien ne va plus'             LIT STR   16:7
 'W'                           LIT STR   15:14    17:14
 0                             NUMBER    10:5     18:12    20:8     23:13
 1                             NUMBER    19:13
 '1 3 5 7 9 12 14 16 18 19 21 2 LIT STR   42:9
 3 25 27 30 32 34 36'
 18                            NUMBER    27:11
 2                             NUMBER    23:11    31:17
 36                            NUMBER    18:14
 6                             NUMBER    31:28

 ----- Simple Variables -----

 MP                            SIMP VAR  28:7(s)  30:7(s)  31:35
 PI                            SIMP VAR  24:7(s)  26:7(s)  31:25
 R                             SIMP VAR  18:3(s)  19:6     19:11    20:6     23:8     27:8     31:15    31:33
 ROUGE                         SIMP VAR  42:3(s)  43:12    44:15    44:23(s)
 T                             SIMP VAR  44:21(s) 45:7

 ----- Stems and Compound Variables -----

 C.                            STEM      41:3(s)
   C.R                         COMP VAR  31:31
   C.T                         COMP VAR  45:5(s)
 RR.                           STEM      10:1(s)
   RR.R                        COMP VAR  19:3(s)  19:8
1                                         ROULETTE EXEC     A1
 IBM Compiler for REXX on System z 4.0  LVL -NONE--    Time: 15:40:14       Date: 2003-05-20                     Page:    4
  If  Do Sel  Line C ----+----1----+----2----+----3----+----4----+----5----+----6----+----7----+----8----+---- Incl  Recd


 ----- Optimizing Stoppers -----

                9   Call set_color                    /* initialize c.i with color of i */                          9
                      |
      1        15    Call pause('W')                   /* wait for input to proceed      */                         15
                     |
      1        17    Call pause('W')                   /* wait for input to proceed      */                         17
                     |
      1        33    If pause('E')='END' Then          /* check if termination request   */                         33
                       |
               40   set_color:          /* Set up c.i to contain the color of each number */                    1     1
                    |
```

Figure 9. Extract of Cross-Reference Listing as Printed under z/VM

Compilation Statistics

The compilation statistics at the end of the source listing provide the following information:

- Number of lines in the source program
- Size of the compiled program in bytes, if compiled code was generated
- Message statistics
- Flagged source lines, if any source lines were flagged
- List of included z/OS data set names or CMS file names, if any %INCLUDE directives are found

Note: The message statistics and the flagged source lines are produced regardless of the FLAG compiler-option setting. For more information refer to "FLAG" on page 26.

An example of compilation statistics is shown in Figure 10. The numbers indicate how many messages were produced for each particular message severity.

```
1===> Compilation Statistics     SYS03140.T154049.RA000.RXTLISTS.SYSINPDS.H01(ROULETTE)
 IBM Compiler for REXX on System z 4.0  LVL -NONE--    Time: 15:40:51       Date: 2003-05-20                  Page:    6


 REXX Lines  48

 Total messages   Informational   Warning   Error   Severe   Terminating
              3               0         1       0        2             0

 The following lines have been flagged

  19:5 33:16 41:1

 Error No.  Line:Col

        71  41:1

       561  33:16

       566  19:5

 Included files
       1  RXT.FB80.PEXEC(SETCOLOR)                          RECFM=FB,LRECL=80,BLKSIZE=6160

 Finishing time of compilation:  15:40:51
```

Figure 10. Extract of Compiler Listing Showing Compilation Statistics as Printed under z/OS

Examples with Column Numbers

The following examples show the Compiler listings where FORMAT(C) and option SAA are in effect. The column numbers and the line numbers appear in the cross-reference listing of the variables and in the statistics listing that contains the flagged lines. The line numbers precede, and the column numbers follow, the colon (:) sign.

The program to be compiled also contains several host commands. They are printed in the cross-reference listing in the same format and sequence as in the source listing. Appendix D, "The z/OS Cataloged Procedures Supplied by IBM," on page 231 contains another version of this program without errors.

Figure 11 on page 60 shows an extract of a source listing printed under z/VM.

```
1===> Source Listing              SYS03140.T160625.RA000.RXTLISTS.SYSINPDS.H01(MVS2OE)
 IBM Compiler for REXX on System z 4.0  LVL -NONE--    Time: 16:06:28       Date: 2003-05-20                    Page:    2
  If  Do Sel  Line C ----+----1----+----2----+----3----+----4----+----5----+----6----+----7----+----8----+----9----+----0

                 1   /* ------------------------------ REXX ------------------------------ */
                 2   /*                                MVS2OE                               */
                 3   /*               Copy an MVS data set to OpenEdition                   */
                 4   /*                                                                     */
                 5   /* MVS2OE:  This EXEC will copy a sequential data set or a member in  */
                 6   /* a library to OpenEdition.  It will run in a TSO environment.       */
                 7   /* However this version is designed to illustrate how the REXX        */
                 8   /* Compiler lists the deliberate errors found herein. It is not       */
                 9   /* meant to run this example.                                         */
                10   /*********************************************************************/
                11
                12   /* try to retrieve previous values                                    */
                13   address ISPEXXXXXXXXXXXXEC "VGET (OEDSN,OEPATH,OEBIN)"
                                 |
 +++FANGAO0583S Environment name longer than 8 characters
                14   if (rc = 0) then do                    /* vget o.k., confirm values */
   1   1        15     say 'MVS data set name';    oedsn  = check(oedsn)
   1   1        16     say 'OE path name';         oepath = check(oepath, 'lower')
   1   1        17     say 'Binary file (Y or N)'; oebin  = check(oebin)
   1            18   end
                19   else do                               /* vget not o.k., read in values */
   1   1        20     say 'please key in the complete DSNAME with High Level Qualifier
                           |
 +++FANPAR0855W SAA: Literal strings must be completely on one line
   1   1        21         '
   1   1        22     pull oedsn
   1   1        23     say 'please key in the OE path'
   1   1        24     parse pull oepath
   1   1        25     say 'is it an executable (binary) program (Y or N)?'
   1   1        26     pull oebin
   1            27   end
                28
                29   say 'Abort run? "Y" aborts, anything else performs copy'
                30   say 'from' oedsn 'to' oepath
                31   pull answer
                32   if (answer = 'Y') then exit
                33
                34   if (oebin = 'Y') then DO   /* set up some of the file's OE attributes */
   1   1        35     mode == 'SIXUSR'
                          |
 +++FANPAR0182S Assignment operator must not be followed by another "="
   1   1        36     bin == 'BINARY'
                           |
 +++FANPAR0182S Assignment operator must not be followed by another "="
   1            37   end
                38   else do
   1   1        39     mode = ''
   1   1        40     bin = 'TEXT'
   1            41   end
                42
                43   msg_status = msg('OFF')              /* suppress msgs from FREE etc. */
                                  |
 +++FANGAO0857W SAA: Built-in function not part of SAA Procedures Language
                44   "FREE  DDNAME(OEIN)"             /* make sure OEIN and OEOUT are free */
                45   "FREE  DDNAME(OEOUT)"
                46   msg_status = msg(msg_status)            /* restore to previous value */
                                  |
```

Figure 11. Extract of Source Listing as Printed under z/VM (Part 1 of 2)

```
 +++FANGAO0857W SAA: Built-in function not part of SAA Procedures Language
                47
                48   "ALLOC DDNAME(OEIN)  DSN('"oedsn"') SHR"
                49   "ALLOC DDNAME(OEOUT) PATH('"oepath"') PATHDISP(KEEP KEEP)" ,
                50                "PATHOPTS(ORDWR OCREAT) PATHMODE(SIRUSR SIWUSR" mode")"
                51
                52   "OCOPY INDD(OEIN) OUTDD(OEOUT)" bin         /* perform copy operation */
                53   if (rc <> 0) then say 'RC from OCOPY=' rc        /* check return code */
                54   "FREE  DDNAME(OEIN)"
                55   "FREE  DDNAME(OEOUT)"
                56
                57   /* save values for next invocation                                    */
                58   address ISPEXEC "VPUT (OEDSN,OEPATH,OEBIN) PROFILE"
                59   exit 0                                          /* leave this exec */
                60
                61   /* subprogram to request user to confirm or overwrite a value         */
                62   /* -----------------------------------------------------------        */
                63   check:
                64    say 'Use <ENTER> to use' arg(1) 'or key in new value'
1===> Source Listing              SYS03140.T160625.RA000.RXTLISTS.SYSINPDS.H01(MVS2OE)
 IBM Compiler for REXX on System z 4.0  LVL -NONE--    Time: 16:06:28       Date: 2003-05-20                    Page:    3
  If  Do Sel  Line C ----+----1----+----2----+----3----+----4----+----5----+----6----+----7----+----8----+----9----+----0

                65    if (arg(2) = 'lower') then do
   1   1        66      parse pull answer                       /* keep case as typed in */
   1            67    end
                68    else do
   1   1        69      parse upper pull answer                       /* uppercase input */
   1            70    end
                71    if (answer = '') then return arg(1); else return answer
                72    Say 'end of program'
                      |
 +++FANGAO0773I Instruction might never be executed
```

Figure 12. Extract of Source Listing as Printed under z/VM (Part 2 of 2)

Figure 13 on page 62 shows an extract of a cross-reference listing printed under z/OS.

```
1===> Cross Reference Listing     SYS03140.T160625.RA000.RXTLISTS.SYSINPDS.H01(MVS2OE)
 IBM Compiler for REXX on System z 4.0  LVL -NONE--    Time: 16:06:28       Date: 2003-05-20                  Page:    4
 Item                           Attribute Line References


 ----- Labels, Built-in Functions, External Routines -----

 ARG                            BUILT-IN  64:27(f) 65:6(f)  71:31(f)
 CHECK                          LABEL     15:40(f) 16:40(f) 17:40(f) 63:1(d)
 MSG                            SYSTM RTN 43:14(f) 46:14(f)

 ----- Constants -----

 ''                             LIT STR   39:10    71:15
 ')'                            LIT STR   50:67
 ''') PATHDISP(KEEP KEEP)'      LIT STR   49:35
 ''') SHR'                      LIT STR   48:33
 'end of program'               LIT STR   72:6
 'from'                         LIT STR   30:5
 'is it an executable (binary)  LIT STR   25:7
 program (Y or N)?'
 'lower'                        LIT STR   16:54    65:15
 'or key in new value'          LIT STR   64:34
 'please key in the complete DS LIT STR   20:7
 NAME with High Level Qualifier
                  '
 'please key in the OE path'    LIT STR   23:7
 'to'                           LIT STR   30:18
 'Abort run? "Y" aborts, anythi LIT STR   29:5
 ng else performs copy'
 'ALLOC DDNAME(OEIN)  DSN('''   LIT STR   48:1
 'ALLOC DDNAME(OEOUT) PATH('''  LIT STR   49:1
 'Binary file (Y or N)'         LIT STR   17:7
 'BINARY'                       LIT STR   36:10
 'FREE  DDNAME(OEIN)'           LIT STR   44:1     54:1
 'FREE  DDNAME(OEOUT)'          LIT STR   45:1     55:1
 ISPEXEC                        CONST SYM 58:9
 ISPEXXXXXXXXXXXXEC             CONST SYM 13:9
 'MVS data set name'            LIT STR   15:7
 'OCOPY INDD(OEIN) OUTDD(OEOUT) LIT STR   52:1
 '
 'OE path name'                 LIT STR   16:7
 'OFF'                          LIT STR   43:18
 'PATHOPTS(ORDWR OCREAT) PATHMO LIT STR   50:15
 DE(SIRUSR SIWUSR'
 'RC from OCOPY='               LIT STR   53:23
 'SIXUSR'                       LIT STR   35:11
 'TEXT'                         LIT STR   40:9
 'Use <ENTER> to use'           LIT STR   64:6
 'VGET (OEDSN,OEPATH,OEBIN)'    LIT STR   13:28
 'VPUT (OEDSN,OEPATH,OEBIN) PRO LIT STR   58:17
 FILE'
 'Y'                            LIT STR   32:14    34:13
 0                              NUMBER    14:10    53:11    59:6
 1                              NUMBER    64:31    71:35
 2                              NUMBER    65:10

 ----- Simple Variables -----

 ANSWER                         SIMP VAR  31:6(s)  32:5     66:15(s) 69:21(s) 71:6     71:51
 BIN                            SIMP VAR  36:3(s)  40:3(s)  52:33
 MODE                           SIMP VAR  35:3(s)  39:3(s)  50:63
 MSG_STATUS                     SIMP VAR  43:1(s)  46:1(s)  46:18
 OEBIN                          SIMP VAR  17:31(s) 17:46    26:8(s)  34:5
 OEDSN                          SIMP VAR  15:31(s) 15:46    22:8(s)  30:12    48:28
 OEPATH                         SIMP VAR  16:31(s) 16:46    24:14(s) 30:23    49:29
 RC                             SIMP VAR  14:5     53:5     53:40
```

Figure 13. Extract of Cross-Reference Listing as Printed under z/OS (Part 1 of 2)

```
 ----- Commands -----
  If  Do Sel  Line C ----+----1----+----2----+----3----+----4----+----5----+----6----+----7----+----8----+----9----+----0

                13   address ISPEXXXXXXXXXXXXEC "VGET (OEDSN,OEPATH,OEBIN)"
                44   "FREE  DDNAME(OEIN)"            /* make sure OEIN and OEOUT are free */
                45   "FREE  DDNAME(OEOUT)"
                48   "ALLOC DDNAME(OEIN)  DSN('"oedsn"') SHR"
                49   "ALLOC DDNAME(OEOUT) PATH('"oepath"') PATHDISP(KEEP KEEP)" ,
                50                "PATHOPTS(ORDWR OCREAT) PATHMODE(SIRUSR SIWUSR" mode")"
                52   "OCOPY INDD(OEIN) OUTDD(OEOUT)" bin         /* perform copy operation */
                54   "FREE  DDNAME(OEIN)"
                55   "FREE  DDNAME(OEOUT)"
1                           SYS03140.T160625.RA000.RXTLISTS.SYSINPDS.H01(MVS2OE)
 IBM Compiler for REXX on System z 4.0  LVL -NONE--    Time: 16:06:28       Date: 2003-05-20                    Page:    5
  If  Do Sel  Line C ----+----1----+----2----+----3----+----4----+----5----+----6----+----7----+----8----+----9----+----0

                58   address ISPEXEC "VPUT (OEDSN,OEPATH,OEBIN) PROFILE"

 ----- Optimizing Stoppers -----

                13   address ISPEXXXXXXXXXXXXEC "VGET (OEDSN,OEPATH,OEBIN)"
                     |
   1   1        15     say 'MVS data set name';    oedsn  = check(oedsn)
                                                     |
   1   1        16     say 'OE path name';         oepath = check(oepath, 'lower')
                                                     |
   1   1        17     say 'Binary file (Y or N)'; oebin  = check(oebin)
                                                     |
                43   msg_status = msg('OFF')              /* suppress msgs from FREE etc. */
                               |
                44   "FREE  DDNAME(OEIN)"             /* make sure OEIN and OEOUT are free */
                     |
                45   "FREE  DDNAME(OEOUT)"
                     |
                46   msg_status = msg(msg_status)            /* restore to previous value */
                               |
                48   "ALLOC DDNAME(OEIN)  DSN('"oedsn"') SHR"
                     |
                49   "ALLOC DDNAME(OEOUT) PATH('"oepath"') PATHDISP(KEEP KEEP)" ,
                     |
                52   "OCOPY INDD(OEIN) OUTDD(OEOUT)" bin         /* perform copy operation */
                     |
                54   "FREE  DDNAME(OEIN)"
                     |
                55   "FREE  DDNAME(OEOUT)"
                     |
                58   address ISPEXEC "VPUT (OEDSN,OEPATH,OEBIN) PROFILE"
                     |
                63   check:
                     |
```

Figure 14. Extract of Cross-Reference Listing as Printed under z/OS (Part 2 of 2)

Figure 15 shows an extract of a statistics listing that was created with the FORMAT compiler option.
Figure 16 on page 64 shows an extract of a statistics listing that was created with

```
1===> Compilation Statistics      SYS03140.T160625.RA000.RXTLISTS.SYSINPDS.H01(MVS2OE)
 IBM Compiler for REXX on System z 4.0  LVL -NONE--    Time: 16:06:28       Date: 2003-05-20                    Page:    6


 REXX Lines  72

 Total messages   Informational   Warning   Error   Severe   Terminating
              7               1         3       0        3             0

 The following lines have been flagged

  13:9 20:7 35:8 36:7 43:14 46:14 72:2

 Error No.  Line:Col

       182  35:8 36:7

       583  13:9

       773  72:2

       855  20:7

       857  43:14 46:14


 Finishing time of compilation:  16:06:28
```

Figure 15. Extract of Statistics Listing as Printed under z/VM (using FORMAT)

the NOFORMAT compiler option.

```
1===> Compilation Statistics      SYS03140.T161009.RA000.RXTLISTS.SYSINPDS.H01(MVS2OE)
 IBM Compiler for REXX on System z 4.0  LVL -NONE--    Time: 16:10:13       Date: 2003-05-20                   Page:    6


 REXX Lines  72

 Total messages   Informational   Warning   Error   Severe   Terminating
             7               1         3       0        3             0

 The following lines have been flagged

  13 20 35 36 43 46 72

 Error No.  Line

       182  35 36

       583  13

       773  72

       855  20

       857  43 46


 Finishing time of compilation:  16:10:13
```

Figure 16. Extract of Statistics Listing as Printed under z/VM (using NOFORMAT)

Example of a Complete Compiler Listing

```
1===> Compilation Summary        SYS03175.T171043.RA000.RXTLISTS.SYSINPDS.H01(CMPEXAMP)
 IBM Compiler for REXX on System z 4.0  LVL -NONE--    Time: 17:10:47       Date: 2003-06-24                   Page:    1


 13 message(s) reported. Highest severity code was 12 - Severe

 Compiler Options

      NOALTERNATE
        CEXEC     (SYS03175.T171043.RA000.RXTLISTS.SYSCEXEC.H01(CMPEXAMP))
      NOCOMPILE   (S)
      NOCONDENSE
      NODDNAMES
      NODLINK
      NODUMP
        FLAG      (I)
      NOFORMAT
      NOIEXEC
        LIBLEVEL  (*)
        LINECOUNT (0)
        MARGINS   (1 *)
        OBJECT    (SYS03175.T171043.RA000.RXTLISTS.SYSPUNCH.H01(CMPEXAMP))
        OPTIMIZE
        PRINT     ()                                        RECFM=VBA,LRECL=125,BLKSIZE=1250
      NOSAA
        SLINE     (A)
        SOURCE
        SYSIN     (SYS03175.T171043.RA000.RXTLISTS.SYSINPDS.H01(CMPEXAMP)) RECFM=VB,LRECL=4092,BLKSIZE=4096
      NOTERMINAL
      NOTESTHALT
      NOTRACE
        XREF
      Minimum Library Level required: N/A

      SLINE(AUTO) in effect, no source lines included
```

Figure 17. A Complete Compiler Listing as Printed under z/OS (Part 1 of 6)

```
1===> Source Listing             SYS03175.T171043.RA000.RXTLISTS.SYSINPDS.H01(CMPEXAMP)
 IBM Compiler for REXX on System z 4.0  LVL -NONE--    Time: 17:10:47       Date: 2003-06-24                    Page:    2
  If  Do Sel  Line C ----+----1----+----2----+----3----+----4----+----5----+----6----+----7----+----8----+----9----+----0

 +++FANENV0673S LINECOUNT value not 0 or a whole number in the range 10-99: LC(100)

                 1   /* REXX ***************************************************************
                 2   * Name    : CMPEXAMP  EXEC
                 3   * Purpose : Example for a Compilation Listing
                 4   *********************************************************************/
                 5   Parse source . . exfn exft exfm synfn .;
                 6   Parse version all_ver;
                 7
                 8   /* Call - Signal - Function  or multiple                            */
                 9   id=0001; Call   L01;
                10   id=0002; x    = L02(1);
                11   id=0003; Signal L03;
                12   L01:;
                13     say 'id0001' Call L01;
                14     Return;
                15   L02:;
                16     Arg num . ;
                17     Select;
           1    18        When num = 1 then y = 'Test1';
           1    19        When num = 2 then y = 'Test2';
           1    20        Otherwise y = 'no';
                21     End;
                22     say 'id0002 Function L02 returns' y;
                23     Return y;
                24   L03:;                              /* first occurrence of label     */
                25   L03:;                              /* second occurrence of label    */
                     |
 +++FANPAR0071W Duplicate label: Only first occurrence on line 24 used
                26     say 'id0003' Signal L03;
                27
                28   /* compound variables  ********************************************/
                29   id=0010; stema.='';
                30   id=0011; stema.tail1 = L02(1);
                31   id=0012; stema.tail2 = L02(2);
                32   id=0013; stema.tail3 ,
                33                        = L02(2);
                34   id=0014; stema.tail4.aaaaaaaa.bbbbbbbbbbbbbbbbbbbbbbbbbbb.5=0;
                35            /* CV > 250 */
                36   id=0015; stema.tail9.00000001.00000002.00000003.00000004.00000005.00000006.00000
                37            /* 240 < CV > 250 */
                38   id=0016; stema.taila.00000001.00000002.00000003.00000004.00000005.00000006.00000
                39   id=0017; stema.tAILb.GRkl = 'test';
                40   id=0018; stema.tailc = ,
                41                        ;
                42   id=0019; Say stema.taild..00000002.;
                43   id=0020; stemb.tail1 = stema.tail1;
                44
                45   /* flagged lines (multiple)                                        */
                46   id=0021; RANDOM(20,10,4,5)  DATE('X');
                              |
 +++FANGAO0770S Invalid number of arguments in built-in function
                             |
 +++FANGAO0868S RANDOM() BIF: either min>max or (max-min)>100000
                                                  |
 +++FANGAO0866S Invalid option in built-in function invocation
                47   id=0022; U =  'A' / 'B';
                                   |
 +++FANGAO0659S Nonnumeric term
                                        |
 +++FANGAO0659S Nonnumeric term
                48
                49   /* Created a long list of errors                                   */
                50   id=0023;
                51     Say MIN(33,55,'l');
                                     |
```

Figure 18. A Complete Compiler Listing as Printed under z/OS (Part 2 of 6)

```
+++FANGAO0659S Nonnumeric term
               52
               53   /* Error in column 149                                          */
               54   id=0024;
               55
               56   /* drop                                                         */
               57   id=0025; vars = 'stema.tail1 stema.taila'; drop (vars); stema.tail1 = 1;
               58
                  C                             Say MIN(33,55,'l');
                                                                  |
+++FANGAO0659S Nonnumeric term
               59   /* Verify some DATE(5)                                          */
               60   id=0026;
               61     Say DATE('N','1 Jan 1000','X');
                                                 |
+++FANGAO0866S Invalid option in built-in function invocation
               62     Say DATE('X','1 Jan 1000','N');
                             |
+++FANGAO0866S Invalid option in built-in function invocation
               63     Say DATE('N','1 Jan 1000','N','-','q');
                                                      |
+++FANGAO0866S Invalid option in built-in function invocation
               64     Say DATE('E','01/01/1000','U','-','//');
                                                       |
+++FANGAO0879S Separator arg (4 or 5) of DATE exceeds one character
1===> Cross Reference Listing    SYS03175.T171043.RA000.RXTLISTS.SYSINPDS.H01(CMPEXAMP)
 IBM Compiler for REXX on System z 4.0  LVL -NONE--    Time: 17:10:47       Date: 2003-06-24                    Page:    3
 Item                          Attribute Line References


 ----- Labels, Built-in Functions, External Routines -----

 DATE                          BUILT-IN  46(f) 61(f) 62(f) 63(f) 64(f)
 L01                           LABEL      9(c) 12(d)
 L02                           LABEL     10(f) 15(d) 30(f) 31(f) 33(f)
 L03                           LABEL+++  11(s) 24(d) 25(m)
 MIN                           BUILT-IN  51(f) 58(f)
 RANDOM                        BUILT-IN  46(f)
```

Figure 19. A Complete Compiler Listing as Printed under z/OS (Part 3 of 6)

```
----- Constants -----

 ''                             LIT STR   29
 '-'                            LIT STR   63    64
 '//'                           LIT STR   64
 'id0001'                       LIT STR   13
 'id0002 Function L02 returns'  LIT STR   22
 'id0003'                       LIT STR   26
 'l'                            LIT STR   51    58
 'no'                           LIT STR   20
 'q'                            LIT STR   63
 'stema.tail1 stema.taila'      LIT STR   57
 'test'                         LIT STR   39
 'A'                            LIT STR   47
 'B'                            LIT STR   47
 'E'                            LIT STR   64
 'N'                            LIT STR   61    62    63    63
 'Test1'                        LIT STR   18
 'Test2'                        LIT STR   19
 'U'                            LIT STR   64
 'X'                            LIT STR   46    61    62
 0                              NUMBER    34
 00000                          NUMBER    36    38
 00000001                       NUMBER    36    38
 00000002                       NUMBER    36    38    42
 00000003                       NUMBER    36    38
 00000004                       NUMBER    36    38
 00000005                       NUMBER    36    38
 00000006                       NUMBER    36    38
 0001                           NUMBER     9
 0002                           NUMBER    10
 0003                           NUMBER    11
 0010                           NUMBER    29
 0011                           NUMBER    30
 0012                           NUMBER    31
 0013                           NUMBER    32
 0014                           NUMBER    34
 0015                           NUMBER    36
 0016                           NUMBER    38
 0017                           NUMBER    39
 0018                           NUMBER    40
 0019                           NUMBER    42
 0020                           NUMBER    43
 0021                           NUMBER    46
 0022                           NUMBER    47
 0023                           NUMBER    50
 0024                           NUMBER    54
 0025                           NUMBER    57
 0026                           NUMBER    60
 '01/01/1000'                   LIT STR   64
 1                              NUMBER    10    18    30    57
 '1 Jan 1000'                   LIT STR   61    62    63
 10                             NUMBER    46
 2                              NUMBER    19    31    33
 20                             NUMBER    46
 33                             NUMBER    51    58
 4                              NUMBER    46
 5                              NUMBER    34    46
 55                             NUMBER    51    58
```

Figure 20. A Complete Compiler Listing as Printed under z/OS (Part 4 of 6)

```
----- Simple Variables -----

AAAAAAAA                        SIMPV+++  34
ALL_VER                         SIMP VAR   6(s)
BBBBBBBBBBBBBBBBBBBBBBBBBBB     SIMPV+++  34
CALL                            SIMPV+++  13
EXFM                            SIMP VAR   5(s)
EXFN                            SIMP VAR   5(s)
EXFT                            SIMP VAR   5(s)
GRKL                            SIMPV+++  39
ID                              SIMP VAR   9(s) 10(s) 11(s) 29(s) 30(s) 31(s) 32(s) 34(s) 36(s) 38(s) 39(s) 40(s) 42(s)
                                           43(s) 46(s) 47(s) 50(s) 54(s) 57(s) 60(s)
L01                             SIMPV+++  13
L03                             SIMPV+++  26
NUM                             SIMP VAR  16(s) 18    19
SIGNAL                          SIMPV+++  26
SYNFN                           SIMP VAR   5(s)
TAILA                           SIMPV+++  38
TAILB                           SIMPV+++  39
TAILC                           SIMPV+++  40
TAILD                           SIMPV+++  42
TAIL1                           SIMPV+++  30    43    43    57
TAIL2                           SIMPV+++  31
TAIL3                           SIMPV+++  32
TAIL4                           SIMPV+++  34
TAIL9                           SIMPV+++  36
U                               SIMP VAR  47(s)
VARS                            SIMP VAR  57(s) 57
X                               SIMP VAR  10(s)
Y                               SIMP VAR  18(s) 19(s) 20(s) 22    23

----- Stems and Compound Variables -----

STEMA.                          STEM      29(s)
  STEMA.TAILA.00000001.0000000 COMP VAR  38
    2.00000003.00000004.000000
    05.00000006.00000
  STEMA.TAILB.GRKL              COMP VAR  39(s)
  STEMA.TAILC                   COMP VAR  40(s)
  STEMA.TAILD..00000002.        COMP VAR  42
  STEMA.TAIL1                   COMP VAR  30(s) 43    57(s)
  STEMA.TAIL2                   COMP VAR  31(s)
  STEMA.TAIL3                   COMP VAR  32(s)
  STEMA.TAIL4.AAAAAAAA.BBBBBBB COMP VAR  34(s)
    BBBBBBBBBBBBBBBBBBB.5
  STEMA.TAIL9.00000001.0000000 COMP VAR  36
    2.00000003.00000004.000000
    05.00000006.00000
STEMB.                          STEM
  STEMB.TAIL1                   COMP VAR  43(s)
```

Figure 21. A Complete Compiler Listing as Printed under z/OS (Part 5 of 6)

```
----- Commands -----
 If  Do Sel  Line C ----+----1----+----2----+----3----+----4----+----5----+----6----+----7----+----8----+----9----+----0

               36   id=0015; stema.tail9.00000001.00000002.00000003.00000004.00000005.00000006.00000
               38   id=0016; stema.taila.00000001.00000002.00000003.00000004.00000005.00000006.00000
               46   id=0021; RANDOM(20,10,4,5)  DATE('X');

----- Optimizing Stoppers -----

                9   id=0001; Call   L01;
                                    |
               10   id=0002; x    = L02(1);
                                    |
               12   L01:;
                    |
               15   L02:;
                    |
               24   L03:;                              /* first occurrence of label     */
                    |
               30   id=0011; stema.tail1 = L02(1);
                                         |
               31   id=0012; stema.tail2 = L02(2);
                                         |
               33                       = L02(2);
                                         |
               36   id=0015; stema.tail9.00000001.00000002.00000003.00000004.00000005.00000006.00000
                    |
               38   id=0016; stema.taila.00000001.00000002.00000003.00000004.00000005.00000006.00000
                    |
               46   id=0021; RANDOM(20,10,4,5)  DATE('X');
                    |
               57   id=0025; vars = 'stema.tail1 stema.taila'; drop (vars); stema.tail1 = 1;
                                                                            |
1===> Compilation Statistics     SYS03175.T171043.RA000.RXTLISTS.SYSINPDS.H01(CMPEXAMP)
 IBM Compiler for REXX on System z 4.0  LVL -NONE--    Time: 17:10:47       Date: 2003-06-24                    Page:    4


 REXX Lines  64

 Total messages   Informational   Warning   Error   Severe   Terminating
             13               0         1       0       12             0

 The following lines have been flagged

  25 46 47 51 58 61 62 63 64

 Error No.  Line

        71  25

       659  47 51 58

       673  0

       770  46

       866  46 61 62 63

       868  46

       879  64


 Finishing time of compilation:  17:10:47
```

Figure 22. A Complete Compiler Listing as Printed under z/OS (Part 6 of 6)

Chapter 6. Using Object Modules and TEXT Files

This chapter describes circumstances in which you may want to use OBJECT output rather than CEXEC output. It also describes how to generate executable modules from the compiler output generated when you select the OBJECT compiler option.

Initial Considerations

Usually you choose the CEXEC option to compile REXX programs because compiled programs of this type can replace interpreted REXX programs transparently and in all circumstances. However, you may want to consider the OBJECT option for:

- Invoking a REXX program as a command or a program (z/OS)
- Improving the packaging and performance of your application
- Building function packages
- Writing parts of applications in REXX
- Placing programs in a discontiguous saved segment (DCSS) (z/VM)
- Invoking a REXX program from JCL (VSE/ESA)

If you decide to use object output, you may have to:

- Change the invocation of the compiled REXX program if it is invoked by other programs
- Change the processing of the information obtained with the PARSE SOURCE instruction
- Check for storage overlaps with other modules (z/VM)

Whether you run object output or CEXEC output for single programs, you can expect the same runtime performance when the program starts running. The time required to locate and load the program, however, may be different.

Object modules and TEXT files do not contain operating system dependencies, and can, therefore, be moved between operating systems. The generated code and the REXX Library are reentrant and can, therefore, be placed in read-only storage.

Object modules and TEXT files do not normally contain relocation information. If you want to have relocation information, you must generate the object module or TEXT file with the DLINK compiler option. This option enables you to link external functions and subroutines directly to an object module or to a TEXT file. See the compiler option DLINK at “DLINK” on page 24 and “DLINK Example” on page 208.

The name of the TEXT file or the object module in the external symbol dictionary (ESD) record is derived from the name of the input file or input data set when the REXX program is compiled. For z/VM, it is the CMS file name of the input file. For z/OS, it is one of the following:

- The member name of the partitioned input data set
- The last qualifier of the name of the sequential input data set
- Or else, COMPREXX (for example, if the source file is part of the job stream)

To run either type of object code, the Library must be installed on z/VM or z/OS. REXX/VSE must be installed on VSE/ESA. (See Chapter 4, “Runtime Considerations,” on page 45 for information on the use of the Alternate Library.)

Object Modules (z/OS)

Generating load modules: Before you can use an object module, you must link it to the appropriate stub (a stub transforms input parameters into a form understandable by the compiled REXX program). This can be done with the REXXL cataloged procedure supplied by IBM, which is listed under “REXXL (EAGL)” on page 238, with the REXXL EXEC explained in “REXXL (z/OS)” on page 74, or with the REXXC EXEC as described in “Invoking the Compiler with the REXXC (FANC) EXEC” on page 9.

Stubs are provided for the parameter-passing conventions as described in “Stubs” on page 211.

After you have linked the modules to the appropriate stubs, you can use the modules in the same way you use modules of other high-level language compilers.

Note:

1. By default, the link-edit step adds a dollar ($) sign to the beginning of the temporary name. If the name consists of 8 characters, the last character is dropped. That is why you must not use 8-character names that differ only in the eighth character, for load modules that are made up of multiple object modules.

   To avoid renaming it is recommended that you use a %STUB definition as described in “%STUB” on page 41.
2. Compiled programs linked with **RENT** modules located in an **APF** library can cause a system abend in the module **IRXSTAMP**. To avoid this problem, compile the program using the CONDENSE option. The compiled program is uncondensed at runtime and the storage is getmained in the TSO subpool 78 for execution of the program. For information on the CONDENSE compiler option, see “CONDENSE” on page 22.
3. Object modules generated with STUB code terminate abnormally if they are run under z/VM.

Invoking a REXX Program as a Command or a Program

A program linked with the:

- CPPL stub can be invoked as a command under TSO/E. The command is usually found earlier in the search order than the same command executed as either a compiled or interpreted REXX EXEC.
- MVS or the CALLCMD stub enables you to invoke a REXX program just as you would invoke a program written in another high-level language.
- EFPL stub enables you to store an external function or subroutine in a load library, where it is usually earlier in the search order than the same function or subroutine executed as a compiled or interpreted REXX EXEC. It can also be in a function package that is loaded when the environment is initialized. The EFPL stub can also be used with the DLINK option, see “DLINK” on page 24 and “DLINK Example” on page 208 for more information.
- CPPLEFPL stub can be invoked both as a TSO/E command or as a REXX external routine. The CPPLEFPL stub determines whether the REXX program

has been invoked as TSO/E command or as a REXX external routine, then gives control to the compiled REXX program with the appropriate parameters.

- MULTI stub enables you to link-edit and package compiled REXX applications under z/OS. It is recommended that you use this stub, because it combines nearly all of the above stub conventions. For more information refer to “Stubs” on page 211.

Programs and commands can be stored and cached wherever a load module can be stored and cached.

For an example, see “Link-Editing of Object Modules” on page 207.

Improving Packaging and Performance

You can improve packaging as follows:

- If your application includes many REXX programs, you can create one module that contains all the REXX programs. You can package it more compactly, thereby reducing the system load, because the application spends less time searching for and invoking external functions and subroutines. To generate a single module:
  1. Specify the DLINK compiler option when you compile programs that invoke external subroutines and functions whose references are intended to be resolved.
  2. Link-edit the main program with the appropriate stub for the intended invocation.
  3. Link-edit each external subroutine and function with an EFPL stub.
  4. Link-edit all the programs together into a single module.

  For an example, see “DLINK Example” on page 208.
- You can use the %STUB prelink control directive to simplify packaging. The following example shows a "Hello World" REXX application that can be started as a TSO/E command. It is assumed that:
  – The REXX source is named 'HELLO' and resides in the partitioned data set '*upref*.REXX'.
  – The user has partitioned data sets for object and load modules called '*upref*.OBJ' and '*upref*.LOAD'.
  – The REXX Compiler is included in the JOBLIB. If not, you must specify a STEPLIB statement with the data set where the REXX Compiler is located.

  In this example you must only insert a %STUB control directive into the REXX source code. For example, to use the CPPL stub, you must define the following anywhere in the source code and use a simple link step in the JCL:

```
/*%STUB CPPL*/
```

  The compile step remains the same as before:

```
//*-------------------------------------------------------------
//COMPSTEP EXEC PGM=REXXCOMP,PARM='NOCEXEC OBJECT'
//SYSPRINT   DD SYSOUT=*
//SYSIN      DD DSN=upref.REXX(HELLO),DISP=SHR
//SYSPUNCH   DD DSN=upref.OBJ(HELLO),DISP=SHR
//*-------------------------------------------------------------
//LINKSTEP EXEC PGM=HEWL,PARM='LIST,AMODE=31,RMODE=ANY,RENT,MAP'
//SYSUT1     DD UNIT=SYSDA,SPACE=(1024,(200,20))
//SYSPRINT   DD SYSOUT=*
//SYSLMOD    DD DSN=upref.LOAD,DISP=SHR
//SYSLIN     DD DSN=upref.OBJ(HELLO),DISP=SHR
```

The object output of the compile step includes the stub code at the beginning and the names of stub and program code adapted to proper values. Extra INCLUDE or CHANGE cards in the link step are not required.

Note: You can still use the previous linkage method. The name of the new multi-purpose STUB load module is 'EAGSTMP'.

Building Function Packages

The parts of a function package can be written in REXX, compiled, linked with the EFPL stub, and then linked to function packages, in which they must be defined as external routines. See the *TSO/E REXX/MVS: Reference* manual for details about function packages.

Writing Parts of Applications in REXX

You can link-edit load modules that are already link-edited with the appropriate stub with applications written in another programming language. The language used must be able to provide the parameters in one of the supported parameter-passing conventions. Otherwise, you can write your own stub to support the parameter-passing convention of the language in question, modeled after one of the existing stubs. See "Stubs" on page 211 for more information.

REXXL (z/OS)

There are two possible uses of the REXXL command:

- REXXL can be used in batch to create a load module. REXXL generates the control cards for the linkage editor to link together a stub and a compiled REXX program of type OBJECT. The compiled REXX program is read from the data set allocated to SYSIN. The control cards, including the compiled REXX program, are written to a data set allocated to SYSOUT.
- REXXL can be used interactively to create a load module. REXXL links together a stub and the compiled REXX program of type OBJECT and builds a load module. The SYSPRINT output of the linkage editor is stored in a sequential data set with a low-level qualifier of LINKLIST.
- REXXL supports both names, EAGSTMP and MULTI. The existing procedures to link-edit stubs can be used in the same way for the MULTI stub.

See also "Link-Editing of Object Modules" on page 207 for more information.

Enter the REXXL command in the following format:

REXXL *stub obj-data-set-name [load-data-set-name]*

where:

stub Is one of the following:

- A predefined stub name. Refer to "Stubs" on page 211 for a list of stub names.
- A member name. The member will be searched for in the default data set. Refer to "Stubs" on page 211 for a list of member names in the sample data set.
- The name of a partitioned data set including a member name.

obj-data-set-name
Is a partitioned or a sequential data set containing the compiled REXX program of type OBJECT. If it is a partitioned data set, the member name has to be specified.

load-data-set-name
Is the partitioned data set in which the load module will be stored. If the member name is not specified, it defaults to the *csect* name that the Compiler puts in the ESD from the OBJECT output. If *load-data-set-name* is not specified, a default name is used.

Default names of the output data sets:

	Partitioned Data Set pref.cccc.qual(member)	**Sequential Data Set pref.cccc.qual**
load data set name	*upref.cccc.LOAD(csect)*	*upref.cccc.qual.LOAD(csect)*
listing data set name	*upref.cccc.csect.LINKLIST*	*upref.cccc.qual.LINKLIST*

where:

pref and *qual* represent the prefix and the last level qualifier of *obj-data-set-name*. *csect* represents the name that the compiler puts in the ESD from the OBJECT output.

Note:

1. The user's default prefix *upref* (as set by the PROFILE PREFIX command) is used for the output data sets. If the prefix of *obj-data-set-name* is different, it is replaced.
2. If you include a stub at compilation time, you do not need the batch job step for REXXL to create the object output for the final link. You must only run the final link step, which creates the executable load module.

   REXXL detects if a stub was included at compilation time. Due to this, you may keep and run your existing procedures. The arguments for REXXL remain the same. You must still specify the stub name as the first argument. REXXL compares the name specified by the argument with the stub name included in the object. If they do not match, an error message is raised and the program terminates.

TEXT Files (z/VM)

The OBJECT output that the Compiler generates has the same properties as TEXT files that are generated by other high-level language compilers, with the following exceptions:

- The compiled program cannot run in the transient program area (TPA).
- The compiled program cannot be invoked from a program that is running in the TPA.
- A module generated from a TEXT file expects SVC parameter-passing conventions. See Appendix B, “Interface for TEXT Files (z/VM),” on page 221 for additional information. You can invoke such a module as a command from the CMS command line or from a REXX program, but the parameter-passing convention is different from that used by other high-level language compilers.

Generating modules: To generate a relocatable module from a TEXT file, use the LOAD command followed by the **GENMOD** command. For example:

load *progname* **(rldsave**
genmod *progname*

Under CMS Release 5.5 or later, relocatable modules are loaded in free storage, thereby reducing the probability that one module may overwrite part of another module that was invoked by a compiled REXX program.

Improving performance: In the REXX search order for external functions and subroutines, the first step is to search for a program whose name is prefixed with RX and truncated to 8 characters. If this program is invoked many times, you can improve its performance if you:

1. Generate a module from the OBJECT output and name it RX*myprog*.
2. Load the module as a nucleus extension. For example, enter the NUCXLOAD command in the following way:

   nucxload rx*myprog*
3. Invoke the program without the prefix RX. For example:

   call *myprog*
   a=*myprog***()**

   The nucleus extension RX*myprog* is searched for and found first.

Improving packaging: If your application contains a REXX program and several external subroutines, you can create one module that includes all these programs. When you do so, your programs are more compactly packaged, thereby reducing system load, because the application spends less time searching for and invoking external functions and subroutines. You also eliminate the possibility of invoking REXX programs that have the same name but are not part of the application. To generate a single module:

1. Specify the DLINK compiler option when you compile the programs that invoke external subroutines and functions whose references are intended to be resolved.
2. Link together the TEXT files to create one relocatable module. For example:

   load *myprog mysub1 mysub2 mysub3* **(rldsave**
   genmod *myprog*
3. Optionally, load the resulting module as a nucleus extension before it is invoked, to avoid storage overlaps with other programs.

Building function packages: The *VM/ESA REXX/VM: Reference* manual includes a coding example of a function package whose functions are included in the code. You can, however, build a function package in which some or all of the functions are compiled REXX programs of OBJECT type. These functions must be linked to the function package and their names declared as external. Additionally, to find out the size of such a function package, you need to link a dummy external program to the end of the function package.

Writing parts of applications in REXX: You can link a compiled REXX program of OBJECT type to a program written in another language. If the language enables

you to invoke programs that require REXX parameter-passing conventions (see Appendix B, "Interface for TEXT Files (z/VM)," on page 221), you can:
1. Declare the REXX program as an external program.
2. Link the REXX program to the application.
3. Invoke the REXX program from within the application.

Placing programs in a DCSS: You can load TEXT files into a DCSS located above 16 MB in virtual storage. If you decide to do this, you first need to write additional code that attaches the DCSS and identifies the REXX programs residing in the DCSS as nucleus extensions.

Object Modules (VSE/ESA)

Generating phases: Before you can use an object module, you must combine it with the appropriate stub (a stub transforms input parameters into a form understandable by the compiled REXX program), then you must link-edit it to generate a phase.

With the cataloged procedure REXXLINK supplied by IBM, you can create a phase consisting of a single program in one step (see "REXXLINK Cataloged Procedure (VSE/ESA)" on page 79).

To create a phase consisting of multiple programs (if you have used the DLINK compiler option), you must combine each object module with the appropriate stub by means of the cataloged procedure REXXPLNK supplied by IBM (See "REXXPLNK Cataloged Procedure (VSE/ESA)" on page 78). You must then link-edit the resulting object modules in an additional step to generate a phase.

Stubs are provided for the following parameter-passing conventions:

- VSE for invocation by means of VSE JCL.
- EFPL (external function parameter list) for invocation with the REXX CALL instruction or as a function. This must be used when building a function package.

Note: Do not use 8-character names that differ only in the eighth character, for phases that are made of multiple object modules. The eighth character of the program name is lost during the prelink step.

After you have combined the object modules with the appropriate stubs and linked them together, you can use the resulting phases in the same way you use phases of high-level language compilers.

Invoking a REXX program as a phase: A program linked with the VSE stub enables you to invoke a REXX program just as you would invoke a program written in another high-level language.

Improving packaging and performance: If your application includes many REXX programs, you can create one phase that contains all the REXX programs. You can package it more compactly, thereby reducing system load, because the application spends less time searching for and invoking external functions and subroutines. To generate a single phase:

1. Specify the DLINK compiler option when you compile programs on z/OS or z/VM that invoke external subroutines and functions whose references are intended to be resolved.
2. Generate the object module on z/VM or z/OS and send it to VSE/ESA.

3. Use the REXXPLNK cataloged procedure to combine the main program with the appropriate stub for the intended invocation.
4. Use the REXXPLNK cataloged procedure to combine each external subroutine and function with an EFPL stub.
5. Link-edit all the combined object modules together into a single phase.

Building function packages: The parts of a function package can be written in REXX, compiled, combined with the EFPL stub using REXXPLNK, and then linked to the function packages, in which they are defined as external routines. See the *IBM VSE/ESA REXX/VSE Reference* manual for details about function packages.

Writing parts of applications in REXX: You can link-edit the object modules that are already combined with the appropriate stub with other object modules written in another programming language. The language used must be able to provide the parameters in one of the supported parameter-passing conventions. Otherwise, you can write your own stub to support the parameter-passing convention of the language in question, modeled after one of the existing stubs. See "Stubs" on page 225 for more information.

Including a copyright notice in your program: You can provide stubs containing a copyright notice. The stubs supplied by IBM contain comments that show where the copyright notice can be easily added. The member names of the stubs are **EAGSDVSE** and **EAGSDEFP**.

REXXPLNK Cataloged Procedure (VSE/ESA)

The cataloged procedure REXXPLNK builds as output an object module that contains the stub combined with the input object module. The resulting object module can be combined with other object modules to create a phase.

Invoke REXXPLNK in the following format:

```
// EXEC PROC=REXXPLNK,[STUBLIB='lib.sublib',]
                      STUBNAM=mn,
                      INLIB='lib.sublib',
                      INNAME=mn,
                      OUTLIB='lib.sublib',
                      OUTNAME=mn
```

where:

STUBLIB=*'lib.sublib'*
: Is the name of the sublibrary where the stub resides. If stublib is not specified, a default name is assumed. (The default name is set in the cataloged procedure.)

STUBNAM=*mn*
: Is the member name of the stub residing in stublib. Member type is always **OBJ**. You can also use one of the predefined stub names:

  VSE The program is invoked by VSE JCL as a program.

  EFPL The program is invoked as a REXX external routine. This is the default stub name.

  For more information refer to "Stubs" on page 225.

INLIB=*'lib.sublib'*
: Is the name of the sublibrary where the input object module resides.

INNAME=*mn*
Is the member name of the input object module residing in inlib. Member type is always **OBJ**.

OUTLIB=*'lib.sublib'*
Is the name of the sublibrary where the output object module will be stored.

OUTNAME=*mn*
Is the member name of the output object module that will be stored in outlib. Member type is always **OBJ**.

See also "REXXPLNK" on page 241.

REXXLINK Cataloged Procedure (VSE/ESA)

The cataloged procedure REXXLINK is used to create a phase. REXXLINK does the following:

1. Builds as output an object module that contains the stub combined with the input object module
2. Link-edits the resulting object module
3. Catalogs the phase in the sublibrary specified by a LIBDEF PHASE,CATALOG=*lib.sublib* statement

Invoke REXXLINK in the following format:

```
// EXEC PROC=REXXLINK,[STUBLIB='lib.sublib',]
                      STUBNAM=mn,
                      INLIB='lib.sublib',
                      INNAME=mn,
                      OUTLIB='lib.sublib',
                      OUTNAME=mn
                     [,PHASNAM=mn]
```

where:

STUBLIB=*'lib.sublib'*
Is the name of the sublibrary where the stub resides. If stublib is not specified, a default name is assumed. (The default name is set in the cataloged procedure.)

STUBNAM=*mn*
Is the member name of the stub residing in stublib. Member type is always **OBJ**. You also can use one of the predefined stubnames:

VSE The program is invoked by VSE JCL as a program.

EFPL The program is invoked as a REXX external routine. This is the default sub name.

For more information refer to "Stubs" on page 225.

INLIB=*'lib.sublib'*
Is the name of the sublibrary where the input object module resides.

INNAME=*mn*
Is the member name of the input object module residing in inlib. Member type is always **OBJ**.

OUTLIB=*'lib.sublib'*
Is the name of the sublibrary where the output object module will be stored.

OUTNAME=*mn*
Is the member name of the output object module that will be stored in outlib. Member type is always **OBJ**.

PHASNAM=*mn*
Is the member name of the phase that will be cataloged in the sublibrary specified by a LIBDEF PHASE,CATALOG=*lib.sublib* statement. The default member name is that specified in the outname parameter. Member type is always PHASE.

See also "REXXLINK" on page 242.

REXXL Cataloged Procedure (VSE/ESA)

The REXXL EXEC builds as output an object module that contains the stub combined with the input object module. The resulting object module can be link-edited with other object modules to create a phase.

Invoke REXXL in the following format:

```
// EXEC REXX=REXXL,PARM='stublib stubnam inlib inname outlib outname'
```

REXXL can also be called from a REXX program as a subroutine:

```
CALL REXXL 'stublib stubnam inlib inname outlib outname'
```

where:

stublib Is the name of the sublibrary, in the form *lib.sublib*, where the stub resides.

stubnam
Is the member name, in the form *mn*, of the stub residing in *stublib*. Member type is always **OBJ**. You also can use one of the predefined stub names:

VSE The program is invoked by VSE JCL as a program.

EFPL The program is invoked as a REXX external routine.

For more information refer to "Stubs" on page 225.

inlib Is the name of the sublibrary, in the form *lib.sublib*, where the input object module resides.

inname Is the member name, in the form *mn*, of the input object module residing in *inlib*. Member type is always **OBJ**.

outlib Is the name of the sublibrary, in the form *lib.sublib*, where the output object module will be stored.

outname
Is the member name of the output object module, in the form *mn*, that will be stored in *outlib*. Member type is always **OBJ**.

See also "REXXL" on page 243.

Linking External Routines to a REXX Program

A REXX program can invoke external routines by means of either the REXX CALL instruction or a function invocation if a routine of that name is neither an internal routine nor a built-in function. Note that the DBCS routines behave identically to built-in functions in terms of the REXX search order. Whenever an external routine is invoked, the standard REXX search for external routines is performed.

Using the standard REXX search may lead to two problems:

- Invoking external routines frequently may affect performance, because each invocation follows the search order.
- Name conflicts may occur in applications that invoke external routines whose names are identical. The external routine that is earlier in the search order is executed, which is not necessarily what you want to occur.

The DLINK compiler option enables you to create self-contained modules and avoid these problems. You can selectively link external routines to the main program. Alternatively, you can turn the main program into a self-contained module by linking to it all externally referenced routines.

When the DLINK option is specified, the OBJECT output contains references to all external functions and subroutines. These references are in the form of weak external references, which means that during the link-edit or load steps the libraries are not automatically searched to resolve these references.

Under z/OS, the linkage editor resolves the addresses only if you link and load the referenced module with the module containing the external reference.

Under z/VM, the loader resolves the addresses only if you load the referenced modules with the module containing the external reference, or if you bring in the referenced module by means of an INCLUDE command.

Under VSE/ESA, the linkage editor resolves the addresses only if you link and load the referenced object module with the object module containing the external reference. If you do not link and load the referenced object module, the linkage editor ends with return code 4, which indicates unresolved external references.

Resolving External References—An Example

The following example illustrates how to resolve external references selectively. For the purposes of the example, assume the following:

- Your main program is **MYAPPL**; that is:
  TEST.**EXEC(MYAPPL)** under z/OS
  MYAPPL EXEC under z/VM
- Your main program contains a call to your external routine **MYEXTR**; that is:
  TEST.**EXEC(MYEXTR)** under z/OS
  MYEXTR EXEC under z/VM

  It also contains a call to the external routine **OTHRPROG** contained in some function package.

  Note: If you are working on VSE/ESA, **MYAPPL** and **MYEXTR** are REXX EXECs compiled on either z/OS or z/VM.
- You want to link **MYEXTR** directly to **MYAPPL**, but you want the standard search order performed for **OTHRPROG**.

To accomplish this:

1. Compile **MYAPPL EXEC** with the DLINK, NOCEXEC, and OBJECT compiler options to get:
   TEST.**OBJ(MYAPPL)** under z/OS
   MYAPPL TEXT under z/VM
2. Compile **MYEXTR EXEC** with the NOCEXEC and OBJECT compiler options to get:

TEST.OBJ(MYEXTR) under z/OS
MYEXTR TEXT under z/VM

3. Generate load modules as follows:

Under z/OS

1. Determine the appropriate parameter convention for **MYAPPL**. If, for example, **MYAPPL** is called either from the TSO/E command line or from another EXEC as a host command with ADDRESS TSO, the appropriate stub is CPPL.
2. Link the CPPL stub with **TEST.OBJ(MYAPPL)** and store the result in **TEST.LOAD(MYAPPL)**. Use the REXXL cataloged procedure or the REXXL command to perform this task.
3. Because **MYEXTR** is called as a subroutine, link the EFPL stub with **TEST.OBJ(MYEXTR)** and store the result in **TEST.LOAD(MYEXTR).**
4. Link together the two linked modules from **TEST.LOAD(MYAPPL)** and **TEST.LOAD(MYEXTR)**, and store the result in **TEST.LOAD(MYAPPL)**.

Assuming you have allocated the data set **TEST.LOAD** to ddname **INFILE**, the appropriate control statements for the linkage editor are:

```
INCLUDE INFILE(MYAPPL)
INCLUDE INFILE(MYEXTR)
ENTRY   MYAPPL
NAME    MYAPPL(R)
```

Now you have an executable module that can be invoked from the TSO/E command line or with ADDRESS TSO, where each invocation of **MYEXTR** from **MYAPPL** passes control to **MYEXTR** directly instead of using the REXX search order. Recursive calls from **MYEXTR** to **MYEXTR** use the REXX search order, because **MYEXTR** was not compiled with the DLINK option. Therefore, the OBJECT output for **MYEXTR** does not contain external references. Calls from **MYAPPL** to **OTHRPROG** also use the REXX search order, because **OTHRPROG** was not included explicitly during the link-edit step.

Under z/VM

Link **MYAPPL** with **MYEXTR**, without resolving the reference to **OTHRPROG**, and generate a module in either of these ways:

```
LOAD MYAPPL MYEXTR (RLDSAVE
GENMOD MYAPPL
```

or:

```
LOAD MYAPPL (RLDSAVE
INCLUDE MYEXTR (SAME
GENMOD MYAPPL
```

Now you have an executable module that can be invoked from the CMS command line as a host command or from another EXEC as an external routine. It can be loaded as a nucleus extension (by using NUCXLOAD) to avoid address conflicts when invoking another program that also runs in the CMS user area. Each invocation of **MYEXTR** from **MYAPPL** passes control to **MYEXTR** directly instead of following the REXX search order. Recursive calls from **MYEXTR** to **MYEXTR** use the REXX search order, because **MYEXTR** was not compiled with the DLINK option. Therefore, the OBJECT output for **MYEXTR** does not contain external references. Calls from **MYAPPL** to **OTHRPROG** also use the REXX search order, because **OTHRPROG** was not included explicitly during the load step.

Under VSE/ESA

1. Send the object modules MYAPPL and MYEXTR from z/OS or z/VM to VSE/ESA, and store them in the sublibrary **REXXLIB**.**OBJECT** under the names **MYAPPL**.**OBJ** and **MYEXTR**.**OBJ**
2. Determine the appropriate parameter convention for MYAPPL. If, for example, MYAPPL is invoked from VSE JCL by means of an EXEC MYAPPL statement, the appropriate stub is VSE.
3. Combine the appropriate stub with **REXXLIB**.**OBJECT**.**MYAPPL**.**OBJ** and store the result in the sublibrary **REXXLIB**.**OBJECT** under the name **CMYAPPL**.**OBJ**. Use the REXXPLNK cataloged procedure to perform this task.
4. Because MYEXTR is called as a subroutine, combine the EFPL stub with **REXXLIB**.**OBJECT**.**MYEXTR**.**OBJ** and store the result in the sublibrary **REXXLIB**.**OBJECT** under the name **CMYEXTR**.**OBJ**. Use the REXXPLNK cataloged procedure to perform this task.
5. Link together the two object modules **REXXLIB**.**OBJECT**.**CMYAPPL**.**OBJ** and **REXXLIB**.**OBJECT**.**CMYEXTR**.**OBJ** and store the result in the sublibrary **REXXLIB**.**MODULE** under the name **MYAPPL**.**PHASE**.

Specify the sublibrary where the phase should reside with a LIBDEF PHASE,CATALOG=**REXXLIB**.**MODULE** statement, and the sublibrary where the object modules reside with a LIBDEF **OBJ**,**SEARCH**=**REXXLIB**.**OBJECT** statement. The appropriate control statements for the linkage editor are:

```
PHASE MYAPPL,*,SVA
INCLUDE CMYAPPL
INCLUDE CMYEXTR
```

Now you have an executable phase that can be invoked from VSE JCL, where each invocation of MYEXTR from MYAPPL passes control to MYEXTR directly, instead of using the REXX search order. Recursive calls from MYEXTR to MYEXTR use the REXX search order, because MYEXTR was not compiled with the DLINK option. Therefore, the OBJECT output for MYEXTR does not contain external references. Calls from MYAPPL to **othrprog** also use the REXX search order, because **othrprog** was not included explicitly during the link-edit step.

Chapter 7. Converting CEXEC Output between Operating Systems

This chapter describes what to do to run CEXEC output on the operating system other than the one on which you generated the output. To do this, you may have to convert the record format and record length of the compiled EXEC. Use the REXXF EXEC to perform the conversion on z/OS or z/VM. If you want to run your compiled programs on VSE/ESA, you must prepare the CEXEC file for transmission from z/OS or z/VM to VSE/ESA. Use the REXXV EXEC to perform this task.

This chapter also explains how to copy, under z/OS, CEXEC output from one data set to another. You must use the REXXF EXEC to copy CEXEC output.

The EXECs are described in "REXXF (FANCMF) under z/OS" on page 87, "REXXF under z/VM" on page 87, "REXXV (FANV) under z/OS" on page 88, and "REXXV under z/VM" on page 89.

Compiling on One System and Running on Another System

You can compile a REXX program on one operating system, convert the CEXEC output by using either the REXXF or the REXXV EXEC, as appropriate, and then run the converted EXEC under the other operating system. You can do this because the generated code does not contain operating system dependencies.

Converting from z/OS to MVS OpenEdition

Compiled EXECs of type CEXEC can run under MVS OpenEdition. They behave the same as interpreted REXX programs.

To transfer the CEXEC output to an OpenEdition file system, use the OCOPY command with the BINARY parameter. See *OpenEdition MVS Command Reference* for a description of the OCOPY command, the cataloged procedure REXXOEC, and the REXX procedure MVS2OE in Appendix D, "The z/OS Cataloged Procedures Supplied by IBM," on page 231, for an example.

Compiled EXECs in load module format cannot run under MVS OpenEdition.

Converting from z/OS to z/VM

The two methods for converting CEXEC output from z/OS to z/VM are:

- Method 1:
  1. Transfer the CEXEC output to z/VM, maintaining the same record length and record format that the CEXEC had on z/OS.
  2. Use REXXF to convert the CEXEC output to record format F and record length 1024.
- Method 2:
  1. Use REXXF to convert the CEXEC output to record format F or FB with a record length of 1024.
  2. Transfer the CEXEC output to z/VM.

Converting from z/OS to VSE/ESA

1. Use REXXV on z/OS to prepare the CEXEC output for transmission to VSE/ESA. The resulting record format must be F or **FB**, and the record length must be 80.
2. Create a job containing the following control statements and send it to VSE/ESA:

```
// LIBDEF PROC,SEARCH=lib.sublib
// EXEC REXX=REXXV,PARM='SYSIPT outlib outname [(option]'
   .
   . prepared CEXEC output from step 1
   .
/*
```

where:

lib.sublib
: Specifies the sublibrary where the EXEC REXXV resides.

outlib
: Is the name of the sublibrary, in the form *lib.sublib*, where the output file will reside on VSE/ESA.

outname
: Is the member name and member type of the output file that will reside in *outlib*, in the form *mn.mt*. If the member type is not specified, it defaults to PROC.

option
: Can be **DATA** or **NODATA**. Nested procedures must be cataloged all in the same way, either all with **DATA=YES**, or all with **DATA=NO**. You cannot mix procedures cataloged with **DATA=YES** and **DATA=NO** in one nesting.

  DATA
  : Indicates that the member outname is cataloged with **DATA=YES**

  NODATA
  : Indicates that the member outname is cataloged with **DATA=NO**

  The default for a new member is **NODATA**. For an existing member that is cataloged with **DATA=YES**, the default is **DATA**, if it is cataloged with **DATA=NO**, the default is **NODATA**.

  DATA and **NODATA** are used as parameters by the **EXECIO** command in VSE/ESA. For further information about the **EXECIO** command, refer to *IBM VSE/ESA REXX/VSE Reference*.

Converting from z/VM to z/OS

The two methods for converting CEXEC output from z/VM to z/OS are:

- Method 1:
  1. Transfer the CEXEC output to z/OS. Receive the CEXEC output in a data set with a record format F or FB and a record length of 1024.
  2. Use REXXF to copy the CEXEC output to the target data set.
- Method 2:
  1. Use REXXF to convert the CEXEC output to record format F or V: F if the receiving data set has record format F or FB, and V if the receiving data set has record format V or VB.

- For record format F or FB, set the record length equal to the record length of the receiving data set.
- For record format V or VB, set the record length equal to the record length of the receiving data set minus 4.

2. Transfer the CEXEC output to z/OS.

Converting from z/VM to VSE/ESA

1. Use REXXV on z/VM to prepare the CEXEC output for transmission to VSE/ESA. The resulting record format must be F or **FB**, and the record length must be 80.
2. Continue with step 2 on page 86 of "Converting from z/OS to VSE/ESA" on page 86.

Copying CEXEC Output

To avoid having characters inserted into CEXEC output when copying it from one data set to another, use the REXXF EXEC. Use the REXXV EXEC to prepare compiled REXX programs (CEXEC type) for transmission to VSE/ESA, and then to reformat them on VSE/ESA.

REXXF (FANCMF) under z/OS

The REXXF EXEC (FANCMF) converts a CEXEC output to a different record format or a different record length, or both. This EXEC must run in a TSO/E address space. (See also "Alias Definitions and Member Names under z/OS" on page 8.)

Enter the REXXF command in the following format:

REXXF *input-data-set-name output-data-set-name* [**REPlace**]

where:

input-data-set-name
: Is the name of the input data set that contains the CEXEC output. If the data set has a partitioned organization, a member name must be specified. The data set can have one of the following record formats: F, FB, V, or VB with an arbitrary logical record length.

output-data-set-name
: Is the name of the output data set that is to contain the converted CEXEC output. If the data set has a partitioned organization, a member name must be specified. The data set can have one of the following record formats: F, FB, V, or VB with an arbitrary logical record length equal to or greater than 20 and equal to or less than 32 767.

REPlace
: Specifies that an existing output data set that is not empty is to be overwritten. The minimum abbreviation is **REP**.

REXXF under z/VM

The REXXF EXEC converts a CEXEC output to a different record format or a different record length, or both.

Enter the REXXF command in the following format:

REXXF *input-file-identifier* [*output-file-identifier*] [**(***options*]

where:

input-file-identifier
Is the name of the input file. The file name must be specified. If the file type is not specified, it defaults to CEXEC. If the file mode is not specified, it defaults to **A**. If you want to specify an output file identifier, you must specify all parts of the input file identifier.

output-file-identifier
Is the name of the output file. If a part of the file name is not specified, it defaults to the corresponding part of the input file identifier. Similarly, an = character used to specify a part of the output file identifier is replaced by the corresponding part of the input file identifier. Note that the *output-file-identifier* and the *input-file-identifier* can be the same, if the **REPlace** option is used.

options Options can be specified in any order. Each option can be specified only once. The choices are:

F or V Indicates the record format of the output file. The default record format is F.

n Indicates the record length of the output file. The default record length is 1024. The minimum record length is 20.

REPlace
Specifies that an existing output file is to be overwritten. The minimum abbreviation is **REP**.

REXXV (FANV) under z/OS

The REXXV EXEC (FANV) prepares a compiled REXX program (CEXEC type) for transmission to VSE/ESA. It must run in a TSO/E address space. (See also "Alias Definitions and Member Names under z/OS" on page 8.)

Enter the REXXV command in the following format:

REXXV *input-data-set-name output-data-set-name* [**REPlace**]

where:

input-data-set-name
Is the name of the input data set that contains the CEXEC output. If the data set has a partitioned organization, a member name must be specified. The data set can have one of the following formats: **F, FB, V,** or **VB** with an arbitrary logical record length.

output-data-set-name
Is the name of the output data set that is to contain the resulting CEXEC output. If the data set has a partitioned organization, a member name must be specified. The data set must have record format F or **FB** and the record length must be 80. To protect the input data set, the *output-data-set-name* must differ from the *input-data-set-name*.

REPlace
Specifies that an existing output data set that is not empty is to be overwritten. The minimum abbreviation is the string **REP**.

REXXV under z/VM

The REXXV EXEC prepares a compiled REXX program (CEXEC type) for transmission to VSE/ESA.

Enter the REXXV command in the following format:

REXXV *input-file-identifier* [*output-file-identifier*][**(REPlace**]

where:

input-file-identifier
: Is the name of the input file. The file name must be specified. If the file type is not specified, it defaults to CEXEC. If the file mode is not specified, it defaults to A. If you want to specify an output file identifier, you must specify all parts of the input file identifier.

output-file-identifier
: Is the name of the output file. The file will have record format F and the record length will be 80. The file identifier does not need to be fully specified. For every missing part, the corresponding part of the *input-file-identifier* is used. An = character is replaced by the corresponding part of the *input-file-identifier*. Note that the *output-file-identifier* and the *input-file-identifier* can be the same, if the **REPlace** option is used.

REPlace
: Specifies that an existing output file is to be overwritten. The minimum abbreviation is the string **REP**.

Chapter 8. Language Differences between the Compiler and the Interpreters

This chapter describes the differences between the language processed by the Compiler and by the interpreters. Programs that run with the Alternate Library are interpreted, therefore they behave like normal interpreted programs.

For a complete description of the language definition and the other programming interfaces provided by each of these implementations, refer to *TSO/E REXX/MVS: Reference*, *IBM VSE/ESA REXX/VSE: Reference*, or to the corresponding z/VM documentation. Under CMS, use the HELP REXXCOMP command to get complete descriptions of the REXX language elements.

Differences from the Interpreters on VM/ESA Release 2.1, TSO/E Version 2 Release 4, and REXX/VSE

The language accepted by the Compiler and Library is:

- REXX language level 4.02 on CMS on VM/ESA Release 2.1 and subsequent releases
- REXX language level 3.48 everywhere else.

This section describes the items that the Compiler and Library handle differently from the Interpreter. In programs that are affected by these differences, the effects can usually be eliminated by minimal program changes. The support of some commands is also different.

Compiler Control Directives

Valid control directives are:

%COPYRIGHT
%INCLUDE
%PAGE
%STUB
%SYSDATE
%SYSTIME
%TESTHALT

The Compiler supports control directives, which are contained in comments. The interpreter treats them as normal comments. See “Control Directives” on page 38 for an explanation of how to use the control directives.

Halt Condition

The HI (Halt Interpretation) immediate command sets the halt condition. This may terminate all currently running REXX programs without affecting the operation of any other programs (as would the HE command under z/OS and the HX command under CMS).

The HI command and testing for the halt condition are supported only for programs that are compiled with the TESTHALT option or %TESTHALT control directive. A program compiled with the NOTESTHALT option and no %TESTHALT directive continues to run if the HI command is entered; the HALT condition is not raised.

Note:

1. A REXX program compiled with the TESTHALT option tests for the HALT condition:
   - At the beginning of a program
   - After each host command
   - At each label
   - At the beginning of the body of a repetitive DO loop
   - After the END of each iterative DO
   - At the first instruction following a clause containing either the invocation of an external function or the call (by means of a CALL instruction) to an external routine:
     – If an IF expression contains an invocation of an external function:
       - At the beginning of the THEN
       - At the beginning of the ELSE or, if there is no ELSE, after the THEN
     – If a WHEN expression contains an invocation of an external function:
       - At the beginning of the THEN
       - At the beginning of the following WHEN or, if there is no following WHEN, at the beginning of the OTHERWISE or, if there is no OTHERWISE, before the code that raises the SYNTAX condition.

   When you compile a program with the TESTHALT option, the compiled output may be slightly larger and the runtime performance may be slightly degraded.
2. If a HALT condition is detected at a label, the compiled program stores the line number of the label in the **SIGL** special variable, whereas the interpreter stores the line number of the instruction following the label in the **SIGL** special variable.

   To avoid this problem, put the label and the beginning of the following instruction on the same line.
3. When an EXEC runs under NetView®, NetView issues a Halt Immediate command. An interpreted EXEC will stop the execution. For a compiled EXEC to show the same behavior, it must be compiled with the TESTHALT compiler option or the %TESTHALT control directive. If compiled with the default NOTESTHALT, the HALT condition is not raised and the program continues.

If the expression following a RETURN, EXIT, or SIGNAL VALUE instruction contains a reference to an external function, the value stored in the special variable **SIGL** might be different, depending on whether a program is run compiled or interpreted.

Hint: Assign the expression to an intermediate variable and use this variable in the RETURN, EXIT, or SIGNAL VALUE instruction. For example, instead of coding this:

```
Return ext_rtn()
```

code this:

```
a = ext_rtn()
Return a
```

NOVALUE Condition

If a program contains a SIGNAL ON NOVALUE instruction but no NOVALUE label, the interpreter issues the `Label not found` message if the NOVALUE condition is raised. The message indicates the line number.

If a program contains a SIGNAL ON NOVALUE instruction but no NOVALUE label, the Compiler issues a message and does not generate compiled code.

You can correct this error by adding to the program a routine that handles the NOVALUE condition. The routine should indicate which line caused the NOVALUE condition and display the name of the uninitialized variable. The following code shows an example of a NOVALUE routine:

```
...
Exit                          /* End of routine                      */

NOVALUE:
  Say 'NOVALUE raised at line' sigl
  Say 'The referenced variable is' "CONDITION"('D')
Exit
```

The %SYSDATE and %SYSTIME control directives do not raise a NOVALUE condition.

The Compiler supports the %SYSDATE and %SYSTIME control directives, which generate the variables SYSDATE and SYSTIME, which contain the compilation date and time. The generated code ensures that the NOVALUE condition is not raised for these variables. If you did not explicitly assign a value to the variables SYSDATE and SYSTIME the interpreter raises the NOVALUE condition. Therefore, always assign a value to these variables:

```
sysdate = ''
/*%sysdate */
if (sysdate = '') then say 'interpreted'
                  else say 'compiled on' sysdate
```

OPTIONS Instruction

The ETMODE option requests checking of any double-byte character set (DBCS) string, literal string, or comment in the program for proper use of DBCS representation conventions, and enables the use of DBCS characters in symbols.

The ETMODE option of the OPTIONS instruction is recognized only if:

- It is enclosed within quotes (single or double) by itself, that is, no other option is enclosed within the quotes.
- Any other options in the same instruction are also enclosed within quotes by themselves.

If the OPTIONS instruction is not the first non-comment, non-label clause of the program, the Compiler ignores the ETMODE option. In the same situation, the interpreter raises the SYNTAX condition.

Examples of valid OPTIONS instructions are:

```
Options "ETMODE"
Options 'ETMODE' 'EXMODE'
```

PARSE SOURCE Instruction

PARSE SOURCE returns information describing the source of the program being executed.

The PARSE instruction with the SOURCE option returns the same tokens as returned by the interpreter, except in the following cases:

- Under z/OS, when an object module is linked with the EFPL stub, it always shows the string 'SUBROUTINE' as the second token, even when it is invoked as a function. See “PARSE SOURCE” on page 219 for details.
- Under z/VM, when the compiled program is a TEXT file, the file type and file mode (the fourth and fifth tokens) are * characters.

  When a module is generated from a TEXT file and is invoked using a synonym, the file name (the third token) is the synonym (which is also provided in the sixth token). See also “What the REXX Program Gets” on page 222.
- Under VSE/ESA, when an object module is linked with the EFPL stub, it always has the string **'SUBROUTINE'** as the second token, even if it is invoked as a function. See “PARSE SOURCE” on page 229 for more information.
- Under z/OS and VSE/ESA, link-edited modules with stubs insert a question mark (?) for the third, fourth, fifth, and sixth tokens (see “PARSE SOURCE” on page 219 and “PARSE SOURCE” on page 229).

PARSE VERSION Instruction

PARSE VERSION returns information describing the language level and the date of the language processor.

The PARSE instruction with the VERSION option returns five tokens:

1. The string `REXXC370` (interpreters produce `REXX370`).
2. The language level description. The language level depends on the Operating System:
   - Under VM/ESA Release 2.1 and subsequent releases, the language level is 4.02. This language level supports stream I/O. Programs containing stream I/O that have been compiled with an earlier release of the Compiler need to be recompiled.
   - Under z/OS, under z/VM (releases earlier than VM/ESA Release 2.1), and under VSE/ESA, the language level is 3.48.
3. Three tokens describing the release date of the Compiler that was used to generate the code (for example, 27 Oct 1994).

The general format of the PARSE VERSION information is the same as that provided by the interpreter, although the values of the tokens differ.

RANDOM Built-In Function

The Compiler and the Interpreter might not produce the same sequence, if you use the RANDOM built-in function with the third argument 'seed' to obtain a predictable sequence of quasi-random numbers, even if you use identical values of 'seed'.

SOURCELINE Built-In Function

In the interpreter, the SOURCELINE built-in function works like this:

- SOURCELINE() returns the line number of the final line in the source program.
- SOURCELINE(*n*) returns the *n*th line of the source program.

The full functions of the SOURCELINE function are available only if the program is compiled with the SLINE compiler option.

If you use the SLINE compiler option (described in “SLINE” on page 35), the source program is included in the compiled program and SOURCELINE continues to work as just described.

Note:

1. Any implied or specified EXEC compression for the interpreter (specifying %NOCOMMENT) will not be reflected by the Compiler.
2. The string returned by SOURCELINE(*n*) from a compiled program contains only the text within the specified margins. The string returned from a program interpreted with the system product interpreter, however, contains the complete line.
3. If source files are included using the %INCLUDE directive (see "%INCLUDE" on page 39), SOURCELINE() from a compiled program returns the total number of source lines including those from the included files.

If the NOSLINE compiler option is specified or defaulted to, however, SOURCELINE works like this:

- SOURCELINE() returns a value of 0.
- SOURCELINE(*n*) raises the SYNTAX condition at runtime.

To find out whether the SLINE or NOSLINE option is in effect, test whether SOURCELINE() is 0. The following code shows an example of this test:

```
Signal On Error
'COPY' ...                              /* This command may give a     */
                                        /* nonzero return code         */
...
Exit                                    /* End of main program         */
/*--------------------------------------------------------------------*/
/* Error handler:  common exit for command errors                     */
/*--------------------------------------------------------------------*/
ERROR:
Say "Unexpected return code" rc "from command"
/* If the SL option was used, display the source line.                */
If Sourceline() ¬= 0 Then
  Say "      " Sourceline(sigl)
/* Display the line number as shown in the listing.                   */
Say "at line" sigl"."
```

Start of Clause

The interpreter considers the line that consists of only a continuation comma (and possibly comments) as the start of a clause.

The Compiler considers the line where the actual instruction starts as the start of the clause.

This might lead to different output in the traceback (in case of an error) and in a different value of the special variable **SIGL**.

Example:

```
,                   /* 16 interpreter sets SIGL here */
SIGNAL X            /* 17 Compiler sets SIGL here    */
  .
  .
  .
X: SAY SIGL         /* interpreter says 16 */
                    /* Compiler says 17    */
```

SYSVAR Function

If ISPF variables are accessed with REXX programs running under TSO, note the following: If SYSICMD is retrieved using the SYSVAR function, link-edited REXX EXECs return a null string. For compiled EXECs that are not link-edited and are

therefore equal to interpreted REXX EXECs, SYSVAR('sysicmd') contains the EXEC name. The name of the link-edited REXX EXEC can be retrieved using SYSVAR('syspcmd') provided that it is obtained before any other subcommand is issued. In interpreted REXX EXECs and compiled REXX EXECs that are not link-edited, the initial value in SYSVAR('syspcmd') is 'EXEC'.

TRACE Instruction and TRACE Built-In Function

The TRACE instruction and the TRACE built-in function are supported (except for trace setting SCAN) only for programs compiled with the TRACE and SLINE options in effect.

Programs that have been compiled with the NOTRACE option behave the same as interpreted programs that run with TRACE set to **Off**. All valid options in the TRACE instructions or built-in functions are changed to OFF.

In a compiled program, interactive tracing starts immediately after the clause requesting it, provided that the clause is eligible. In an interpreted program, the clause following the eligible clause is executed before tracing is started. `Trace ?R`, for example, causes a first pause immediately after this trace instruction (unless the program is in interactive debug already). `Trace ?C` causes pauses only after host commands encountered after this instruction. Interactive debug is, however, entered immediately after a host command that contains this reference to the TRACE built-in function: `Trace('?C')`.

When tracing Intermediates (with TRACE setting I) of an expression that contains more than one adjacent concatenation, all intermediate results of the operands are shown before the intermediate results of the concatenations.

This example shows the difference between the output of the Compiler and that of the interpreter when the following program is run:

```
/*REXX*/Trace I
Say 'Tracing' 'a' 'concatenation'
```

Interpreter output	Compiler output

`2 *-* Say 'Tracing' 'a' 'concatenation'`
`>L> "Tracing"`
`>L> "a"`
`>O> "Tracing a"`
`>L> "concatenation"`
`>O> "Tracing a concatenation"`
`Tracing a concatenation`
	`2 *-* Say 'Tracing' 'a' 'concatenation'`
	`>L> "Tracing"`
	`>L> "a"`
	`>L> "concatenation"`
	`>O> "Tracing a"`
	`>O> "Tracing a concatenation"`
	`Tracing a concatenation`

The indentation of traced clauses reflects only function and subroutine invocations of internal routines and INTERPRET instructions.

Note: Any implied or specified EXEC compression for the interpreter (specifying `%NOCOMMENT`) will not be reflected by the Compiler.

TS (Trace Start) and TE (Trace End) Commands

The TS (Trace Start) and TE (Trace End) immediate commands are used to start and stop interactive tracing. TS and TE are supported in programs that have been compiled with the TRACE option.

Note:

1. TS and TE are not supported on VSE/ESA with REXX/VSE Version 1 Release 1.
2. The TS and TE commands have no effect on programs that have been compiled with the NOTRACE option.
3. Interactive tracing is started immediately after the TS command has been executed.
4. If interactive tracing is active and the TE command is executed, no interactive pause takes place after TE.

Differences to Earlier Releases of the Interpreters

This section describes the differences between the language supported by the Compiler and by releases of the interpreters earlier than those described in the preceding section.

SIGNAL Instruction

The SIGNAL instruction changes the flow of control. The VALUE option specifies an expression, and the result of evaluating this expression determines the label to get control.

Note:

1. The label name specified on a SIGNAL VALUE instruction must be in uppercase, because all labels defined in the program are translated to uppercase. The comparison is case-sensitive, and the result of the expression is not translated to uppercase.
2. A literal string specified as a label name on a SIGNAL *labelname* instruction must also be in uppercase for the same reason. For example:

```
SIGNAL 'LABEL1'
```

This restriction is for compatibility with the SAA REXX interface.

Integer Divide (%) and Remainder (//) Operations

The ratio of the operands in integer divide (%) and remainder (//) operations is checked.

The following condition must be true for integer divide and remainder operations:

first operand < *second operand* * (10**d*)

where *d* is the current setting of NUMERIC DIGITS. The absolute values of the terms in the formula are used.

This ensures that the quotient is a whole number within the current setting of NUMERIC DIGITS. Therefore, the result of an integer division is never rounded.

Exponentiation (**) Operation

In exponentiation (**) operations, the NUMERIC DIGITS setting is increased by *k* + 1, where *k* is the number of digits in the second operand. Then, the result is rounded to NUMERIC DIGITS, if necessary.

For example, in the operation a**500 with NUMERIC DIGITS 9, all intermediate results are rounded to 13 significant digits (9 + 3 + 1).

This restricts the possible error in the result to a maximum of 1 in the least significant position.

If the first operand cannot be expressed precisely within the current setting of NUMERIC DIGITS, it may be rounded (as the result of a previous operation) or truncated (as input to the exponentiation). In such cases, the precision of the first operand must be the precision of the result + *k* + 1, and the NUMERIC DIGITS setting must be raised accordingly.

Location of PROCEDURE Instructions

The PROCEDURE instruction:

- Sets up a local environment for the variables in an internal subroutine.
- If used, it must be the first instruction executed after the CALL or function invocation—that is, it must be the first instruction following the label.

Ensure that your programs contain no "deferred" PROCEDURE instructions when you compile them.

Binary Strings

Binary strings may not be supported by your Interpreter. A binary string is any sequence of zero or more binary digits (0 or 1) grouped in fours. The first group may have fewer than four digits.

The groups of digits are optionally separated by one or more blanks, and the whole sequence is delimited by single quotes or double quotes and immediately followed by the symbol b or **B**.

```
Examples: '11110000'b    "101 1101"B
```

Templates Used by PARSE, ARG, and PULL

The templates used by the **PARSE, ARG**, and **PULL** instructions may contain variable column numbers.

A variable within parentheses, where the open parenthesis is preceded by an equal, plus, or minus sign, means that the value of the variable is used as absolute or relative positional pattern.

```
Examples: =(v)   +(v)   -(v)   =(v.1)   +(v.1)   -(v.1)
```

PROCEDURE EXPOSE and DROP

The **PROCEDURE EXPOSE** and DROP instructions are enhanced to support subsidiary lists.

```
Examples: Procedure Expose (list)
          Drop (list)
```

DO LOOPs

If variables are named TO, BY, and FOR, they can be used within the expressions following WHILE and UNTIL, and within the repetitor expression immediately following the **DO**.

DBCS Symbols

Symbols may contain DBCS characters, if **OPTIONS 'ETMODE'** is in effect.

VALUE Built-In Function

The VALUE built-in function may have up to three arguments.

Three arguments for the VALUE built-in function are supported in compiled REXX only in CMS Release 6 and subsequent releases.

Argument Counting

Omitted trailing arguments are ignored. The number of arguments passed to a function or a subroutine is the largest number for which the **ARG** built-in function `ARG(n,'e')` returns 1. Where:

- `n` is the position of the last argument string specified.
- `'e'` is the existence test for the `n`th argument.

Options of Built-In Functions

The following options of built-in functions are supported by the Compiler, but may not be supported by your Interpreter.

Function	Option	Definition and Example

DATATYPE	**D**bcs	Returns 1 if the given string is a pure DBCS string enclosed within a shift-out (SO) and shift-in (SI). For example:
		`DATATYPE('<AABB>','D') → 1`
		`DATATYPE('a<AABB>b','D') → 0`

DATATYPE	**C**	Returns 1 if the given string is a valid mixed DBCS string. For example:
		`DATATYPE('<AABB>','C') → 1`
		`DATATYPE('a<AABB>b','C') → 1`
		`DATATYPE('abcde','C') → 0`

DATE	**N**ormal	Specifies the default date format, which returns the date in the format dd mon yyyy. For example: `DATE('N') → '30 Jun 1991'`

DATE	2nd to 5th	The second to fifth arguments represent an input date that can be converted to a specific output format. The fourth and fifth arguments specify the separation characters of the output and input strings, respectively. For example:
		`DATE('U','28 02 90','E','*',' ')`
		`→ '02*28*90'`

TIME	**C**ivil	Returns the time in the format hh:mmxx, where the hours are 1 through 12, and the minutes are 00 through 59. The minutes are immediately followed by the letters `am` or `pm`. For example: `TIME('C') → '4:54pm'`
TIME	**N**ormal	Specifies the default time format, which returns the time in the format hh:mm:ss. For example: `TIME('N') → '16:54:22'`
VERIFY	**N**omatch	Specifies the default option, which returns the position of the first character in the given string that is not also in the given reference. For example: `VERIFY('AB4T','1234567890','N') → 1`
Note: < represents shift-out (SO), and > represents shift-in (SI).		

Built-In Functions

The following built-in functions are supported by the Compiler, but may not be supported by your Interpreter.

Function	Definition and Example
B2X	Converts a string of binary digits into an equivalent string of hexadecimal characters.
CONDITION	Returns condition information associated with the most recently trapped condition. For example: `CONDITION('I') → 'SIGNAL'`
DIGITS	Returns the current setting of NUMERIC DIGITS. For example: `DIGITS() → 9`
FORM	Returns the current setting of NUMERIC FORM. For example: `FORM() → 'SCIENTIFIC'`
FUZZ	Returns the current setting of NUMERIC FUZZ. For example: `FUZZ() → 0`
WORDPOS	Returns the word number of the first word of a given phrase found in a given string. Returns 0 if `phrase` is not found. `WORDPOS('is the','now is the time') → 2`
X2B	Converts a string of hexadecimal characters into an equivalent string of binary digits.

Options of Instructions

The following options of instructions are supported by the Compiler, but may not be supported by your Interpreter.

Instruction	Options	Definition
CALL	ON/OFF	Controls the trapping of certain conditions.
NUMERIC FORM	VALUE	Enables specification of the SCIENTIFIC or ENGINEERING form as an expression.
OPTIONS	'EXMODE' 'NOEXMODE'	Enables or disables DBCS data operations capability[1].
SIGNAL ON	FAILURE	Traps negative return codes from host commands. (These are trapped by SIGNAL ON ERROR if trapping of the failure condition is not enabled.)
SIGNAL ON	NAME	Specifies the name of a label to get control if a specified condition occurs.

Strict Comparison Operators

The strict comparison operators carry out a simple character-by-character comparison. Unlike the other comparison operators, they never pad either of the strings being compared and never attempt to perform a numeric comparison. The strict comparison operators that may not be supported by your Interpreter are:

<< Strictly less than

1. The support of DBCS data operations affects all functions that deal with delimiting words and determining length. For example, the LENGTH function counts each double-byte character between SO and SI as 1 character.

<<= Strictly less than or equal to

¬<< Strictly not less than

>> Strictly greater than

>>= Strictly greater than or equal to

¬>> Strictly not greater than

The backslash (\) is synonymous with the logical NOT character (¬). The two characters may be used interchangeably in operators.

LINESIZE Built-In Function in Full-Screen CMS

The LINESIZE built-in function returns the current line width of the terminal. In full-screen CMS, the LINESIZE function invoked by a compiled REXX program always returns a value of `999999999`.

Enhancement to the EXECCOMM Interface

The EXECCOMM interface enables called commands to access and manipulate the current generation of REXX variables. The Fetch Private Information operation has been extended to return information for the following requests:

PARM Fetch the number of parameters (arguments) supplied to the program.

PARM.*n*
Fetch the *n*th parameter (argument string).

Chapter 9. Limits and Restrictions

This chapter provides information both on the maximum implementation limits and on technical restrictions imposed by the Compiler and Library.

If a program runs with the Alternate Library, all the limits and restrictions of the appropriate interpreter apply.

Implementation Limits

None of the following limits is lower than the corresponding interpreter limit:

Table 7. Compiler Implementation Limits

Item	Limit
Literal strings	250 bytes
Symbol (variable name) length	250 bytes
Nesting control structures	999
Clause length	Virtual storage
Variable value length	16 megabytes [2]
Call arguments	16000
MIN and MAX function arguments	16000
Number of PARSE templates	16000
PROCEDURE EXPOSE items	16000
Queue entries	Virtual storage
Queue entry length	Same as interpreter
NUMERIC DIGITS value	999 999 999
Notational exponent value	999 999 999
Hexadecimal strings	250 bytes
Binary strings	250 bytes
C2D input string	250 bytes
D2C output string	250 bytes
X2D input string	500 bytes
D2X output string	500 bytes
active PROCEDURES	30000

Technical Restrictions

Restrictions common to all systems:

- The number of lines of the source program is restricted to 99 999. The logical record length of the source program is restricted:
  - Under z/VM, to 65 535

2. If the length of a variable's value exceeds 16 megabytes, the results are unpredictable.

  - Under z/OS, to 32 760 for fixed length data sets, and to 32 756 for variable length data sets
- The maximum number of external routines that can be referenced in a program when compiled with the DLINK option is 65 534.
- The length of the value of variables is restricted to 16MB. If the length of a variable's value exceeds 16MB, the results are unpredictable.
- Compiled EXECs or object programs are restricted to **16MB** in size.
- Checking of pad characters: some built-in functions that perform string operations have an argument that specifies a pad character. If a program contains an OPTIONS or an INTERPRET instruction, the pad characters on built-in functions are not checked until runtime.

z/OS Restrictions

- You cannot invoke compiled REXX programs as authorized.
- The storage replaceable routine is not used by the Library.
- If the **NOESTAE** flag is set in the **PARM BLOCK**, no clean-up can be performed by the Library in case an **ABEND** occurs.

z/VM restrictions

- You cannot run compiled programs in the transient program area (TPA). A program running in the TPA cannot invoke a compiled REXX program.
- A **NUCXDROP EAGRTPRC** command must be issued before purging the segment that contains the Library, otherwise an **ABEND** will occur.
- Under VM/ESA Release 1.1 and subsequent releases, if the command **NUCXDROP EAGRTPRC** is issued while a compiled REXX program is running, unpredictable results may occur.

VSE/ESA restrictions

- The storage replaceable routine is not used by the Library.
- National Language Support: the messages are supported only in English.

C restriction

- The compilation of a program might be abended with the following messages:

  ```
  DMSABE155T User abend 2100 called from 002BCEB0 reason code 00007203 CMS
  DMSMOD109S Virtual storage capacity exceeded
  ```

  Reason code 7203 states an error when extending the stack.

  When such an error occurs, refer to the book *IBM C/370™ Programming Guide* for information on how to proceed.

Chapter 10. Performance and Programming Considerations

This chapter is intended to help you to improve the performance of your compiled programs. It also explains how to find out whether the IBM Library for REXX on System z is available on a system—an important programming consideration.

Performance Considerations

The performance improvements that you can expect when you run compiled REXX programs depend on the type of program. A program that performs large numbers of arithmetic operations of default precision shows the greatest improvement. A program that mainly issues commands to the host shows limited improvement, because REXX cannot decrease the time taken by the host to process the commands.

Compiled programs that include many ...	Run this much faster
Arithmetic operations	6 to 10 times
String and word processing operations	
Constants and variables	4 to 6 times
References to procedures and built-in functions	
Changes to values of variables	
Assignments	2 to 4 times
Reused compound variables	
Host commands	Minimal improvement

Note: This is true only when:

- The IBM Library for REXX on System z is used. With the Alternate Library, the performance of compiled REXX programs is similar to that of interpreted programs.
- The program has been compiled with the NOTRACE option.

Optimization, Optimization Stoppers, and Error Checking

The compiler performs the optimization procedures on a REXX program to improve error checking at compilation time and performance at runtime. Certain REXX constructs do not allow the compiler to optimize. They are called optimization stoppers.

The optimization procedures and stoppers are described in the following sections.

Keeping Track of Variables

After a value is assigned to a variable or the variable is used in an assignment, such as a target in a PARSE template, the variable is no longer in a dropped state. For example, in:

```
SIGNAL ON NOVALUE; X = Y; SAY X
```

the SAY instruction does not need to include code to test for, and raise, the NOVALUE condition although such code is needed for the evaluation of the expression Y in the assignment.

After a constant is assigned to a simple variable, the compiler can use the constant instead of the variable. This improves performance and enables the compiler to find more errors. For example, in:

```
I = 'A'; SAY SUBSTR(X, I)
```

the compiler can detect that the argument I for SUBSTR has a value that is not numeric and therefore not valid.

Even if the compiler cannot predict the exact value of a variable, it can derive properties of the value from the context in which the variable is used. For example, in:

```
X = Y + Z; SAY DATE(X) DATE(Y) DATE(Z)
```

the compiler can report that the arguments X, Y, and Z for the DATE function are not valid because they must all be numeric if the assignment is successful.

Performing Operations at Compilation Time

In many cases, the compiler can replace an expression involving only constants with the result of the expression. Together with keeping track of variables, this procedure can improve both the performance and error checking.

Note, however, that in the expression `X + 1 + 2`, for example, the subexpression `1 + 2` cannot be optimized. The reason for this is that, depending on the constants involved and the NUMERIC DIGITS setting, the expressions `X + (1 + 2)` and `(X + 1) + 2` can have different results.

Eliminating Several Evaluations

If an expression occurs more than once in a REXX program, it is not always necessary to evaluate the expression more than once. For example, the compiler treats `SAY X * Y + X * Y` like `T = X * Y; SAY T + T` where multiplication is performed only once at runtime.

This optimization procedure is even more effective if a compound variable is involved. For example, for `A.I = X; SAY A.I` the compiler generates only once the code for searching the tree belonging to stem A. and the variable belonging to tail I. In addition, the search is performed only once at runtime.

Improving Access to Compound Variables

In a loop where the tail of the compound variable is the control variable of the loop, such as:

```
DO I = 1 TO 1000
       SAY I A.I
END
```

all compound variables belonging to stem A. might be accessed sequentially. In this case, performing the general tree search for stem A. each time would be inefficient. Therefore, the code generated for A.I always first checks whether the next compound variable in stem A. is the one required. It then either uses it or continues its search.

If the tail is the control variable of an outer loop instead of the immediately enclosing loop, the same variable might be accessed repeatedly. In many such cases, the compiler can apply the usual optimization for compound variables. If this is not possible, it generates code that checks whether the compound variable used previously is the one required and only continues its search if not.

Note: This optimization procedure is not possible if a loop contains an optimization stopper.

Optimization Stoppers

An *optimization stopper* is a point in the REXX program where the compiler's information about the state of the variables or the expressions evaluated previously becomes unreliable.

Such optimization stoppers are:

- A point where the EXECCOMM interface can be invoked because any variable in the REXX program can be changed by this interface. Examples are the start of the program, invocations of external procedures, host commands, and TESTHALT hooks.
- Label definitions.
- INTERPRET instructions.
- Calls of the VALUE built-in function with a second argument.
- NUMERIC instructions. They cause information derived from, or about, arithmetic or comparison expressions to become unreliable, but do not affect information about compound variables.

The removal or introduction of an optimization stopper can cause the compiler to issue more or fewer warnings or error messages. In addition, the performance of the compiled program is affected if an optimization stopper is introduced into an inner loop.

Because the TESTHALT compiler option introduces TESTHALT hooks, at least one in every loop, using this option reduces the possibilities for optimization and error checking. It is, therefore, recommended that you first compile without the TESTHALT option to improve error checking, and compile with the option after you corrected the errors. Similarly, use the %TESTHALT directive after correcting the errors.

Optimization Limitations

The compiler's optimization procedures are designed to be compatible with the interpreter. Therefore, sometimes no optimization occurs where, at first glance, it seems possible. For example, in the following instruction:

```
SAY A X Y; SAY B X Y
```

the generated code evaluates the concatenation `X Y` only once, whereas no optimization occurs in:

```
SAY A + X + Y; SAY B + X + Y
```

To understand this, add the parentheses implied by the REXX evaluation order. The expression `A X Y` is equivalent to `(A X) Y`. The rules for REXX concatenation guarantee that the expressions `(A X) Y` and `A (X Y)` always produce the same result. However, in the case of the addition, the expressions `(A + X) + Y` and `A + (X + Y)` can produce different results because of the rounding rules required by NUMERIC DIGITS. Therefore, there is no common subexpression `X + Y` in these two expressions, and the optimizer cannot treat them alike. However, the compiler can optimize these expressions if they are rewritten as:

```
SAY X + Y + A ; SAY X + Y + B
```

Arithmetic

Compiled REXX programs normally use binary arithmetic for whole numbers. But for NUMERIC DIGITS settings of less than 9, and for whole numbers in exponential notation, arithmetic operations are performed using string arithmetic, which is slower. String arithmetic is also used for whole numbers written with decimal points, such as `'2.'` and `'3.0'`.

Hints: Do not set NUMERIC DIGITS to a value less than 9, unless necessary. Do not write whole numbers with decimal points, unless necessary.

Literal Strings

A string in quotes is considered to be a literal constant; its contents are never modified. Other symbols can also be used as constants: if no value has been assigned to a symbol, the defined value is the symbol itself, translated to uppercase. If a value has been assigned to a symbol the line number in the Compiler's cross-reference listing (see page "Cross-Reference Listing" on page 55) is followed by the characters `'(s)'`.

The Compiler does not know whether you intend to use a nonquoted symbol that could be a variable as a constant, a variable, or both. Therefore, every nonquoted symbol that could be a variable is checked for a value each time it is referenced. (No check can be made for value assignment during compilation, because values can be assigned to variables through the variable pool interface at runtime.)

Hint: Enclose all literal constants in quotes. For example, instead of coding this:

```
reportheader = customers         /* No value assigned to */
                                 /* "customers" yet      */
```

code this:

```
reportheader = "CUSTOMERS"
```

Variables

Simple variables and stems are addressed from a static symbol table created during compilation, whereas compound variables are held in a binary tree created at runtime. This tree has to be searched to retrieve a compound variable. Therefore, simple variables and stems are accessed faster than are compound variables.

Hint: Use compound variables only for structures, such as arrays and lists, for which they are appropriate.

Compound Variables

Compound variables that have three or fewer numeric tail parts can be accessed faster than compound variables that have nonnumeric characters in their tail.

Hint: If you need tails with nonnumeric and numeric tail parts, the first tail part should be nonnumeric.

For the best performance, use three or fewer numeric tail parts.

Labels within Loops

If there is a label between a DO and its corresponding END, the performance of the loop is adversely affected; control may jump incorrectly into the body of the loop, thus requiring more runtime checking at the end of each pass through the loop.

Hint: Avoid putting labels within DO loops. Structure your code so that there is no need for such labels.

Procedures

The EXPOSE option of the PROCEDURE instruction is used to ensure that references to specified variables within the internal routine refer to the variables environment owned by the caller.

If you expose a stem, the entire array of compound variables is available to the internal routine. This is much more efficient than exposing individual compound variables of the same stem.

Hint: If you expose a compound variable in an internal routine, expose the entire stem, if practical. For example, instead of coding this:

```
Procedure Expose x.j
```

code this:

```
Procedure Expose x.
```

TESTHALT Option

When a program is compiled with the TESTHALT option, the Compiler generates code in several places in the program to check for the HALT condition (see "Halt Condition" on page 91.) This extra code may adversely affect the performance of the program.

Hints:

- Compile with the TESTHALT option only when it is necessary.
- Instead of the TESTHALT compiler option, use the %TESTHALT control directive to check for the HALT condition only at points in the program that affect the performance less, for example not inside inner loops.

Frequently Invoked External Routines

If your program frequently invokes external routines or functions, consider linking them to the program that invokes them. This will improve performance by eliminating the search time. See the compiler option DLINK at "DLINK" on page 24 in topic "DLINK" on page 24 and "DLINK Example" on page 208.

Programming Considerations

This section explains:

- How to find out whether the Library is available on your system
- The different ways in which the z/OS and the z/VM Compilers handle the **VALUE** built-in function
- The different ways in which different systems support stream I/O
- How to determine whether an EXEC is compiled or interpreted
- How to create programs that run with the Alternate Library
- The upper and lower limits on the absolute value of numbers

Verifying the Availability of the Library

To find out whether the Library is available on a system, use the following code sequence in an interpreted program for the system you wish to query:

Under z/OS:

```
Trace 'O'                      /*  Suppress trace messages          */
Address Linkmvs 'EAGRTPRQ'     /*  Check for the Library            */
If rc¬=-3 Then                 /* -3 means the Library is not there */
  Say 'IBM Library for REXX on System z available'
```

Under z/VM:

```
Trace 'O'                      /*  Suppress trace messages          */
Address Command 'EAGRTPRC'     /*  Check for the Library            */
If rc¬=-3 Then                 /* -3 means the Library is not there */
  Say 'IBM Library for REXX on System z available'
```

Under z/OS, the EAGQRLIB EXEC (REXXQ) is located in the data set *prefix*.SEAGCMD. Under z/VM, you can find the EAGQRLIB EXEC on the installation minidisk of the IBM Library for REXX on System z. The source of the EAGQRLIB EXEC contains the following definition:

```
/* REXX ---------------------------------------------------------------*/
/*      Diagnose to query the REXX Runtime Library Symptom string.    */
/*                                                                     */
/*    Licensed Materials - Property of IBM                             */
/*    5695-014 IBM REXX Library                                        */
/*    (C) Copyright IBM Corp. 1989, 2003                               */
/*                                                                     */
/* Change Activity:                                                    */
/*   03-05-28       Release 4.0                                        */
/*                                                                     */
/*---------------------------------------------------------------------*/

  Trace 'O';
  Parse source src;
  Say 'Query the REXX Runtime Library symptom string';
  Say 'Source:' src;
  If word(src,1)='CMS' then eagname='EAGRTPRC';
  Else                      eagname='EAGRTPRQ';
  Say '  Calling query entry' eagname;
  If word(src,1)='CMS' then do;
    'NUCXDROP' eagname;
    ADDRESS COMMAND eagname;
  End;
  Else ADDRESS LINKMVS eagname;
  If rc>0 then do;
    Say '  Address: ' right(d2x(rc),8,0);
    ids=c2d(storage(d2x(rc+16),4));
    lvl=storage(d2x(ids+2),c2d(storage(d2x(ids),2)));
    Say '  Symptom:' lvl;
    Say '  Descrpt:  Name       Rel     APAR    LibLevel';
    If word(lvl,1)='EAGRTALT' then,
         Say '  The REXX Alternate Library is in effect.';
    Else Say '  The REXX Runtime Library is in effect.';
  End;
  Else Do;
    Say '  'eagname' returned RC='rc;
    Say '  A REXX Runtime System was not found.';
  End;
Exit;
```

Figure 23. Source of the EAGQRLIB EXEC

Note: Under VSE/ESA, no checking is necessary because the IBM Library for REXX in REXX/VSE, is always available if REXX/VSE is installed.

VALUE Built-In Function

When cross-compiling, the z/OS and z/VM Compilers treat the VALUE built-in function as follows:

- The z/OS compiler issues message **FANGAO0600W** if the VALUE built-in function has been coded with the *selector* argument. If you are compiling a REXX program with the z/OS compiler for execution under z/VM, you should ignore this message.
- When compiling under z/VM for execution under z/OS or VSE/ESA, no message is issued if the *selector* argument has been coded even though the z/OS or VSE/ESA runtime support for *selector* is not available.

Stream I/O

When cross-compiling, you should bear in mind that stream I/O is supported for execution only under VM/ESA Release 2.1 and subsequent releases.

Table 8 illustrates how stream I/O is supported on the different systems.

Table 8. Stream I/O Support

Function	VM/ESA 2.1	Other systems
LINEIN LINEOUT LINES CHARIN CHAROUT CHARS STREAM	Built-in function	External function
PARSE LINEIN SIGNAL ON/OFF NOTREADY CALL ON/OFF NOTREADY	Executed	Raise SYNTAX condition at runtime

The z/OS and z/VM Compilers act as follows:

- The z/OS compiler issues message **FANPAR0465W** for PARSE LINEIN and message **FANPAR0466W** for SIGNAL ON/OFF NOTREADY, and CALL ON/OFF NOTREADY. If you are compiling a REXX program with the z/OS compiler for execution under VM/ESA Release 2.1 and subsequent releases, you should ignore this message.
- When compiling under z/VM for execution under z/OS or VSE/ESA, no message will be issued for PARSE LINEIN, SIGNAL ON/OFF NOTREADY, and CALL ON/OFF NOTREADY even though no z/OS or VSE/ESA runtime support for them is available.

The following Parse statement is flagged by the Compiler because TSO/E does not support the Stream I/O (CMS supports the Stream I/O):

```
Parse LineIn x
```

If you have installed the Stream I/O function package for z/OS (described in Part 3, "Stream I/O for TSO/E REXX," on page 127), you can code the Parse statement as follows:

```
Parse value LineIn(data_set_name) with aline
```

The Stream I/O function package for z/OS is described in Part 3, "Stream I/O for TSO/E REXX," on page 127.

Determining whether a Program is Interpreted or Compiled

Use the PARSE VERSION instruction to determine whether the EXEC is running compiled or interpreted. This makes it possible to choose different logic paths depending on whether the EXEC is compiled or interpreted.

Example:

```
Parse Version v .          /* Use Parse Version to see if compiled  */
If left(v,5)='REXXC' Then what='compiled'
                     Else what='interpreted'
Say what
```

Creating REXX Programs for Use with the Alternate Library (z/OS, z/VM)

Not all programs are good candidates to run with the Alternate Library. This is because programs that run with the Alternate Library are in fact interpreted.

To create a REXX program that can run with both the Library and the Alternate Library, do the following:

- Compile the REXX program.

  At compilation time, you must consider these options:

  ALTERNATE
  Is required. It enables the program to run with the Alternate Library. The program can also run with the Library.

  SLINE
  Is required. It enables the creation of the control structures required by the interpreter.

  CONDENSE
  Is not required. However, because the SLINE option includes the program source in the compiled program, CONDENSE can be used to create compacted output, which is unreadable when using ISPF/PDF browse, view, and edit under z/OS, or browse and XEDIT in CMS.

  DLINK
  Requires special care. The DLINK option of a single module requires the Library. To run a program that uses the DLINK option with the Alternate Library, you must supply the external functions and subroutines that are in the single module as separate programs. In this way, the interpreter can locate them and invoke them.

  TESTHALT
  When running with the Alternate Library the Halt condition is always tested for by the REXX Interpreter, regardless of whether you specified TESTHALT, NOTESTHALT or neither as a compiler option.

- Continue with the preparation of the compiled program as explained in Chapter 6, “Using Object Modules and TEXT Files,” on page 71, if necessary.
- Document that the IBM Library for REXX on System z is not a prerequisite, but if it is available, using it will result in better runtime performance.

Limits on Numbers

There are upper and lower limits on the absolute values of numbers. These limits apply regardless of the setting of NUMERIC DIGITS or NUMERIC FORM. If a string that represents a number exceeds one of the limits, it is treated as non-numeric (data type CHAR):

- A number is within the **upper limit** if the following conditions are true:
  - The exponential part does not exceed +999999999. Leading zeros in the exponent are ignored.
  - The absolute value of the number does not exceed 9E+999999999.

  Examples:
  - 0.1E1000000000 is not numeric, because the exponent is too large.
  - 9.1E+999999999 is not numeric, because the value is too large. If this number is the result of an arithmetic operation, an **OVERFLOW** occurs and the **SYNTAX** condition is raised.
- A number exceeds the **lower limit** if the following is true for any operand or for the result:

  exponent - number of fractional digits in the mantissa < -999999999

  That is: the difference between the exponent and the number of fractional digits in the mantissa is less than -999999999.

  Note that trailing zeros in the fractional part of the mantissa are significant in REXX.

  For example, 1.23E-999999998 causes an UNDERFLOW error and raises the SYNTAX condition because -999999998 - 2 is less than -999999999. (The exponent relative to the trailing digit of the mantissa would be -1000000000.)

Part 2. Customizing the Compiler and Library

Chapter 11. Customizing the IBM Compiler and Library for REXX on z/OS

This chapter describes how to customize the IBM Compiler for REXX on z/OS and the IBM Library for REXX on z/OS, when they are installed or later. For instructions on how to install either the Compiler or the Library under z/OS refer to the corresponding program directories. For more information visit the home page at: `http://www.ibm.com/software/awdtools/rexx/`

Modifying the Cataloged Procedures Supplied by IBM

Modify the data set names and parameters (shown in Appendix D, "The z/OS Cataloged Procedures Supplied by IBM," on page 231) as necessary for your system, and store your cataloged procedures in **SYS1.PROCLIB.**

Customizing the REXXC EXEC

You can set up installation defaults for the compiler options by assigning the required options to the variable *instopts* in the customizing section of the REXXC EXEC.

Other specifications that you can customize in this EXEC include:

- The UNIT specification and the size of data sets that are allocated by the REXXC EXEC, if they are specified to receive output and do not already exist
- Data set attributes for these data sets (adhering to the limits shown in Table 4 on page 14)
- The default data set names used for compiler output (see the routine **MKDSN** in the REXXC EXEC)
- The text of messages issued by the EXEC

For more information refer to "Invoking the Compiler with the REXXC (FANC) EXEC" on page 9.

The defaults specified in the REXXC EXEC apply when users invoke the Compiler from both the command line and from the foreground and background compilation panels. The defaults do not apply when users use the cataloged procedures, or if they invoke the Compiler directly.

Customizing the REXXL EXEC

Assign the default name of the data set where stubs in load module form reside to variable `g.0lib` in the customizing section of the REXXL EXEC. This is also the name of the data set where predefined stubs reside. Refer to "Stubs" on page 211 for a list of stub names and member names in the sample data set names.

Other specifications that you can customize in this EXEC include:

- The member names of the predefined stubs
- The names of the predefined stubs that can be used as parameters of REXXL
- The **UNIT** specification and the size of data sets that are allocated by the REXXL EXEC, if they are specified to receive output and do not already exist
- The data set attributes for these data sets

- The linkage editor and the linkage editor options
- The text of messages issued by the EXEC

Message Repository

The Compiler, the Library, and the Alternate Library use the MVS message service (MMS). Installation message files are provided for U.S. English (**FANUMENU** and **EAGUMENU**) and Japanese (**FANUMJPN** and **EAGUMJPN**). For languages other than U.S. English, Japanese, and Upper Case English, you must supply a version of the installation message file with the appropriate translated message skeletons. For information on how to translate messages and on how to activate these translated messages, see the corresponding z/OS documentation.

The Compiler and Library can run on MVS SP Version 3 systems that have TSO/E Version 2 Release 4 installed. The Compiler and the Library use MVS Message Services (MMS) to provide National Language Support (NLS) on z/OS. These services are not available on a MVS SP Version 3 system, therefore only English is supported when running the Compiler or the Library on a MVS SP Version 3 system.

Systems that use U.S. English or Upper Case English do not require the MMS. In these cases, the installation message file for U.S. English is not used.

Chapter 12. Customizing the IBM Compiler and Library for REXX on z/VM

This chapter describes how to customize the IBM Compiler for REXX on z/VM and the IBM Library for REXX on z/VM, either when they are installed or later. For instructions on how to install either the Compiler or the Library under z/VM refer to the corresponding program directories. For more information visit the home page at: `http://www.ibm.com/software/awdtools/rexx/`

Customizing the Compiler Invocation Shells

Users can invoke the Compiler from a Compiler invocation shell. Two sample Compiler invocation shells are supplied with the Compiler: a full-screen interactive dialog, and an EXEC that operates in line mode. Customization tasks, which are normally done immediately after installation but can also be done later, are:

- Modify the function of the invocation shells to suit your system's requirements.
- Set up the installation defaults for the Compiler options.

Modifying the Function of the Compiler Invocation Shells

You can use the sample Compiler invocation shells as supplied. If you want to customize them, modify the following files:

Compiler invocation EXEC:
REXXC EXEC

Compiler invocation dialog:
REXXD EXEC

REXXDX XEDIT

The shells are written in REXX and can be compiled.

The REXXCOMP Command

Use the REXXCOMP command if you plan to write your own compiler invocation shell. The Compiler invocation shells use this command to invoke the Compiler. The syntax of the REXXCOMP command is as follows:

REXXCOMP *source-file-identifier* [**(***options-list***)**]

where:

source-file-identifier
Is the file identifier of the source program. The source file identifier need not be fully specified. If the file type is not specified, EXEC is used. If the file mode is not specified, it defaults according to the CMS search order. The REXXCOMP command does not translate the file identifier to uppercase.

options-list
Is a list of Compiler options to be used, separated by blanks. The Compiler invocation shell must process any user-defined defaults and explicitly selected options and pass them to the REXXCOMP command. The default

values supplied by IBM are used for any options that are not specified. For information on the syntax of the Compiler options, see "Compiler Options" on page 19.

Note: The enhanced form of the options must not be passed directly to the REXX compiler.

Setting Up Installation Defaults for the Compiler Options

The installation default values for the Compiler options are specified in the Compiler invocation EXEC.

To set up the installation default values:

1. Read the descriptions of the Compiler options in "Compiler Options" on page 19, and decide which options you want.
2. Edit the Compiler invocation EXEC (REXXC EXEC).
3. Find the place near the beginning of the file where the variable for the Compiler options, *InstOpts*, is initialized. A comment box after the variable assignment shows the default values supplied by IBM and the valid values.
4. In the assignment with the target *InstOpts*, specify any default values that you want to change.

   For the PRINT, CEXEC, OBJECT, and IEXEC options, you can use an equals (=) sign as the file name or file mode; this specifies that the file name or file mode are to be the same as the corresponding part of the source file identifier. You can also use an asterisk at the beginning or end of the file type; this specifies that part of the file type is to be the same as the corresponding part of the source file type.

   The following example shows a valid specification of installation defaults:

   ```
   InstOpts='NOC(E) PRINT(= LIST =) TERM'
   ```

Note: This procedure does not change the defaults supplied by IBM in the REXXCOMP module.

Customizing the Compiler Invocation Dialog

Some customization of the compiler invocation dialog may be required. REXXDX XEDIT, the XEDIT macro that controls the dialog, contains a section in which you can specify:

- The compilation command
- The **GLOBALV** group name for saving dialog information
- The commands for editing, printing, and invoking help
- The REXX file types that are acceptable
- The character set for file names and file types
- The naming convention for compiled and source EXECs

The installation defaults for compiler options are usually those that are specified in REXXC.

Customizing the Library

This section describes how to customize the Library. For detailed information refer to *z/VM Saved Segments Planning and Administration*.

Defining the Library as a Physical Segment

The IBM Library for REXX on z/VM, which is required to run compiled REXX programs, can be run in a DCSS. Here is an example of how to define the Library as a physical segment:

1. Define the segment by using the **DEFSEG** command. For example:

   ```
   DEFSEG EAGRTSEG 900-94F SR
   ```

   The segment can be above 16MB in virtual storage.
2. Ensure that the DCSS will not overlap any other DCSS or saved system.

Note: For detailed information refer to *z/VM Saved Segments Planning and Administration*.

Saving the Physical Segment

1. For an SP system, ensure that your virtual machine has class-E privilege and a virtual storage size at least 0.5MB greater than the address of the end of the segment.

   For an XA system, round up this value to the nearest megabyte boundary.
2. Invoke the **EAGDCSS** EXEC with the DCSS name as an argument. If you do not supply an argument, **EAGRTSEG** is used. While the segment is being saved, the EAGRTPRC module is updated to contain the name of the DCSS. Therefore, if the segment name you give it is different than the name contained in the first EAGRTPRC module in the search order, this module must reside on a disk accessed in read/write mode. For an explanation of how to load the Library, see "Other Runtime Considerations" on page 47.

Defining the Library as a Logical Segment

With CMS Release 6 or a subsequent release, the Library can be contained in a logical segment.

Note: For more information refer to *z/VM Saved Segments Planning and Administration*.

Here is an example of how to define the Library:

1. Define the physical segment to CP.
2. In file *eagrtseg* **PSEG**, define the physical segment contents by means of the following record:

   ```
   LSEGMENT NLSxxxxx LSEG
   ```

   Note: Throughout this section, *eagrtseg* and xxxxx have the following meaning:

   eagrtseg
   : Is the name of the segment

   xxxxx
   : Is **AMENG** for American English, or **KANJI** for Kanji.
3. In file `NLSxxxxx LSEG`, define the logical saved segment contents by means of the following records:

   ```
   MODULE EAGRTLIB (SYSTEM PERM NAME EAGRTPRC)
   LANGUAGE EAG xxxxx
   ```

   Note: The logical segment that contains a language information must be called `NLSxxxxx LSEG` regardless of its contents.

4. Create a LANGMERG control file called EAGxxxxx LANGMCTL that contains the following records:

   ```
   ETMODE OFF
   MESSAGE EAGUME
   ```

5. Enter the LANGMERG command to build EAGNLS TXTxxxxx:

   ```
   LANGMERG xxxxx EAG
   ```

6. Enter the SEGGEN command to save the segment:

   ```
   SEGGEN eagrtseg PSEG (MAP GEN
   ```

7. Access your system disk in read/write mode and copy the updated system segment identification file SYSTEM SEGID.

To make the logical segment and its contents available, put the following **SEGMENT** command into the SYSPROF EXEC:

```
SEGMENT LOAD NLSxxxxx
```

Note: If your installation has another logical segment named NLSxxxxx LSEG, you should add the SEGMENT ASSIGN command to this procedure to select the appropriate physical segment from which the logical segment will be used:

```
SEGMENT ASSIGN NLSxxxxx eagrtseg
```

Selecting the Version of the Library

You may want to have multiple versions of the Library on one z/VM system. For example, after applying a program temporary fix (**PTF**), you may want to try the new version while all other users continue to use the old version.

The product is shipped with a library loader (EAGRTPRC MODULE), which does not search for the Library in a DCSS and which assumes that the name of the Library is **EAGRTLIB MODULE**.

You can customize the library loader to search for the Library in a named DCSS or to suppress any DCSS search. You can also specify the name under which the Library is searched for on disk. See "Other Runtime Considerations" on page 47 for a description of how the Library is loaded under CMS.

When the first compiled REXX program is run, the first library loader in the search order loads the Library. If a new **PTF** is installed, you can:

1. Use the EAGCUST EXEC to generate a customized version of EAGRTPRC that searches for the **EAGRTNEW** library and does not search the DCSS.
2. Copy the new EAGRTLIB MODULE to **EAGRTNEW** MODULE.
3. Place the customized version of EAGRTPRC ahead of the production version of EAGRTPRC in the search order. Make sure that other users cannot access it.
4. IPL your CMS system.

Using the EAGCUST EXEC

With the EAGCUST EXEC you can:

- Query the current customization of EAGRTPRC.
- Specify a DCSS that is to be searched for the Library.
- Specify that the Library not be loaded from a DCSS.
- Specify the file name of the module that contains the Library.

These tasks are explained in the following paragraphs. The following definition applies to all the syntax descriptions in those paragraphs:

file-identifier
Is the file identifier of the file. The file name defaults to EAGRTPRC; the file type defaults to MODULE; the file mode defaults to that of the first file in the search order.

When you generate a customized version of EAGRTPRC, ensure that you have the EAGRTPRC MODULE on a disk accessed in read/write mode.

To query the current customization of EAGRTPRC, enter:

EAGCUST [*file-identifier*]

To specify that the Library is to be searched for in a DCSS, enter:

EAGCUST [*file-identifier*] **(S** *segname*

where:

segname
Specifies the name of the DCSS that contains the Library to be used.

To specify that the Library is not to be loaded from a DCSS, enter:

EAGCUST [*file-identifier*] **(NOS**

To specify the file name of the module that contains the Library, enter:

EAGCUST [*file-identifier*] **(L** *libname*

where:

libname
Specifies the name of the module that contains the Library.

Customizing the Message Repository to Avoid a Read/Write A-Disk

The message repository is distributed as EAGUME TXT*xxxxx*, where *xxxxx* indicates the language; it is **AMENG** (American English) in the base product. In this form, the SET LANGUAGE command (issued when the Library is loaded) copies the message repository to your A-disk and loads it from there. Your A-disk must be accessed in read/write mode.

To avoid the need for an A-disk accessed in read/write mode when a compiled REXX program is first invoked, change the message repository file type to TEXT. See the description of the SET LANGUAGE command in the *VM/SP CMS: Command Reference* manual for further explanation.

You can load the message repository into a DCSS that contains system-provided language files, because the repository is loaded with the **ALL** option of the SET LANGUAGE command when the Library is loaded. In this case, the message repository need not be accessible on disk.

Files Needed to Run Compiled REXX Programs

If neither the Library nor the message repository is in a DCSS, you need the following files to run compiled REXX Programs:

EAGUME	TXTAMENG	Message repository
EAGRTPRC	MODULE	Library loader
EAGRTLIB	MODULE	Library

If you need to work with compiled REXX CEXECS (not object files) in z/OS background mode, you need the following files:

EAGRTPRC	Runtime library
EAGUME	English language messages
EAGUME2	Kanji message repository

If you need to work with compiled REXX EXECS under TSO/E, you must also have the following file:

IRXCMPTM	Compiler programming table

Chapter 13. Customizing the Library under VSE/ESA

This chapter describes how to customize the IBM Library for REXX in REXX/VSE:

- Modify the data set names and parameters (shown in Appendix E, "The VSE/ESA Cataloged Procedures Supplied by IBM," on page 241) as necessary for your system, and store your cataloged procedures in **REXXLIB.PROCLIB.**
- The specifications that you can customize in this REXXL EXEC include:
  - The member names of the predefined stubs
  - The names of the predefined stubs that can be used as parameters of REXXL
  - The text of messages issued by the EXEC

For more information about REXXL refer to "REXXL Cataloged Procedure (VSE/ESA)" on page 80 and "REXXL" on page 243.

Part 3. Stream I/O for TSO/E REXX

Chapter 14. How to Read the Syntax Diagrams

The structure of the syntax diagrams shown in Part 3, "Stream I/O for TSO/E REXX," on page 127 is described below:

- Read the syntax diagrams from left to right, from top to bottom, following the path of the line.

  The ►►— symbol indicates the beginning of a statement.

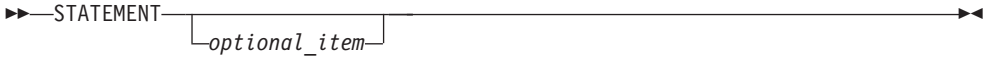
  The —► symbol indicates that the statement syntax is continued on the next line.

  The ►— symbol indicates that a statement is continued from the previous line.

  The —►◄ symbol indicates the end of a statement.

  Diagrams of syntactical units other than complete statements start with the ►— symbol and end with the —► symbol.
- Required items appear on the horizontal line (the main path).

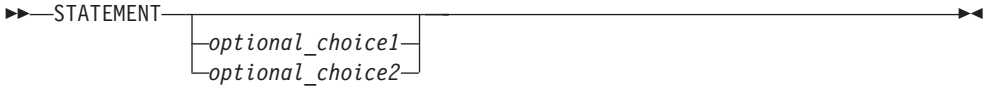
- Optional items appear below the main path.



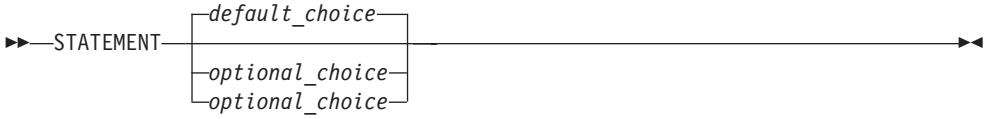
- If you can choose from two or more items, they appear vertically, in a stack.

  If you *must* choose one of the items, one item of the stack appears on the main path.

- If choosing one of the items is optional, the entire stack appears below the main path.



- If one of the items is the default, it appears above the main path and the remaining choices are shown below.



- An arrow returning to the left above the main line indicates an item that can be repeated.

A repeat arrow above a stack indicates that you can repeat the items in the stack.

- A set of vertical bars around an item indicates that the item is a *fragment*, a part of the syntax diagram that appears in greater detail below the main diagram.

fragment:

- Keywords appear in uppercase (for example, PARM1). They must be spelled exactly as shown, but you can type them in uppercase, lowercase, or mixed case. Variables appear in all lowercase letters (for example, *parmx*). They represent user-supplied names or values.
- If punctuation marks, parentheses, arithmetic operators, or such symbols are shown, you must enter them as part of the syntax.

The following example shows how the syntax is described.

Chapter 15. Installing the Function Package

This function package is a collection of I/O functions that follow the stream I/O concept. It extends and enhances the I/O capabilities of REXX for TSO/E, and shields the complexity of z/OS data set I/O to some degree. Further, the use of stream I/O functions provides for easier coding syntax and leads to better portability of REXX programs among different operating system platforms. The stream I/O concept is introduced in Chapter 16, "Understanding the Stream I/O Concept," on page 133.

This function package can be used with TSO/E REXX on z/OS, OS/390, and MVS systems that provide the MVS Name/Token Services, which are required to hook the function package into an existing TSO/E REXX installation. It is a loadable file that contains multiple object files bound together. Before its functions can be accessed and executed, the function package must be properly integrated into TSO/E REXX. Perform the following steps to install the package.

Note: It is assumed that you are familiar with the REXX language, the TSO/E environment, and the logical organization of data sets in the z/OS environment.

Preparation

1. The z/OS TSO/E REXX Stream I/O function package is shipped together with the IBM Library for REXX on System z and installed with SMP/E. The executable load libraries are in the data set `prefix.SEAGFUP`.
2. To activate the function package it is necessary to assemble and link-edit the TSO/E parameter modules IRXPARMS and IRXTSPRM.

   Customize the JCL job EAGSIOAS, which is in data set `prefix.SEAGJENU`. It contains predefined steps to automate the assembly of the TSO/E parameter modules. You must customize the PROC section as follows:

   - Specify the load library data set that has already been allocated. Replace the uid.REXX with the appropriate naming.
   - Ensure that `SYS1.MACLIB` and `SYS1.CSSLIB` are referenced in your SYSLIB concatenation. `SYS1.CSSLIB` must contain the modules `IEANTRT`, `IEANTCR`, and `IEANTDL`.

     `SYS1.CSSLIB` contains the stubs for z/OS Name/Token Services that the stream I/O functions require to share data with TSO/E REXX.
3. The parameter modules `IRXPARMS` and `IRXTSPRM` provided with this function package are modified exclusively for the needs of the REXX Stream I/O function package. Do not modify them. They are used by the `EAGSIOAS` job.

Assembly, Link-Edit, and Verification

1. Submit the `EAGSIOAS` job to assemble and link-edit the modules, and place the load module into a load library that is accessible by your system.

   Upon completion the load library `.SEAGFUP` should contain these load modules:

   ```
   EAGEFSIO     EAGIOHKP      IRXPARMS      IRXTSPRM
   ```
2. For the functional verification of the z/OS TSO/E REXX Stream I/O function package customize job `RZSIOVER`:
   - Specify the load library as for `EAGSIOAS`.

- A small EXEC is run that issues several REXX Stream I/O function calls. For more information refer to the corresponding output in the job output.

Note: The load modules IRXPARMS and IRXTSPRM provided with this function package can only be used if the REXX Stream I/O function package is the only function package to be used on your system.

Installations with Multiple Function Packages

Your installation might already use other function packages. These are defined in the parameter modules IRXPARMS and IRXTSPRM installed on your system. You need to add the definitions for the REXX Stream I/O function package to these modules to make all function packages work.

1. Inspect the parameter modules IRXPARMS and IRXTSPRM provided with this function package. They contain the TSO/E default definitions and the definitions for the REXX Stream I/O function package.
2. Incorporate the modifications for the REXX Stream I/O function package into the modules IRXPARMS and IRXTSPRM that are installed on your system.
3. Assemble and link-edit the updated parameter modules IRXPARMS and IRXTSPRM.
4. Copy the IRXPARMS, IRXTSPRM, EAGEFSIO, and EAGIOHKP modules to an LPA library. It is recommended that you copy these modules to a user LPA library instead of the SYS1.LPALIB.
5. Make sure that the user LPA library is the first in the LPALSTxx parmlib member and that a SYSLIB LPALIB (userlib.lpalib) is in the PROGxx parmlib member.

More detailed information about function packages is described in *z/OS TSO/E REXX Reference*.

Usage Considerations

Your TSO/E REXX installation might use the EXECTERM exec termination exit to customize the processing after REXX execs complete their processing. This customized processing can include closing of data sets, and freeing of resources that were allocated during the exec initialization step. If exec termination is used, a REXX exec does not necessarily need to close data sets it has opened.

On the other hand, the stream I/O function package provides the STREAM function, which can issue a CLOSE ALL stream command. If CLOSE ALL is used in a REXX exec, it also closes the data sets and frees the resources that were allocated with the first use of stream functions.

If you prefer relying on exec termination functionality (without using CLOSE ALL in your REXX exec), ensure that exec termination is active and APF-authorized, otherwise you might receive abend 066D.

To avoid these dependencies, use CLOSE ALL in your REXX execs regardless of the use of exec termination.

Chapter 16. Understanding the Stream I/O Concept

This chapter introduces the stream I/O concept and the implementation for TSO/E REXX. The terminology, the functions, and the common elements are described. Further, attention is given to the aspects of TSO/E and z/OS data set handling from the view of the stream I/O functions.

This knowledge lets you effectively use the information in Chapter 17, "Stream I/O Functions," on page 143.

The Basic Elements of Stream I/O

A stream is a popular concept for how to perform input/output to and from a program. Basically, a stream is a *sequence of characters* with functions to take characters out of one end, and put characters into the other end. In the case of input/output streams, one end of the stream is connected to a physical or logical I/O device, such as a keyboard, display, file, or queue. If it is an *output stream*, your program puts characters into one end of the stream, and an output device takes characters out of the other end. If it is an *input stream*, an input device puts characters into one end of the stream, and your program takes characters out of the other end.

The purpose of stream I/O is to simplify a programmer's view of input and output devices. The physical characteristics of I/O devices and the organization of data remain hidden. The data organization of devices is reduced to two simple forms:

- A sequence of characters that can be read or written character by character
- A sequence of lines that can be read or written line by line. A line in this context is defined as a sequence of characters that are terminated by means of any special character, or by means of the organizational form of the storage media.

A simple set of functions performs stream I/O operations from within a program.

- Housekeeping functions *declare streams* as input or output streams, *open* and *close* streams before and after using them, and allow to query their existence and characteristics.
- *Character input* and *character output* functions let the program read and write data character by character from input streams or to output streams.
- *Line input* and *line output* functions let the program read and write data line by line from input streams or to output streams.
- Further functions let the program check for the availability of input data from input streams.

During stream I/O operations a pointer is maintained for each stream. The pointer references the *current position* in a stream where a read operation or a write operation takes place. The position in a stream is relative to its beginning, counted as number of characters. Position 1 is always the first character. Pointers increase automatically during read operations and write operations. This eases the sequential reading from or writing to streams. The stream I/O functions can modify the position pointers by specifying explicit character or line positions to be read or written.

When streams are declared, they are given *names*. The input and output functions refer to these names to distinguish among multiple streams in a REXX program.

Generally, a stream can be any source or destination of external data that a program uses. Typical streams are files and data sets, and consoles for interactive input and output. The stream I/O concept also allows to view other sources and destinations as streams, for example, a reader, puncher, printer, program stack, queue, or a communication path. Programming environments that support stream I/O usually provide a *default input stream*, which is often the terminal input buffer, and a *default output stream*, often the display.

Data streams have two distinctive traits; they are either finite or conceptually unbound. An input stream from a file is finite because of the known quantity of characters; an input stream from a keyboard or communication path is unbound because of the unknown quantity. Stream I/O functions generally provide a mechanism of determining that an input stream is exhausted – that all data was read, and no more data is available. For finite streams they can detect the end of a file, for example. For unbound streams they might interpret special characters of the stream as delimiters.

The TSO/E REXX Stream I/O Implementation

Stream I/O is a concept already implemented in REXX for various operating system platforms. However, on z/OS and its predecessors, programmers needed to use the EXECIO command to access z/OS data sets. EXECIO requires programmers to consider many parameters, and to care about the allocation and deallocation of data sets. The TSO/E REXX Stream I/O functions hide this complexity. Programmers can use these easy-to-use functions to access z/OS data. Further, the use of stream I/O functions makes REXX programs more portable among platforms that support stream I/O.

The following sections describe the implementation of stream I/O for TSO/E REXX. A good understanding of this information is necessary to effectively use the individual functions described in Chapter 17, “Stream I/O Functions,” on page 143.

The Stream I/O Functions

The function package provides the following functions:

- The STREAM function controls streams and their status. It opens and closes streams, declares the type of operation (either read or write), and queries the existence and details of streams.
- The CHARIN and LINEIN functions are the stream input functions. They perform character input or line input.
- The CHAROUT and LINEOUT functions are the stream output functions. They perform character output and line output.
- The CHARS and LINES functions determine whether data exists in input streams for further read operations.

In this context, the following terms require definitions:

- The term “character” is any single byte in the range of X'00'...X'FF', respectively 0...255. So, a “character stream” is synonymous with a “binary stream” or a “byte stream”.
- The term “line” is defined as a sequence of characters that makes up the smallest unit that can be processed by the LINEIN and LINEOUT functions. A line read by LINEIN or written by LINEOUT does not process any additional

line-terminating characters (such as new-line character or carriage return character) if they are not part of the string to be read or written. You might think of a line as a record of a z/OS data set.

The function calling mechanism for the stream I/O functions is identical to the REXX built-in functions. Thus, they are called in REXX programs as functions, with the result being assigned to a variable, like in `rexx_variable = CHARS()`.

Naming Streams

TSO/E REXX provides a *default input stream* and a *default output stream*, which are used implicitly whenever a stream I/O function does not name a stream. In z/OS these default streams are associated with the console:

- For TSO/E background and z/OS:
  - ddname SYSTSIN represents the default input stream.
  - ddname SYSTSPRT represents the default output stream.
- For TSO/E foreground:
  - ddname SYSIN represents the default input stream.
  - ddname SYSOUT represents the default output stream.

If functions are to be performed on other streams, the streams must be named explicitly. The naming follows the rules and conventions for z/OS data sets as follows:

- A stream name can be the name of a data set, or the name of a data set member, for example:
  - A fully qualified data set name, for example `bill.january.data`.
  - A partially qualified data set name, for example `january.data`, where TSO/E adds a system-defined prefix to the data set name, as in `<user_id>.january.data`.
  - A fully qualified or partially qualified name of a data set member, for example `bill.year2001.data(january)`.

  Data set names should be enclosed in single quotation marks to avoid a modification by TSO/E, such as `'year2001.data(january)'`.
- A stream name can also be a ddname that is known to TSO/E and has the required data sets or resources allocated to it, for example, SYSPRINT or SYSOUT.
- A stream name can be a ddname that is generated from a data set name through the STREAM function. Each time the STREAM function opens a stream that is specified as a data set name, it automatically generates enumerated ddnames of the form `&SYS`*xxxxx*. The leading ampersand distinguishes them from data set names, and *xxxxx* is an enumeration. These unique ddnames can be used in a REXX program to explicitly name a stream with the stream I/O functions.

  The following example shows how the STREAM function opens the data set member `SYS1.MACLIB(PARM)`, generates a ddname, assigns this ddname to the variable *infile*, and uses this variable in the following CHARIN function call to name the stream. To recognize the generated ddname you could add `SAY infile`, which displays something similar to `&SYS00004`.

  ```
  /* Open the file. */
  infile = STREAM("'SYS1.MACLIB(PARM)'",'C','OPEN')
  if infile ¬= "-ERROR" then
     parm = CHARIN(infile,,20)
  ```

  As an alternative to this example you can also define the following routine:

```
 .
 .
if left(infile,1)="&" then
  parm = CHARIN(infile,,20)
```

Note the specification of the data set member name; the inner single quotation marks avoid a modification by TSO/E, the outer double quotation marks are the REXX convention for literal strings that include single quotation marks.

A second use of this side effect is more sophisticated. You can open the same data set multiple times with this method, and the STREAM function will provide a respective number of unique ddnames. Using these ddnames with the stream I/O functions lets you maintain multiple position pointers in the same data set. See "Multiple Read Operations" on page 140 for a detailed description.

After streams are given names, the stream I/O functions use these names to specify on which stream an operation is to be performed.

Transient and Persistent Streams

Streams might have a variety of sources and destinations, but they are either transient or persistent. Both types have certain characteristics that should be known when using the stream I/O functions.

- Transient streams usually communicate with the human user. The default input stream and the default output stream, if they represent the keyboard and the display, are typical examples. A communication path in a network is another example of a transient stream because of its similar behavior.

  The distinctive feature of a transient stream is that after a specific character or line was read from or written to a stream this process cannot be repeated. For example, if your REXX program reads user input from the default input stream, the characters are read as they are typed. You cannot change the position in a stream and read again the same character or line without the character being typed again.
- Persistent streams are usually files or data sets or equivalent media.

  The distinctive feature of a persistent stream is that you can repeatedly change the position in a persistent stream and read or write from and to different positions, within the boundaries of the stream.

When you use the stream I/O functions you will find that several parameters, such as the *start* position for the CHARIN function, are applicable only for persistent streams. In transient streams, read positions and write positions always default to the *next* character or line in a stream. In persistent streams, read positions and write positions can generally be changed within the boundaries of a stream.

Note: The current implementation of the TSO/E REXX stream I/O functions is limited with respect to randomly changing the positions in persistent streams. See the description of the individual functions for these capabilities. The LINEIN function might provide the most flexibility.

Opening and Closing Streams

A stream needs to be opened before it can be used, as a means to make the stream known to a REXX program, and to gain access to this stream for read and write operations.

The default input stream and the default output stream are opened when TSO/E REXX is started. Any stream I/O function that does not specify a stream by name performs its read operation or write operation on a default stream.

Implicit versus Explicit Opening of Streams

Streams are opened either implicitly or explicitly. All stream I/O functions open a named stream *implicitly* upon their first use within a REXX program. The named stream remains open for further function calls.

Streams can also be opened *explicitly* with the STREAM function. Explicit opening (as well as closing) of streams has some advantages. For the sake of a few lines, your program is more understandable, and you can easily recognize the type of operation (read or write) allowed on a stream.

You must explicitly open a stream to perform multiple read operations on the same data set. See also "Naming Streams" on page 135 and "Multiple Read Operations" on page 140.

Opening Streams for Read or Write Operations

A stream is opened for either read operations or write operations. It is not recommended to have a stream concurrently open for both types of operations.

If a stream is opened *implicitly*, the stream I/O function that is used at first decides the type of operation. A CHARIN, CHARS, LINEIN, or LINES function call opens a stream for read operations. A CHAROUT or LINEOUT function call opens a stream for write operations.

If a stream is opened *explicitly* through the STREAM function, the type of operation is specified as a parameter of the STREAM function.

After the type of operation is determined for a specific stream, you can use only the corresponding stream I/O functions, otherwise an error occurs.

To change the type of operation allowed for a stream, you first need to close the stream, then open it again for a different type of operation.

Note that opening a stream with the stream I/O functions in a TSO/E REXX program implies an allocation of the corresponding resource. You do not need to allocate a resource with the TSO/E ALLOCATE command, or by any other means.

Opening Nonexistent Streams

Persistent streams like data sets or files might not exist at the time they are opened. An attempt to open such a stream for read operations, either implicitly or explicitly, will fail. An attempt to open such a stream for write operations, either implicitly or explicitly, allocates an empty data set with VB 255, or whatever the operating system has defined as default. If you require a different record format, allocate the data set through the TSO/E ALLOCATE command, ISPF option 3.2, or a DD statement in batch.

If you name a nonexisting member (directly as data set member, or indirectly through a ddname) with a stream output function, the member is created and receives all subsequent output data.

You can use the STREAM function to query the existence of a stream before it is opened for read operations or write operations.

Closing Streams

All opened streams are closed implicitly when the REXX program ends. You can also use the STREAM function to explicitly close all or specific streams. You might want to do so for clarity, to free dynamically allocated working storage, or to change the type of operations on a stream (from read to write, or vice versa).

Closing a stream causes all pending write operations on this stream to be executed first. Pending write operations can be, for example, partially written lines on fixed block data sets.

Stream Formats

The z/OS TSO/E REXX stream I/O functions can work with the following files and data sets:

- QSAM (queued sequential access method) files
- z/OS sequential data sets and single members of partitioned data sets with the following record formats:
  - Fixed block formats (FB), and fixed length with ASA control characters (FBA)
  - Variable length (VB), and variable length with ASA control characters (VBA)

As a rule, the stream I/O functions do not write any *additional* formatting or control characters (other than what is specified as *string* with the stream I/O function) to a data set. Vice versa, the stream I/O functions read whatever is considered data from a data set. The read functions do not hide or remove anything.

Note that the record formats of data sets influence how the output stream functions succeed:

- The LINEOUT function attempts to write a specified string to a data set as a single line.

  If the length of the string fits in to the LRECL of the data set, the line is written. For data sets with a fixed record length the line is padded with blanks up to the logical record length.

  If the length of the string exceeds the LRECL of the data set, the line is *truncated*. The function returns a 1 as an indication that data remains to be written to the stream.

  Note that, if the LINEOUT function writes a null string, the stream is closed. Nothing is written to a data set.
- The CHAROUT function attempts to write a specified string to a data set character by character.

  No truncation takes place. Subsequent strings of characters are concatenated to previously written strings. If a fixed or maximum LRECL is exceeded, the characters wrap around to the next record. Thus, a large string can cause several records to be written. A partially filled record is retained internally until it is filled by subsequent CHAROUT (or LINEOUT) function calls, or until the output stream is closed. For data sets with fixed record length a partial record is padded with blanks.

The LINEIN function and the CHARIN function attempt to read a line or a number of characters from a persistent stream. For data sets with a fixed record length the string returned includes the padded blanks.

You can combine the use of the CHARIN and LINEIN functions for whatever purpose. This also applies to the CHAROUT and LINEOUT functions. For

example, you can write a few characters with CHAROUT, followed by a line written with LINEOUT. The basic rule is that the line starts at the position where the character string ended. To use these combinations, understand how the position pointers in stream work, as described in "Position Pointer Details."

Position Pointer Details

Each persistent stream maintains a position pointer to mark the position where a read operation or write operation takes place. By definition, position 1 marks the first character in a stream, and the positions are counted in number of characters relative to the beginning of a stream. When a stream is opened, the position pointer is set to position 1 of the stream.

The general use of position pointers is to ease the sequential reading and writing of streams. By default the first read operation starts at position 1, reads a number of characters or a line, and automatically increments the read position to the next unread character or line. A subsequent read operation starts at the incremented read position (the *current read position*). Similarly, a first write operation starts at position 1, writes a number of characters or a line, and automatically increments the write position behind the last character written. A subsequent write operation starts at the incremented write position (the *current write position*). The current position is maintained automatically. Thus, for sequential processing of a persistent stream, the stream I/O functions do not require the specification of a stream position.

The position pointer in a persistent stream can be manipulated to a certain degree to set it to a specific position where the next read operation or write operation should take place.

- A CHARIN or CHAROUT *start* value of 1 sets the current position to the beginning of a stream.
- A LINEIN *line* value can be set to any line number within a stream, which sets the current position to the beginning of this line.
- A LINEOUT *line* value of 1 sets the current position to the beginning of the stream (the beginning of the first line).

Note that the *line* parameter specifies a line, not the position of a character. Lines are counted from 1 to *n*, where line 1 is the first line in a stream.

Each open stream has its own position pointer. If a stream is opened for read operations, the pointer is either automatically set by any sequence of CHARIN and LINEIN function calls, or it is explicitly manipulated as described. Likewise, if a stream is opened for write operations, the pointer is either automatically set by any sequence of CHAROUT and LINEOUT function calls, or it is explicitly manipulated as described.

The CHARIN and LINEIN functions manipulate the same read position in a stream; while the CHAROUT and LINEOUT functions manipulate the same write position in a stream. For example, if two lines of 80 characters each were written to a fixed length data set by LINEOUT, followed by a CHAROUT of five characters, the current write position is 166 (the position where the next write operation would start). A subsequent LINEOUT with 80 characters would not succeed because only 75 characters would fit in the record. The line would be truncated. Conversely, if a line of 50 characters was written by LINEOUT to a fixed length (80) data set, the line is padded with blanks, and the current write position is 81

(the position where the next write operation would start). A subsequent CHAROUT or LINEOUT function starts at position 81.

End-of-Stream Treatment

For transient and persistent input streams use the CHARS function or the LINES function to detect the end of an input stream. These functions return 0 if no more characters or lines are available for reading, or they return 1 if at least one character or line is available for reading.

For transient streams, 0 means that the user has terminated the input to the stream by means of the two-character sequence `/*`, followed by the Enter key.

For persistent streams, 0 means that the input stream is either empty, or a previous read operation has already read the last character or line, or repeated read operations have triggered an end-of-file condition.

An attempt to read beyond the end of a stream returns a null string and triggers an error message. If this happens, the stream should be closed and reopened. Do not try to manipulate the position pointer after the end-of-file condition was triggered.

Error Treatments

Stream I/O Processing Errors

The current implementation of the z/OS TSO/E REXX Stream I/O function package supports only the `SIGNAL ON SYNTAX` condition trap. This means that a `SYNTAX` condition is raised if a language processing error, a syntax error, or a runtime error occurs during the execution of a stream I/O function call.

Note that it is not possible to trap `NOTREADY` conditions. Therefore, before using a stream, query its existence with the `STREAM ... QUERY EXISTS` function call.

If a syntax condition is raised because of a stream I/O function call, it is recommended to exit the REXX program. The recovering from such a syntax condition might cause unpredictable results.

Messages

The z/OS TSO/E REXX Stream I/O function package adds its own set of messages to TSO/E REXX. Similar to TSO/E REXX messages, each message consists of a message identifier and a message text. The message identifier is EAGSIO.

See Chapter 21, “Stream I/O Messages,” on page 197, if required.

Multiple Read Operations

As already described, each open stream maintains its own position pointer. This is sufficient for most sequential operations on a persistent stream. However, if you work with a sequential data set and you must perform multiple read operations on the same stream, you can use the following method. (Multiple write operations as well as concurrent read and write operations on the same stream are not supported.)

Use the STREAM function to open a stream explicitly for read operations. Name the data set to work with by its fully qualified or partially qualified data set name. The STREAM function returns a ddname, for example `&SYS00001`. You have now a stream open with its own position pointer.

Repeat this step with the *same* data set name. The next ddname might be `&SYS00002`. You have now a second stream open with its own position pointer.

Both streams represent the same data set. Both streams have their position pointers, each set to position 1 at the beginning.

You can now perform various CHARIN and LINEIN function calls on the streams `&SYS00001` and `&SYS00002` in any combination, and each stream pointer is maintained independently.

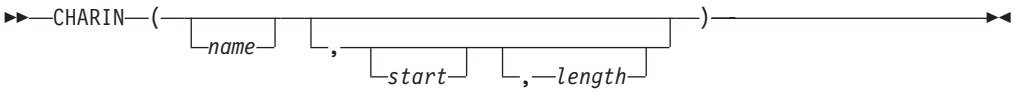
Note: `STREAM OPEN` returns either a valid ddname preceded by an ampersand (&), or in case of an open error the string `-ERROR`. The error string `ERROR` was prefixed with a minus sign (-) to avoid further processing if the `STREAM` return value is not verified for correctness. If the string `-ERROR` is used together with `CHARIN`, `CHAROUT`, `LINEIN`, `LINEOUT`, the data set allocation fails. If you define `ERROR`, the results may be unpredictable depending on the system installation.

Chapter 17. Stream I/O Functions

This chapter lists the stream I/O functions and shows their syntax elements. For each function, the basic function, the boundary conditions, the parameters, and the results are described. Examples show possible uses and return values.

CHARIN (Character Input)

Format



Purpose

Returns a string of up to *length* characters read from the character input stream *name*.

For persistent streams, a read position is maintained for each stream. Any read operation from the stream will by default start at the current read position. When the read operation is completed, the read position is increased by the number of characters read.

A *start* value of 1 can be given, together with a *length* of 0, to refer to the first character in a persistent stream. The read position is set to the beginning of the stream, no characters are read, and the null string is returned.

For transient streams (SYSIN in TSO/E foreground) only: If there are fewer than *length* characters available, then the execution of the program will normally stop until sufficient characters become available.

Parameters

name
Specifies the name of the character input stream. If it is not specified, the default input stream is assumed.

start
For a persistent stream, specify a value of 1 (and a *length* of 0) to set the read position to the first character in the stream. No other value is supported.

For a transient stream do not specify a read position.

length
Specifies the number of characters to be returned. The default is 1.

If *length* is 0, no characters are read, a null string is returned, and the read position is set to the value specified by *start*.

Comments

If a *length* of 0 is given (to specify an explicit read position, without reading from the stream) you must also specify *start* or let *start* default to 1 (the first character in

a stream). This combination is only applicable to persistent streams, because for transient streams you cannot specify an explicit read position.

Results

A string of characters, or a null string.

Examples

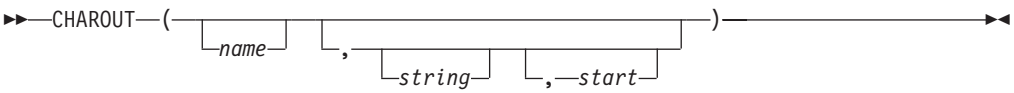
```
CHARIN(myfile,1,3)  -> 'MFC'   /* First 3 characters are read.           */


CHARIN(myfile,1,0)  -> ''        /* Read position set to start position.    */
CHARIN(myfile)      -> 'M'       /* 1 character read from start position.   */
CHARIN(myfile,,2)   -> 'FC'      /* Next 2 characters read.                 */


/* Reading from default input stream (here, the keyboard).                 */
/* The user types 'abcd efg'.                                              */
CHARIN()            -> 'a'      /* Default is one character.               */
CHARIN(,,5)         -> 'bcd e'  /* Next 5 characters.                      */
```

CHAROUT (Character Output)

Format



Purpose

Returns the result (0 or 1) of the write operation after attempting to write *string* to the character output stream *name*. *string* can be the null string, then no characters are written to the stream and 0 is returned.

For persistent streams, a write position is maintained for each stream. Any write operation to the stream will by default start at the current write position. When the write operation is completed, the write position is increased by the number of characters that are written. The initial write position is the beginning of the stream, so that calls to CHAROUT will append characters to the beginning of the stream.

A *start* value of 1 can be given, together with *string* being omitted (or specified as a null string), to refer to the first character in a persistent stream. The write position is set to the beginning of the stream, no characters are written to the stream, and 0 is returned.

If neither *start* nor *string* is given, the output stream is closed, and 0 is returned.

The execution of the CHAROUT function will normally stop until the output operation is effectively complete. If it is impossible for a character to be written, CHAROUT returns with a result of 1, and a corresponding error message is shown.

Parameters

name
Specifies the name of the output stream. If it is not specified, the default output stream is assumed.

string
specifies the string to write.

For transient streams, the length of the string is limited by the capabilities of your input device, usually 80 characters.

For persistent strings, the length is limited to a maximum of 32760 characters.

start
For a persistent stream, specify a value of 1 and omit *string* to set the write position to the first character in the stream. No other value is supported.

For a transient stream do not specify a write position. (If a value is specified, it is ignored.)

Results

Returns 0 after the specified characters are successfully written, or 1 if the specified characters could not be written.

Examples

```
CHAROUT(myfile,'Hi') -> 0      /* */
CHAROUT(myfile)      -> 0
CHAROUT(,'Hi')       -> 0
CHAROUT(V90,'29 BYTES FOR a V90 FILE LRECL')  -> 0    /* Variable format */
CHAROUT(V20,'29 BYTES FOR a V20 FILE LRECL')  -> 9    /* Variable format */
```

If a string of 29 characters is written to a data set with RECFM=F and LRECL=20, 20 bytes are written to record *n*, and nine bytes are written to record *n*+1.

If a string of 29 characters is written to a data set with RECFM=V or VB and LRECL=20, four bytes are reserved for the RDW, 16 bytes are written to record *n*, and 13 bytes are written to record *n*+1.

In both cases CHAROUT returns 0, as no truncation takes place.

CHARS (Characters Remaining)

Format

Purpose

Returns 0, or 1 if characters are remaining in the character input stream *name*.

name
Specifies the name of the input stream. If it is not specified, the default input stream is assumed.

Results

Returns 1, if one or more characters are available.

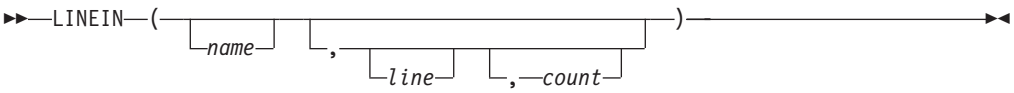
Returns 0, if no character is available. The data set or data set member is empty, or a previous read operation has already read the last character, or a previous read operation has triggered an EOF condition.

Examples

```
CHARS(myfile)  -> 1      /* EOF not reached. */
CHARS(empty)   -> 0      /* Empty data set.  */
CHARS()        -> 1      /* TSO/E console.   */
```

LINEIN (Line Input)

Format



Purpose

Returns *count* (0 or 1) lines read from the character input stream *name*.

For persistent streams, a read position is maintained for each stream. Any read operation from the stream will by default start at the current read position.[3] When the read operation is completed, the read position is increased by the number of characters read.

A *line* number can be given to set the read position to the start of a specified line. This line number must be positive and within the boundaries of the stream, and it must not be specified for a transient stream. A value of 1 for *line* refers to the first line in the stream.

If a *count* of 0 is given, then the read position is set to the start of the specified *line*, but no characters are read, and the null string is returned.

For transient streams (SYSIN in TSO/E foreground) only: If a complete line is not available in the stream, then the execution of the program will normally stop until the line becomes available.

Parameters

name
Specifies the name of the input stream. If it is not specified, the default input stream is assumed.

line
For a persistent stream, it specifies an explicit read position. The default is 1, or the position set by a previous read operation.

For a transient stream do not specify a read position.

3. Under certain circumstances, therefore, a call to LINEIN will return a partial line if the stream has already been read with the CHARIN function, and part but not all of the line has been read.

count
Specifies the number of lines to be returned. Only 0 or 1 is allowed. The default is 1.

If *count* is 0, no lines are read, a null string is returned, and the read position is set to the value specified by *line*.

Comments

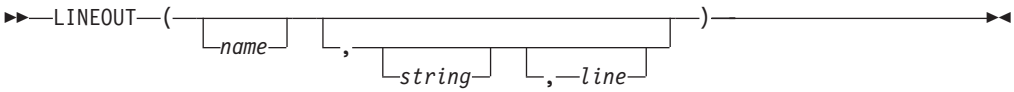
If a *count* of 0 is given (to specify an explicit read position, without reading from the stream), you must also specify *line*. This combination is only applicable to persistent streams, because for transient streams you cannot specify an explicit read position.

Results

A line or a null string.

LINEOUT (Line Output)

Format



Purpose

Returns the result (0 or 1) of the write operation after attempting to write *string* as a line to the character output stream *name*. The result is either 0 (the line was successfully written) or 1 (an error occurred while writing the line). *string* can be the null string, then no characters are written to the stream and 0 is returned.

For persistent streams, a write position is maintained for each stream. Any write operation will by default start at the current write position.[4] When the write operation is completed, the write position is increased by the length of the line written. The initial write position is the beginning of the stream, so that calls to LINEOUT will append lines to the beginning of the stream.

Note: The *line* parameter is provided for compatibility reasons, but does not allow to set the write position in this implementation.

If neither *line* nor *string* is given, the output stream is closed, and 0 is returned.

The execution of the LINEOUT function will normally stop until the output operation is effectively complete. If it is impossible for a line to be written, LINEOUT returns with a result of 1, and a corresponding error message is shown.

Parameters

name
Specifies the name of the output stream. If it is not specified, the default output stream is assumed.

4. Under certain circumstances, therefore, the characters written by a call to LINEOUT might be added to a partial line previously written to the stream with the CHAROUT routine. LINEOUT conceptually terminates a line at the end of each call.

string
Specifies the string to write as a line.

line
Specify value of 1, or specify no value. See the previous note.

Results

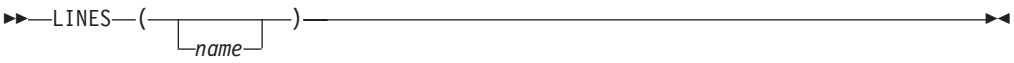
Returns 0 after the specified line is successfully written, or 1 if the line could not be written or is only partially written.

Examples

```
LINEOUT(myfile,'Hi')                 -> 0    /* Writes the string.          */
LINEOUT(myfile,,                     -> 0    /* No action.                  */
LINEOUT(myfile)                      -> 0    /* Output stream is closed.    */
LINEOUT(myfile,'String longer than lrecl')   -> 1   /* Truncated.          */
```

LINES (Lines Remaining)

Format



Purpose

Returns 0, or 1 if lines are remaining in the character input stream *name*. If the stream has already been read with the CHARIN function, this might include an initial partial line.

Parameters

name
Specifies the name of the input stream. If it is not specified, the default input stream is assumed.

Results

Returns 1, if one or more lines are available.

Returns 0, if no line is available. The data set or data set member is empty, or a previous read operation has already read the last line, or a previous read operation has triggered an EOF condition.

Examples

```
LINES(myfile)  -> 0      /* EOF encountered. */
LINES(empty)   -> 0      /* Empty data set.  */
LINES()        -> 1      /* TSO/E console.   */
```

STREAM (Operations)

Format

Purpose

Returns a string describing the state of the character stream *name*, or the result of an operation upon the character stream *name*.

This function is used to request information on the state of an input or output stream, or to carry out some particular operation on the stream.

Parameters

name
Specifies the name of the stream. Use a fully or partially qualified data set name (with or without a member specification), or a ddname known to TSO/E.

Note that the STREAM function returns enumerated ddnames of type `&SYSxxxxx` when it performs an OPEN, OPEN READ, or OPEN WRITE command. You can use these ddnames to name a stream in the stream I/O function calls. For more information see "Naming Streams" on page 135.

operation
Specifies the type of operation. This parameter must be the string **Command**, or its leading character **C**. The first character must be uppercase, and subsequent characters are ignored.

stream_command
Specifies one of the following commands (in capital letters) to be performed on the named stream:

CLOSE
Closes the named stream.

CLOSE ALL
Closes all streams that have been opened so far in this REXX exec, and frees resources that are bound to opened streams. For this stream command the first parameter *name* is ignored and can be omitted, such as in `STREAM(,'Command','CLOSE ALL')`.

It is recommended that you use this stream command in your REXX execs, regardless of external exec termination exits providing similar functions. See "Usage Considerations" on page 132 for a detailed description.

OPEN Is identical with **OPEN READ**.

OPEN READ
Opens the named stream for input and read operations. The input stream must already exist.

Note that input functions (CHARIN, CHARS, LINEIN, LINES) implicitly open streams for input at their first usage. You can explicitly open a stream for clarity reasons. You need to explicitly open a stream if you want to maintain multiple position pointers. See "Multiple Read Operations" on page 140, if required.

OPEN WRITE
Opens a named stream for output and write operations. If the output stream does not exist, a data set is allocated with the system defaults. See "Opening Nonexistent Streams" on page 137, if required.

Note that output functions (CHAROUT, LINEOUT) implicitly open streams for output at their first usage. You can explicitly open a stream for clarity reasons.

When the output stream is opened, the position pointer is initially set to position 1. Thus, the contents are overwritten. See "Position Pointer Details" on page 139, if required).

QUERY EXISTS
Queries the existence of a named stream and returns the fully qualified data set name that is allocated to this stream.

QUERY REFDATE
Queries the date when the named stream was last referenced. The date is returned in Julian form.

Note: You can also use the STREAM function to query the level of the installed function package. This might be required if you need to report problems. If required, type `STREAM(,'COMMAND','QUERY SERVICELEVEL')`. The function returns the service level of the installed function package in the form `REXXSIO <v><r><m> FIX<nnnn> <yyyy><mm><dd>`, for example `REXXSIO 140 FIX0000 20030801`.

Results

- **CLOSE** returns 0. If unsuccessful, RC = 4 is returned, and a message is issued. The named stream might not exist, or it has already been closed.
- **OPEN** returns a ddname as a string `&SYS`*xxxxx*, with *xxxxx* being an enumeration. If unsuccessful, the string `ERROR` is returned.
- **QUERY REFDATE** returns the date in Julian form (like `2003/360`), or a null string if the stream does not exist or the date cannot be determined.

Examples

```
STREAM('MYDATA FILE','C','CLOSE')  /* Closes the named data set. */

STREAM(strinp,'C','OPEN')         /* Opens an input stream.     */

STREAM(strout,'C','OPEN WRITE')   /* Opens an output stream.    */

STREAM('YOURDATA.FILE','C','QUERY EXISTS')
                                  /* Request the fully qualified */
                                  /* data set name.              */

STREAM('MY.DATA.FILE','C','QUERY REFDATE')
                                  /* Requests the date when last */
                                  /* referenced.                 */
```

Part 4. Messages

Chapter 18. Message Format and Return Codes

This chapter introduces you to the message format and lists the return codes displayed in messages.

Message Format

Compilation messages are prefixed by the message identifier. Under z/OS, runtime messages include the identifier only if the TSO/E command PROFILE MSGID ON has been issued. Under z/VM, runtime messages include the identifier only if the CP command SET EMSG ON has been issued. The format of the message identifier is as follows:

ppp Compiler (FAN) or Library (EAG) product prefix

xxx REX (runtime), ALT (Alternate Library, refer to the message with the REX identifier), SIO (REXX Stream I/O), or the Compiler phase identifier. The following list shows the identifiers of the Compiler phases:

COD Coder

CON Controller

ENV Environment interface

FLA Flattener

FMU Final make-up

GAO Global analyzer and optimizer

LIS Lister

PAR Parser

POP Post-optimizer

TOK Tokenizer

nnnn Message number

For example, EAGREX**3300E** is the main message for an error **33**. EAGREX**3301I** is a secondary message providing more information about error **33**.

s Severity code, it can be:

I Informational

W Warning

E Error

S Severe error

T Terminating error

In runtime messages, the first two digits of the message number are the REXX error number, and the last two digits are the subcode. The subcode is used in secondary messages to identify the error more specifically.

Return Codes

The return code indicates the maximum severity of any messages issued, as follows:

Return Code	Meaning
0	No messages or only informational messages
4	Warning
8	Error
12	Severe error
16	Terminating error
>16	Indicates that the Compiler has terminated abnormally and that it receives an internal error denoted by a reason code. Contact your IBM representative.

Note:

1. No compiled code is generated if one of the following occurs:
   - NOTRACE is in effect and a severe or terminating error is detected
   - TRACE is in effect and a terminating error is detected
   - NOCOMPILE is in effect
   - Warnings or errors have been issued and the appropriate options, such as NOCOMPILE(W) or NOCOMPILE(E), apply.
2. You can get unpredictable results if one of the following occurs:
   - NOTRACE is in effect and an error is detected
   - TRACE is in effect and an error or severe error is detected.
3. If the Compiler issues warning or informational messages, the program might still run correctly. However, you should examine the source code to assess the likely effects. For example, if the Compiler detects more than one definition of the same label, check whether some occurrences are misspellings.
4. It is good programming practice to correct all compilation errors.
5. A program that can be interpreted successfully may give compilation errors. There could be errors in parts of the program that are rarely, or never, executed. Also, the program may contain language elements that are either not supported by the Compiler or that must be coded differently. Refer to Chapter 8, “Language Differences between the Compiler and the Interpreters,” on page 91 for details.

Chapter 19. Compilation Messages

FANCON0050T Source file cannot be opened

Explanation: The source file could not be opened. You might have mistyped the file name, file type, or file mode. This problem can also occur when you are attempting to compile a program from a minidisk for which you have read-only access, while someone with read/write access to that minidisk has altered the program so that it no longer exists in the same place on the minidisk. Another possibility is that a lowercase file identifier has been passed to the REXXCOMP command.

User response: Ensure that you specify the source file correctly. If necessary, reaccess the minidisk on which the program resides.

FANFMU0051T Source file cannot be read

Explanation: The source file could not be read from the minidisk. This problem can occur when you are compiling a program from a minidisk for which you have read-only access, while someone with read/write access to that minidisk has altered the program so that it no longer exists in the same place on the minidisk.

User response: Reaccess the minidisk on which the program resides.

FANCON0052T Compiler listing cannot be printed

Explanation: An error occurred when creating the compiler listing. The most likely cause is insufficient virtual storage.

User response: Obtain more free storage by releasing a minidisk or SFS directory (to recover the space used for the file directory) or by deleting a nucleus extension. Alternatively, define a larger virtual storage size for the virtual machine and re-IPL CMS.

FANTOK0053T Required comment not found in line 1

Explanation: The first line of the program does not begin with a comment (delimited by /* and */) within the specified margins setting.

User response: Start the program with a comment.

FANxxx0054T Virtual storage exhausted

Explanation: The Compiler was unable to get the space needed for its work areas.

User response: Under z/OS, increase your region size.

Under z/VM, obtain more free storage by releasing a minidisk or SFS directory (to recover the space used for the file directory) or by deleting a nucleus extension. Alternatively, define a larger virtual storage size for the virtual machine and re-IPL CMS.

FANxxx0055T Compiler error: Reason code *nnn*

Explanation: An internal verification check in the Compiler failed.

User response: Report any occurrence of this message to your IBM representative. See the *IBM Compiler and Library for REXX on System z: Diagnosis Guide* for more information.

FANPAR0056I No comment found at start of program

Explanation: The first line of the program does not begin with a comment within the margins setting.

User response: Start the program with a comment.

FANCON0060T Limit of 99999 source lines exceeded

Explanation: Your program contains more source lines than the limit of 99999. The limit includes the lines in the source files, the lines in the included files, and the lines resulting from the splitting of source lines that contain %INCLUDE statements.

User response: Reduce the size of the program or split it into several smaller programs.

FANPAR0071W Duplicate label: Only first occurrence on line *nn* used

Explanation: The Compiler found more than one occurrence of the same label. After a CALL or SIGNAL instruction with this label as a target, control is always passed to the first occurrence of the label - namely that whose line number is shown in the message.

User response: Check whether one of the occurrences of the label is a misspelling.

FANGAO0072S Label not found

Explanation: The Compiler could not find the label specified by a SIGNAL instruction or the label matching an enabled condition.

User response: Check if the label is spelled correctly, or if you forgot to include it.

FANPAR0073S PROCEDURE not preceded by label

Explanation: The Compiler found a PROCEDURE instruction that is not immediately preceded by a label. The PROCEDURE instruction, if used, must be the first instruction within a routine.

User response: Move the PROCEDURE instruction to the beginning of the routine.

FANPAR0074W Label precedes THEN

Explanation: The Compiler found one or more labels before a THEN clause. This causes a runtime error if you use the label to transfer control to the THEN clause.

User response: If the label is for tracing, continue as planned. Otherwise, remove the label.

FANPAR0075W Label precedes ELSE

Explanation: The Compiler found one or more labels before an ELSE clause. This causes a runtime error if you use the label to transfer control to the ELSE clause.

User response: If the label is for tracing, continue as planned. Otherwise, remove the label.

FANPAR0076W Label precedes WHEN

Explanation: The Compiler found one or more labels before a WHEN clause. This causes a runtime error if you use the label to transfer control to the WHEN clause.

User response: If the label is for tracing, continue as planned. Otherwise, remove the label.

FANPAR0077W Label precedes OTHERWISE

Explanation: The Compiler found one or more labels before an OTHERWISE clause. This causes a runtime error if you use the label to transfer control to the OTHERWISE clause.

User response: If the label is for tracing, continue as planned. Otherwise, remove the label.

FANPAR0078W Label precedes END

Explanation: The Compiler found one or more labels before an END clause. This causes a runtime error if you use the label to transfer control.

User response: If the label is for tracing, continue as planned. Otherwise, remove the label. If you used a label because you wanted to stop the current iteration of a DO loop, use the ITERATE instruction instead.

FANPAR0079S ":" not preceded by label name

Explanation: The Compiler found a colon that is not used as a label terminator where it expects the beginning of a clause. You might have used a colon in a literal string without enclosing the string in quotes.

User response: Check your code and correct it.

FANPAR0080S More than 16000 arguments/operands/templates

Explanation: A function invocation or a CALL has more than 16000 arguments, or an EXPOSE has more than 16000 operands, or a PARSE has more than 16000 templates.

User response: Reduce the number of arguments/operands/templates.

FANPAR0081W Label before ITERATE

Explanation: The Compiler found one or more labels before an ITERATE instruction. This causes a runtime error if you use the label to transfer control to the ITERATE instruction.

User response: If the label is for tracing, continue as planned. Otherwise, remove the label.

FANPAR0082W Label before LEAVE

Explanation: The Compiler found one or more labels before a LEAVE instruction. This causes a runtime error if you use the label to transfer control to the LEAVE instruction.

User response: If the label is for tracing, continue as planned. Otherwise, remove the label.

FANGAO0083S Label would match (line *nn*) if uppercased

Explanation: The label referred to in a SIGNAL, SIGNAL VALUE, or SIGNAL ON clause is not defined. The label contains lowercase characters and would match the label defined in the indicated line if it were changed to uppercase.

User response: Change the program such that the label reference is in uppercase.

FANGAO0084W Label corresponds to a BIF name

Explanation: The label is equal to the name of a built-in function.

User response: No response is required. However, always put a function name in quotes if it refers to a built-in function and specify it without quotes if it refers to an internal label.

FANPAR0090S Maximum nesting level of 999 exceeded

Explanation: You have exceeded the limit of 999 levels of nesting of control structures such as DO-END and IF-THEN-ELSE and their components such as IF clauses and ELSE clauses.

User response: Check your code and correct it.

FANPAR0150S Mismatched DO control variable

Explanation: The variable specified on the END clause does not match the control variable of the related DO clause. The most common cause of this message is incorrect nesting of loops.

User response: See the Do column of the source listing, which shows the nesting level of each instruction, to find the incorrectly matched DO instruction.

FANPAR0151S Incomplete DO instruction: END not found

Explanation: The Compiler has reached the end of the source file without finding a matching END for an earlier DO.

User response: See the Do column of the source listing, which shows the nesting level of each instruction, to find the incorrectly matched DO instruction.

FANPAR0152S FOREVER not followed by WHILE/UNTIL/";"

Explanation: The Compiler found incorrect data after DO FOREVER. The only valid subkeywords after DO FOREVER are WHILE and UNTIL.

User response: Check your code and correct it.

FANPAR0153S TO/BY/FOR found in a DO after DO FOREVER

Explanation: A BY, TO, or FOR subkeyword has been found after FOREVER. The only valid subkeywords after DO FOREVER are WHILE and UNTIL.

User response: Check your code and correct it.

FANPAR0154S TO occurs more than once in a DO

Explanation: A DO clause contains more than one TO phrase.

User response: Check your code and correct it.

FANPAR0155S BY occurs more than once in a DO

Explanation: A DO clause contains more than one BY phrase.

User response: Check your code and correct it.

FANPAR0156S FOR occurs more than once in a DO

Explanation: A DO clause contains more than one FOR phrase.

User response: Check your code and correct it.

FANPAR0157S TO not followed by expression

Explanation: The Compiler expects an expression after the TO subkeyword in a DO clause.

User response: Check your code and correct it.

FANPAR0158S BY not followed by expression

Explanation: The Compiler expects an expression after the BY subkeyword in a DO clause.

User response: Check your code and correct it.

FANPAR0159S FOR not followed by expression

Explanation: The Compiler expects an expression after the FOR subkeyword in a DO clause.

User response: Check your code and correct it.

FANPAR0160S WHILE not followed by expression

Explanation: The Compiler expects an expression after the WHILE subkeyword in a DO clause.

User response: Check your code and correct it.

FANPAR0161S UNTIL not followed by expression

Explanation: The Compiler expects an expression after the UNTIL subkeyword in a DO clause.

User response: Check your code and correct it.

FANPAR0162S WHILE or UNTIL not allowed after WHILE phrase

Explanation: The compiler found the subkeyword WHILE or UNTIL in the WHILE phrase of a DO clause.

User response: If WHILE or UNTIL is the name of a variable, change the name or use the VALUE built-in function (for example, write VALUE('WHILE') instead of WHILE). If it is meant as a constant string, enclose it in quotes. If you intended to use both an UNTIL phrase and a WHILE phrase, you must modify the program logic to eliminate one of the phrases.

FANPAR0163S WHILE or UNTIL not allowed after UNTIL phrase

Explanation: The Compiler found the subkeyword WHILE or UNTIL in the UNTIL phrase of a DO clause.

User response: If WHILE or UNTIL is the name of a variable, change the name or use the VALUE built-in function (for example, write VALUE('WHILE') instead of WHILE). If it is meant as a constant string, enclose it in quotes. If you intended to use both an UNTIL phrase and a WHILE phrase, you must modify the program logic to eliminate one of the phrases.

FANPAR0164S Unexpected END

Explanation: The Compiler has found more END clauses in your program than DOs or SELECTs, or the ENDs were placed so that they did not match the DOs or SELECTs.

User response: Use the Do and Sel columns of the source listing, which show the nesting level of each instruction, to check the program's structure.

FANPAR0180S Initial expression missing in controlled DO loop

Explanation: The Compiler expects an expression to be assigned to the control variable after the assignment operator (=) in a DO.

User response: Check your code and correct it.

FANPAR0181S Variable required to the left of "="

Explanation: The symbol to the left of the "=" in an assignment begins with a period or digit, hence does not represent a variable.

User response: If the clause was intended as a command, enclose the expression in parentheses.

FANPAR0182S Assignment operator must not be followed by another "="

Explanation: The Compiler found a second "=" immediately after the first one of an assignment.

User response: Delete one "=" to form a correct assignment, or, if the clause was intended as a command, enclose the expression in parentheses.

FANPAR0190S THEN expected

Explanation: The Compiler expects a THEN clause after an IF or WHEN clause.

User response: Insert a THEN clause between the IF or WHEN clause and the following clause.

FANPAR0191S IF not followed by expression

Explanation: The Compiler expects an expression in an IF clause.

User response: Check your code and correct it.

FANPAR0192S Unexpected THEN

Explanation: The Compiler has found a THEN that does not match an IF clause or the WHEN clause of a SELECT instruction.

User response: Check your code and correct it.

FANPAR0193S Unexpected ELSE

Explanation: The Compiler has found an ELSE that does not match a corresponding IF clause. This situation can be caused by a DO-END in the THEN part of a complex IF-THEN-END construct. For example:

WRONG	RIGHT

`If a=b Then Do`
`  Say 'EQUALS'`
`  Exit`
`Else`
` Say 'NOT EQUALS'`
	`If a=b Then Do`
	`  Say 'EQUALS'`
	`  Exit`
	`  End`
	`Else`
	`  Say 'NOT EQUALS'`

User response: Check your code and correct it.

FANPAR0194S Instruction expected after ELSE

Explanation: The next clause after ELSE (not counting label clauses) must be an instruction or the start of an instruction. The Compiler found instead a non-instruction clause (such as END) or the end of the source program.

User response: Remove the ELSE or insert an instruction. As an explicit indication that no action is needed in the ELSE case, you can use a NOP instruction.

FANPAR0250I No OTHERWISE found in SELECT instruction ending in line *nn*

Explanation: The Compiler found a SELECT instruction that does not contain an OTHERWISE phrase. This causes a runtime error if all WHEN expressions are found to be false.

User response: If it is possible that none of the WHEN expressions will be true, insert an OTHERWISE that handles this condition.

FANPAR0253S SELECT not followed by ";" (WHEN follows instead)

Explanation: The Compiler expects a semicolon or implied semicolon between a SELECT and the first WHEN.

User response: Insert a semicolon or begin a new line between the SELECT and WHEN.

FANPAR0254S Incomplete SELECT instruction: END not found

Explanation: The Compiler has reached the end of the source file and has found a SELECT without a matching END.

User response: See the Sel column of the source listing, which shows the nesting level of each instruction.

FANPAR0255S WHEN expected

Explanation: The Compiler expects a WHEN after a SELECT.

User response: Insert one or more WHEN clauses after the SELECT.

FANPAR0256S WHEN/OTHERWISE/END expected

Explanation: The Compiler expects a series of WHENs, an OTHERWISE, and a terminating END within a SELECT instruction. This message is issued when any other instruction is found. The error can be caused by forgetting to enclose the list of instructions following a THEN within a DO and END. For example:

WRONG	RIGHT

Select
  When a=b Then
  Say 'A equals B'
  Exit
  Otherwise Nop
  End
	Select
	  When a=b Then Do
	  Say 'A equals B'
	  Exit
	  End
	  Otherwise Nop
	  End

User response: Check your code and correct it.

FANPAR0257S WHEN not followed by expression

Explanation: The Compiler expects an expression after the WHEN in a SELECT instruction.

User response: Check your code and correct it.

FANPAR0258S Unexpected WHEN

Explanation: The Compiler has found a WHEN clause that does not match a SELECT clause. You might have accidentally enclosed the WHEN in a DO-END construct by forgetting the matching END.

User response: Check whether the END is missing.

FANPAR0259S Unexpected OTHERWISE

Explanation: The Compiler has found an OTHERWISE clause that does not match a SELECT clause. You might have accidentally enclosed the OTHERWISE in a DO-END construct by forgetting the matching END.

User response: Check whether the END is missing.

FANPAR0260S Instruction expected after THEN

Explanation: The next clause after THEN (not counting label clauses) must be an instruction or the start of an instruction. The Compiler found instead a non-instruction clause (such as END) or the end of the source program.

User response: Remove the THEN or insert an instruction. As an explicit indication that no action is needed in the THEN case, you can use a NOP instruction.

FANPAR0270S Unexpected data in template

Explanation: The Compiler found unexpected data, for example, a symbol that is neither a number nor a variable, within a parsing template.

User response: Check your code and correct it.

FANPAR0271S "+" not followed by a whole number or "("

Explanation: The Compiler found an incorrect positional pattern in a parsing template: a plus sign must be followed by a whole number or by the name of a variable in parentheses.

User response: Check your code and correct it.

FANPAR0272S "-" not followed by a whole number or "("

Explanation: The Compiler found an incorrect positional pattern in a parsing template: a minus sign must be followed by a whole number or by the name of a variable in parentheses.

User response: Check your code and correct it.

FANPAR0273S "(" not followed by a variable

Explanation: The Compiler found an incomplete pattern in a parsing template: an open parenthesis must be followed by the name of a variable and a close parenthesis.

User response: Check your code and correct it.

FANPAR0274S PARSE not followed by a valid subkeyword

Explanation: The Compiler found a PARSE keyword that is not followed by the UPPER subkeyword or by one of the subkeywords ARG, EXTERNAL, LINEIN, NUMERIC, PULL, SOURCE, VALUE, VAR, or VERSION.

User response: Check your code and correct it.

FANPAR0275S PARSE UPPER not followed by a valid subkeyword

Explanation: The Compiler found a PARSE UPPER that is not followed by one of the subkeywords ARG, EXTERNAL, LINEIN, NUMERIC, PULL, SOURCE, VALUE, VAR, or VERSION.

User response: Check your code and correct it.

FANPAR0276S PARSE VAR not followed by a variable

Explanation: The Compiler expects the name of a variable at this position in a PARSE VAR instruction.

User response: Check your code and correct it.

FANPAR0277S Incomplete PARSE VALUE: WITH not found

Explanation: The Compiler found a PARSE VALUE instruction that does not contain a WITH subkeyword.

User response: Check your code and correct it.

FANPAR0278S Variable expected

Explanation: The Compiler found something other than the name of a variable in the operand list of an UPPER instruction. The variables can be simple or compound, but not stems.

User response: Check your code and correct it.

FANPAR0279S Variable pattern not terminated by ")"

Explanation: The Compiler found an open parenthesis in a parsing template but no corresponding close parenthesis. Each open parenthesis must be followed by the name of a variable and a close parenthesis.

User response: Ensure that you close all parentheses.

FANPAR0280S Unexpected ")" in template

Explanation: In a parsing template, the Compiler found a close parenthesis which does not match an open parenthesis.

User response: Check your code and correct it.

FANPAR0281S Unexpected ":" in template

Explanation: In a parsing template, a colon was found. Only variable names, patterns, and periods are accepted.

User response: Check your code and correct it.

FANPAR0282S Unexpected operator in template

Explanation: An operator, such as ¬ or || was found. Only variable names, patterns, and periods are accepted.

User response: Check your code and correct it.

FANPAR0283S DROP list must not be empty

Explanation: DROP must be followed by at least one variable name or at least one variable name in parentheses.

User response: Check your code and correct it.

FANPAR0284S UPPER list must not be empty

Explanation: The UPPER instruction needs at least one variable as an operand. The variable must be simple or compound. No stem variables are accepted.

User response: Check your code and correct it.

FANPAR0285W Variable name WITH found on PARSE VAR

Explanation: A WITH was found after the variable operand of a PARSE VAR. The WITH is assumed to be a variable.

User response: None if you intended WITH to be a variable. Otherwise, remove it.

FANPAR0290S Expression expected after OPTIONS

Explanation: The keyword OPTIONS must be followed by an expression.

User response: If you want to write an OPTIONS instruction, you must add an expression. If you want to use OPTIONS as a command, do one of the following:

- Enclose OPTIONS in parentheses or quotes.
- Prefix OPTIONS with a null string.
- Choose another name.

FANPAR0350S CALL not followed by routine name/ON/OFF

Explanation: The Compiler expects the name of a routine, or ON with a condition name, or OFF with a condition name at this position in a CALL instruction.

User response: Check your code and correct it.

FANPAR0352S CALL ON/OFF not followed by ERROR/FAILURE/HALT/NOTREADY

Explanation: The Compiler expects one of the conditions ERROR, FAILURE, HALT, or NOTREADY at this position in a CALL ON or CALL OFF instruction.

User response: Check your code and correct it.

FANPAR0353S NAME not followed by routine name

Explanation: The Compiler expects the name of a routine at this position in a CALL ON instruction. This error can occur if the routine name is in quotes.

User response: Check your code and correct it.

FANPAR0354S ";" or subkeyword NAME expected

Explanation: The Compiler found incorrect data at the end of a CALL ON instruction. The only subkeyword accepted after the condition name is NAME.

User response: Check your code and correct it.

FANPAR0371S No stem permitted in UPPER instruction

Explanation: The Compiler found a stem in an UPPER instruction. A stem cannot be converted to uppercase.

User response: Issue an UPPER instruction for each variable referred to by the stem.

FANPAR0381S INTERPRET not followed by expression

Explanation: The Compiler found an INTERPRET instruction that does not contain an expression to be interpreted.

User response: Check your code and correct it.

FANPAR0390S LEAVE not valid outside repetitive DO loop

Explanation: The Compiler found a LEAVE instruction outside a repetitive DO loop.

User response: Check your code and correct it.

FANPAR0391S ITERATE not valid outside repetitive DO loop

Explanation: The Compiler found an ITERATE instruction outside a repetitive DO loop.

User response: Check your code and correct it.

FANPAR0392S Variable does not match control variable of an active DO loop

Explanation: The symbol specified on a LEAVE or ITERATE instruction does not match the control variable of a currently active DO loop. You might have mistyped the name.

User response: Check your code and correct it.

FANPAR0393S Name of DO control variable expected

Explanation: The Compiler expects the name of the control variable of a currently active DO loop after a LEAVE or ITERATE instruction. Some other characters were found.

User response: Check your code and correct it.

FANPAR0394S ";" expected: corresponding DO not controlled by a variable

Explanation: An END clause specifies a symbol, but the related DO instruction does not have a control variable. The most common cause of this message is incorrect nesting of DO groups.

User response: Check your code and correct it.

FANPAR0450S NUMERIC not followed by DIGITS/FORM/FUZZ

Explanation: The Compiler expects one of the subkeywords DIGITS, FORM, or FUZZ after the keyword NUMERIC.

User response: Check your code and correct it.

FANPAR0451S NUMERIC FORM not followed by expression/valid subkeyword/";"

Explanation: The Compiler found incorrect data at the end of a NUMERIC FORM. The only data recognized after FORM is an expression or one of the subkeywords VALUE, SCIENTIFIC, or ENGINEERING.

User response: Check your code and correct it.

FANPAR0452S NUMERIC FORM VALUE not followed by expression

Explanation: The Compiler expects an expression after the subkeyword VALUE.

User response: Supply the missing expression or, if you are using VALUE as the name of a variable, enclose it in parentheses or write VALUE VALUE.

FANPAR0460S PROCEDURE not followed by EXPOSE or ";"

Explanation: The Compiler found incorrect data in a PROCEDURE instruction. The only subkeyword recognized on a PROCEDURE instruction is EXPOSE.

User response: Check your code and correct it.

FANPAR0465W PARSE LINEIN not supported under z/OS

Explanation: PARSE LINEIN is supported only under VM/ESA Release 2.1 and subsequent releases. The SYNTAX condition is raised if the program runs under systems other than VM/ESA Release 2.1 or subsequent releases.

User response: Check your code and correct it.

Note: If you are compiling a REXX program with the z/OS compiler for execution under VM/ESA Release 2.1 and subsequent releases, you should ignore this message.

FANPAR0466W NOTREADY condition not supported under z/OS

Explanation: The NOTREADY condition is supported only under VM/ESA Release 2.1 and subsequent releases. The SYNTAX condition is raised if the program runs under systems other than VM/ESA Release 2.1 or subsequent releases.

User response: Check your code and correct it.

Note: If you are compiling a REXX program with the z/OS compiler for execution under VM/ESA Release 2.1 and subsequent releases, you should ignore this message.

FANPAR0469S SIGNAL VALUE not followed by expression

Explanation: The Compiler expects an expression after the subkeyword VALUE.

User response: Supply the missing expression or, if you are using VALUE as the name of a variable, enclose it in parentheses or write VALUE VALUE.

FANPAR0470S SIGNAL not followed by label name or VALUE/ON/OFF

Explanation: After the keyword SIGNAL the compiler expects one of the subkeywords ON, OFF or VALUE, or a symbol, literal string or expression for a label. The end of the clause (or source program) was found instead.

User response: If you intended to use SIGNAL as a command, enclose it in quotes or parentheses. Otherwise complete the instruction or delete the clause.

FANPAR0471S SIGNAL ON/OFF not followed by condition name

Explanation: The Compiler expects the name of a condition (ERROR, FAILURE, HALT, NOTREADY, NOVALUE, or SYNTAX) after the subkeyword ON or OFF.

User response: Supply the missing condition or, if you are using ON or OFF as a label, write it in uppercase and enclose it in quotes.

FANPAR0472S NAME not followed by label name

Explanation: The subkeyword NAME in a SIGNAL ON instruction must be followed by a symbol. It is not permitted at this point to enclose the label name in quotes or to obtain it by evaluating an expression.

User response: Check your code and correct it.

FANPAR0490S ADDRESS VALUE not followed by expression

Explanation: The Compiler expects an expression after the subkeyword VALUE.

User response: Check your code and correct it.

FANPAR0550W Unsupported TRACE options will default to OFF

Explanation: REXX programs that have been compiled with Compiler option NOTRACE support no TRACE options other than OFF. The Compiler has found a TRACE instruction or a use of the TRACE built-in function which might require a different option.

User response: Compile your program with Compiler option TRACE or use an interpreter if you wish to trace.

FANPAR0560S Left operand missing

Explanation: The Compiler found an expression that does not have a term before the operator. Only the following can be used as prefix operators:

```
+ - ¬ \
```

User response: Check your code and correct it.

FANPAR0561S Right operand missing

Explanation: The Compiler found an expression that does not have a term after the operator.

User response: Check your code and correct it.

FANPAR0562S Prefix operator not followed by operand

Explanation: The Compiler found an expression that does not have a term after a prefix operator.

User response: Check your code and correct it.

FANPAR0564S "(" not followed by an expression or subexpression

Explanation: The Compiler expects an expression or subexpression after an open parenthesis, unless it is the open parenthesis of a function invocation.

User response: Check your code and correct it.

FANPAR0565S Unmatched "(" in expression

Explanation: The Compiler found an unmatched open parenthesis in an expression. This message is also displayed if a single parenthesis is included in a command without being enclosed in quotes. For example, the instruction:

```
COPY A B C A B D (REP
```

should be written as:

```
COPY A B C A B D '('REP
```

User response: Check your code and correct it.

FANPAR0566S Unexpected "," in expression

Explanation: The Compiler found a comma outside a routine invocation. This message is also displayed if a comma is included in a character expression without being enclosed in quotes. For example, the instruction:

```
Say Enter A, B, or C
```

should be written as:

```
Say 'Enter A, B, or C'
```

User response: Check your code and correct it.

FANPAR0567S Unexpected ")" in expression

Explanation: The Compiler found too many close parentheses in an expression.

User response: Check your code and correct it.

FANPAR0568S Unexpected ":" in expression

Explanation: The Compiler found a colon in an expression. This message is also displayed if a colon is included in a character expression without being enclosed in quotes. For example, the instruction:

```
Say Enter address: city and state
```

should be written as:

```
Say 'Enter address: city and state'
```

User response: Check your code and correct it.

FANPAR0569S Invalid operator

Explanation: The Compiler found an incorrect sequence of operator tokens in an expression. There might be two adjacent operators with no data in-between, or the characters might be in the wrong order, or special characters might be included in a character expression without being enclosed in quotes. For example, the instruction:

```
LISTFILE * * *
```

should be written as:

```
'LISTFILE * * *'
```

or, if LISTFILE is a variable, as:

```
LISTFILE '* * *'
```

User response: Check your code and correct it.

FANPAR0570S Invalid use of NOT operator

Explanation: The Compiler found a logical **NOT** operator (¬ or \), which is not part of a longer (comparison) operator, after a term in an expression. You might have meant to write a comparison operator but omitted the =, < or > characters.

User response: If you intend to concatenate the result of a **NOT** operation to the result of the preceding term, write an explicit concatenation operator (||) before the **NOT** operator. If you intend a comparison, append one or two =, < or > characters to the **NOT** operator.

FANPAR0580S Variable name longer than 250 characters

Explanation: A symbol used as a variable name is longer than the limit of 250 characters.

User response: Reduce the length of the variable name.

FANPAR0581S Invalid hexadecimal constant

Explanation: Hexadecimal constants cannot have leading or trailing blanks and can have embedded blanks only at byte boundaries.

The following are all valid hexadecimal constants:

```
'13'x
'A3C2 1c34'x
'1de8'x
```

User response: If you want to have a literal (quoted) string followed by the symbol X, but you do not want it to be interpreted as a hexadecimal constant, you must insert a concatenation operator (||) between the string and the symbol X. Otherwise, ensure that no digits are mistyped and remove any blanks that do not correspond to byte boundaries.

FANPAR0582S Resulting string longer than 250 characters

Explanation: The Compiler tried to convert a binary string, a hexadecimal string, or a literal string into internal format. The length of the resulting string exceeds the limit of 250 characters. Binary strings are limited to 2000 binary digits, hexadecimal strings are limited to 500 hexadecimal digits, and literal strings are limited to 250 characters.

This error can be caused by a missing ending quote or by a single quote in a string. For example, the string `'don't'` must be written as `'don''t'` or `"don't"`.

User response: To specify a string longer than 250 characters, concatenate two or more smaller strings, each with fewer than 250 characters.

FANGAO0583S Environment name longer than 8 characters

Explanation: The Compiler found an environment name longer than the limit of 8 characters specified on an ADDRESS instruction.

User response: Correct the environment name.

FANPAR0584S Name longer than 250 characters

Explanation: A symbol used as a label is longer than the limit of 250 characters.

User response: Reduce the length of the label.

FANPAR0590S Invalid binary constant

Explanation: The Compiler has found a literal string that is immediately followed by a symbol consisting only of the letter B, and tries to interpret it as a binary constant. No leading or trailing blanks are allowed in the string. Blanks can occur only at four-digit boundaries.

User response: If you want to have a literal (quoted) string followed by the symbol B, but you do not want it to be interpreted as a binary constant, you must insert a concatenation operator (`||`) between the string and the symbol B. Otherwise, ensure that no digits are mistyped and remove any blanks that do not correspond to four-digit boundaries.

FANPAR0591S EXPOSE list must not be empty

Explanation: A PROCEDURE instruction contains the subkeyword EXPOSE but no further data. EXPOSE must be followed by at least one variable name or one variable name in parentheses.

User response: If you wish to expose no variables, omit the subkeyword EXPOSE.

FANPAR0592S "=" not followed by a whole number or "("

Explanation: The Compiler found an incorrect positional pattern in a parsing template: an equal sign must be followed by a whole number or by the name of a variable in parentheses.

User response: Check your code and correct it.

FANPAR0593S Unmatched "(" in DROP list

Explanation: After each open parenthesis in a DROP instruction there must be the name of a variable and a close parenthesis.

User response: Check your code and correct it.

FANPAR0594S Unmatched "(" in EXPOSE list

Explanation: After each open parenthesis in the EXPOSE list of a PROCEDURE instruction there must be the name of a variable and a close parenthesis.

User response: Check your code and correct it.

FANPAR0595S Variable expected after "(" in DROP list

Explanation: After each open parenthesis in a DROP instruction there must be the name of a variable and a close parenthesis.

User response: Check your code and correct it.

FANPAR0596S Variable expected after "(" in EXPOSE list

Explanation: After each open parenthesis in the EXPOSE list of a PROCEDURE instruction there must be the name of a variable and a close parenthesis.

User response: Check your code and correct it.

FANPAR0597S Variable or "(" expected in DROP list

Explanation: Each entry in the list following DROP must be the name of a variable optionally enclosed in parentheses. The Compiler has found some other token, such as a symbol that does not begin with a letter.

User response: Check your code and correct it.

FANPAR0598S Variable or "(" expected in EXPOSE list

Explanation: Each entry in the EXPOSE list of a PROCEDURE instruction must be the name of a variable optionally enclosed in parentheses. The Compiler has found some other token, such as a symbol that does not begin with a letter.

User response: Check your code and correct it.

FANPAR0599S TRACE VALUE not followed by expression

Explanation: The Compiler expects an expression after the subkeyword VALUE.

User response: Supply the missing expression or, if you are using VALUE as the name of a variable, enclose it in parentheses or write VALUE VALUE.

FANGAO0600W Third argument of VALUE built-in function not supported

Explanation: VALUE built-in functions with three parameters are only supported under z/VM. This message is displayed, for example, if you have coded the VALUE built-in function with the *selector* argument.

User response: Check your code and correct it.

Note: If you are compiling a REXX program with the z/OS Compiler for execution under z/VM, you should ignore this message.

FANPAR0601W Invalid DBCS data in comment

Explanation: The first instruction of the program is OPTIONS 'ETMODE', and the Compiler has detected an invalid DBCS string in a comment. The number of bytes between shift-out and shift-in is odd.

User response: Correct the comment.

FANPAR0648S Invalid data after SELECT

Explanation: The Compiler expects a semicolon or implied semicolon after a SELECT.

User response: Remove the incorrect data after the SELECT, and insert a semicolon or begin a new line when appropriate.

FANPAR0650S Invalid data at end of clause

Explanation: The Compiler has found extra tokens after those allowed in the clause. You might have omitted a semicolon or not have started a new line after the offending clause.

User response: Insert a semicolon if necessary, or put the next clause into a new line.

FANPAR0651S Clause not completed before end of program

Explanation: The Compiler reached the end of the source program without finding the end of the last clause. This often occurs because of some other error, such as an unmatched start of comment or an invalid DBCS string.

User response: Terminate all quoted strings, comments and DBCS strings correctly. Do not use a continuation comma on the last line of the program.

FANPAR0652S Unmatched quote

Explanation: The Compiler reached the end of the source program without finding the close quote for a literal string.

User response: Add the close quote.

FANPAR0653S Unmatched shift-out character

Explanation: The Compiler found a character string or a comment that has unmatched shift-out/shift-in pairs (that is, a shift-out character without a shift-in character) with OPTIONS 'ETMODE' in effect.

User response: Supply the appropriate shift-in character.

FANPAR0654S Unmatched "/*"

Explanation: The Compiler reached the end of the source program without finding the ending */ for a comment.

User response: Add the missing */ characters.

FANPAR0655S Invalid character in program

Explanation: The Compiler found an unexpected character outside a literal (quoted) string or comment that is not a blank or one of the following:

- `A-Z a-z 0-9 (Alphanumerics)`
- `@ # $ ¢ . ? ! _ (Name Characters)`
- `& * ( ) - + = \ ¬ ' " ; : < , > / | % (Special Characters)`
- `Any DBCS character when OPTIONS 'ETMODE' is in effect`

In case the program was imported from another system: Verify that the translation of the characters was correct.

User response: Check your code and correct it.

FANPAR0656E Invalid DBCS data in string

Explanation: A character string that has an odd number of bytes between the shift-out/shift-in characters was encountered with OPTIONS 'ETMODE' in effect.

User response: Correct the character string.

FANGAO0657S Invalid whole number

Explanation: The Compiler found a parsing positional pattern or the right-hand term of the exponentiation (**) operator that did not evaluate to a whole number within the current setting of NUMERIC DIGITS, or that

was greater than the limit, for these uses, of 999 999 999.

User response: Check your code and correct it.

FANGAO0658S Logical value not 0 or 1

Explanation: The Compiler found a logical expression that does not result in a 0 or 1. Any term operated on by a logical operator (¬, \, |, &, or &&) must result in a 0 or 1. The expression in an IF clause, in a WHEN clause, or in a WHILE or UNTIL phrase must result in a 0 or 1.

User response: Check your code and correct it.

FANGAO0659S Nonnumeric term

Explanation: The Compiler found a nonnumeric term in an arithmetic expression or as an argument of a built-in function, or in a DO clause.

User response: Check your code and correct it.

FANPAR0660S Program ends with ","

Explanation: The last line of the source file ends with the line continuation character (a comma).

User response: Check your code and correct it.

FANPAR0661S Invalid DBCS data in symbol

Explanation: With OPTIONS 'ETMODE' in effect invalid DBCS data in a symbol was detected. DBCS data in a symbol is considered invalid if:

- A shift-in character immediately follows a shift-out character
- A shift-out character immediately follows a shift-in character
- The number of bytes between any shift-out character and shift-in character is odd
- Any byte between shift-out character and shift-in character has a value outside the range '41'X through 'FE'X.

User response: Correct the symbol.

FANPAR0662S Unmatched shift-out character in symbol

Explanation: With **OPTIONS 'ETMODE'** in effect, a symbol that has shift-out and possibly shift-in characters was detected. The shift-in character for symbols must be defined on the same line as the symbol.

User response: Correct the symbol.

FANENV0663S Recursive %INCLUDE directives not allowed

Explanation: A sequence of %INCLUDE directives was detected that lead to an already included file. This would cause an endless include activity. For example, an included file contains a %INCLUDE directive specifying itself; or, file **A** includes file **B** which in turn includes file **A**. The Compiler breaks the recursion and does not execute any more %INCLUDEs within that recursion.

User response: Correct the erroneous %INCLUDE directives.

FANENV0669T *fileID* **output file ID must not be identical with %INCLUDE file ID**

Explanation: The file name, file type, and file mode of one of the %INCLUDE files is equal to the file name, file type, and file mode of one of the output files. The value of *fileID* shows which output file ID is wrong:
CEXEC refers to the compiled EXEC.
IEXEC refers to the expanded IEXEC output.
OBJECT
refers to the TEXT file.
PRINT refers to the compiler listing.

User response: Specify a different file ID for the output file.

FANENV0670S Compiler option not recognized: *option*

Explanation: The command used to invoke the Compiler contains incorrect data in the options string. The name of an option might be mistyped.

User response: Invoke the Compiler again with a valid options list.

FANENV0671T No "(" found to mark start of compiler options

Explanation: The command used to invoke the Compiler did not contain an open parenthesis to mark the start of the options list.

User response: Reissue the command with an open parenthesis between the source file identifier and the options list.

FANENV0672T File name, file type, or file mode too long: *fileID-part*

Explanation: The identifier you specified for the source file or for one of the output files is incorrect. Either the file name or the file type is longer than 8 characters or the file mode is longer than 2 characters.

User response: Invoke the Compiler again with a valid file identifier.

FANENV0673S LINECOUNT value not 0 or a whole number in the range 10-99: *value*

Explanation: The value of the LINECOUNT (LC) compiler option is not 0 or a whole number in the range 10 through 99.

User response: Invoke the Compiler again with a valid value for the LINECOUNT option.

FANENV0674T *option***: no ")" found after parameter**

Explanation: A keyword parameter in a compiler option does not contain a close parenthesis.

User response: Add the missing close parenthesis.

FANENV0675T No file ID for REXX source found

Explanation: The command used to invoke the Compiler did not specify a source file.

User response: Invoke the Compiler again with a source file identifier.

FANENV0676T *option* **output file ID must not be identical with source file ID**

Explanation: The file name, file type, and file mode of one of the output files is the same as the file name, file type, and file mode specified for the source file. The value of *option* indicates which output file identifier is in error: CEXEC refers to the compiled EXEC, IEXEC refers to the expanded (IEXEC) output, OBJECT refers to the TEXT file, and PRINT refers to the compiler listing.

User response: Specify a different file identifier for the output file.

FANENV0677S Option *option* **ignored because of missing ")"**

Explanation: A compiler option is ignored because a previous keyword parameter in a compiler option does not contain a close parenthesis.

User response: Add the missing close parenthesis.

FANENV0678T *option1*/*option2* **output file IDs must not be identical**

Explanation: The same file name, file type, and file mode has been specified for more than one of the output files. The values of *option1* and *option2* indicate which output file identifiers are identical: CEXEC refers to the compiled EXEC, IEXEC refers to the expanded (IEXEC) output, OBJECT refers to the TEXT file, and PRINT refers to the compiler listing.

User response: Specify a unique file identifier for each output file.

FANENV0679T Invalid file ID: *fileID*

Explanation: The *fileID* specified for the source file or one of the output files is not a valid CMS file name. The *fileID* contains one or more asterisks or the file mode is not in the range A0 to Z6 or A to Z.

User response: Invoke the compiler again with a valid CMS *fileID*.

FANFMU0680T Error opening CEXEC file

Explanation: The Compiler could not open the compiled EXEC file specified in the CEXEC compiler option. This problem can occur if your virtual machine does not have read/write access to the minidisk.

User response: Use a minidisk to which your virtual machine has read/write access.

FANFMU0681T Error opening OBJECT file

Explanation: The Compiler could not open the TEXT file specified in the OBJECT compiler option. This problem can occur if your virtual machine does not have read/write access to the minidisk.

User response: Use a minidisk to which your virtual machine has read/write access.

FANFMU0682T Error writing to CEXEC file

Explanation: An error occurred when writing to the compiled EXEC file specified in the CEXEC compiler option. The most likely cause of this message is a full disk.

User response: Obtain more free disk space.

FANFMU0683T Error closing CEXEC file

Explanation: The Compiler could not close the compiled EXEC file specified in the CEXEC compiler option.

User response: If the problem persists, notify your system support personnel.

FANFMU0684T Error writing to OBJECT file

Explanation: An error occurred when writing to the object file specified in the OBJECT compiler option. The most likely cause of this message is a full disk.

User response: Obtain more free disk space.

FANFMU0685T Error closing OBJECT file

Explanation: The Compiler could not close the TEXT file specified in the OBJECT compiler option.

User response: If the problem persists, notify your system support personnel.

FANCON0686T Error closing source file

Explanation: The Compiler could not close the source file.

User response: If the problem persists, notify your system support personnel.

FANLIS0687T Error opening file or virtual printer for PRINT output

Explanation: The Compiler could not open the compiler listing specified in the PRINT compiler option. This problem can occur if your virtual machine does not have read/write access to the minidisk, if the virtual printer is not operational, or if the Compiler was unable to get the space needed for the work areas.

User response: Use a minidisk to which your virtual machine has read/write access, direct the print output to the virtual printer, make the virtual printer operational, or obtain more storage by releasing a minidisk or SFS directory, or by deleting a nucleus extension. Alternatively, define a larger virtual storage size for the virtual machine and re-IPL CMS.

FANLIS0688T Error writing to file or virtual printer for PRINT output

Explanation: An error occurred when writing to the compiler listing file specified in the PRINT compiler option. The most likely causes of this error are a full disk, or a non-operational virtual printer.

User response: Obtain more free disk space, or use the PRINT compiler option to send the file to another disk or to the virtual printer, or make the virtual printer operational.

FANxxx0689T Error closing file or virtual printer for PRINT output

Explanation: The Compiler listing specified in the PRINT compiler option could not be closed. The most likely cause of this message is that a release-storage request has failed.

User response: If the problem persists, notify your system support personnel.

FANENV0690T Source file cannot be opened: record length greater than number *number*

Explanation: The source file could not be opened, because the record length is greater than the specified *number* of bytes:

For z/OS:
32 760 bytes

For z/VM:
65 535 bytes

User response: Reduce the record length of the source file.

FANENV0691W CEXEC file type truncated: source file type has 8 characters

Explanation: The file type of the compiled EXEC, which is C concatenated with the source file type, was truncated because it was longer than 8 characters.

User response: Either specify a valid file type for the compiled EXEC on the CEXEC option or change the file type of the source file.

FANENV0692S No blank between ")" and next option; next option ignored

Explanation: There is no blank between the close parenthesis of a keyword parameter in a compiler option and the next compiler option. The next compiler option is ignored.

User response: Insert a blank between the compiler options.

FANENV0693T DUMP value not a whole number in the range 0-2047: *value*

Explanation: The value of the DUMP (DU) compiler option is not a whole number in the range 0 through 2047.

User response: Invoke the Compiler again with a valid value for DUMP.

FANENV0694T Incorrect RECFM value for *ddname*

Explanation: See “Standard Data Sets Provided for the Compiler” on page 14 for a list of the valid RECFM values.

User response: Specify an appropriate output data set.

FANENV0695T Incorrect BLKSIZE value for *ddname*

Explanation: See “Standard Data Sets Provided for the Compiler” on page 14 for a list of the valid BLKSIZE values.

User response: Specify an appropriate output data set.

FANENV0696T Incorrect LRECL value for *ddname*

Explanation: See “Standard Data Sets Provided for the Compiler” on page 14 for a list of the valid LRECL values.

User response: Specify an appropriate output data set.

FANENV0697T DSORG=PO but no member name given: *ddname*

Explanation: The data set name associated with the *ddname* corresponds to a partitioned data set, but no member name has been given.

User response: Either specify a member name or correct the data set name to correspond to a sequential data set.

FANENV0698T DSORG=PS but member name given: *ddname*

Explanation: The data set name associated with the *ddname* corresponds to a sequential data set, but a member name has been given.

User response: Either omit the member name or correct the data set name to correspond to a partitioned data set.

FANENV0703S LIBLEVEL not a whole number in range 2-6, or "*": *value*

Explanation: The value of the LIBLEVEL (LL) compiler option is neither a whole number in the range 2 through 6 nor "*".

User response: Invoke the compiler again with a valid LIBLEVEL option.

FANCON0704S Full TRACE support requires runtime level *value*

Explanation: You have requested full TRACE support for your compiled program (TRACE and SLINE compiler options) but the Library level specified in the LIBLEVEL option is too low.

User response: Do one of the following:

- Invoke the compiler again using a higher value for the LIBLEVEL option. The "value" value in the message indicates the minimum Library level required for full TRACE support. For more information refer to “LIBLEVEL” on page 28.
- Compile the program without the TRACE and SLINE options.

FANCOD0705S CONDENSE requires runtime level *value*

Explanation: You have requested that your compiled program be condensed (CONDENSE compiler options) but the Library level specified in the LIBLEVEL option is too low.

User response: Do one of the following:

- Invoke the compiler again using a higher value for the LIBLEVEL option. The "value" value in the message indicates the minimum Library level required for full CONDENSE support. For more information refer to “LIBLEVEL” on page 28.
- Compile the program without the CONDENSE option.

FANxxx0706S Runtime level *value* **needed**

Explanation: You specified the LIBLEVEL(x) compiler options and the compiler has detected a language feature that requires a higher level of the Library. The error marker symbol usually points to the start of the clause containing the language feature.

User response: Do one of the following:

- Invoke the compiler again using a higher value for the LIBLEVEL option. The "value" value in the message indicates the minimum Library level required for the language feature. For more information refer to “LIBLEVEL” on page 28.
- Rewrite the clause indicated by the error message.

FANENV0708T The ALTERNATE option requires the SLINE option

Explanation: When specifying the ALTERNATE Compiler option, the SLINE option is required. The Alternate Library cannot prepare the control blocks needed by the interpreter if the source of the REXX program is not included at compilation time using the SLINE option.

User response: Compile the REXX program again, specifying both the ALTERNATE and SLINE Compiler options.

FANENV0709W DLINK has no effect when running with the Alternate Library

Explanation: The DLINK option supports a direct link of an external subroutine or function when a module is generated from OBJECT output. This option is supported by the Library, but not by the Alternate Library. The Alternate Library runs the compiled REXX program by invoking the interpreter; the standard system search order is used.

User response: When distributing the compiled REXX programs, include the external subroutines and functions that are directly linked for the Library as separate modules for the Alternate Library.

FANENV0710T The TRACE option requires the SLINE option

Explanation: When specifying the TRACE option, the SLINE (or **SLINE(AUTO))** option is required.

User response: Recompile the program specifying both the TRACE and SLINE options.

FANENV0711T DLINK and TRACE must not be specified together

Explanation: These options are mutually exclusive.

User response: Omit one of the two options.

FANENV0712T DLINK and CONDENSE must not be specified together

Explanation: These options are mutually exclusive. A condensed program cannot be used with DLINK.

User response: Omit one of the two options.

FANxxx0713I Message repository not found. Hardcoded messages used.

Explanation: The Compiler could not load the message repository. This means that it could not locate the file containing the error and informational messages and make it available to the compiler run.

Note:

1. Subsequent messages are issued exactly as if the user had installed the delivered FANUME REPAMENG repository.
2. If the repository cannot find the message number, no error indication occurs. Instead the same message is issued as if the FANUME REPAMENG repository has been installed.

User response: Ask your administrator for help.

FANENV0718T Left MARGINS value not a whole number in the range *range***:** *margins*

Explanation: The left margin specified by the MARGINS compiler option must be a whole number. For more information refer to “MARGINS” on page 30.

User response: Invoke the compiler again with a valid MARGINS option.

FANENV0719T Right MARGINS value not a whole number in the range left margin *margin* **or "*":** *margins*

Explanation: The right margin specified by the MARGINS compiler option is neither '*' nor a whole number in the range left *margin*. For more information refer to “MARGINS” on page 30.

User response: Invoke the compiler again with valid values for the MARGINS option.

FANGAO0770S Invalid number of arguments in built-in function

Explanation: The number of arguments you passed to a built-in function is either of the following:

- Less than the number of required arguments for the function
- Greater than the number of arguments defined for the function.

User response: Check your code and correct it.

FANENV0771S *option* **ignored because of missing "("**

Explanation: The *option* is ignored because the command used to invoke the Compiler did not contain an open parenthesis to mark the start of the options list.

User response: Reissue the command after typing an open parenthesis between the source-file identifier and the options list.

FANGAO0772W SOURCELINE built-in function used and SL option not specified

Explanation: The Compiler found a reference to the SOURCELINE built-in function and the SLINE compiler option (abbreviation: SL) was not specified. The full functions of the SOURCELINE function are available only if the program is compiled with the SLINE or SLINE(AUTO) compiler option. For more information on using the SOURCELINE function with the Compiler, see “SOURCELINE Built-In Function” on page 94.

User response: To use the full functions of the SOURCELINE function, recompile the program with the SLINE option.

FANGAO0773I Instruction might never be executed

Explanation: The compiler has found that a section of code starting at the marked point cannot be reached during execution of the program. Such cases occur when the code is not labelled or the label is not valid or is defined several times, and the preceding instruction transfers control to another part of the program. Instructions that transfer control are EXIT, ITERATE, LEAVE, RETURN, and SIGNAL (without ON or OFF), as well as IF and SELECT instructions that contain such instructions after every THEN and ELSE/OTHERWISE.

User response: If the code is unreachable because you have forgotten a label, misspelled it, or defined it several times, or because of mismatched DO/END clauses, correct the error. If you do not want the code to be executed, but do not wish to remove it completely, it is more efficient to enclose it in a comment. Code that is not normally executable can still be executed using the SOURCELINE built-in function in connection with INTERPRET, for example.

FANGAO0774I Number of arguments in standard function not valid

Explanation: A function of an IBM supplied standard function package is used with the wrong number of arguments.

User response: Correct the number of arguments.

FANENV0800E Invalid suboption in OLDDATE option is not C, P, I or O

Explanation: Only the C, P, I, and O are valid for the OLDDATE option.

User response: Specify the correct letters C, P, I, or O as described in "OLDDATE" on page 32.

FANENV0801E OLDDATE option not supported if system is not Y2K ready

Explanation: You have specified the OLDDATE option, but your system is not yet enabled for the year 2000.

User response: You can only use the OLDDATE option, if your system has been enabled for the year 2000.

FANENV0802E OLDDATE option not supported, missing DMSPLU service

Explanation: The OLDDATE option is only supported, if the DMSPLU service is provided by your z/VM installation.

User response: You must not use the OLDDATE option. Ask your system administrator for help.

FANENV0803E Error processing OLDDATE for *file*

Explanation: An error occurred when processing the OLDDATE option for the specified file.

User response: Check your code and correct the error. For more information refer to "OLDDATE" on page 32.

FANPAR0849W SAA: Source expression in assignment is missing

Explanation: This message warns of noncompliance with SAA guidelines. The Compiler did not find an expression after the assignment operator (=).

User response: To assign a null string ('') to the variable, code it after the =.

FANPAR0850W SAA: UPPER instruction not part of SAA Procedures Language

Explanation: This message warns of noncompliance with SAA guidelines. The Compiler found an UPPER instruction in the program. The UPPER instruction is supported by the Compiler, but is not part of the SAA REXX interface.

User response: Use the TRANSLATE built-in function instead.

FANPAR0851W SAA: PARSE EXTERNAL not part of SAA Procedures Language

Explanation: This message warns of noncompliance with SAA guidelines. The Compiler found a PARSE EXTERNAL instruction in the program. PARSE EXTERNAL is supported by the Compiler, but is not supported by the SAA REXX interface.

User response: Use PARSE PULL instead.

FANPAR0852W SAA: PARSE NUMERIC not part of SAA Procedures Language

Explanation: This message warns of noncompliance with SAA guidelines. The Compiler found a PARSE NUMERIC instruction in the program. PARSE NUMERIC is supported by the Compiler, but is not supported by the SAA REXX interface.

User response: Use the DIGITS, FORM, or FUZZ built-in functions instead.

FANPAR0854W SAA: "@", "#", "$", "¢" might not be used in symbols

Explanation: This message warns of noncompliance with SAA guidelines. The Compiler found one of the following characters in a symbol:

```
@  # $ ¢
```

The use of these characters in symbols is supported by the Compiler, but is not supported by the SAA REXX interface. This message is not issued when compiling with the SAA compiler option while OPTIONS 'ETMODE' is in effect.

User response: Change the symbol.

FANPAR0855W SAA: Literal strings must be completely on one line

Explanation: This message warns of noncompliance with SAA guidelines. The Compiler found a literal string that crosses a line boundary. Such strings are supported by the Compiler, but are not supported by the SAA REXX interface.

User response: Either put the entire string on one line of the source file, or divide the string into smaller strings and concatenate those strings. For example, the assignment:

```
title = 'Director of European Sales and Marketing'
```

could be written as:

```
title = 'Director of '||,
        'European Sales and Marketing'
```

FANPAR0856W SAA: "/" must not be used in a comparison operator

Explanation: This message warns of noncompliance with SAA guidelines. The Compiler found a / character being used as part of a comparison operator. This use of the / character is supported by the Compiler, but is not supported by the SAA REXX interface.

User response: Use ¬ or \ instead.

FANGAO0857W SAA: Built-in function not part of SAA Procedures Language

Explanation: This message warns of noncompliance with SAA guidelines. The Compiler found a built-in function that is supported by the Compiler, but is not supported by the SAA REXX interface.

User response: For FIND, use WORDPOS instead. For INDEX, use POS instead. For any other function, change the program to avoid using the function.

FANGAO0858W SAA: Trace prefix ! not part of SAA Procedures Language

Explanation: The message warns of noncompliance with SAA guidelines. The Compiler found a ! character being used as trace prefix. This use of the ! trace prefix is supported by the Compiler, but is not supported by the SAA REXX interface.

User response: Correct the trace prefix.

FANGAO0859S Division by zero

Explanation: The Compiler detected an attempt to divide by zero (/, %, //), which is not valid. The zero divisor can be a constant, a variable, or an expression, which the Compiler recognizes to have a value of zero.

User response: Correct the expression.

FANGAO0860S Not a positive whole number

Explanation: The Compiler expects a number greater than zero in the indicated position. The number can be the operand of a NUMERIC DIGITS instruction or an argument of a built-in function. This operand or argument can be a constant or a variable which the Compiler recognizes to have a value equal to or less than zero; or, it is no number at all.

User response: Correct the operand or argument.

FANGAO0861S Positive whole number or zero required

Explanation: REXX requires a nonnegative numeric value at the indicated position, which can be the operand of a DO, DO FOR, or NUMERIC FUZZ instruction or an argument of a built-in function. This operand or argument can be a constant, variable, or expression. The Compiler recognizes that its value cannot be numeric or, if numeric, cannot be a whole number or be positive or zero.

User response: Correct the operand or argument.

FANGAO0862S Not a whole number in the range 0-99

Explanation: The Compiler expects a number from 0 through 99 as the argument of the ERRORTEXT built-in function. This argument can be a constant or a variable from which the Compiler recognizes that it has a value outside this range.

User response: Correct the argument.

FANGAO0863S Required argument in built-in function missing

Explanation: A required argument of a built-in function has not been specified.

User response: Supply the argument.

FANGAO0864S Argument of built-in function is not a single character

Explanation: A built-in function requires an argument that must be a single character. An argument of another length has been specified.

Note: If the program contains an OPTIONS instruction, the Compiler checks only whether this argument has a length greater than zero.

User response: Supply an argument of 1 character.

FANGAO0865S Argument of built-in function is not a hexadecimal string

Explanation: An X2C or X2D built-in function requiring a hexadecimal first argument has been supplied with a wrong argument. This argument is a constant, a variable, or an expression which the Compiler recognizes to have an invalid value.

User response: Supply a hexadecimal argument.

FANGAO0866S Invalid option in built-in function invocation

Explanation: An option of a built-in function has an incorrect value, for example:

```
TIME('G'),TIME('GMT')
```

User response: Supply a correct option.

FANGAO0867W SAA: Option in built-in function invocation invalid under SAA

Explanation: This message warns of noncompliance with SAA guidelines. An option of a built-in function has a value that is supported by the Compiler, but not by the SAA REXX interface. For example:

```
DATE('C'),DATE('Century')
```

User response: Supply a correct option.

FANGAO0868S RANDOM() BIF: either min>max or (max-min)>100000

Explanation: The values found for the max argument, the min argument, or both in an invocation of a RANDOM built-in function are not valid for one of the following reasons:

- The min argument is greater than the max argument.
- The difference max-min is greater than 100000.

Either one or both of the arguments might have resulted because of defaulting. For example, RANDOM(2000,) is not a valid min argument because the max argument defaults to 999.

User response: Specify values for the arguments or allow them to default so as to comply with the rules specified above.

FANGAO0869S Expression must evaluate to SCIENTIFIC or ENGINEERING

Explanation: The expression following NUMERIC FORM must evaluate to SCIENTIFIC or ENGINEERING.

User response: Correct the expression.

FANFMU0870S More than 65534 external routine invocations

Explanation: When the DLINK option is specified, the Compiler cannot process a program containing invocations of more than 65 534 external procedures or functions.

User response: Reduce the number of external routines or specify the NODLINK option.

FANFMU0871T Size of object module exceeds 16MB

Explanation: The size of an object module (that is, core image) created by the REXX compiler is limited to **16MB**. This restriction applies to both CEXEC and OBJECT output.

User response: If you have not used the SOURCELINE built-in function in your program, you should compile with NOSLINE to avoid incorporating the source statements into your object module.

Try to reduce the size of the REXX source program by dividing it into several sources that can be compiled individually. Obvious candidates for forming new sources are any PROCEDURE subprograms without EXPOSE.

FANGAO0872I Positive whole number or zero expected

Explanation: This argument of the function GETMSG must be a positive number or zero.

User response: Correct the argument.

FANGAO0873I Asterisk, blank, or nonnegative number expected

Explanation: This argument of the function OUTTRAP must be a positive number or zero, or a string consisting of one asterisk.

User response: Correct the argument.

FANGAO0874I Argument should have 8 hexadecimal digits

Explanation: This argument of the function STORAGE must be a string in the range of 1 to 8 hexadecimal digits.

User response: Correct the argument.

FANGAO0875I Argument should be a nonnegative whole number

Explanation: This argument of the function STORAGE must be a positive whole number or zero.

User response: Correct the argument.

FANGAO0878S Separator arg of DATE incompatible with argument *argument*

Explanation: You specified a separator for the output or input date, although the corresponding date format does not allow for a separator. The formats permitting no separator are B, C, D, J, M and W. A zero-length string, too, is a separator and therefore not permitted.

User response: Remove the separator argument. For example, `DATE("C", X, Y, "", Z)` is wrong, but `DATE("C", X, Y, , Z)` is correct.

FANGAO0879S Separator arg (4 or 5) of DATE exceeds one character

Explanation: The separator for a date format must not be longer than one character.

User response: Replace the invalid argument with a string that contains no or a single character.

FANGAO0880S Argument of built-in function is not a binary string

Explanation: A B2X built-in function requiring a binary first argument has been supplied with a wrong argument. This argument is a constant, a variable, or an expression, which the Compiler recognizes to have an invalid value.

User response: Supply a binary argument.

FANGAO0881S TRACE option is not valid

Explanation: The option in a TRACE instruction or a use of the TRACE built-in function is not valid. One of the following options contain a value that is not valid: a constant, a variable, or an expression.

User response: Check your code and correct it.

FANGAO0882E Derived variable name longer than 250 characters

Explanation: The Compiler predicts that at runtime after substitution of values of variables into a compound symbol, the length of the resulting name will be greater than the limit of 250 characters.

User response: Check your code and correct it.

FANGAO0883S Argument is not an unbracketed DBCS string

Explanation: The DBCS processing function DBBRACKET requires an argument that consists of at least one pair of bytes, each pair being a valid EBCDIC DBCS character. The SO and SI characters must not be present. Valid pairs are:

- Two EBCDIC blanks
- Two characters with hexadecimal values in the range of '41'X to, and including, 'FE'X

User response: Correct the argument value.

FANGAO0884S Argument is not a valid DBCS string

Explanation: This argument to a DBCS processing function must be a valid DBCS string or mixed string. The argument can contain SBCS parts, in which any character other than SO and SI is permitted, and DBCS parts. A DBCS part starts with SO and ends with SI. Between SO and SI there must be pairs of bytes, each pair being a valid EBCDIC DBCS character. Valid pairs are:

- Two EBCDIC blanks
- Two characters with hexadecimal values in the range of '41'X to, and including, 'FE'X

User response: Correct the argument value.

FANGAO0885S Argument is not a single bracketed DBCS string

Explanation: The DBCS processing function DBUNBRACKET requires an argument consisting of a single pure DBCS string. A valid argument value starts with SO and ends with SI. Between SO and SI there must be pairs of bytes, each pair being a valid EBCDIC DBCS character. Valid pairs are:

- Two EBCDIC blanks
- Two characters with hexadecimal values in the range of '41'X to, and including, 'FE'X

User response: Correct the argument value.

FANGAO0886I Argument is not one of the permitted values

Explanation: The value defined for the first argument of a system function is not allowed. For example, the value of the first argument of the ASSGN built-in function (BIF) must be "STDIN" or "STDOUT". The following system BIFs are verified:

z/VM specific functions:
APILOAD, CMSFLAG, CSL, DIAG, DIAGRC, SOCKET, STORAGE

z/OS specific functions:
STORAGE, GETMSG, LISTDSI, MSG, MVSVAR, OUTTRAP, PROMPT, SETLANG, SYSCPUS, SYSDSN, SYSVAR

VSE specific functions:
STORAGE, OUTTRAP, ASSGN

User response: Correct the argument.

FANGAO0887I Incompatible arguments to ASSGN

Explanation: The first argument to the function ASSGN is "STDIN"and the second is "SYSLST", or the first is "STDOUT" and the second is "SYSIPT".

User response: Change one of the arguments.

FANGAO0888W Argument must be a single SBCS or DBCS character

Explanation: This argument to a DBCS processing function must consist of a single SBCS or DBCS character. It must be a single character other than SO and SI, or four bytes consisting of SO, a pair of bytes representing a valid DBCS character, and SI. Valid pairs are:

- Two EBCDIC blanks
- Two characters with hexadecimal values in the range of '41'X to, and including, 'FE'X

User response: Correct the argument.

FANGAO0889I Argument should be name of a simple variable or stem

Explanation: This argument to the function GETMSG or OUTTRAP must be a string containing a valid name for a simple variable or stem. A valid string can contain alphanumeric characters, exclamation marks (!), question mark (?), and underscores (_). It must start with an alphabetic character and can end with a period.

User response: Correct the argument.

FANENV0890T Incorrect LRECL value for SYSIEXEC, expected/found: *option1/option2*

Explanation: The LRECL value found for the SYSIEXEC output (*option2*) is incorrect. The compiler expected *option1*. Refer to "IEXEC" on page 27 for information on how to calculate the record length.

User response: Specify an output data set with the correct LRECL.

FANENV0891T Incorrect RECFM value (F|FB) for SYSIEXEC, input records vary in length

Explanation: The data set specified for the SYSIEXEC output has fixed-length records but the input contains records of different length. Input means, in z/OS, all data sets in the SYSIN concatenation and, in z/OS and z/VM, files inserted into the compilation using %INCLUDE directives. Records of different length are the result of a split of the source lines if the source text is found on the same line as the %INCLUDE directive.

User response: Specify a data set for SYSIEXEC with variable-length records.

FANENV0892T Incorrect RECFM value (F|FB) for SYSIEXEC, input with RECFM=V|VB

Explanation: The data set specified for the SYSIEXEC output has fixed-length records but one or more of the input data sets has a record format of V or VB (variable length).

User response: Specify a data set for SYSIEXEC with variable-length records or change the input data sets such that they all have a record format of F or FB and the record lengths (LRECL) are identical.

FANENV0893T Incorrect RECFM value (F|FB) for SYSIEXEC, input with/without seq no

Explanation: The data set specified for the SYSIEXEC output has fixed-length records, but some of the input data sets contain sequence numbers and some do not.

User response: Either specify an output data set with variable-length records or change the input data sets such that either all or none of them have sequence numbers.

FANCON0900T Source data set cannot be opened

Explanation: The Compiler was unable to open the SYSIN data set.

User response: Check that:

- If the Compiler was invoked from a batch job, a DD statement with DD name SYSIN was provided in the job step in which the Compiler was invoked.
- If the Compiler was invoked in a TSO session, a TSO ALLOC command for DD name SYSIN is in effect when the compiler is invoked.
- The data set is accessible when the Compiler is invoked.

FANxxx0901T Source data set cannot be read

Explanation: The Compiler was unable to read the SYSIN data set containing the REXX source program to be compiled.

User response: Check that the data set is accessible when the Compiler is invoked.

FANENV0902T *dataset-name* **output data set must not be identical with %INCLUDE data set**

Explanation: The data set name of one of the %INCLUDE data sets is equal to the data set name of one of the output data sets. The value of *dataset-name* shows which output data set name is wrong:
CEXEC refers to the compiled EXEC.
IEXEC refers to the expanded IEXEC output.
OBJECT refers to the object data set.
PRINT refers to the compiler listing.
TERM refers to the terminal output.
DUMP refers to the DUMP output.

User response: Specify a different name for the output data set.

FANENV0903T *option* **output data set name must not be identical with source data set name**

Explanation: One of the output data sets and the source data set have the same data set name. The value of *option* indicates which output data set name is in error. CEXEC refers to the compiled EXEC, IEXEC refers to the expanded (IEXEC) output, OBJECT refers to the OBJECT data set, PRINT refers to the compiler listing, TERM refers to the terminal output, and DUMP refers to the DUMP output.

User response: Specify a different name for the output data set.

FANENV0904T *option1*/*option2* **output data set names must not be identical**

Explanation: You have specified the same name for more than one of the output data sets. The message indicates which data set names are identical. CEXEC refers to the compiled EXEC, IEXEC refers to the expanded (IEXEC) output, OBJECT refers to the OBJECT data set, PRINT refers to the compiler listing, TERM refers to the terminal output, and DUMP refers to the DUMP output.

User response: Specify a unique name for each output data set.

FANFMU0906T Error opening CEXEC data set

Explanation: The Compiler was unable to open the SYSCEXEC data set.

User response: Check that:

- If the Compiler was invoked from a batch job, a DD statement with DD name SYSCEXEC was provided in the job step in which the Compiler was invoked.
- If the Compiler was invoked in a TSO session, a TSO ALLOC command for DD name SYSCEXEC is in effect when the compiler is invoked.
- The data set is accessible when the Compiler is invoked.

FANFMU0907T Error opening OBJECT data set

Explanation: The Compiler was unable to open the SYSPUNCH data set.

User response: Check that:

- If the Compiler was invoked from a batch job, a DD statement with DD name SYSPUNCH was provided in the job step in which the Compiler was invoked.
- If the Compiler was invoked in a TSO session, a TSO ALLOC command for DD name SYSPUNCH is in effect when the compiler is invoked.
- The data set is accessible when the Compiler is invoked.

FANFMU0908T Error writing to CEXEC data set

Explanation: The Compiler was unable to write to the SYSCEXEC data set.

User response: Check that the data set is accessible when the Compiler is invoked.

FANFMU0909T Error closing CEXEC data set

Explanation: The Compiler was unable to close the SYSCEXEC data set.

User response: Check that the data set is accessible when the Compiler is invoked.

FANFMU0910T Error writing to OBJECT data set

Explanation: The Compiler was unable to write to the SYSPUNCH data set.

User response: Check that the data set is accessible when the Compiler is invoked.

FANFMU0911T Error closing OBJECT data set

Explanation: The Compiler was unable to close the SYSPUNCH data set.

User response: Check that the data set is accessible when the Compiler is invoked.

FANCON0912T Error closing source data set

Explanation: The Compiler was unable to close the SYSIN data set containing the REXX source program to be compiled.

User response: Check that the data set is accessible when the Compiler is invoked.

FANLIS0913T Error opening PRINT data set

Explanation: The Compiler was unable to open the SYSPRINT data set.

User response: Check that:

- If the Compiler was invoked from a batch job, a DD statement with DD name SYSPRINT was provided in the job step in which the Compiler was invoked.
- If the Compiler was invoked in a TSO session, a TSO ALLOC command for DD name SYSPRINT is in effect when the compiler is invoked.
- The data set is accessible when the Compiler is invoked.

FANLIS0914T Error writing to PRINT data set

Explanation: The Compiler was unable to write to the SYSPRINT data set.

User response: Check that the data set is accessible when the Compiler is invoked.

FANLIS0915T Error closing PRINT data set

Explanation: The Compiler was unable to close the SYSPRINT data set.

User response: Check that the data set is accessible when the Compiler is invoked.

FANENV0916T Source data set cannot be opened: record length greater than 32760

Explanation: The Compiler was unable to open the source file, because it contains records longer than 32 760 characters (bytes).

User response: Reorganize the source file so that the value of the LRECL parameter of the DCB statement is less than or equal to 32 760. See "Standard Data Sets Provided for the Compiler" on page 14.

FANENV0917T Error opening DUMP data set

Explanation: The Compiler was unable to open the SYSDUMP data set.

User response: Check that:

- If the Compiler was invoked from a batch job, a DD statement with DD name SYSDUMP was provided in the job step in which the Compiler was invoked.
- If the Compiler was invoked in a TSO session, a TSO ALLOC command for DD name SYSDUMP is in effect when the compiler is invoked.
- The data set is accessible when the Compiler is invoked.

FANENV0918T Error writing to DUMP data set

Explanation: The Compiler was unable to write to the SYSDUMP data set.

User response: Check that the data set is accessible when the Compiler is invoked.

FANENV0919T Error closing DUMP data set

Explanation: The Compiler was unable to close the SYSDUMP data set.

User response: Check that the data set is accessible when the Compiler is invoked.

FANTOK0920T Source data set is empty

Explanation: The SYSIN data set contains no records at all.

User response: Makes sure that the SYSIN data set contains the source program you want to compile.

FANLIS0921T Error opening TERM output

Explanation: The Compiler could not open the target destination for terminal output.

Under z/OS, the output is directed to the destination specified in the SYSTERM DD statement (or TSO ALLOC command).

Under z/VM, the output is directed to the user's terminal unless the Compiler is running in a batch machine, in which case output is directed to the Console Log. The error can occur if the Compiler was unable to get the space needed for work areas.

Note: You will only see this message in the printed output. However, even if there is no printed output, for example if NOPRINT is in effect, the return code passed from the Compiler to the system, at the end of the Compiler run, will correspond to the severity of this message.

User response:

- Under z/OS, check that:
  - If the compiler was invoked in a batch job, a DD statement with DD name SYSTERM was provided in the job step in which the compiler is invoked.
  - If the Compiler was invoked in a TSO session, a TSO ALLOC command for DD name SYSTERM is in effect when the Compiler is invoked.
  - The data set is accessible when the Compiler is invoked.
- Under z/VM, compile without the TERM compiler option, obtain more storage by releasing a minidisk or SFS directory, or by deleting a nucleus extension. Alternatively, define a larger virtual storage size for the virtual machine and re-IPL CMS.

FANLIS0922T Error writing to TERM

Explanation: The Compiler was unable to write to the target destination for terminal output.

Under z/OS, the output is directed to the destination specified in the SYSTERM DD statement (or TSO ALLOC command).

Under z/VM, the output is directed to the user's terminal. The most likely cause of the error is that the virtual screen is not defined or insufficient storage was available to execute the request.

Note: It is very unlikely that you will ever see this message. The Compiler first writes the **PRINT** output, then closes it. Only after the **PRINT** output has been closed, the Compiler writes the **TERM** output. If an error occurs while writing the **TERM** output, there is nowhere to write this error message. However, the return code that the Compiler passes back to the system at the end of the Compiler run corresponds to the severity of this message.

User response:

- Under z/OS, check that the data set is accessible when the Compiler is invoked.
- Under z/VM, compile without the TERM compiler option, define the virtual screen, or obtain more storage by releasing a minidisk or SFS directory, or by deleting a nucleus extension. Alternatively, define a larger virtual storage size for the virtual machine and re-IPL CMS.

FANxxx0923T Error closing TERM output

Explanation: The Compiler was unable to close the target destination for terminal output.

Under z/OS, the output is directed to the destination

specified in the SYSTERM DD statement (or TSO ALLOC command).

Under z/VM, the output is directed to the user's terminal unless the Compiler is running in a batch machine in which case the output is directed to the Console Log. The most likely cause of the error is that a release storage request has failed.

User response:

- Under z/OS, check that the data set is accessible when the Compiler is invoked.
- Under z/VM, compile without the TERM compiler option or notify your system support personnel if the problem persists.

Note: It is very unlikely that you will ever see this message. The Compiler first writes the **PRINT** output, then closes it. Only after the **PRINT** output has been closed, the Compiler writes the **TERM** output. If an error occurs while closing the **TERM** output, there is nowhere to write this error message. However, the return code that the Compiler passes back to the system at the end of the Compiler run corresponds to the severity of this message.

FANENV0924T Error opening virtual printer for DUMP

Explanation: The Compiler could not open the virtual printer for DUMP output. This problem can occur if the virtual printer is not operational or if the Compiler was unable to get the space needed for work areas.

User response: Compile with the NODUMP compiler option, make the virtual printer operational, or obtain more storage by releasing a minidisk or SFS directory, or by deleting a nucleus extension. Alternatively, define a larger virtual storage size for the virtual machine and re-IPL CMS.

FANENV0925T Error writing to virtual printer for DUMP

Explanation: An error occurred when writing to the virtual printer. The most likely cause of this message is a full disk or a non operational virtual printer.

User response: Compile with the NODUMP compiler option or make the virtual printer operational.

FANCON0926T Error closing virtual printer for DUMP

Explanation: The virtual printer could not be closed. The most likely cause of this is that a release storage request has failed.

User response: Compile with the NODUMP compiler option or notify your system support personnel if the problem persists.

FANENV0927S Error opening %INCLUDE input

Explanation: The Compiler was unable to open a file specified in a %INCLUDE directive. Either the file specified does not exist or the file specification contains characters that are invalid in your Operating System.

Under z/VM, the problem can occur if:

- The file you are including does not exist with file type **COPY**, **REXXINCL**, or EXEC on:
  - The accessed disks, for `/*%INCLUDE fn*/` directives
  - The specified collection, for `/*%INCLUDE ddname(filename) /*` with `FILEDEF ddname DISK fn ft [fm]*/` directives
- The file you are including does not exist on:
  - The specified **MACLIB**, for `/*%INCLUDE maclib(fn)*/` directives
  - The **MACLIB**s established with the **GLOBAL MACLIB** command, for `/*%INCLUDE SYSLIB(fn)*/` directives.
- The specification of the file to be included contains invalid characters
- You are including a file from a minidisk for which you have read-only access, while someone with read/write access to that minidisk has altered the file so that it no longer exists in the same place on the minidisk.

User response:

- Under z/OS, check that:
  - If the Compiler was invoked in a batch job, a DD statement with a DD name identical with the DD name given or defaulted in the %INCLUDE directive is present.
  - If the Compiler was invoked in a TSO session, a TSO ALLOC command for a DD name identical with the DD name given or defaulted in the %INCLUDE directive is in effect when the Compiler is invoked.
  - The data set is accessible when the Compiler is invoked.
  - Check that a member with the specified name is present in one of the libraries concatenated under the DD name specified or defaulted in the %INCLUDE directive at the time the Compiler is invoked.
- Under z/VM, make sure that the file exists, the file specification contains valid characters, or reaccess the minidisk on which the file to be included resides.

FANENV0928S Error reading %INCLUDE input

Explanation: The Compiler was unable to read from a file specified in a %INCLUDE directive.

Under CMS, the problem can occur when you are including a file from a minidisk to which you have read-only access, while someone with read/write access to that minidisk has altered the file so that it no longer exists in the same place on the minidisk.

User response:

- Under z/OS, check that the specified member is accessible when the Compiler is invoked.
- Under z/VM, reaccess the minidisk that contains the file to be included.

FANENV0929S Error closing %INCLUDE input

Explanation: The Compiler was unable to close a file specified in a %INCLUDE directive.

User response:

- Under z/OS, check that the data set is accessible when the Compiler is invoked.
- Under z/VM, reaccess the minidisk that contains the file to be included.

FANENV0930T Error opening IEXEC output

Explanation: The compiler was unable to open the target destination for IEXEC output.

Under z/OS, the output is directed to the destination specified in the SYSIEXEC DD statement (or TSO ALLOC command).

Under z/VM, this problem can occur if your virtual machine does not have read/write access to the specified minidisk.

User response:

- Under z/OS, check that the data set is accessible when the Compiler is invoked.
- Under z/VM, use a minidisk to which your virtual machine has read/write access.

FANENV0931T Error writing to IEXEC output

Explanation: The compiler was unable to write to the target destination for IEXEC output.

Under z/OS, the output is directed to the destination specified in the SYSIEXEC DD statement (or TSO ALLOC command).

Under z/VM, this problem can occur if your virtual machine does not have read/write access to the specified minidisk.

User response:

- Under z/OS, check that the data set is accessible when the Compiler is invoked.
- Under z/VM, use a minidisk to which your virtual machine has read/write access.

FANENV0932T Error closing IEXEC output

Explanation: The compiler was unable to close the target destination for IEXEC output.

User response:

- Under z/OS, check that the data set is accessible when the Compiler is invoked.
- Under z/VM, compile with the NOIEXEC compiler option or notify your system support personnel if the problem persists.

FANENV0934E Invalid %INCLUDE directive

Explanation: The file specification in the %INCLUDE directive contains embedded blanks, or the length of the name specified for `member`, `ddname`, or `filename` exceeds 8 characters.

User response: Correct the %INCLUDE directive.

FANPAR0935E Option for %SYSDATE or %SYSTIME not valid

Explanation: The option specified for %SYSDATE or %SYSTIME is too complex for the compiler. The option must be a single symbol or quoted string and must only contain alphanumeric characters of which only the first character is significant.

User response: Simplify or correct the option.

FANPAR0936E Options R and E not valid for %SYSTIME

Explanation: The elapsed-time options R and E cannot be used for the compilation time.

User response: Specify a different option.

FANPAR0937E %SYSDATE/%SYSTIME is not allowed within a clause

Explanation: A %SYSDATE or %SYSTIME control directive can only be used where a REXX statement is allowed.

User response: Insert a semicolon in front of the control directive or write the control directive on a separate line.

FANxxx0938W IEXEC file type truncated: source file type has 8 characters

Explanation: The default file type is the letter I concatenated with the source file type. In this case, the resulting file type exceeds 8 characters in length, that is why it is truncated.

User response: Specify a correct file type. For more information refer to "LIBLEVEL" on page 28.

FANENV0939E Stub name too long

Explanation: The stub name is too long. It can consist of up to 8 characters.

User response: Enter a correct stub name. For more information refer to "Stubs" on page 211.

FANENV0940E Stub name missing

Explanation: The stub name could not be found.

User response: Enter a correct stub name. For more information refer to "Stubs" on page 211.

FANENV0941W Duplicate %STUB directive: Only first occurrence on line *number* used

Explanation: A duplicate %STUB directive was found. Only the first occurrence on the specified line is used.

User response: Check and correct your source code.

FANENV0942E Stub name not one of the allowed values

Explanation: The name of the stub is not allowed.

User response: Enter a correct stub name. For more information refer to "Stubs" on page 211.

FANENV0943I Stub *name* included

Explanation: The stub name is included.

User response: None.

FANENV0944S Error opening DDNAMES input

Explanation: An error occurred when the ddnames input was opened. The data set specified after the ddname is not correct.

User response: Check the input and specify a correct DDNAMES definition. Check if the ddname specified in the DDNAMES definition points to a valid data set that contains the alternate DDNAMES.

FANENV0945S Error reading DDNAMES input

Explanation: An error occurred when the DDNAMES input was read. The data set specified after the DDNAME is not correct.

User response: Check the input and specify a correct DDNAMES definition. Check if the DD name specified in the DDNAMES definition points to a valid data set that contains the alternate DDNAMES.

FANENV0946S Error closing DDNAMES input

Explanation: An error occurred when the DDNAMES input was closed. The data set specified after the DDNAME is not correct.

User response: Check the input and specify a correct DDNAMES definition. Check if the DD name specified in the DDNAMES definition points to a valid data set that contains the alternate DDNAMES.

FANENV0947T Error in alternate DDNAMES file: line *line number*

Explanation: An error occurred in the alternate DDNAMES file that you have created. The line number where the error was found is displayed.

User response: Check the displayed line of the alternate DDNAMES file and correct the error.

FANxxx9999 FAN repository not found, message *nnn* cannot be retrieved

Explanation: An internal error in the Compiler occurred. Message number *nnn* was not issued, because the message is not defined in the internal Compiler message table and an external message repository is not available.

User response: Contact your system administrator.

FANxxx9999 Message number *nnn*, format 1, line 1, was not found; it was called from REXX in application FAN

Explanation: An internal error in the Compiler occurred. Message number *nnn* was not issued, because the message is neither defined in the external message repository, nor in the internal Compiler message table.

User response: Contact your system administrator.

Chapter 20. Runtime Messages

The Library and the Alternate Library have the same error messages. If you have both libraries installed, do one of the following:

- Change the message prefix for the Alternate Library from **EAGREX** to **EAGALT**. If you are using the z/OS Message Repository, you must recompile the messages.

  Note: Some of the messages coming from the Alternate Library start with **EAGALT** instead of **EAGREX**. However, they are equal to the **EAGREX** messages. For example, if you get message EAGALT0248E, you will find the explanation for this message under EAGREX0248E in this book.
- Move the member **EAGKMENU** into a save data set, and run MMS without the member **EAGKMENU**.

EAGREX0248E Unable to load IBM REXX Library

Explanation: The program cannot be executed, because the Library could not be loaded as a nucleus extension, by means of the NUCXLOAD command. This error occurs if your virtual machine does not have access to the Library or does not have sufficient storage. You cannot run any compiled REXX programs until this problem is corrected.

User response: Ensure that you have access to the disk that contains the Library (EAGRTLIB MODULE). If you already have access, obtain more storage by releasing a minidisk or SFS directory, or by deleting a nucleus extension. Alternatively, define a larger virtual storage size for the virtual machine and re-IPL CMS.

EAGREX0249E Unable to load EAG Message Repository

Explanation: The program cannot be executed for one of the following reasons. In the following text, * is the language identifier.

- The message repository is not installed in the language DCSS, and neither EAGUME TXT* nor EAGUME TEXT was found on an accessed disk.
- You do not have a read/write A-disk, and the message repository has the file type TXT*.
- You do not have enough space on your read/write A-disk, and the message repository has the file type TXT*.

User response: Check that the message repository is available either in the language DCSS or on disk. If it is not available in the language DCSS and its file type is TXT*, check that your read/write A-disk is large enough to store the message repository. If the problem remains unresolved, report it to your IBM representative. See the *IBM Compiler and Library for REXX on System z: Diagnosis Guide* for more information. The values for the language identifier (*) can be found in the corresponding z/VM documentation.

EAGREX0300E Error 3 running compiled *program*, line *nn*: Program is unreadable

Explanation: Refer to the secondary message if one is displayed. Under z/VM, the REXX program could not be read from the minidisk. This problem can occur if you attempt to run a program from a minidisk for which you have read-only access, while someone with read/write access to that minidisk has altered the program so that it no longer exists in the same place on the minidisk.

Under z/OS and VSE/ESA, this message is always followed by a secondary message.

User response: Under z/VM, reaccess the minidisk on which the program resides.

Note: If the length in the REXX LPA library and the length of what was loaded did not match, check if there are old LPA modules in one of your existing libraries. Remove the old library to fix the problem.

EAGREX0301I Compiled EXEC does not have fixed length records

Explanation: The compiled EXEC does not have fixed-length records. The Compiler always uses the fixed-length record format for compiled EXEC files in z/VM, but the record format might have been changed later.

User response: Recompile the program or format it for z/VM by using the REXXF EXEC if the program was imported from z/OS.

EAGREX0302I Program is not a valid compiled EXEC

Explanation: The compiled code in the program file is

not in the format that the Compiler generates.

User response: Recompile the program.

EAGREX0303I Level of IBM REXX Library too low

Explanation: The program cannot be run, because it was compiled for a more recent version of the Library than the one installed on your system, or it contains language features that are not supported by the specified level of the Library.

User response: Do one of the following:

- Run the program on a system with a version of the Library that corresponds to the version of the Compiler used to compile the program.
- If you have access to the source file, recompile the program on the system on which you want to run it.
- Recompile the program with the recommended minimum library level (LIBLEVEL compiler option). For more information refer to "LIBLEVEL" on page 28.

If the error persists after recompilation, notify your system support personnel.

Note: If the length in the REXX LPA library and the length of what was loaded did not match, check if there are old LPA modules in one of your existing libraries. Remove the old library to fix the problem.

EAGREX0304I The program cannot run with the Alternate Library

Explanation: The program has been compiled with the NOALTERNATE compiler option.

User response: Do one of the following:

- Compile the program with the ALTERNATE compiler option.
- Check your installation to make sure that you use the Library.

EAGREX0400E Error 4 running compiled *program*, line *nn*: Program interrupted

Explanation: The system interrupted execution of the REXX program. This is usually caused by your issuing the HI (Halt Interpretation) immediate command under z/OS or z/VM, or the EXECUTIL HI command under z/OS.

User response: Check your code and correct it.

EAGREX0500E Error 5 running compiled *program*, line *nn*: Machine storage exhausted or request exceeds limit

Explanation: The Library was unable to get the storage needed for its work areas and variables. This might have occurred because the program that invoked the compiled program has already used up most of the available storage.

User response: Under z/OS, use a larger region size.

Under z/VM, you can obtain more free storage by releasing a minidisk or SFS directory (to recover the space used for the file directory) or by deleting a nucleus extension. Alternatively, define a larger virtual storage size for the virtual machine and re-IPL CMS.

Under VSE, use a larger partition size.

EAGREX0600E Error 6 running compiled *program*, line *nn*: Unmatched "/*" or quote

Explanation: A comment or literal string was started but never finished.

User response: See the secondary message for more specific information. Correct the literal string or comment.

EAGREX0601I Unmatched quote

Explanation: A literal string was started but never finished.

User response: Check your code and correct it.

EAGREX0602I Unmatched "/*"

Explanation: A comment was started but never finished.

User response: Check your code and correct it.

EAGREX0603I Unmatched shift-out character in DBCS string

Explanation: A literal string or a comment that has unmatched shift-out/shift-in pairs (that is, a shift-out character without a shift-in character or an odd number of bytes between the shift-out and shift-in characters) was processed with OPTIONS 'ETMODE' in effect.

User response: Check your code and correct it.

EAGREX0700E Error 7 running compiled *program*, line *nn*: WHEN or OTHERWISE expected

Explanation: Within a SELECT instruction, at least one WHEN clause (and possibly an OTHERWISE clause) is expected. If any other instruction is found (or no WHEN clause is found before the OTHERWISE) then this message is issued.

User response: Insert one or more WHEN clauses after the SELECT.

EAGREX0800E Error 8 running compiled *program*, line *nn*: Unexpected THEN or ELSE

Explanation: The program tried to execute a THEN or ELSE clause without first executing the corresponding IF or WHEN clause. This error occurs when control is transferred within or into an IF or WHEN construct, or if a THEN or an ELSE is outside the context of an IF or WHEN construct.

User response: See the secondary message for more specific information.

EAGREX0801I Unexpected THEN

Explanation: The program tried to execute a THEN clause without first executing the corresponding IF or WHEN clause. This error occurs when control is transferred to the THEN clause.

User response: Check your code and correct it.

EAGREX0802I Unexpected ELSE

Explanation: The program tried to execute an ELSE clause without first executing the corresponding IF clause. This error occurs when control is transferred to the ELSE clause.

User response: Check your code and correct it.

EAGREX0900E Error 9 running compiled *program*, line *nn*: Unexpected WHEN or OTHERWISE

Explanation: The program tried to execute a WHEN or OTHERWISE clause without first executing the corresponding SELECT instruction. This error occurs when control is transferred to a WHEN or OTHERWISE clause, or if a WHEN or an OTHERWISE appears outside of the context of a SELECT instruction.

User response: See the secondary message for more specific information.

EAGREX0901I Unexpected WHEN

Explanation: The program tried to execute a WHEN clause without first executing the corresponding SELECT instruction. This error occurs when control is transferred to a WHEN clause.

User response: Check your code and correct it.

EAGREX0902I Unexpected OTHERWISE

Explanation: The program tried to execute an OTHERWISE clause without first executing the corresponding SELECT instruction. This error occurs when control is transferred to an OTHERWISE clause.

User response: Check your code and correct it.

EAGREX1000E Error 10 running compiled *program*, line *nn*: Unexpected or unmatched END

Explanation: The program reached an END clause when the corresponding DO loop or SELECT clause was not active. This error can occur if you transfer control into a loop, or if there are too many ENDs in the program. Note that the SIGNAL instruction terminates any current loops, so it cannot be used to transfer control from one place inside a loop to another. Another cause for this message is placing an END immediately after a THEN or ELSE subkeyword or specifying a name on the END keyword that does not match the name of the control variable in a DO clause.

User response: Check your code and correct it.

EAGREX1100E Error 11 running compiled *program*, line *nn*: Control stack full

Explanation: This message is issued if the program exceeds a Library runtime limit.

User response: Check your code and correct it.

EAGREX1101I PROCEDURE nesting exceeds 30000

Explanation: This message is issued if you exceed the limit of 30 000 active procedures. A recursive subroutine that does not terminate correctly could loop until it causes this message to be issued.

User response: Check your code and correct it.

EAGREX1200E Error 12 running compiled *program*, line *nn*: Clause too long

Explanation: The Compiler encountered a clause that exceeds the allowed limit.

User response: Rewrite the clause.

EAGREX1300E Error 13 running compiled *program*, line *nn*: Invalid character in program

Explanation: The string to be interpreted includes an unexpected character outside a literal (quoted) string or comment that is not a blank or one of the following:

- `A-Z a-z 0-9 (Alphanumerics)`
- `@ # $ ¢ . ? ! _ (Name Characters)`
- `& * ( ) - + = \ ¬ ' " ; : < , > / | % (Special Characters)`
- `Any DBCS character when OPTIONS 'ETMODE' is in effect`

User response: Check your code and correct it. In case the program was imported from another system: Verify that the translation of the characters was correct.

EAGREX1400E Error 14 running compiled *program*, line *nn*: Incomplete DO/SELECT/IF

Explanation: On reaching the end of the program (or end of the string in an INTERPRET instruction), it has been detected that there is a DO or SELECT without a matching END, or that a THEN clause or an ELSE clause is not followed by an instruction.

User response: See the secondary message for more specific information.

EAGREX1401I Incomplete DO instruction: END not found

Explanation: No matching END for an earlier DO was found.

User response: Check your code and correct it.

EAGREX1402I Incomplete SELECT instruction: END not found

Explanation: No matching END for an earlier SELECT was found.

User response: Check your code and correct it.

EAGREX1403I Instruction expected after THEN

Explanation: A THEN clause is not followed by an instruction.

User response: Check your code and correct it.

EAGREX1404I Instruction expected after ELSE

Explanation: An ELSE clause is not followed by an instruction.

User response: Check your code and correct it.

EAGREX1500E Error 15 running compiled *program*, line *nn*: Invalid hexadecimal or binary string

Explanation: Hexadecimal strings might not have leading or trailing blanks, and might only have embedded blanks at byte boundaries. Only the digits 0-9 and the letters a-f and A-F are allowed. Similarly, binary strings might only have blanks added at the boundaries of groups of four binary digits, and only the digits 0 and 1 are allowed.

User response: Check your code and correct it.

EAGREX1600E Error 16 running compiled *program*, line *nn*: Label not found

Explanation: The label specified in a SIGNAL instruction, or specified by the result of the expression on a SIGNAL VALUE instruction, could not be found. There might be an error in the expression or the label might not have been defined.

User response: Check your code and correct it.

EAGREX1601I Label reference in SIGNAL is mixed case, but label is uppercase

Explanation: The label specified in a SIGNAL instruction, or by the result of the expression on a SIGNAL VALUE instruction is a mixed-case string, but the name of the label that probably is intended to be referenced is defined in uppercase.

User response: Change the expression so that it results in an uppercase string.

EAGREX1700E Error 17 running compiled *program*, line *nn*: Unexpected PROCEDURE

Explanation: A PROCEDURE instruction was encountered in an incorrect position. This error is caused by "dropping through" into a PROCEDURE instruction, rather than invoking it properly by a CALL instruction or a function reference.

User response: Check your code and correct it.

EAGREX1800E Error 18 running compiled *program*, line *nn*: THEN expected

Explanation: All IF clauses and WHEN clauses in REXX must be followed by a THEN clause. Some other clause was found when a THEN clause was expected.

User response: Check your code and correct it.

EAGREX1900E Error 19 running compiled *program*, line *nn*: String or symbol expected

Explanation: On a SIGNAL or CALL instruction a literal string or a symbol was expected but neither was found.

User response: See the secondary message for more specific information.

EAGREX1901I CALL not followed by routine name/ON/OFF

Explanation: The name of a routine, or ON with a condition name, or OFF with a condition name is expected in a CALL instruction.

User response: Check your code and correct it.

EAGREX1902I SIGNAL not followed by label name or VALUE/ON/OFF or expression

Explanation: SIGNAL is not followed by a label name, or by ON, or OFF, or VALUE, or an expression.

User response: Check your code and correct it.

EAGREX2000E Error 20 running compiled *program*, line *nn*: Symbol expected

Explanation: In the clauses CALL ON, END, ITERATE, LEAVE, and SIGNAL ON, a single symbol is expected. Either it was not present when required, or some other token was found, or a symbol followed by some other token was found.

Alternatively, the DROP, UPPER, and PROCEDURE EXPOSE instructions expect a list of symbols or variable references. Some other token was found.

User response: See the secondary message for more specific information.

EAGREX2001I Variable expected

Explanation: Some other token was found where a variable was expected.

User response: Check your code and correct it.

EAGREX2002I UPPER list can contain only simple or compound variables

Explanation: The list of variables for the UPPER instruction contains items other than the permitted ones.

User response: Check your code and correct it.

EAGREX2003I NAME not followed by routine name

Explanation: In a CALL ON clause the subkeyword NAME must be followed by the name of a routine.

User response: Check your code and correct it.

EAGREX2004I NAME not followed by label name

Explanation: In a SIGNAL ON clause the subkeyword NAME must be followed by a label name.

User response: Check your code and correct it.

EAGREX2100E Error 21 running compiled *program*, line *nn*: Invalid data at end of clause

Explanation: A clause is followed by some token other than a comment, where no other token was expected.

User response: Check your code and correct it.

EAGREX2200E Error 22 running compiled *program*, line *nn*: Invalid character string

Explanation: Under OPTIONS 'ETMODE' a symbol was detected which contains characters or character combinations not allowed for symbols containing DBCS characters.

User response: Check your code and correct it.

EAGREX2300E Error 23 running compiled *program*, line *nn*: Invalid SBCS/DBCS mixed string

Explanation: A character string that has unmatched shift-out—shift-in pairs (that is, a shift-out character without a shift-in character) or an odd number of bytes between the shift-out—shift-in characters was processed with OPTIONS 'EXMODE' in effect or was passed to a DBCS function.

User response: Correct the character string.

EAGREX2400E Error 24 running compiled *program*, line *nn*: Invalid TRACE request

Explanation: The setting specified on a TRACE instruction starts with a character that does not match one of the valid TRACE settings.

User response: Check your code and correct it.

EAGREX2500E Error 25 running compiled *program*, line *nn*: Invalid subkeyword found

Explanation: The language processor expected a particular subkeyword in an instruction but found something else. For example, in the NUMERIC instruction the second token must be the subkeyword DIGITS, FORM, or FUZZ. If NUMERIC is followed by anything else, this message is issued.

User response: Check your code and correct it.

EAGREX2501I PARSE not followed by a valid subkeyword

Explanation: A PARSE keyword was found that is not followed by the UPPER subkeyword, or by one of the subkeywords ARG, EXTERNAL, LINEIN, NUMERIC, PULL, SOURCE, VALUE, VAR, or VERSION.

Note: LINEIN is a valid subkeyword only on VM/ESA Release 2.1 or subsequent releases.

User response: Check your code and correct it.

EAGREX2502I PARSE UPPER not followed by a valid subkeyword

Explanation: A PARSE UPPER was found that is not followed by one of the subkeywords ARG, EXTERNAL, LINEIN, NUMERIC, PULL, SOURCE, VALUE, VAR, or VERSION.

User response: Check your code and correct it.

EAGREX2503I CALL ON/OFF not followed by supported condition name

Explanation: One of the conditions: ERROR, FAILURE, HALT or, on VM/ESA Release 2.1 or subsequent releases, NOTREADY is expected in a

CALL ON or CALL OFF instruction.

User response: Check your code and correct it.

EAGREX2504I ";" or subkeyword NAME expected

Explanation: Incorrect data was found at the end of a CALL ON instruction. The only subkeyword accepted after the condition name is NAME.

User response: Check your code and correct it.

EAGREX2505I NUMERIC not followed by DIGITS/FORM/FUZZ

Explanation: One of the subkeywords DIGITS, FORM, or FUZZ is expected in a NUMERIC instruction.

User response: Check your code and correct it.

EAGREX2506I NUMERIC FORM not followed by expression/valid subkeyword/";"

Explanation: Incorrect data was found at the end of a NUMERIC FORM. The only data recognized after FORM is an expression or one of the subkeywords VALUE, SCIENTIFIC, or ENGINEERING.

User response: Check your code and correct it.

EAGREX2507I PROCEDURE not followed by EXPOSE or ";"

Explanation: Incorrect data were found in a PROCEDURE instruction. The only subkeyword recognized on a PROCEDURE instruction is EXPOSE.

User response: Check your code and correct it.

EAGREX2508I SIGNAL ON/OFF not followed by supported condition name

Explanation: One of the conditions: **ERROR, FAILURE, HALT, NOVALUE, SYNTAX** or, on VM/ESA Release 2.1 or subsequent releases, NOTREADY is expected in a SIGNAL ON or SIGNAL OFF instruction.

User response: Check your code and correct it.

EAGREX2600E Error 26 running compiled *program*, **line** *nn*: **Invalid whole number**

Explanation: An expression that was expected to evaluate to a whole number either did not evaluate to a whole number within the current setting of NUMERIC DIGITS or was greater than the limit, for the intended use, of 999 999 999.

User response: Check your code and correct it.

EAGREX2601I Exponent not a whole number

Explanation: The right-hand term of the exponentiation (**) operator did not evaluate to a whole number within the current setting of NUMERIC DIGITS or was greater than the limit, for the intended use, of 999 999 999.

User response: Check your code and correct it.

EAGREX2602I Returned value not a whole number

Explanation: The return code passed back from an EXIT or RETURN instruction (when a REXX program is invoked as a command) is not a whole number in the range from -2147483648 through 2147483647.

User response: Check your code and correct it.

EAGREX2603I NUMERIC setting not a whole number

Explanation: An expression in the NUMERIC instruction did not evaluate to a whole number within the current setting of NUMERIC DIGITS or was greater than the limit, for the intended use, of 999 999 999.

User response: Check your code and correct it.

EAGREX2604I Quotient from integer division not a whole number

Explanation: The result of an integer division (%) is not a whole number within the current setting of NUMERIC DIGITS.

User response: Check your code and correct it.

EAGREX2605I Quotient from remainder operation not a whole number

Explanation: The result of the integer division performed to obtain the remainder (//) is not a whole number within the current setting of NUMERIC DIGITS.

User response: Check your code and correct it.

EAGREX2606I Repetition value in DO not a whole number

Explanation: The repetition value in a DO clause did not evaluate to a whole number within the current setting of NUMERIC DIGITS or was greater than the limit, for the intended use, of 999 999 999.

User response: Check your code and correct it.

EAGREX2607I Column number in PARSE not a whole number

Explanation: A column number in an absolute positional pattern or the value of a variable specified in a variable pattern used as absolute positional pattern on a PARSE instruction is either not a whole number within the current setting of NUMERIC DIGITS, or is greater than the limit, for the intended use, of 999 999 999.

User response: Check your code and correct it.

EAGREX2608I Relative position in PARSE not a whole number

Explanation: A number specified as a relative positional pattern or the value of a variable specified in a variable pattern used as relative positional pattern on a PARSE instruction is either not a whole number within the current setting of NUMERIC DIGITS, or is greater than the limit, for the intended use, of 999 999 999.

User response: Check your code and correct it.

EAGREX2609I Input to stream I/O function not a whole number

Explanation: A number specified as input to a stream I/O function is not a whole number.

User response: Check your code and correct it.

EAGREX2700E Error 27 running compiled *program*, line *nn*: Invalid DO syntax

Explanation: Some syntax error was found in the DO clause.

User response: See the secondary message for more specific information.

EAGREX2701I FOREVER not followed by WHILE/UNTIL/";"

Explanation: Incorrect data were found after DO FOREVER. The only valid subkeywords after DO FOREVER are WHILE and UNTIL.

User response: Check your code and correct it.

EAGREX2703I TO/BY/FOR phrase occurs more than once in a DO

Explanation: A DO clause contains more than one TO, BY, or FOR-phrase.

User response: Check your code and correct it.

EAGREX2706I TO/BY/FOR not followed by expression

Explanation: An expression is expected after a TO, BY, or FOR subkeyword in a DO clause.

User response: Check your code and correct it.

EAGREX2800E Error 28 running compiled *program*, line *nn*: Invalid LEAVE or ITERATE

Explanation: The program tried to execute a LEAVE or ITERATE instruction when no loop was active. This error occurs when control transfers within or into a loop, or if the LEAVE or ITERATE was encountered outside a repetitive DO loop. A SIGNAL instruction terminates all active loops; any ITERATE or LEAVE instruction issued then causes this message to be issued.

User response: See the secondary message for more specific information.

EAGREX2801I Invalid LEAVE

Explanation: The program tried to execute a LEAVE instruction when no loop was active.

User response: Check your code and correct it.

EAGREX2802I Invalid ITERATE

Explanation: The program tried to execute an ITERATE instruction when no loop was active.

User response: Check your code and correct it.

EAGREX2803I LEAVE not valid outside repetitive DO loop

Explanation: A LEAVE instruction was found outside a repetitive DO loop.

User response: Check your code and correct it.

EAGREX2804I ITERATE not valid outside repetitive DO loop

Explanation: An ITERATE instruction was found outside a repetitive DO loop.

User response: Check your code and correct it.

EAGREX2805I Variable does not match control variable of an active DO loop

Explanation: The symbol specified on a LEAVE or ITERATE instruction does not match the control variable of a currently active DO loop.

User response: Check your code and correct it.

EAGREX2806I Name of DO control variable expected

Explanation: The name of the control variable of a currently active DO loop is expected after a LEAVE or ITERATE instruction. Some other token was found.

User response: Check your code and correct it.

EAGREX2900E Error 29 running compiled *program*, line *nn*: Environment name too long

Explanation: The environment name on an ADDRESS instruction was specified as the value of an expression, and the result of evaluating the expression is longer than the limit of 8 characters.

User response: Check your code and correct it.

EAGREX3000E Error 30 running compiled *program*, line *nn*: Name or string > 250 characters

Explanation: A name or string that is longer than the limit of 250 characters was found.

User response: See the secondary message for more specific information.

EAGREX3001I Name of compound variable > 250 characters

Explanation: The name of a compound variable, after substitution, is longer than the limit of 250 characters.

User response: Check your code and correct it.

EAGREX3002I Label name > 250 characters

Explanation: The name of a label specified as an expression on a SIGNAL VALUE instruction is longer than the limit of 250 characters.

User response: Check your code and correct it.

EAGREX3004I String > 250 characters

Explanation: A quoted string, after substitution of hexadecimal or binary strings, exceeds the limit of 250 characters.

User response: Check your code and correct it.

EAGREX3005I Name > 250 characters

Explanation: The name of a symbol exceeds the limit of 250 characters.

User response: Check your code and correct it.

EAGREX3100E Error 31 running compiled *program*, line *nn*: Name starts with number or "."

Explanation: A value must not be assigned to a variable whose name starts with a digit or a period. Similarly, a symbol whose name starts with a digit or a period can not be contained in the list of variables of a DROP, EXPOSE, or UPPER instruction, and cannot follow the VAR subkeyword of the PARSE instruction.

User response: See the secondary message for more specific information.

EAGREX3101I "(" not followed by a variable name

Explanation: A variable name denoting a subsidiary list was expected in a DROP instruction or after the subkeyword EXPOSE of a PROCEDURE instruction.

User response: Check your code and correct it.

EAGREX3102I Variable name expected

Explanation: A name starting with a digit or a period was found in the list of a DROP instruction or after the subkeyword EXPOSE of a PROCEDURE instruction.

User response: Check your code and correct it.

EAGREX3104I Variable required to the left of "="

Explanation: The target of an assignment was found to be a symbol starting with a digit or a period.

User response: Check your code and correct it.

EAGREX3200E Error 32 running compiled *program*, line *nn*: Invalid use of stem

Explanation: The name of a stem has been found in the list of an UPPER instruction.

User response: Check your code and correct it.

EAGREX3300E Error 33 running compiled *program*, line *nn*: Invalid expression result

Explanation: An expression result was encountered that is incorrect in its particular context.

User response: Check your code and correct it.

EAGREX3301I Invalid NUMERIC expression result

Explanation: The result of an expression on the NUMERIC instruction is incorrect. The most common cause of this error is a DIGITS or FUZZ value that is not a whole number.

User response: Check your code and correct it.

EAGREX3302I NUMERIC DIGITS not greater than NUMERIC FUZZ

Explanation: The program issued a NUMERIC instruction that would make the current NUMERIC DIGITS value less than or equal to the current NUMERIC FUZZ value. The DIGITS value must be greater than the FUZZ value.

User response: Check your code and correct it.

EAGREX3304I SIGNAL VALUE not followed by expression

Explanation: In a SIGNAL VALUE instruction the required expression is missing.

User response: Check your code and correct it.

EAGREX3305I ADDRESS VALUE not followed by expression

Explanation: In the ADDRESS VALUE instruction the required expression is missing.

User response: Check your code and correct it.

EAGREX3306I NUMERIC FORM VALUE not followed by expression

Explanation: In the NUMERIC FORM VALUE instruction the required expression is missing.

User response: Check your code and correct it.

EAGREX3400E Error 34 running compiled *program*, line *nn*: Logical value not 0 or 1

Explanation: The expression in an IF-, WHEN-, DO WHILE-, or DO UNTIL-phrase must result in a 0 or 1, as must any term operated on by a logical operator (that is, ¬, \, |, &, or &&). For example, the phrase:

```
If result Then Exit rc
```

fails if `result` has a value other than 0 or 1. Thus, the phrase might be better written as:

```
If result¬=0 Then Exit rc
```

User response: Check your code and correct it.

EAGREX3401I WHILE not followed by expression

Explanation: The subkeyword WHILE must be followed by an expression.

User response: Check your code and correct it.

EAGREX3402I UNTIL not followed by expression

Explanation: The subkeyword UNTIL must be followed by an expression.

User response: Check your code and correct it.

EAGREX3403I IF not followed by expression

Explanation: The keyword IF must be followed by an expression.

User response: Check your code and correct it.

EAGREX3404I WHEN not followed by expression

Explanation: The keyword WHEN must be followed by an expression.

User response: Check your code and correct it.

EAGREX3500E Error 35 running compiled *program*, line *nn*: Invalid expression

Explanation: An expression contains a grammatical error.

User response: See the secondary message for more specific information.

EAGREX3501I Assignment operator must not be followed by another "="

Explanation: A second "=" was found immediately after the first one of an assignment.

User response: Delete one "=" to form a correct assignment, or, if the clause was intended as a command, enclose the expression in parentheses.

EAGREX3502I Left operand missing

Explanation: An operator was found that is not a prefix operator, and whose left operand is missing.

User response: Check your code and correct it.

EAGREX3503I Right operand missing

Explanation: An operator is not followed by an operand.

User response: Check your code and correct it.

EAGREX3504I Prefix operator not followed by operand

Explanation: A prefix operator was found that is not followed by a symbol or by a literal string or by an open parenthesis.

User response: Check your code and correct it.

EAGREX3505I "(" not followed by an expression or subexpression

Explanation: An open parenthesis was found that is not followed by a valid expression or subexpression.

User response: Check your code and correct it.

EAGREX3506I Invalid operator

Explanation: An expression contains an invalid sequence of operator characters.

User response: Check your code and correct it.

EAGREX3507I Invalid use of NOT operator

Explanation: An expression or subexpression of the form a¬b or (a)¬b was found.

User response: If you want to concatenate a negated term:

- To some other operand, enclose it into parentheses, for example: `left(a,3)(¬b)`.
- To a symbol or a literal string, use the concatenation operator, for example: `a||(¬b)`.

EAGREX3508I Missing expression

Explanation: An expression is missing where one is expected. Example: `INTERPRET;`

User response: Check your code and correct it.

EAGREX3600E Error 36 running compiled *program*, line *nn*: Unmatched "(" in expression

Explanation: The parentheses in an expression are not paired correctly. There are more open parentheses than close parentheses.

User response: Check your code and correct it.

EAGREX3700E Error 37 running compiled *program*, line *nn*: Unexpected "," or ")"

Explanation: In an expression, either a comma was found outside a function invocation, or there are too many close parentheses.

User response: Check your code and correct it.

EAGREX3800E Error 38 running compiled *program*, line *nn*: Invalid template or pattern

Explanation: Within a parsing template, a special character that is not allowed was found, or the syntax of a variable pattern is incorrect. This message is also issued if the WITH subkeyword is omitted in a PARSE VALUE instruction.

User response: Check your code and correct it.

EAGREX3801I Incomplete PARSE VALUE: WITH not found

Explanation: The WITH subkeyword is omitted in a PARSE VALUE instruction.

User response: Check your code and correct it.

EAGREX3900E Error 39 running compiled *program*, line *nn*: Evaluation stack overflow

Explanation: INTERPRET or TRACE caused a stack overflow. You exceeded the maximum number of nesting levels.

User response: Check your code and correct it.

EAGREX4000E Error 40 running compiled *program*, line *nn*: Incorrect call to routine

Explanation: The program invoked a built-in function with incorrect parameters, or invoked an external routine, which ended with a SYNTAX condition that was not trapped.

If you were not trying to invoke a routine, you might have a symbol or a string adjacent to a left parenthesis when you meant it to be separated by a space or an operator. A symbol or a string in this position causes the phrase to be read as a function call. For example, `TIME(4+5)` should be written as `TIME*(4+5)` if a multiplication was intended.

User response: Check your code and correct it.

EAGREX4001I Null string specified as option

Explanation: The program invoked a built-in function that has an *option* argument, and passed a null string as the option.

User response: Specify a valid value for the option.

EAGREX4002I Invalid option

Explanation: The program invoked a built-in function that has an *option* argument, and passed an incorrect value for the option.

User response: Specify a valid value for the option.

EAGREX4003I Argument not positive

Explanation: The program invoked a built-in function with an argument whose value is less than or equal to zero.

User response: Check your code and correct it.

EAGREX4004I Argument not a single character

Explanation: A built-in function expected an argument of length 1; one of a different length was supplied.

User response: Check your code and correct it.

EAGREX4005I Argument not a whole number

Explanation: The value of an argument on the invoked built-in function must be a whole number, but the program supplied something else. For example, a *length* argument is expected to be a whole number.

User response: Check your code and correct it.

EAGREX4006I First argument negative and second argument not supplied

Explanation: The program did not supply the second argument of the D2C or D2X function, but this argument is required when the first argument is a negative number.

User response: Check your code and correct it.

EAGREX4007I String longer than 250 characters (500 hexadecimal digits)

Explanation: The program invoked the C2D or X2D function with an input string that exceeds one of the following limits:

- The input string for the C2D function must not have more than 250 characters that are significant in forming the result of the function.
- The input string for the X2D function must not have more than 500 hexadecimal digits that are significant in forming the final result.

User response: Check your code and correct it.

EAGREX4008I Argument not a valid hexadecimal string

Explanation: The value of an argument on the invoked built-in function must be a hexadecimal string, but the program supplied something else. A hexadecimal string can contain only the characters 0-9, a-f, and A-F. Blanks may only occur only at byte boundaries and are not allowed at the beginning or the end of the string.

User response: Check your code and correct it.

EAGREX4009I Output string longer than 250 characters (500 hexadecimal digits)

Explanation: The output string on an invocation of the D2C or D2X function would exceed one of the following limits:

- The output string of the D2C function must not have more than 250 significant characters.
- The output string of the D2X function must not have more than 500 significant hexadecimal characters.

User response: Check your code and correct it.

EAGREX4010I Result not a whole number

Explanation: The data returned by the invoked built-in function is not a whole number and cannot be formatted without an exponent. This can occur if the NUMERIC DIGITS value is not large enough. For example, this error occurs if you set NUMERIC DIGITS to 2 and then invoke the C2D function with `C2D(1)`; the result is 241, which needs three digits, but only two digits are allowed for.

User response: Check your code and correct it.

EAGREX4011I Result too long

Explanation: The data returned by the invoked built-in function is too large for the available memory. This error can occur if you use, for example, the COPIES, INSERT, OVERLAY, or SPACE built-in functions.

User response: Specify smaller *string* or *count* arguments, or obtain more storage.

EAGREX4012I Failure in system service, no clock available

Explanation: The invoked built-in function was unable to obtain the system time, due to a failure in a system service.

User response: If the problem persists, notify your system support personnel.

EAGREX4013I "min" > "max" on RANDOM function

Explanation: The program invoked the RANDOM built-in function with a value for the *min* argument greater than the value for the *max* argument. The *min* argument must be less than or equal to the *max* argument.

User response: Check your code and correct it.

EAGREX4014I "max" - "min" exceeds 100000 in RANDOM function

Explanation: The range between the *min* and *max* arguments in an invocation of the RANDOM built-in function is greater than the limit of 100 000.

User response: Check your code and correct it.

EAGREX4015I Error number out of range in ERRORTEXT function

Explanation: The program invoked the ERRORTEXT built-in function with an incorrect value for the error number argument. The error number must be in the range of 0 through 99.

User response: Check your code and correct it.

EAGREX4017I Argument not positive or zero

Explanation: The program invoked a built-in function with a value less than zero for an argument that must be greater than or equal to zero.

User response: Check your code and correct it.

EAGREX4018I Invalid pad character

Explanation: The value of the *pad* argument on the invoked built-in function must be a single character, but the program supplied something else.

User response: Check your code and correct it.

EAGREX4019I Elapsed-time clock out of range in TIME function invocation

Explanation: The elapsed-time clock was out of range in an invocation of the TIME built-in function. This error occurs if the number of seconds in the elapsed-time clock exceeds nine digits.

User response: This error might be caused by a system problem; notify your system support personnel.

EAGREX4020I Line number out of range in SOURCELINE function

Explanation: An invocation of the SOURCELINE built-in function was incorrect for one of these reasons:

- The program passed an incorrect line number to the function.
- The program was compiled with the NOSLINE (NOSL) option.

User response: If the program was compiled with the SLINE option, ensure that the line number does not exceed the number of the final line in the source file. If the program was compiled with the NOSLINE option, either change the program or recompile with the SLINE option.

EAGREX4021I Invalid symbol in name argument of VALUE function

Explanation: The value of the *name* argument in the VALUE built-in function must be a valid REXX symbol, but the program supplied something else. The most common cause of this message is the use of special characters that are not valid within symbols.

User response: Check your code and correct it.

EAGREX4022I Incorrect call to built-in function or DBCS function package

Explanation: An error occurred when a function was invoked with OPTIONS 'EXMODE' in effect. This error can occur for functions in the DBCS function package and for built-in functions that perform string operations.

User response: If the cause of the problem is not obvious, debug the program using the interpreter.

EAGREX4023I Argument not a number

Explanation: The value of an argument on the invoked built-in function must be a number, but the program supplied something else.

User response: Check your code and correct it.

EAGREX4024I Exponent exceeds specified digits in FORMAT function

Explanation: The value supplied for the exponent argument of the FORMAT built-in function is out of range for the result. This error occurs if the FORMAT built-in function is invoked with an exponent size too small for the number to be formatted.

User response: Check your code and correct it.

EAGREX4025I Integer part exceeds specified digits in FORMAT function

Explanation: The program invoked the FORMAT built-in function with a value for the *before* argument that is not large enough to contain the integer part of the number to be formatted. For example, this error occurs if the function is invoked with `FORMAT(225.1,2);` there are three integer digits in the number, but space has been specified for only two digits.

User response: Check your code and correct it.

EAGREX4026I External routine returned with non-zero return code

Explanation: An external routine returned with a nonzero return code.

User response: Correct the external routine.

EAGREX4027I External routine could not obtain an EVALBLOCK

Explanation: An external routine could not obtain an EVALBLOCK control block, because there was not enough storage.

User response: Use a larger region size.

EAGREX4028I External routine could not locate language processor environment

Explanation: An external routine could not locate a language processor environment.

User response: Notify your system support personnel.

EAGREX4029I External routine encountered an ABEND

Explanation: An external routine abnormally ended.

User response: Correct the external routine.

EAGREX4030I Invalid number of arguments on built-in function invocation

Explanation: A built-in function was invoked, but the number of arguments passed is not in the range of arguments expected by the function.

User response: Check your code and correct it.

EAGREX4031I Required argument missing in built-in function invocation

Explanation: A built-in function was invoked, but an argument required by this function was not provided.

User response: Check your code and correct it.

EAGREX4032I Argument not a valid binary string

Explanation: The value of an argument on the invoked built-in function must be a binary string, but the program supplied something else. A binary string can contain only the digits 0 and 1. Blanks may only occur at the boundaries of groups of four binary digits and are not allowed at the beginning or the end of the string.

User response: Check your code and correct it.

EAGREX4033I Selector not supported for VALUE function

Explanation: A selector for the VALUE built-in function is only supported on CMS Release 6 or subsequent releases.

User response: Check your code and correct it.

EAGREX4034I Global variable name longer than 255 characters

Explanation: The VALUE built-in function was invoked with a selector on CMS Release 6 or subsequent releases, but the length of the name of the variable exceeds the allowed maximum of 255 characters.

User response: Check your code and correct it.

EAGREX4035I New global variable value longer than 255 characters

Explanation: The VALUE built-in function was invoked with a selector on CMS Release 6 or subsequent releases, but the length of the value exceeds the allowed maximum of 255 characters.

User response: Check your code and correct it.

EAGREX4036I Invalid selector

Explanation: The VALUE built-in function was invoked with a selector on CMS Release 6 or a subsequent release, but the first token in the *selector* is not valid. Valid tokens are GLOBAL, SESSION, and LASTING.

User response: Check your code and correct it.

EAGREX4037I Error upon invocation of system service in VALUE function

Explanation: The VALUE built-in function was invoked with a selector on CMS Release 6 or subsequent releases, but the attempt to perform the desired action was unsuccessful. This might be caused by a full A-disk, or by an A-disk not accessed in read/write mode, or by not having accessed an A-disk.

User response: Check your code and correct it.

EAGREX4038I Variable expected

Explanation: The first argument on an invocation of the VALUE built-in function was a symbol starting with a numeric digit or a period, and a selector is not supplied.

User response: Check your code and correct it.

EAGREX4039I Start value of CHARIN or CHAROUT function must be 1

Explanation: A value other than 1 was specified as start value of the CHARIN or CHAROUT function.

User response: Check your code and correct it.

EAGREX4040I Count value of the LINEIN function must be 0 or 1

Explanation: A value other than 0 or 1 was specified as count value of the LINEIN function.

User response: Check your code and correct it.

EAGREX4041I Command required for operation 'C'

Explanation: Invocation of the STREAM function with operation 'C' requires a command as third parameter.

User response: Check your code and correct it.

EAGREX4042I Command not allowed with operation other than 'C'

Explanation: A command can be specified only if the STREAM function is invoked with operation 'C'.

User response: Check your code and correct it.

EAGREX4043I Operation value of STREAM function must be 'C', 'D', or 'S'

Explanation: The only valid STREAM function operations are:
- 'C' (command)
- 'D' (description)
- 'S' (state)

User response: Check your code and correct it.

EAGREX4044I Invalid argument value in stream I/O function

Explanation: A stream I/O function (CHARIN, CHAROUT, CHARS, LINEIN, LINEOUT, LINES, or STREAM) returned an error.

User response: Check your code and correct it.

EAGREX4045I Argument 2 is not in the format described by argument 3

Explanation: The second argument specified is not in the format described by the third argument.

User response: Check the format definitions of the built-in function for which the error is reported. Either correct the value of the second argument or change the format specified in the third argument.

EAGREX4046I BIF argument 4/5 must be a single non-alphanumeric character or the null string

Explanation: You specified a wrong date separation character.

User response: Check your code and correct it.

EAGREX4047I BIF argument 1/3 is in a format incompatible with separator in argument 4/5

Explanation: You specified a separator character for a date type that does not allow for separators.

User response: Check your code and correct it.

EAGREX4048I Argument 2 is not in the format described by argument 5

Explanation: The separator character in the input date in argument 2 does not correspond to the date separator character specified in argument 5.

User response: Check your code and correct it.

EAGREX4100E Error 41 running compiled *program*, line *nn*: Bad arithmetic conversion

Explanation: In an arithmetic expression, a term was found that was not a valid number or that had an exponent outside the range of -999 999 999 through +999 999 999.

A variable might have been incorrectly used or an arithmetic operator might have been included in a character expression without being put in quotes. For example, the command `MSG * Hi!` should be written as `'MSG * Hi!'`, otherwise the program will try to multiply `MSG` by `Hi!`.

User response: Check your code and correct it.

EAGREX4101I Initial expression missing in controlled DO loop

Explanation: No initial expression was found in a controlled DO loop where one was expected.

User response: Check your code and correct it.

EAGREX4200E Error 42 running compiled *program*, line *nn*: Arithmetic overflow/underflow

Explanation: A result of an arithmetic operation was encountered that required an exponent greater than the limit of nine digits (more than +999 999 999 or less than -999 999 999). This error can occur during evaluation of an expression or during the stepping of a DO loop control variable.

User response: Check your code and correct it.

EAGREX4201I Overflow occurred during addition or subtraction

Explanation: The result of an addition or subtraction required an exponent greater than 999 999 999.

User response: Check your code and correct it.

EAGREX4202I Overflow occurred during multiplication

Explanation: The result of a multiplication required an exponent greater than 999 999 999.

User response: Check your code and correct it.

EAGREX4203I Underflow occurred during multiplication

Explanation: The result of a multiplication required an exponent less than -999 999 999.

User response: Check your code and correct it.

EAGREX4204I Overflow occurred during division

Explanation: The result of a division required an exponent greater than 999 999 999.

User response: Check your code and correct it.

EAGREX4205I Underflow occurred during division

Explanation: The result of a division required an exponent less than -999 999 999.

User response: Check your code and correct it.

EAGREX4206I Division by zero

Explanation: The program tried to divide a number by zero.

User response: Check your code and correct it.

EAGREX4207I Integer division by zero

Explanation: The program tried to divide a number by zero with the % (integer division) operator.

User response: Check your code and correct it.

EAGREX4208I Remainder of division by zero

Explanation: The program tried to divide a number by zero with the // (remainder) operator.

User response: Check your code and correct it.

EAGREX4209I Overflow occurred during exponentiation

Explanation: The result of an exponentiation operation required an exponent greater than 999 999 999.

User response: Check your code and correct it.

EAGREX4210I Underflow occurred during exponentiation

Explanation: The result of an exponentiation operation required an exponent less than -999 999 999.

User response: Check your code and correct it.

EAGREX4211I Value zero to a negative power

Explanation: The program tried to raise zero to a negative power in an exponentiation operation.

User response: Check your code and correct it.

EAGREX4300E Error 43 running compiled *program*, **line** *nn*: **Routine not found**

Explanation: An external routine called in your program could not be found. The simplest, and probably most common, cause of this error is a mistyped name. Another possibility is that one of the standard function packages is not available.

If you were not trying to invoke a routine, you might have put a symbol or string adjacent to a left parenthesis when you meant it to be separated by a space or operator. The Compiler would see that as a function invocation. A symbol or a string in this position causes the phrase to be read as a function call. For example, the string `3(4+5)` should be written as `3*(4+5)` if a multiplication was intended.

User response: Check your code and correct it.

EAGREX4400E Error 44 running compiled *program*, **line** *nn*: **Function did not return data**

Explanation: The program invoked an external routine as a function within an expression. The routine seemed to end without error, but it did not return data for use within the expression.

User response: Check your code and correct it.

EAGREX4500E Error 45 running compiled *program*, **line** *nn*: **No data specified on function RETURN**

Explanation: A REXX program or internal routine has been called as a function, but an attempt is being made to return (by a RETURN instruction) without passing back any data.

User response: Check your code and correct it.

EAGREX4600E Error 46 running compiled *program*, **line** *nn*: **Invalid variable reference**

Explanation: Within a DROP or PROCEDURE instruction, the syntax of a variable reference (a variable whose value is to be used, indicated by its name being enclosed in parentheses) is incorrect. The close parenthesis that should immediately follow the variable name is missing.

User response: Check your code and correct it.

EAGREX4700E Error 47 running compiled *program*, **line** *nn*: **Unexpected label**

Explanation: A label was found in the string of an INTERPRET instruction.

User response: Check your code and correct it.

EAGREX4800E Error 48 running compiled *program*, **line** *nn*: **Failure in system service**

Explanation: Either a system service, such as user input, output, or manipulation of the console stack, has failed to work correctly, or a system exit detected such an error in a system service.

User response: Ensure that your input is correct and

that your program is working correctly. If the problem persists, notify your system support personnel.

EAGREX4801I Error in EXECINIT invocation

Explanation: The EXECINIT routine specified in the module name table either could not be invoked, or returned a nonzero return code.

User response: Notify your system support personnel.

EAGREX4802I Error in EXECTERM invocation

Explanation: The EXECTERM routine specified in the module name table either could not be invoked, or returned a nonzero return code.

User response: Notify your system support personnel.

EAGREX4803I EVALBLOCK cannot be obtained

Explanation: The Library attempted to obtain an EVALBLOCK control block by calling the **IRXRLT** system routine with the **GETEVAL** function, but did not succeed.

User response: Notify your system support personnel.

EAGREX4804I Error in invocation of global exit for REXX programs

Explanation: A global exit for REXX programs on z/VM was specified, but cannot be invoked due to missing system interfaces. You might be missing a prerequisite z/VM **PTF**.

User response: Notify your system support personnel.

EAGREX4805I System interfaces for invocation of stream I/O function not available

Explanation: Stream I/O on VM/ESA Release 2.1 and VM/ESA Release 2.2 was specified, but cannot be invoked due to missing system interfaces. You might be missing a prerequisite z/VM **PTF**.

User response: Notify your system support personnel.

EAGREX4806I Error in stream I/O function

Explanation: A stream I/O function (CHARIN, CHAROUT, CHARS, LINEIN, LINEOUT, LINES, or STREAM) returned an error.

User response: Check your code and correct it.

EAGREX4900E Error 49 running compiled *program*, **line** *nn*: **Language processor failure**

Explanation: An internal self-consistency check of the INTERPRET processor indicated an error.

User response: Report any occurrence of this message to your IBM representative.

EAGREX9999S Message number nnn

Explanation: The Library was about to issue a message but the message could not be found in the message repository currently allocated. This can occur when you have different product releases or PTF levels installed.

User response:

- Under z/VM

  If this message has been issued only a few times, you are probably using a back-level version of the message repository and the Compiler cannot find the newer messages. Upgrade your repository.

  If this message has been issued several times and the Compiler's listing does not contain correct headers and text, the Compiler cannot find the repository. If you did not customize the repository, see "Customizing the Message Repository to Avoid a Read/Write A-Disk" on page 123 for the correct names supplied by IBM. Issue a FILELIST command to see if one of these repositories is in your current search order. If you wish to customize the repository, make sure you issued the GENMSG and SET LANG commands with the correct parameters and file IDs.
- Under z/OS

  If this message has been issued only a few times, you are probably using a back-level version of the message repository and the Compiler cannot find the newer messages. Check with your Systems Programming staff.

  If this message has been issued several times and the Compiler's listing is mainly in English although you have been trying to use another language, the Compiler cannot find the text in the message repository and has switched to hard-coded English text. Check with your Systems Programming staff and see "Message Repository" on page 118 for more details.

Chapter 21. Stream I/O Messages

One or more of the following messages might occur in response to a problem during a stream I/O function call. If the problem cause is likely to be with your REXX program, detailed information is given with each message. If the problem cause is outside your REXX program, you need to see the appropriate TSO/E or z/OS documentation or to contact the system administrator for further help.

Note: The abbreviation REXXIO refers to the REXX Stream I/O function package.

EAGSIO0001 Invalid numeric parameter for REXXIO.

Explanation: A function call contains a numeric parameter that is not allowed. A *line*, *count*, or *start* parameter value might be negative or outside the boundaries of a stream.

User response: Check your code and correct the value.

EAGSIO0002 Invalid "count" value for LINEIN.

Explanation: The *count* parameter contains a value other than 0 or 1.

User response: Check your code and correct the value. You can only define a value of 0 or 1.

EAGSIO0003 Invalid number of parameters for LINEIN.

Explanation: There might be a syntax problem with the LINEIN function call or more than three parameter values have been specified. Check the use of commas in the function call.

User response: Check your code and correct it. For more information refer to "LINEIN (Line Input)" on page 146.

EAGSIO0004 Invalid file specification for REXXIO.

Explanation: A stream name is not properly specified. The name might contain invalid characters, or a qualifier might be more than eight characters long.

User response: Check your code and correct it. For more information refer to "Naming Streams" on page 135.

EAGSIO0005 Invalid number of parameters for LINES.

Explanation: There might be a syntax problem with the LINES function call or more than one parameter value has been specified (no commas in the function call).

User response: Check your code and correct it. For more information refer to "LINES (Lines Remaining)" on page 148.

EAGSIO0007 Invalid number of parameters for LINEOUT.

Explanation: There might be a syntax problem with the LINEOUT function call or more than three parameter values have been specified. Check the use of commas in the function call.

User response: Check your code and correct it. For more information refer to "LINEOUT (Line Output)" on page 147.

EAGSIO0008 Invalid line number for LINEOUT.

Explanation: The *line* parameter contains a value for a transient stream, or a value other than 1 for a persistent stream.

User response: Check your code and correct it. For more information refer to "LINES (Lines Remaining)" on page 148.

EAGSIO0009 The file failed to open for REXXIO.

Explanation: The specified stream failed to open. The ddname might not be allocated or it might be misspelled.

User response: Check if the ddname is allocated in foreground or background. Then check your code and correct it.

EAGSIO0010 The file is a partitioned data set, but no member name was specified.

Explanation: You must specify the stream name with an explicit member name because the data set is partitioned.

User response: Check your code and specify the stream name with an explicit member name because the data set is partitioned. For more information refer to "Naming Streams" on page 135.

EAGSIO0011 The file is a sequential data set, but a member name was specified.

Explanation: The data set is not partitioned, that is why you must not specify the stream name with an explicit member name.

User response: Check your code and make sure that you do not specify the stream name with an explicit member name, because the data set is not partitioned. For more information refer to "Naming Streams" on page 135.

EAGSIO0012 Record format of *file* is not supported.

Explanation: You have tried to open a persistent stream with a record format that is not supported.

User response: Check your code and correct it. For more information refer to "Stream Formats" on page 138.

EAGSIO0013 Data set organization of *file* is not supported.

Explanation: You have tried to open a persistent stream with a data set organization that is not supported.

User response: Check your code and correct it. For more information refer to "Stream Formats" on page 138.

EAGSIO0014 Warning: Output record truncated.

Explanation: A LINEOUT function call attempted to write a string that exceeds the LRECL of the data set. A preceding CHAROUT function call might have already written several characters, or the string length exceeds the LRECL of the data set.

User response: Check your code and correct it. For more information refer to "Stream Formats" on page 138.

EAGSIO0015 Cannot allocate data set. Insufficient storage.

Explanation: The Region size is too small.

User response: Increase the Region size or contact your administrator.

EAGSIO0016 Cannot allocate data set. Data set cannot be accessed exclusively.

Explanation: Opening a stream for write operations requires exclusive allocation. Someone else has already allocated the data set.

User response: Contact your administrator to check who has allocated the data set.

EAGSIO0017 Cannot allocate data set. Data set in use by another user or job.

Explanation: Someone else has already allocated the data set exclusively. You cannot open the stream for read operations or write operations yet.

User response: Contact your administrator to check who has allocated the data set.

EAGSIO0018 Cannot allocate data set. No unit available.

Explanation: You specified a stream name as a data set that cannot be allocated.

User response: Ask the system administrator for help.

EAGSIO0019 Cannot allocate data set. Volume cannot be mounted.

Explanation: You specified a stream name as a data set that cannot be allocated.

User response: Ask the system administrator for help.

EAGSIO0020 Cannot allocate data set. Volume allocated to another user or job.

Explanation: You specified a stream name as a data set that cannot be allocated.

User response: Contact your administrator to check who has allocated the data set.

EAGSIO0021 Cannot allocate data set. Number of required devices unavailable.

Explanation: You specified a stream name as a data set that cannot be allocated.

User response: Ask the system administrator for help.

EAGSIO0022 Cannot allocate data set. Volume or unit in use by system.

Explanation: You specified a stream name as a data set that cannot be allocated (exclusively used by someone else).

User response: Ask the system administrator for help.

EAGSIO0023 Cannot allocate data set. Volume mounted on an ineligible device.

Explanation: You specified a stream name as a data set that cannot be allocated.

User response: Ask the system administrator for help.

EAGSIO0024 Cannot allocate data set. Specified device in use.

Explanation: You specified a stream name as a data set that cannot be allocated.

User response: Ask the system administrator for help.

EAGSIO0025 Cannot allocate data set. Specified volume is on another device.

Explanation: You specified a stream name as a data set that cannot be allocated.

User response: Ask the system administrator for help.

EAGSIO0026 Cannot allocate data set. Limit of data sets allocated is exceeded.

Explanation: You have already allocated too many data sets. The maximum number of allocations is specified in TSO/E and z/OS.

User response: Either try to free some other allocations, or ask the system administrator for help.

EAGSIO0027 Cannot allocate data set. Maximum number of allocations exceeded.

Explanation: The maximum number of concurrent allocations is reached.

User response: Either try to free some ddnames, or ask the system administrator for help.

EAGSIO0028 Cannot allocate data set. Job Entry Subsystem unavailable.

Explanation: The JES is not available to verify an allocation request. Ask the system administrator for help.

User response: Ask the system administrator for help.

EAGSIO0029 Cannot allocate data set. Number of volumes exceeds limit.

Explanation: You specified a stream name as a data set that cannot be allocated.

User response: Ask the system administrator for help.

EAGSIO0030 Cannot allocate data set. Request cancelled by the operator.

Explanation: You specified a stream name as a data set that cannot be allocated.

User response: Ask the system administrator for help.

EAGSIO0031 Cannot allocate data set. MSS volume not accessible from unit.

Explanation: You specified a stream name as a data set that cannot be allocated.

User response: Ask the system administrator for help.

EAGSIO0032 Cannot allocate data set. MSS volume does not exist.

Explanation: You specified a stream name as a data set that cannot be allocated.

User response: Ask the system administrator for help.

EAGSIO0033 Cannot allocate data set. Data set name not found.

Explanation: You specified a stream name as a data set that cannot be allocated (the data set name is not cataloged).

User response: Ask the system administrator for help.

EAGSIO0034 Cannot allocate data set. Locate I/O error.

Explanation: You specified a stream name as a data set that cannot be allocated (probably a system problem).

User response: Ask the system administrator for help.

EAGSIO0035 Cannot allocate data set. DADSM I/O error.

Explanation: You specified a stream name as a data set that cannot be allocated (probably a system problem).

User response: Ask the system administrator for help.

EAGSIO0036 Cannot allocate data set. Data set not on volume as denoted by catalog.

Explanation: You specified a stream name as a data set that cannot be allocated (probably a system problem).

User response: Ask the system administrator for help.

EAGSIO0037 Cannot allocate data set. OBTAIN I/O error.

Explanation: You specified a stream name as a data set that cannot be allocated (probably a system problem).

User response: Ask the system administrator for help.

EAGSIO0038 Cannot allocate data set. Required catalog not mounted.

Explanation: You specified a stream name as a data set that cannot be allocated (probably a system problem).

User response: Ask the system administrator for help.

EAGSIO0039 The requested member is not in the specified data set.

Explanation: An attempt was made to open a partitioned data set for a read operation, but the data set member does not exist.

User response: Check your code and specify or create the data set member again.

EAGSIO0040 The STOW failed during the close of a data set.

Explanation: Probably a system problem.

User response: Ask the system administrator for help.

EAGSIO0041 Invalid number of parameters for CHARIN.

Explanation: There might be a syntax problem with the CHARIN function call or more than three parameter values have been specified. Check the use of commas in the function call.

User response: Check your code and correct it. For more information refer to "LINEIN (Line Input)" on page 146.

EAGSIO0042 Invalid number of parameters for CHAROUT.

Explanation: There might be a syntax problem with the CHAROUT function call, or more than three parameter values have been specified. Check the use of commas in the function call.

User response: Check your code and correct it. For more information refer to "LINEIN (Line Input)" on page 146.

EAGSIO0043 Invalid number of parameters for CHARS.

Explanation: There might be a syntax problem with the CHARS function call, or more than one parameter value has been specified (no commas in the function call).

User response: Check your code and correct it.

EAGSIO0044 Invalid "start" value for CHAROUT.

Explanation: The *start* parameter contains a value other than 1.

User response: Check your code and correct it. The parameter value must be 1.

EAGSIO0045 Invalid "start" value for CHARIN.

Explanation: The *start* parameter contains a value other than 1.

User response: Check your code and correct it. The parameter value must be 1.

EAGSIO0046 Invalid "line" value for LINEIN.

Explanation: The *line* parameter contains a value that is negative or not within the boundaries of the stream (the specified line might not exist).

User response: Check your code and correct it. The parameter value must be positive and within the boundaries of the stream.

EAGSIO0047 Read error in REXXIO.

Explanation: TSO/E returned with a Read error. See the subsequent messages for further information.

User response: If required, ask the TSO/E support for help.

EAGSIO0048 CLOSE ignored. File already closed, or not found.

Explanation: The stream was already closed by a preceding STREAM CLOSE command or by an implicit close, or the named stream does not exist.

User response: None.

EAGSIO0049 Logic error IEANTRT token retrieval.

Explanation: A severe error occurred.

User response: Contact your IBM representative.

EAGSIO0050 Invalid command for STREAM specified.

Explanation: The *stream_command* parameter of the STREAM function call contains an invalid value.

User response: Check your code and correct it. For more information refer to "STREAM (Operations)" on page 148.

EAGSIO0051 Invalid parameter for STREAM QUERY specified.

Explanation: The *stream_command* parameter of the STREAM function call contains an invalid value.

User response: Check your code and correct it. Only `QUERY EXISTS`, `QUERY REFDATE`, and `QUERY SERVICELEVEL` are supported. For more information refer to "STREAM (Operations)" on page 148.

EAGSIO0052 I/O RC=12, DCB already opened for a different type of I/O operation.

Explanation: Error returned by TSO/E. A data set operation was tried in a mode different from the initial mode.

User response: Check your code, correct it, and rerun it.

EAGSIO0053 I/O RC=16, Output data was truncated for WRITE option.

Explanation: A LINEOUT function call failed to write a string. The LRECL value of the data set in question might be too small.

User response: Check if the LRECL value of the data set is too small or adjust the stream length to the existing LRECL.

EAGSIO0054 I/O RC=20, unsuccessful processing, function not performed.

Explanation: Error returned by TSO/E. Subsequent messages might provide further information.

User response: If required, ask the TSO/E support for help.

EAGSIO0055 I/O RC=24, unsuccessful processing, file cannot be opened.

Explanation: Error returned by TSO/E. Subsequent messages might provide further information.

User response: If required, ask the TSO/E support for help.

EAGSIO0056 I/O RC=28, unsuccessful processing. Language processor cannot be located.

Explanation: Error returned by TSO/E. Subsequent messages might provide further information.

User response: If required, ask the TSO/E support for help.

EAGSIO0057 I/O RC=32, unsuccessful processing. Internal error in REXXIO.

Explanation: Error returned by TSO/E. Subsequent messages might provide further information.

User response: If required, ask the TSO/E support for help.

EAGSIO0058 Record for console must be Fixed 80.

Explanation: An attempt to read from SYSTSIN with IRXJCL failed. The DSN must be F 80.

User response: Verify the data set allocation for record format and LRECL.

EAGSIO0059 SVC99 allocation error in EAGIODYN, see error code below:

Explanation: Check the SVC99 error code listed in the subsequent message.

User response: Contact your system administrator or your IBM representative.

EAGSIO0060 Allocation failed. Too many attempts.

Explanation: Too many unsuccessful allocation attempts from within the REXX program have used up the available storage.

User response: Check your code and correct it.

EAGSIO0061 Allocation failed. Invalid data set name, SVC99 error code 9700.

Explanation: The allocation failed because the data set name is not valid.

User response: Verify in your code whether you used a ddname instead of a data set name. If you used the stream function, you should also check if the return code is zero. Otherwise, contact your system administrator or your IBM representative.

EAGSIO0062 Allocation failed. SMS error, see SVC99 error code below:

Explanation: Check the SVC99 error code listed in the subsequent message.

User response: Contact your system administrator or your IBM representative.

EAGSIO0063 Allocation failed. SMS VTOC error, see SVC99 error code below:

Explanation: Check the SVC99 error code listed in the subsequent message.

User response: Contact your system administrator or your IBM representative.

EAGSIO9999

EAGSIO9999 <-Logic error, this MSG ID is not defined. Call "IBM" service center.

Explanation: A severe error occurred.

User response: Contact your IBM representative.

Part 5. Appendixes

Appendix A. Interface for Object Modules (z/OS)

This appendix explains in detail the preparatory steps for generating a load module from a REXX program that has been compiled to an object module under z/OS, and the **ISPF** restrictions on load modules. It also describes the parameter-passing conventions for the different stubs and how the stubs invoke the EXEC handler, IRXEXEC. This appendix also describes the PARSE SOURCE information, as it appears in the REXX program.

ISPF Restrictions on Load Modules

To run compiled REXX load modules for ISPF you must consider the installed ISPF Version.

The use of a REXX load module as an external routine is supported. This program is created using the EFPL stub. If an application is to be completely packaged, it can use an interpreted REXX program in a **SELECT CMD** statement, and this interpreted REXX program can invoke the packaged external routine with the REXX CALL instruction.

As an example, assume that you have an interpreted REXX program, called **MYISPFRX**, that has many external routines, all written in REXX. Your program can be invoked as follows:

```
SELECT CMD(MYISPFRX)
```

One way of improving the performance is to create a load module containing the **MYISPFRX** program and all its external routines. To do this, use the DLINK option (see "Object Modules (z/OS)" on page 72).

If ISPF variables are accessed with REXX programs running under TSO:

- If SYSICMD is retrieved using the SYSVAR function, link-edited REXX EXECs return a null string.
- For compiled EXECs that are not link-edited and are therefore equal to interpreted REXX EXECs, SYSVAR('sysicmd') contains the EXEC name. The name of the link-edited REXX EXEC can be retrieved using SYSVAR('syspcmd') provided that it is obtained before any other subcommand is issued.
- In interpreted REXX EXECs and compiled REXX EXECs that are not link-edited, the initial value in SYSVAR('syspcmd') is 'EXEC'.

Earlier Releases of ISPF

For earlier releases of ISPF you must use the VDEFINE service to define all variables that will be manipulated by ISPF services, such as VGET. The VDEFINE service cannot be invoked from a REXX program, therefore the creation of a load module from a REXX program is not supported, if the load module is to run directly from the SELECT service.

To run the load module, the **MYISPFRX** program must be linked with either the EFPL or the CPPLEFPL stub. You must write another REXX program that can be either interpreted or of CEXEC type. The source of your new program called, for example, **MYISPFST EXEC** is:

```
/* REXX * MYISPFST ****************************************************
* This EXEC calls MYISPFRX
*********************************************************************/
  CALL MYISPFRX
```

Figure 24. MYISPFST Sample Program

Invoke this program as follows:

```
SELECT CMD(MYISPFST)
```

The load module consisting of REXX programs will now run successfully.

ISPF Version 4 Release 1

Starting with ISPF Version 4.1, compiled REXX load modules are supported through the ISPSTART command and the SELECT service by a new value, CREX, for the LANG parameter of the CMD keyword.

To run the load module, the **MYISPFRX** program must be linked with either the CPPL or the CPPLEFPL stub. Invoke the program as follows:

```
SELECT CMD(MYISPFRX) LANG(CREX)
```

ISPF uses the correct function pool for the variables. For example:

- To copy a variable to the ISPF pool:

  ```
  /* Copy variable to variable to ISPF pool */
     myvar='-TESTING VPUT-';
     ADDRESS ISPEXEC "VPUT MYVAR PROFILE";
     Exit;
  ```

- To get a variable from the ISPF pool:

  ```
  /* Get a variable from the ISPF pool */
     ADDRESS ISPEXEC 'VGET MYVAR PROFILE';
     SAY 'Variable myvar holds:' myvar;
     Exit;
  ```

- To call a load module:

  ```
  /* Call a load module */
     "SELECT CMD(MYVPUT) LANG(CREX)";
     SAY 'VPUT RC='rc;
     "SELECT CMD(MYVGET) LANG(CREX)";
     SAY 'VGET RC='rc;
  ```

The load module consisting of REXX programs will now run successfully.

ISPF for z/OS Version 1 Release 5.5

REXX stubs are required to build a link-edited module from the object output of the REXX Compiler. If ISPF services are used by the module, the invocation parameter LANG(CREX) must be specified. With ISPF for z/OS Version 1 Release 5.5 the value CREX for the LANG parameter is obsolete.

The following items are prepared for this change:

1. The REXX stubs EAGSTCE, EAGSTCPP, and EAGSTMP are updated to denote ISPF that a REXX module is invoked.
2. ISPF will be updated with Release 5.5 to recognize the information of the stub to provide the full service as if it were invoked with parameter LANG(CREX).

Note:

1. To use the update of the stubs, you must rebuild all REXX modules using ISPF with REXXC or REXXL. If they are not rebuilt, the modified stubs are not incorporated.
2. For more information about stubs refer to "%STUB" on page 41.
3. See also "REXXL (z/OS)" on page 74.
4. See also "Stubs" on page 211.

Link-Editing of Object Modules

There are various parameter-passing conventions. Stubs are used to:

- Transform the input parameters into a form understandable by the compiled REXX program
- Invoke the compiled REXX program
- Transform the returned result into a form understandable by the caller

You must link-edit the OBJECT output of the Compiler with a stub.

To compile a program and link-edit the resulting **OBJECT** output with a stub, use the REXXC EXEC with the enhanced **OBJECT** option (see "Compiler Options" on page 19), or the REXXCL cataloged procedure which is supplied with the Compiler. Note that these require that the Library is also installed on your system.

To link-edit a stub and a compiled REXX program, use the REXXL cataloged procedure, which is supplied with the Library, or the REXXL EXEC to link-edit a stub and a compiled REXX program.

When used in a batch job, REXXL EXEC generates the control statements for the linkage editor to link-edit a stub and a compiled REXX program of type OBJECT. The compiled REXX program is read from the data set allocated to SYSIN. The control statements, including the compiled REXX program, are written to a data set allocated to SYSOUT.

When used interactively, REXXL EXEC link-edits a stub and the compiled REXX program of type OBJECT and builds a load module. The SYSPRINT output of the linkage editor is stored in a sequential data set, where the last identifier is LINKLIST.

Note: For object modules, do not use 8-character names that differ only in the eighth character, because the eighth character of the program name is lost during the link-edit step.

The original name of each stub is EAGSTUB. Each stub contains an external reference to the compiled REXX program named EAGOBJ.

The name of the OBJECT module in the external symbol dictionary (ESD) record is derived from the name of the input data set when the REXX program is compiled. It is one of the following:

- The member name of the partitioned input data set
- The last qualifier of the name of the sequential input data set
- Or else, COMPREXX (for example, if the source file is part of the job stream)

To link a stub with a program, REXXL generates the following linkage editor input:

```
  CHANGE  EAGSTUB(csect),EAGOBJ(temp_name)
  INCLUDE SYSLIB(stub_name)
  CHANGE  csect(temp_name)
**compiled REXX program is included here**
  ENTRY   csect
```

An example of this code is provided in "REXXL (EAGL)" on page 238.

For example, if the REXX program **AGOODPGM** is to be link-edited with the EFPL stub, the control statements are as follows:

```
    CHANGE   EAGSTUB(AGOODPGM),EAGOBJ($AGOODPG)
    INCLUDE  SYSLIB(EAGSTEFP)
    CHANGE   AGOODPGM($AGOODPG)
  **compiled REXX program AGOODPGM is included here**
    ENTRY    AGOODPGM
```

With this input, the linkage editor performs the following:

- Changes the external name of the stub to the original name of the compiled REXX program. The name of the compiled REXX program becomes a temporary name, which is the original name contained in the ESD record, prefixed with a $ character, and truncated to eight characters.
- Includes the stub
- Changes the external name of the REXX program to the temporary name
- Includes the compiled REXX program

The *csect* name, which is now the external name of the stub, is the recognized entry point.

Instead of invoking the Compiler and the linkage editor separately, you can create a load module with a single invocation of the REXXC command. Assuming that the source for **AGOODPGM** is located in the partitioned data set *upref*.**REXX.EXEC**, the following statement generates a load module with name **AGOODPGM**, with an EFPL stub in the partitioned data set *upref*.**REXX.LOAD**:

```
REXXC REXX.EXEC(AGOODPGM) OBJECT(,EFPL)
```

Note:

1. You can link different stubs to a compiled REXX program to make a program known under different names for invocation with different parameter-passing conventions. Or you can use your own renaming scheme by preparing the necessary linkage editor control statements yourself.
2. For more information refer to "Compiler Options" on page 19.
3. See also "REXXL (z/OS)" on page 74.

DLINK Example

The use of the DLINK option is discussed in "DLINK" on page 24. The following is a step-by-step example of an application that is packaged using the DLINK and OBJECT options of the Compiler.

This particular application is simply a performance test for the DLINK option. It is made up of the following REXX programs:

DLT: Is the main program. The source code is shown in Figure 25 on page 209.

CPUTIME:
Returns the CPU time that has been used. The source code is shown in Figure 26 on page 210.

ECHO:
Is a simple EXEC that returns the argument that was passed to it. The source code is shown in Figure 27 on page 210.

Note: The names are unique in the first seven characters, to prevent a naming conflict when the stubs are added.
The DLT EXEC was originally stored in a partitioned data set allocated to the ddname SYSPROC. It was invoked using a command equal to its name, DLT. The other two EXECs were included in the same partitioned data set, and were found as external routines only after all function packages and all the appropriate load libraries had been searched.

```
/* REXX * DLT ************************************************************
* Performance Test for DLINK option:
* Invoke external routine ECHO 50 times and tell how long it took
************************************************************************/
  n='DLT'
  Parse Version v .          /* Use Parse Version to see if compiled  */
  If left(v,5)='REXXC' Then what=n 'compiled'
                       Else what=n 'interpreted'
  Say what
  num=50

  t0=cputime()
  Call time 'r'
  Say num 'invocations of ECHO will be measured'
  Do i=1 To num
    Call echo i
    End
  Say 'This took me' (cputime()-t0) 'CPU-seconds.'
  Say '(elapsed:' time('E')')'
```

Figure 25. DLT Sample Program

The CPUTIME program can be used on several operating systems. The CPU time is calculated using an operating-system-dependent facility. Logic is also included to return the output when the program is invoked as an external routine.

```
/* REXX * CPUTIME ******************************************************
* Return the cpu-time used up so far
**********************************************************************/
  Parse Version v
  Parse Source  s

  Parse Var s sys .

  Select                          /* Figure out which system we are on */
    When sys='CMS' Then Do
      qt="DIAG"(8,'Q TIME')
      Parse Var qt . 'VIRTCPU=' mm . ':' +1 ss +6
      cpu=mm*60+ss
      End
    When sys='TSO' Then Do
      cpu=sysvar('SYSCPU')
      End
    When wordpos(sys,'PCDOS OS/2')>0 Then Do
      t=Time()
      Parse Var t hh ':' mm ':' ss
      cpu=(hh*60+mm)*60+ss
      End
    Otherwise Do
      Say 'System' sys 'is unknown to CPUTIME'
      cpu=0
      End
    End
  If word(s,2)='COMMAND' Then
    Say 'CPU time used so far:' cpu
  Else                              /* When an external routine     */
    Return cpu                      /* Return the CPU time          */
```

Figure 26. CPUTIME Sample Program

ECHO is a simple EXEC that returns its first argument.

```
/* REXX * ECHO ********************************************************
* Performance Test for DLINK option:
* Return the argument
**********************************************************************/
  Return arg(1)
```

Figure 27. ECHO Sample Program

To package this application, the following steps are required:

1. Compile all the routines that will be included in the application with both the OBJECT and DLINK options. In our example, DLT, CPUTIME, and ECHO are the appropriate routines.
2. Create a load module with the OBJECT code for the main routine and the appropriate stub, using either the REXXL cataloged procedure, or the REXXL command provided with the Library. In our example, we create a load module with DLT and the CPPL stub.
3. Once again, using the REXXL cataloged procedure, or the REXXL command provided with the Library, create a load module with the OBJECT code for each of the external routines and the EFPL stub. In our example, we combine both the CPUTIME and the ECHO routine with an EFPL stub. This creates two separate load modules both having their own EFPL stub.

4. Combine all these load modules into a single load module using the linkage editor. The entry point for this load module is DLT. In our example, BJVLIB is the ddname of the load library containing the programs. The control statements for the linkage editor are:

```
INCLUDE BJVLIB(DLT)
INCLUDE BJVLIB(ECHO)
INCLUDE BJVLIB(CPUTIME)
ENTRY   DLT
NAME    DLT(R)
```

   Place the load module in the appropriate load library so that it will get control before the REXX EXEC. The application is packaged and ready to run.

Notes on recursive routines that are compiled with the DLINK option:

- Routines that are called from other external routines recursively must be linked to the appropriate EFPL or CPPLEFPL stub.
- Routines that call themselves recursively must be renamed to a temporary name before compilation, otherwise the internal recursive call resolves to the beginning of the OBJECT module instead of the beginning of the stub.

  If, for example, DLT contained a `Call DLT` instruction, the following actions would be required:

  1. Rename DLT to a temporary name, for example: **DLT1**
  2. Compile **DLT1** with compiler options DLINK, NOCE, and OBJ
  3. Link **DLT1** to the CPPLEFPL stub:

```
CHANGE  EAGSTUB(DLT),EAGOBJ(DLT1)
INCLUDE SYSLIB(EAGSTCE)
INCLUDE OBJECTS(DLT1)
ENTRY   DLT
NAME    DLT(R)
```

Stubs

A stub is code that:

- Provides an interface between a certain parameter-passing convention and the parameter-passing convention defined for REXX programs
- Invokes the compiled REXX program using IRXEXEC
- Transforms the result of the compiled REXX program into a form understandable by the caller
- Is selected using a %STUB control directive in the source as described in "%STUB" on page 41.

Stub Names

The following stub names are supplied with the Library to provide interfaces with the following types of parameter-passing conventions (see also "Object Modules (z/OS)" on page 72):

CPPL (command processor parameter list)
: For running REXX applications from the TSO/E command line as a TSO command processor or if the program was invoked from an EXEC that contained ADDRESS TSO. See also "CPPL Parameter List" on page 215.

EFPL (external function parameter list)
: For REXX applications that are invoked by a REXX CALL statement or as function *program_name()*. EFPL must be used when building a function package. See also "EFPL Parameter List" on page 216.

CPPLEFPL
This is a combination of the CPPL and EFPL stubs. It determines if the program is invoked as a TSO/E command or as a REXX external routine. It is recommended for most compiled REXX applications running under TSO/ISPF. See also "CPPLEFPL" on page 217.

MVS For invoking the link-edited REXX load module from z/OS JCL using EXEC PGM=*program_name*, or as a host command from an EXEC with **ADDRESS LINKMVS** or **ADDRESS ATTCHMVS**. See also "MVS Parameter List" on page 217.

CALLCMD
For calling the program from the TSO/E command line using the TSO/E CALL command, or from another REXX EXEC using ADDRESS TSO for invoking the TSO/E CALL command. See also "CALLCMD Parameter List" on page 217.

MULTI (multi-purpose stub)
Simplifies link-edit and packaging (linking together into one load module) of compiled REXX applications under z/OS. It combines:

- EFPL
- CPPLEFPL
- CALLCMD
- Part of the MVS functionality ('EXEC PGM=name')
- ADDRESS LINK
- ADDRESS ATTACH

Note:

1. For ADDRESS LINKMVS and ADDRESS ATTCHMVS the existing MVS stub must be selected.
2. The multi-purpose stub load module member is located as 'EAGSTMP' in the existing SEAGLMD library.

Note: Object modules generated with STUB code terminate abnormally if they are run under z/VM.
If you want to create additional stubs, you can use the stubs shipped in the sample data set as models.

Stub name	Member name in the SEAGLMD data set
CPPL	EAGSTCPP
EFPL	EAGSTEFP
CPPLEFPL	EAGSTCE
MVS	EAGSTMVS
CALLCMD	EAGSTCAL
MULTI	EAGSTMP

Processing Sequence for Stubs

For each stub, the general processing sequence is as follows:

1. Save the registers.
2. Obtain storage required to execute the stub. For an EFPL parameter list, storage is requested from the same subpool as REXX. For CPPL and CALLCMD

parameter lists, storage is requested from subpool 78. For MVS parameter lists, no subpool parameter is supplied for obtaining the required storage.

3. Build a parameter list to invoke IRXEXEC. How the input parameter list maps into the parameter list for the invocation of IRXEXEC is shown separately for each type of parameter list.
4. Invoke IRXEXEC.
5. Convert the result supplied by IRXEXEC to the form needed for a specific type of invocation (described separately for each type of invocation).
6. Free the storage obtained in Step 2 on page 212.
7. Restore the registers and return to the caller.

Parameter List for Invoking IRXEXEC

The parameter list for invoking IRXEXEC is as follows:

Parameter 1
: The address of an **EXECBLK.** An **EXECBLK** address is never supplied; therefore the value of the parameter is 0.

Parameter 2
: The address of the argument list.

Parameter 3
: Specify the type of invocation (COMMAND, SUBROUTINE, or FUNCTION) and whether extended return codes are requested.

  The COMMAND invocation is specified except for EFPL parameter lists where the SUBROUTINE invocation is specified. Extended return codes are always requested.

Parameter 4
: The address of the in-storage control block describing the compiled program. An in-storage control block is always supplied.

Parameter 5
: The address of the CPPL. The value of the parameter is 0 if no CPPL is supplied.

Parameter 6
: The address of the EVALBLOCK control block that is to contain the result.

  For EFPL parameter lists, the passed EVALBLOCK control block is used. In all other cases, an EVALBLOCK control block with a data length of 16 bytes is used. This is large enough to hold any expected result. It holds the result of a COMMAND invocation, which must be numeric and must fit into a fullword.

Parameter 7
: A work area vector address is never supplied; therefore the value of the parameter is 0.

Parameter 8
: The address of a user field or 0. A user field address is never supplied; therefore the value of the parameter is 0.

Parameter 9
: The address of the environment block.

  For EFPL parameter lists, the address of the environment block as passed in register 0 is supplied. Otherwise, no parameter is supplied.

For a complete description of the parameters, refer to the *TSO/E REXX/MVS: Reference*.

In-Storage Control Block

The in-storage control block supplied when IRXEXEC is invoked is as follows (the default values are indicated in parentheses):

ACRONYM
String `'IRXINSTB'`.

HDRLEN
Length of the in-storage control block.

ADDRESS
Address of the vector of entries. A vector of records containing one address and length pair is supplied. The address points to the setup code, and the length is 20; this is the length needed for IRXEXEC to identify the header.

USEDLEN
Length of the vector of records (8).

MEMBER
Name of the EXEC ('? ').

DDNAME
Name of the DD from which the program was loaded (' ').

SUBCOM
Name of the initial host command environment (' ').

DSNLEN
Length of the data set name (0).

DSNAME
Name of the data set (X'00').

If the environment is known (because a program is linked to the EFPL stub), the environment is passed to IRXEXEC when IRXEXEC is called. Otherwise, a value of 0 is passed to register 0. IRXEXEC locates the last non-reentrant environment and uses it when it executes your program.

Parameter Lists

Each of the following sections contains a figure showing, in the upper part, the parameter list that is passed to the stub when the stub is invoked. Register 1 points to this parameter list. The upper part of the figure also shows the relevant surrounding structures. The lower part of the figure shows the parameter list that is passed to IRXEXEC when IRXEXEC is invoked. Register 1 points to this parameter list. The lower part of the figure also shows the surrounding structures built by the stub for the invocation of IRXEXEC.

The following sections also describe, for each type of parameter list, how to obtain the return code (to be passed back in register 15) and, for EFPL, the necessary EVALBLOCK control block processing. For more information about registers refer to "Registers for Stubs."

Note: The MULTI stub combines the parameter lists of the stubs that are shown in this section.

Registers for Stubs

On entry to each stub, registers are set as follows:

Register 0
Address of the environment block (EFPL stub only)

Register 1
Address of the parameter list

Registers 2-12
Unpredictable

Register 13
Address of a register save area

Register 14
Return address

Register 15
Entry point address

On exit of each stub, registers are set as follows:

Registers 0-14
Same as on entry

Register 15
Return code

CPPL Parameter List

A CPPL parameter list is supplied if, on the TSO/E command line, the user issued the command *program_name*, or if the program was invoked from an EXEC that used ADDRESS TSO.

Storage is obtained from subpool 78.

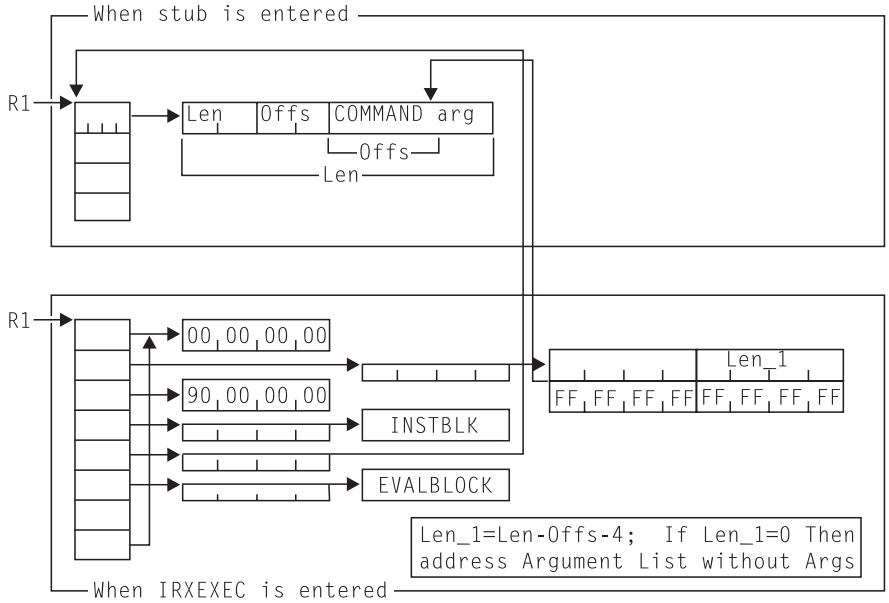


Figure 28. CPPL Parameter List Mapping

If the return code from IRXEXEC is not 0, the return code is passed back in register 15. Otherwise, the value contained in the EVALBLOCK control block is converted to a fullword and passed back in register 15.

EFPL Parameter List

An EFPL parameter list is supplied if, from within an EXEC, either the instruction CALL *program_name* is issued or a program is invoked through the function invocation *program_name*(). The compiled REXX program is always invoked as a subroutine, because the information specifying whether the program is to be invoked as a subroutine or as a command is not accessible.

Storage is obtained from the same subpool as REXX. The subpool number is contained in the parameter block, which is addressed through the environment block. The address of the environment block is passed in register 0 when the stub is entered.

Note: Most NetView applications require the EFPL stub.

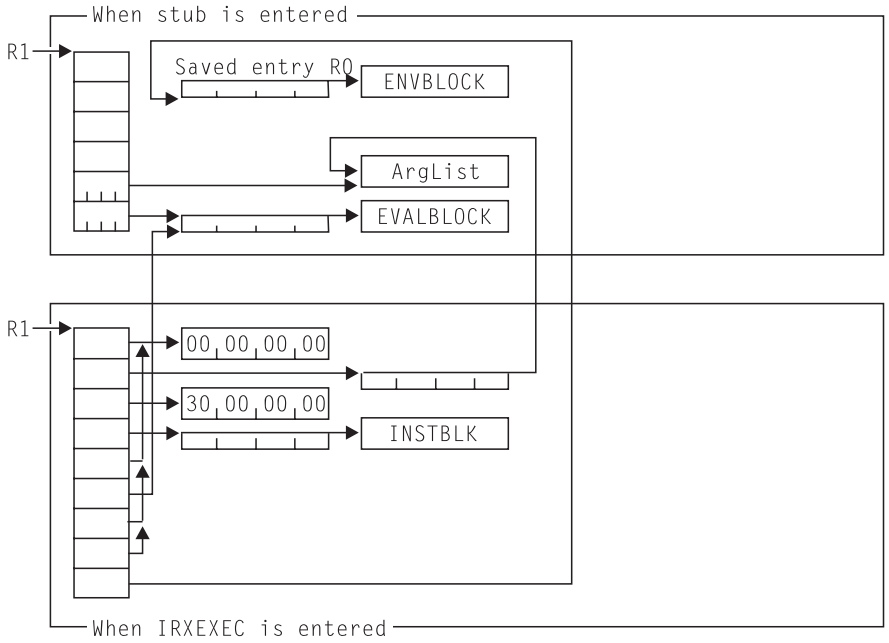


Figure 29. EFPL Parameter List Mapping

The required, final EVALBLOCK control block handling (and the determination of the return code to pass back in register 15) is:

```
rc_to_pass_back = 0
If rc_from_irxexec ¬=0 Then
  rc_to_pass_back=rc_from_irxexec
Else Do
  If evalblock shows truncated result Then Do
    invoke irxrlt 'GETBLOCK'
    If rc ¬= 0 Then
      rc_to_pass_back=rc
    Else Do
      put new evalblock Address INTO parameter list
      invoke irxrlt 'GETRLTE' With new evalblock
      If rc ¬= 0 Then
        rc_to_pass_back=rc
      End
    End
  End
```

If the return code passed back from IRXEXEC is 100 or 104 (which indicates an abend), register 0 contains the value passed back by IRXEXEC (abend code and reason code).

CPPLEFPL

This stub is a combination of the CPPL and EFPL stubs. It contains the logic to determine if the REXX program is being invoked as a TSO/E command or as a REXX external routine. Once this has been determined, the compiled REXX program is given control with the appropriate parameters.

CPPLEFPL is recommended for most compiled REXX programs running under TSO/ISPF.

MVS Parameter List

An MVS parameter list is supplied when a program is invoked from z/OS JCL by means of EXEC PGM=*program_name*, or as a host command from an EXEC with ADDRESS LINKMVS or ADDRESS ATTCHMVS.

The end of the parameter list is indicated by the high-order bit of the last element of the address list being set to 1.

When obtaining storage, no subpool parameter is supplied.

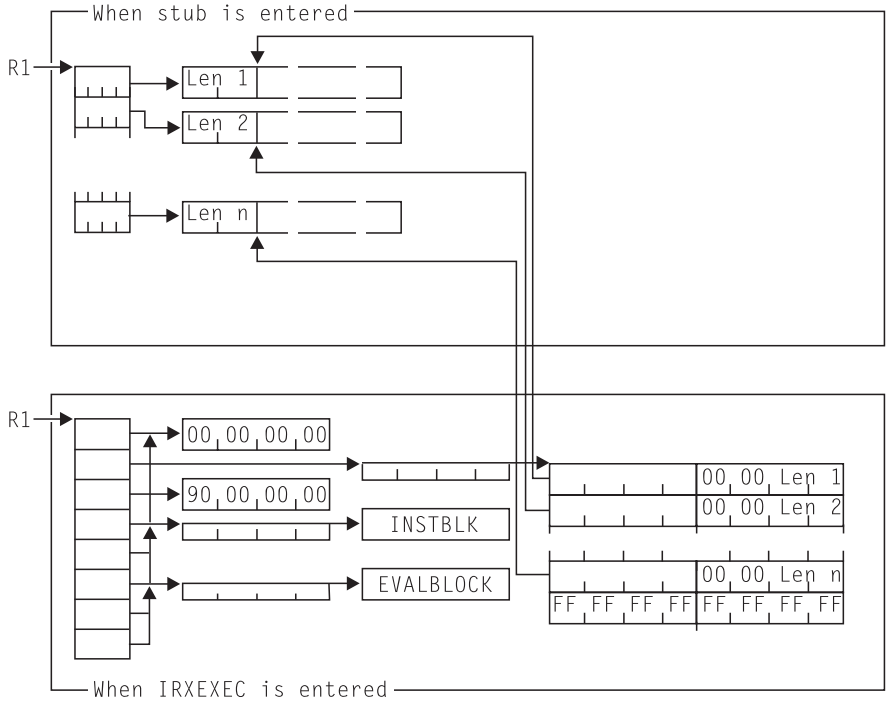


Figure 30. z/OS Parameter List Mapping

If the return code from IRXEXEC is not 0, the return code is passed back in register 15. Otherwise, the value contained in the EVALBLOCK control block is converted to a fullword and passed back in register 15.

CALLCMD Parameter List

A CALLCMD parameter list is supplied when the CALL *program_name* command is issued from the TSO/E command line, or when the CALL *program_name* host command is issued from within an EXEC executing under TSO/E.

The address pointed to by register 1 on entry is an AMODE 24 address (the first byte must be ignored).

Storage is obtained from subpool 78.

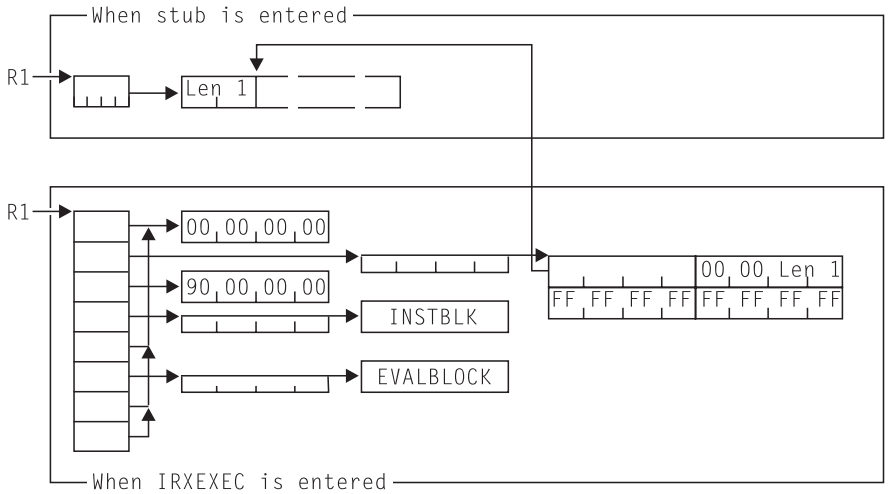


Figure 31. CALLCMD Parameter List Mapping

If the return code from IRXEXEC is not 0, it is passed back in register 15. Otherwise, the value contained in the EVALBLOCK control block is converted to a fullword and passed back in register 15.

Search Order

When an external function or subroutine is invoked from a compiled REXX program of OBJECT type, the standard REXX search order applies. The in-storage control block that is set up in the stubs indicates that the compiled REXX program has been loaded from the default system file in which you can store REXX EXECs.

Testing Stubs

You can use the following program to test that a stub is invoked and the parameter list is passed correctly.

```
/* Tell me who I am */
   Parse source allsrc;
   Arg allp;
   Say 'Source;' allsrc;
   Say 'says hello world...';
   If allp /='' then Say 'Parmlist:' allp;
   Else Say 'No parmlist received...';
   Exit;
```

If you are using the wrong stub, one of the following might happen:

- 0C4 Abend in the stub before calling the compiled program.
- The parameters are not all passed to the compiled program.

In either case, use a different stub. For example, if you used **CALLCMD** stub, use **MVS** stub instead.

PARSE SOURCE

For a REXX program compiled into an object module, the source string that can be obtained by means of the PARSE SOURCE instruction contains the following tokens:

- The characters TSO
- If the program is linked with the EFPL stub, the string SUBROUTINE; otherwise, the string COMMAND
- A question mark (?) to indicate that the name of the EXEC is not known
- A question mark (?) to indicate that the name of the DD statement from which the EXEC was loaded is not known
- A question mark (?) to indicate that the name of the data set from which the EXEC was loaded is not known
- A question mark (?) to indicate that the name of the EXEC as it was invoked is not known
- The initial host command environment in uppercase
- The name of the address space in uppercase
- An 8-character user token

Appendix B. Interface for TEXT Files (z/VM)

This appendix explains how an Assembler program can invoke a REXX program that has been compiled into a TEXT file under z/VM. It also describes the parameters and the PARSE SOURCE information received by the REXX program.

The Call from the Assembler Program

A TEXT file can be linked to an Assembler program and may be called by using any of the standard forms of PLIST.

Call Type

Under z/VM the call type is specified in the byte that follows the 24-word save area.

Registers

On entry to the called program, the following registers are defined:

R0 For call type X'05', the address of a 6-word extended PLIST (see "Extended PLISTs") and, in the high-order bit, an indication of the invocation type.

For call types X'01', X'0B', X'02', and X'06', the address of an extended PLIST (see "Extended PLISTs").

R1 The address of a tokenized PLIST.

R2 User word (meaningful only for non-SVC invocation).

R13 The address of a 24-word save area.

The byte that follows the save area specifies the call type.

For SVC invocations, the SVC handler provides the save area and sets register 13.

R14 The return address.

For SVC invocations, the SVC handler sets this register.

R15 The entry point address.

For SVC invocations, the SVC handler sets this register.

On return to the Assembler program, the following register is defined:

R15 The return code.

For call type X'05', this is the return code produced by the last operation that set the return code during execution of the REXX program. The value specified in the RETURN or EXIT instruction is passed back by means of the 6-word extended PLIST.

For all other call types, this is the return code specified on the RETURN or EXIT instruction.

Extended PLISTs

The extended PLIST has the form:

```
EPLIST   DS    0F                  PLIST with pointers:
         DC    A(COMVERB)          → C'synonym' CL1' '
*                                    (Note that this area must precede
*                                     the area containing the Argstring.)
         DC    A(BEGARGS)          → start of Argstring
         DC    A(ENDARGS)          → character after end of the Argstring
         DC    A(FBLOK)            → file block
*                                    (If there is no file block,
*                                     this pointer must be 0.
*                                     The high-order byte is ignored.)
```

The 6-word extended PLIST has the same four pointers followed by:

```
         DC    AL4(ARGLIST)        → Argument list.
*                                    If there is no argument list,
*                                    this pointer is 0, and BEGARGS/ENDARGS
*                                    are used for the ARG string.
         DC    A(SYSFUNRT)         → SYSFUNRT location, which:
*                                    - contains a 0 on entry
*                                    - will be unchanged if no result is
*                                      returned
*                                    - will contain the address of an
*                                      EVALBLOK if a result is returned
```

What the REXX Program Gets

The arguments accessible through the PARSE ARG instruction and the ARG built-in function, and the information returned by the PARSE SOURCE instruction, depend on the type of PLIST used.

Invocation with a Tokenized PLIST Only

If the program is invoked with only a tokenized PLIST, the argument string is available to the program as a single argument. This is taken from the second token of the parameter list, which is delimited by X'FFFFFFFF'. There is one blank between each token of the argument.

The information returned by PARSE SOURCE is as follows:

Description of Token	Value
—	CMS
Invocation type	COMMAND
File name	The first token of the PLIST or *
File type	*
File mode	*
Synonym	The first token of the PLIST or ?
Initial (default) address for commands	CMS

Invocation with an Extended PLIST or a 6-Word Extended PLIST

If the program is invoked with an extended PLIST, the argument string (as defined by **BEGARGS** and **ENDARGS**) is available to the program as a single argument.

If the program is invoked with a 6-word extended PLIST and an argument list is supplied, the arguments are taken from the argument list. If the address of the argument list is 0, the argument string (as defined by **BEGARGS** and **ENDARGS**) is available to the program as a single argument.

The information returned by PARSE SOURCE is as follows:

Description of Token	Value
—	CMS
Invocation type	For call type X'05', when high-order bit of R0=1: FUNCTION For call type X'05', when high-order bit of R0=0: SUBROUTINE For all other call types: COMMAND
File name	The file name in the file block or, if there is no file block, the first token of the tokenized PLIST.
File type	The file type in the file block. If the file type in the file block is blank: EXEC. If there is no file block: *
File mode	The file mode in the file block. If there is no file block: *
Synonym	For files of type CEXEC: a question mark or the first token (delimited by an open parenthesis, close parenthesis, or blank) from the area identified by BEGARGS and ENDARGS. For files of type OBJ: the synonym from the extended PLIST.
Initial (default) address for commands	If a named PSW is specified in the file block, that name is used. If an unnamed PSW is specified in the file block, ? is used. If the file type is EXEC or blank, or if there is no file block, CMS is used. Otherwise, the file type is used.

Example of an Assembler Interface to a TEXT File

The following code shows an example of how an Assembler program can invoke a TEXT file that has been linked to it. Note that the setting of the high-order bit of register 1 depends on the CMS release. The code in the example works correctly on all the releases of CMS supported by the Compiler and the Library. On XA systems, the example works with both 24-bit and 31-bit addressing.

```
         .
         .                        set up R0 if necessary
         LA    13,SAVE            address save area
         IC    15,TYPE            get call type
         SLL   15,24              to HOB, fill rest with 0s
         LA    1,0(,15)           0 for non-XA or type '00'x or '80'x
         LTR   1,1                is it 0 ?
         LA    1,TOKPL            address tokenized PLIST
         BNZ   $1                 skip for A and not '00'x or '80'x
         OR    1,15               insert HOB of R1 for non-XA machine
                                    or when type is '00'x or '80'x
$1       L     15,PROG            entry point
         BALR  14,15              invoke REXX program
         .                        REXX program will return here
         .
PROG     DC    V(REXXPRG)         entry of compiled program, name of
                                    the source file goes here
TOKPL    DC    CL8'REXXPRG'       tokenized PLIST
         DC    CL8'token 1'       parameter starts here (if passed by means of
         DC    CL8'token 2'         tokenized PLIST)
         .
         .
         DC    8X'FF'             tokenized PLIST ended by fence
SAVE     DS    24F                save area
TYPE     DC    X'00'              call type follows save area, enter
                                    required call type here
```

Note: In this case, **HOB** stands for high order byte.

Appendix C. Interface for Object Modules (VSE/ESA)

This appendix describes the parameter passing conventions for the different stubs and how the stubs invoke the EXEC handler, ARXEXEC. This appendix also describes the PARSE SOURCE information, as it appears in the REXX program.

Stubs

A stub is code that:

- Provides an interface between a certain parameter-passing convention and the parameter-passing convention defined for REXX programs
- Invokes the compiled REXX program
- Transforms the result of the compiled REXX program into a form understandable by the caller

Two stubs are supplied with the Library to provide interfaces with the following types of parameter-passing conventions:

- VSE (refer to "VSE Parameter List" on page 228)
- EFPL (refer to "EFPL Parameter List" on page 228)

If you want to create additional stubs, you can use the supplied stubs as models.

Processing Sequence for Stubs

For each stub, the general processing sequence is as follows:

1. Save the registers.
2. Obtain storage required to execute the stub.
3. Build a parameter list to invoke ARXEXEC. How the input parameter list maps into the parameter list for the invocation of ARXEXEC is shown separately for each type of parameter list.
4. Invoke ARXEXEC.
5. Convert the result supplied by ARXEXEC to the form needed for a specific type of invocation (described separately for each type of invocation).
6. Free the storage obtained in Step 2.
7. Restore the registers and return to the caller.

Parameter List for Invoking ARXEXEC

The parameter list for invoking ARXEXEC is as follows:

Parameter 1

The address of an **EXECBLK.** An **EXECBLK** address is never supplied; therefore the value of the parameter is 0.

Parameter 2

The address of the argument list.

Parameter 3

Specify the type of invocation (COMMAND, SUBROUTINE, or FUNCTION) and whether extended return codes are requested.

The COMMAND invocation is specified except for EFPL parameter lists where the SUBROUTINE invocation is specified. Extended return codes are always requested.

Parameter 4
The address of the in-storage control block describing the compiled program. An in-storage control block is always supplied.

Parameter 5
Reserved, must be 0.

Parameter 6
The address of the EVALBLOCK control block that is to contain the result.

For EFPL parameter lists, the passed EVALBLOCK control block is used. For VSE parameter lists, an EVALBLOCK control block with a data length of 16 bytes is used. This is large enough to hold any expected result. It holds the result of a COMMAND invocation, which must be numeric and must fit into a fullword.

Parameter 7
The address of a work area vector or 0. A work area vector address is never supplied; therefore the value of the parameter is 0.

Parameter 8
The address of a user field or 0. A user field address is never supplied; therefore the value of the parameter is 0.

Parameter 9
The address of the environment block.

For EFPL parameter lists, the address of the environment block as passed in register 0 is supplied. Otherwise, no parameter is supplied.

For a complete description of the parameters, refer to *IBM VSE/ESA REXX/VSE Reference*.

In-Storage Control Block

The in-storage control block supplied when ARXEXEC is invoked is as follows (the default values are indicated in parentheses):

ACRONYM
String `'ARXINSTB'`.

HDRLEN
Length of the in-storage control block.

ADDRESS
Address of the vector of records. A vector of records containing one address and length pair is supplied. The address points to the setup code, and the length is 20; this is the length needed for ARXEXEC to identify the header.

USEDLEN
Length of the vector of records (8).

MEMBER
Name of the EXEC ('? ').

DDNAME
Name of the member that represents the load data set (' ').

SUBCOM
Name of the initial host command environment (' ').

DSNLEN
Length of the data set name (0).

DSNAME
Name of the data set (X'00').

If the environment is known (because a program is linked to the EFPL stub), the environment is passed to ARXEXEC when ARXEXEC is called. Otherwise, a value of 0 is passed to register 0. ARXEXEC locates the last non-reentrant environment and uses it when it` executes your program.

Parameter Lists

Each of the following sections contains a figure showing in the upper part the parameter list that is passed to the stub when the stub is invoked. Register 1 points to this parameter list. The upper part of the figure also shows the relevant surrounding structures. The lower part of the figure shows the parameter list that is passed to ARXEXEC when ARXEXEC is invoked.

The following sections also describe, for each type of parameter list, how to obtain the return code (to be passed back to register 15) and, for EFPL, the necessary EVALBLOCK control block processing.

Registers for VSE/ESA Stubs

On entry to the VSE stub, registers are set as follows:

Register 0
Unpredictable

Register 1
Address of the parameter list if the contents of Register 1 and Register 15 are different.

Registers 2-12
Unpredictable

Registers 13
Address of a register save area

Register 14
Return address

Register 15
Unpredictable

On exit from the VSE stub, registers are set as follows:

Registers 0-14
Same as on entry

Register 15
Return code

On entry to the EFPL stub, registers are set as follows:

Register 0
Address of the environment block

Register 1
Address of the parameter list

Registers 2-12
Unpredictable

Register 13
Address of a register save area

Register 14
Return address

Register 15
Entry point address

On exit from the EFPL stub, registers are set as follows:

Registers 0-14
Same as on entry

Register 15
Return code

VSE Parameter List

A VSE parameter list is supplied when a program is invoked from VSE JCL by means of EXEC *program_name*.

No parameter list is provided if on entry to the stub register 1 and register 15 are set to the same value.

The high-order bit of the fullword addressed by register 1 on entry to the stub is set to 1 if the parameter length is greater than 0, otherwise it is set to 0. The address pointed to by Register 1 on entry is an **AMODE 24** address (the first byte must be ignored).

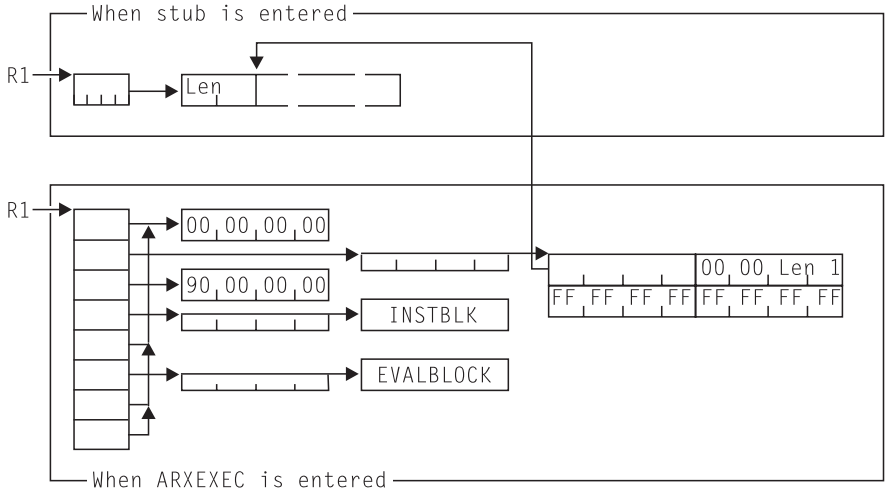


Figure 32. VSE Parameter List Mapping

A return code of 4095 is passed back in register 15 if either storage could not be obtained, or **ARXEXEC** could not be loaded, or **ARXEXEC** issued a return code different from 0 (indicating that the program did not complete successfully). If the return code from **ARXEXEC** is 0, the value contained in the **EVALBLOCK** control block is divided by 4096, and the remainder is passed back in register 15.

EFPL Parameter List

An EFPL parameter list is supplied if, from within an EXEC, either the instruction CALL *program_name* is issued or a program is invoked through the function invocation *program_name*(). The compiled REXX program is always invoked as a subroutine, because the information specifying whether the program is to be invoked as a subroutine or as a command is not accessible.

The address of the environment block is passed in register 0 when the stub is entered.

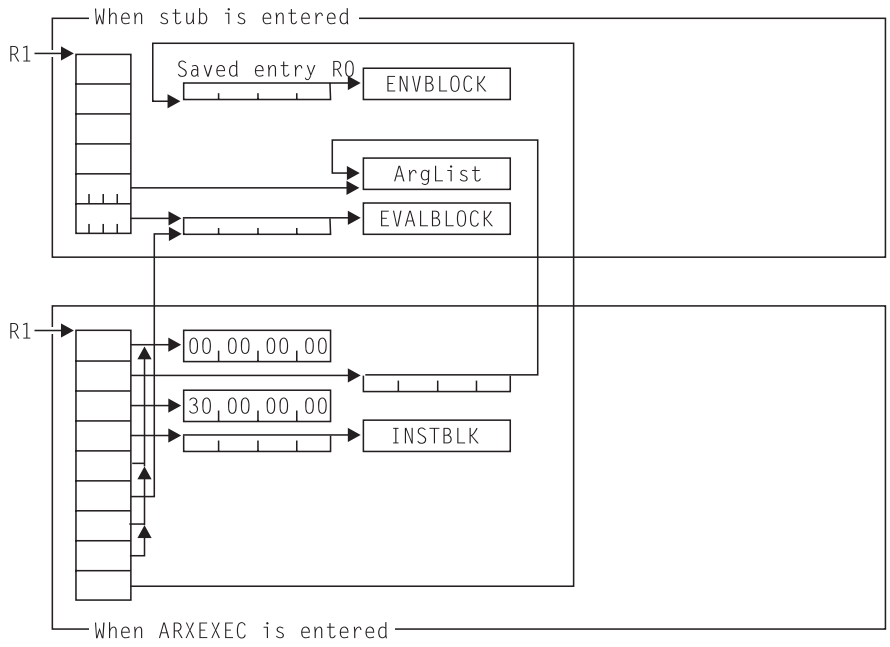


Figure 33. EFPL Parameter List Mapping

The required, final EVALBLOCK control-block handling (and the determination of the return code to pass back in register 15) is:

```
rc_to_pass_back = 0
If rc_from_arxexec ¬=0 Then
  rc_to_pass_back=rc_from_arxexec
Else Do
  If evalblock shows truncated result Then Do
    invoke arxrlt 'GETBLOCK'
    If rc ¬= 0 Then
      rc_to_pass_back=rc
    Else Do
      put new evalblock Address INTO parameter list
      invoke arxrlt 'GETRLTE' With new evalblock
      If rc ¬= 0 Then
        rc_to_pass_back=rc
      End
    End
  End
```

If storage could not be obtained, a return code of 100 is passed back in register 15. In this case, register 0 contains a cancel code of 0. If the return code passed back from **ARXEXEC** is either 100 or 104 (which indicates an abend), register 0 contains the value passed back by **ARXEXEC** (cancel code).

PARSE SOURCE

For a REXX program compiled into an object module, the source string that can be obtained by means of the PARSE SOURCE instruction contains the following tokens:

- The characters VSE
- If the program is linked with the EFPL stub, the string SUBROUTINE; otherwise, the string COMMAND
- A question mark (?) to indicate that the name of the EXEC is not known

- A question mark (?) to indicate that the name of the DD statement from which the EXEC was loaded is not known
- A question mark (?) to indicate that the name of the file from which the EXEC was loaded is not known
- A question mark (?) to indicate that the name of the file as it was passed to the language processor (that is, the name is not translated to uppercase) is not known
- The initial host command environment in uppercase
- The name of the address space in uppercase
- An 8-character user token

Appendix D. The z/OS Cataloged Procedures Supplied by IBM

You can compile a REXX program in a z/OS batch environment by using a cataloged procedure that is invoked by an EXEC statement in your job.

Note: Your system administrator may have customized the cataloged procedures on your system.

The following cataloged procedures are supplied by IBM:

REXXC (FANCMC)

REXXC compiles a REXX program. FANCMC is located in the data set *prefix*.SFANPRC.

```
//*********************************************************************
//*
//* REXXC   Compile a REXX program.
//*
//*    Licensed Materials - Property of IBM
//*    5695-013 IBM REXX Compiler
//*    (C) Copyright IBM Corp. 1989, 2003
//*
//* Change Activity:
//*   03-05-28       Release 4.0
//*
//*********************************************************************
//*
//* Parameters:
//*
//*   OPTIONS     Compilation options.
//*               Default: XREF OBJECT
//*
//*   COMPDSN     DSN of IBM REXX Compiler load library.
//*
//* Required:
//*
//*   REXX.SYSIN  DDNAME, REXX program to be compiled.
//*
//* Example:
//*
//*   To compile MYREXX.EXEC(MYPROG) and to keep the resulting
//*   CEXEC output and OBJECT output in MYREXX.CEXEC(MYPROG) and
//*   MYREXX.OBJ(MYPROG), respectively, use the following
//*   invocation:
//*
//*   //S1 EXEC REXXC
//*   //REXX.SYSCEXEC DD DSN=MYREXX.CEXEC(MYPROG),DISP=SHR
//*   //REXX.SYSPUNCH DD DSN=MYREXX.OBJ(MYPROG),DISP=SHR
//*   //REXX.SYSIN    DD DSN=MYREXX.EXEC(MYPROG),DISP=SHR
//*
```

```
//* Modifications:
//*   Change #HLQREXX to the appropriate high-level qualifier of
//*   your installation.
//*
//*******************************************************************
//*
//REXXC   PROC OPTIONS='XREF OBJECT',         REXX Compiler options
//             COMPDSN='#HLQREXX.SFANLMD'     REXX Compiler load lib
//*
//*------------------------------------------------------------------
//* Compile REXX program.
//*------------------------------------------------------------------
//*
//REXX    EXEC PGM=REXXCOMP,PARM='&OPTIONS'
//STEPLIB   DD DSN=&COMPDSN,DISP=SHR
//SYSPRINT  DD SYSOUT=*
//SYSTERM   DD SYSOUT=*
//*SYSIEXEC DD DUMMY
//*SYSDUMP  DD DUMMY
//SYSCEXEC  DD DSN=&&CEXEC(GO),DISP=(MOD,PASS),UNIT=SYSDA,
//             SPACE=(800,(800,100,1))
//SYSPUNCH  DD DSN=&&OBJECT,DISP=(MOD,PASS),UNIT=SYSDA,
//             SPACE=(800,(800,100))
```

REXXCG (FANCMCG)

REXXCG compiles and runs a REXX program of type CEXEC. FANCMCG is located in the data set *prefix*.SFANPRC.

```
//*******************************************************************
//*
//* REXXCG  Compile and run a REXX program of CEXEC type.
//*
//*    Licensed Materials - Property of IBM
//*    5695-013 IBM REXX Compiler
//*    (C) Copyright IBM Corp. 1989, 2003
//*
//* Change Activity:
//*   03-05-28       Release 4.0
//*
//*******************************************************************
//*
//* Parameters:
//*
//*   OPTIONS     Compilation options.
//*               Default: XREF
//*
//*   COMPDSN     DSN of IBM REXX Compiler load library.
//*
//*   LIBLPA      DSN of IBM REXX Library LPA library.
//*               If &LIBLPA is in the search order, you may deactivate
//*               the GO.STEPLIB and the PROC LIBLPA definition.
//*
//* Required:
//*
//*   REXX.SYSIN  DDNAME, REXX program to be compiled and run.
//*
//* Example:
//*
//*   To compile MYREXX.EXEC(MYPROG), to keep the resulting CEXEC
//*   output in MYREXX.CEXEC(MYPROG), and to run this compiled
//*   program, passing the string MYPARM as parameter for this run,
//*   use the following invocation (note that the first token in the
//*   PARM of the GO step specifies the name of the program):
//*
//*   //S1 EXEC REXXCG,PARM.GO='MYPROG MYPARM'
//*   //REXX.SYSCEXEC DD DSN=MYREXX.CEXEC(MYPROG),DISP=SHR
```

```
//*   //REXX.SYSIN    DD DSN=MYREXX.EXEC(MYPROG),DISP=SHR
//*   //GO.SYSEXEC    DD DSN=MYREXX.CEXEC,DISP=SHR
//*
//* Modifications:
//*   Change #HLQREXX to the appropriate high-level qualifier of
//*   your installation.
//*
//*****************************************************************
//*
//REXXCG  PROC OPTIONS='XREF',                REXX Compiler options
//             COMPDSN='#HLQREXX.SFANLMD',    REXX Compiler load lib
//             LIBLPA='#HLQREXX.SEAGLPA'      REXX Library LPA lib
//*
//*---------------------------------------------------------------
//* Compile REXX program.
//*---------------------------------------------------------------
//*
//REXX    EXEC PGM=REXXCOMP,PARM='&OPTIONS'
//STEPLIB   DD DSN=&COMPDSN,DISP=SHR
//SYSPRINT  DD SYSOUT=*
//SYSTERM   DD SYSOUT=*
//*SYSIEXEC DD DUMMY
//*SYSDUMP  DD DUMMY
//SYSCEXEC  DD DSN=&&CEXEC(GO),DISP=(MOD,PASS),UNIT=SYSDA,
//             SPACE=(800,(800,100,1))
//SYSPUNCH  DD DUMMY
//*
//*---------------------------------------------------------------
//* Run the compiled REXX program.
//*---------------------------------------------------------------
//GO      EXEC PGM=IRXJCL,PARM='GO',
//             COND=(9,LT,REXX)
//*
//STEPLIB   DD DSN=&LIBLPA,DISP=SHR
//SYSEXEC   DD DSN=&&CEXEC,DISP=(OLD,DELETE)
//SYSTSPRT  DD SYSOUT=*
```

REXXCL (FANCMCL)

REXXCL compiles and link-edits a REXX program of type OBJECT. FANCMCL is located in the data set *prefix*.SFANPRC.

```
//*****************************************************************
//*
//* REXXCL  Compile and link edit a REXX program of OBJ type.
//*
//*    Licensed Materials - Property of IBM
//*    5695-013 IBM REXX Compiler
//*    (C) Copyright IBM Corp. 1989, 2003
//*
//* Change Activity:
//*   03-05-28       Release 4.0
//*
//*****************************************************************
//*
//* Parameters:
//*
//*   STUB        Stub type: MVS, CPPL, CALLCMD, EFPL, CPPLEFPL,
//*                          MULTI
//*               Default: EFPL.
//*
//*   OPTIONS     Compilation options.
//*               Default: XREF OBJECT NOCEXEC
//*
//*   COMPDSN     DSN of IBM REXX Compiler load library.
//*
//*   LIBDSN      DSN of IBM REXX Library load library for Stubs.
```

```
//*
//*   LIBXDSN     DSN of IBM REXX Library exec library.
//*
//* Required:
//*
//*   REXX.SYSIN  DDNAME, REXX program to be compiled and link
//*               edited.
//*
//* Example:
//*
//*   To compile MYREXX.EXEC(MYPROG) and to link edit the resulting
//*   OBJECT output together with a stub suitable for invocation
//*   of the program from a REXX EXEC with the CALL instruction or
//*   via function invocation, and to keep the resulting load module
//*   in MYREXX.LOAD(MYPROG), use the following invocation:
//*
//*   //S1 EXEC REXXCL
//*   //REXX.SYSIN   DD DSN=MYREXX.EXEC(MYPROG),DISP=SHR
//*   //LKED.SYSLMOD DD DSN=MYREXX.LOAD(MYPROG),DISP=SHR
//*
//* Modifications:
//*   Change #HLQREXX to the appropriate high-level qualifier of
//*   your installation.
//*
//*********************************************************************
//*
//REXXCL  PROC STUB=EFPL,                      Type of stub
//             OPTIONS='XREF OBJECT NOCEXEC', REXX Compiler options
//             COMPDSN='#HLQREXX.SFANLMD',    REXX Compiler load lib
//             LIBDSN='#HLQREXX.SEAGLMD',     REXX Library stub load
//             LIBXDSN='#HLQREXX.SEAGCMD'     REXX Library exec lib
//*
//*-------------------------------------------------------------------
//* Compile REXX program.
//*-------------------------------------------------------------------
//*
//REXX    EXEC PGM=REXXCOMP,PARM='&OPTIONS'
//STEPLIB   DD DSN=&COMPDSN,DISP=SHR
//SYSPRINT  DD SYSOUT=*
//SYSTERM   DD SYSOUT=*
//*SYSIEXEC DD DUMMY
//*SYSDUMP  DD DUMMY
//*SYSCEXEC DD DUMMY
//SYSPUNCH  DD DSN=&&OBJECT,DISP=(MOD,PASS),UNIT=SYSDA,
//             SPACE=(800,(800,100))
//*
//*-------------------------------------------------------------------
//* Prepare SYSLIN data set for subsequent link step.
//*-------------------------------------------------------------------
//*
//PLKED   EXEC PGM=IRXJCL,PARM='REXXL &STUB',
//             COND=(9,LT,REXX)
//*
//SYSEXEC   DD DSN=&LIBXDSN,DISP=SHR
//SYSIN     DD DSN=&&OBJECT,DISP=(OLD,DELETE)
//SYSTSPRT  DD SYSOUT=*
//SYSOUT    DD DSN=&&SYSOUT,DISP=(MOD,PASS),UNIT=SYSDA,
//             DCB=(RECFM=FB,LRECL=80,BLKSIZE=800),
//             SPACE=(800,(800,100))
//*
//*-------------------------------------------------------------------
//* Link together stub and program.
//*-------------------------------------------------------------------
//*
//LKED    EXEC PGM=HEWL,PARM='LIST,AMODE=31,RMODE=ANY,RENT,MAP',
//             COND=((9,LT,REXX),(0,NE,PLKED))
//*
```

```
//SYSLIN    DD DSN=&&SYSOUT,DISP=(OLD,DELETE)
//SYSLIB    DD DSN=&LIBDSN,DISP=SHR
//SYSUT1    DD UNIT=SYSDA,SPACE=(1024,(200,20))
//SYSPRINT  DD SYSOUT=*
//SYSLMOD   DD DSN=&&GOSET(GO),DISP=(MOD,PASS),UNIT=SYSDA,
//             SPACE=(1024,(50,20,1))
```

REXXCLG (FANCMCLG)

REXXCLG compiles, link-edits, and runs a REXX program of type OBJECT. FANCMCLG is located in the data set *prefix*.SFANPRC.

```
//*****************************************************************
//*
//* REXXCLG Compile, link edit, and run a REXX program of OBJ type.
//*
//*    Licensed Materials - Property of IBM
//*    5695-013 IBM REXX Compiler
//*    (C) Copyright IBM Corp. 1989, 2003
//*
//* Change Activity:
//*   03-05-28       Release 4.0
//*
//*****************************************************************
//*
//* Parameters:
//*
//*   OPTIONS     Compilation options.
//*               Default: XREF OBJECT NOCEXEC
//*
//*   COMPDSN     DSN of IBM REXX Compiler load library.
//*
//*   LIBDSN      DSN of IBM REXX Library load library for Stubs.
//*
//*   LIBLPA      DSN of IBM REXX Library LPA library.
//*               If &LIBLPA is in the search order, you may deactivate
//*               the GO.STEPLIB and the PROC LIBLPA definition.
//*
//*   LIBXDSN     DSN of IBM REXX Library exec library.
//*
//* Required:
//*
//*   REXX.SYSIN  DDNAME, REXX program to be compiled, link edited,
//*               and run.
//*
//* Example:
//*
//*   To compile MYREXX.EXEC(MYPROG), to link edit the resulting
//*   OBJECT output together with a stub suitable for invocation
//*   in MVS batch, to keep the resulting load module in
//*   MYREXX.LOAD(MYPROG), and to run this load module, use the
//*   following invocation:
//*
//*   //S1 EXEC REXXCLG
//*   //REXX.SYSIN   DD DSN=MYREXX.EXEC(MYPROG),DISP=SHR
//*   //LKED.SYSLMOD DD DSN=MYREXX.LOAD(MYPROG),DISP=SHR
//*
//* Modifications:
//*   Change #HLQREXX to the appropriate high-level qualifier of
//*   your installation.
//*
//*****************************************************************
//*
//REXXCLG PROC STUB=MVS,                      Type of stub
//            OPTIONS='XREF OBJECT NOCEXEC', REXX Compiler options
//            COMPDSN='#HLQREXX.SFANLMD',    REXX Compiler load lib
//            LIBDSN='#HLQREXX.SEAGLMD',     REXX Library stub load
```

```
//             LIBLPA='#HLQREXX.SEAGLPA',     REXX Library LPA lib
//             LIBXDSN='#HLQREXX.SEAGCMD'     REXX Library exec lib
//*
//*-------------------------------------------------------------------
//* Compile REXX program.
//*-------------------------------------------------------------------
//*
//REXX    EXEC PGM=REXXCOMP,PARM='&OPTIONS'
//STEPLIB   DD DSN=&COMPDSN,DISP=SHR
//SYSPRINT  DD SYSOUT=*
//SYSTERM   DD SYSOUT=*
//*SYSIEXEC DD DUMMY
//*SYSDUMP  DD DUMMY
//*SYSCEXEC DD DUMMY
//SYSPUNCH  DD DSN=&&OBJECT,DISP=(MOD,PASS),UNIT=SYSDA,
//             SPACE=(800,(800,100))
//*
//*-------------------------------------------------------------------
//* Prepare SYSLIN data set for subsequent link step.
//*-------------------------------------------------------------------
//*
//PLKED   EXEC PGM=IRXJCL,PARM='REXXL &STUB',
//             COND=(9,LT,REXX)
//*
//SYSEXEC   DD DSN=&LIBXDSN,DISP=SHR
//SYSIN     DD DSN=&&OBJECT,DISP=(OLD,DELETE)
//SYSTSPRT  DD SYSOUT=*
//SYSOUT    DD DSN=&&SYSOUT,DISP=(MOD,PASS),UNIT=SYSDA,
//             DCB=(RECFM=FB,LRECL=80,BLKSIZE=800),
//             SPACE=(800,(800,100))
//*
//*-------------------------------------------------------------------
//* Link together stub and program.
//*-------------------------------------------------------------------
//*
//LKED    EXEC PGM=HEWL,PARM='LIST,AMODE=31,RMODE=ANY,RENT,MAP',
//             COND=((9,LT,REXX),(0,NE,PLKED))
//*
//SYSLIN    DD DSN=&&SYSOUT,DISP=(OLD,DELETE)
//SYSLIB    DD DSN=&LIBDSN,DISP=SHR
//SYSUT1    DD UNIT=SYSDA,SPACE=(1024,(200,20))
//SYSPRINT  DD SYSOUT=*
//SYSLMOD   DD DSN=&&GOSET(GO),DISP=(MOD,PASS),UNIT=SYSDA,
//             SPACE=(1024,(50,20,1))
//*
//*-------------------------------------------------------------------
//* Run the compiled REXX program.
//*-------------------------------------------------------------------
//*
//GO      EXEC PGM=*.LKED.SYSLMOD,
//             COND=((9,LT,REXX),(0,NE,PLKED),(0,NE,LKED))
//*
//STEPLIB  DD DSN=&LIBLPA,DISP=SHR
//SYSTSPRT  DD SYSOUT=*
```

REXXOEC (FANCMOEC)

REXXOEC compiles source programs into CEXECs to run under MVS OpenEdition. FANCMOEC is located in the data set *prefix*.SFANPRC.

```
//*****************************************************************
//*
//* REXXOEC Compile a REXX program for OpenEdition MVS
//*
//*    Licensed Materials - Property of IBM
//*    5695-013 IBM REXX Compiler
//*    (C) Copyright IBM Corp. 1989, 2003
```

```
//*
//* Change Activity:
//*   03-05-28       Release 4.0
//*
//*****************************************************************
//*
//* Parameters:
//*
//*   OPTIONS     Compilation options.
//*               Default: XREF
//*
//*   COMPDSN     DSN of IBM REXX Compiler load library.
//*
//* Required:
//*
//*   REXX.SYSIN  DDNAME, REXX program to be compiled.
//*
//* Example:
//*
//*   To compile MYREXX.EXEC(MYPROG) and to keep the resulting
//*   CEXEC output in '/vienna/myprog' and the listing in
//*   '/vienna/myprogl' use the following invocation:
//*
//*   //STEP1    EXEC REXXOEC
//*   //REXX.SYSIN   DD DSN=MYREXX.EXEC(MYPROG),DISP=SHR
//*   //REXX.SYSPRINT DD DSN=&&LIST,DISP=(NEW,PASS),UNIT=SYSDA
//*   //OCOPY.OUT    DD PATH='/vienna/myprog',PATHDISP=(KEEP,DELETE),
//*   //             PATHOPTS=(ORDWR,OCREAT),PATHMODE=(SIRUSR,SIWUSR)
//*   //OCOPY.IN2    DD DSN=&&LIST,DISP=(OLD,DELETE)
//*   //OCOPY.OUT2   DD PATH='/vienna/myprogl',PATHDISP=(KEEP,DELETE),
//*   //             PATHOPTS=(ORDWR,OCREAT),PATHMODE=(SIRUSR,SIWUSR)
//*   //OCOPY.SYSTSIN DD *
//*    OCOPY INDD(IN) OUTDD(OUT) BINARY
//*    OCOPY INDD(IN2) OUTDD(OUT2)
//*   /*
//*
//* Modifications:
//*   Change #HLQREXX to the appropriate high-level qualifier of
//*   your installation.
//*
//*****************************************************************
//*
//REXXOEC PROC OPTIONS='XREF',                REXX Compiler options
//             COMPDSN='#HLQREXX.SFANLMD'     REXX Compiler load lib
//*
//*---------------------------------------------------------------
//* Compile REXX program
//*---------------------------------------------------------------
//*
//REXX    EXEC PGM=REXXCOMP,PARM='&OPTIONS'
//STEPLIB   DD DSN=&COMPDSN,DISP=SHR
//SYSPRINT  DD SYSOUT=*
//SYSTERM   DD SYSOUT=*
//*SYSIEXEC DD DUMMY
//*SYSDUMP  DD DUMMY
//SYSCEXEC  DD DSN=&&CEXEC,DISP=(MOD,PASS),UNIT=SYSDA,
//             SPACE=(800,(800,100))
//OCOPY   EXEC PGM=IKJEFT01,
//             COND=(9,LT,REXX)
//SYSTSPRT  DD SYSOUT=*
//SYSTSIN   DD DUMMY
//IN        DD DSN=&&CEXEC,DISP=(OLD,DELETE)
//OUT       DD DUMMY
```

REXXL (EAGL)

REXXL link-edits a REXX program of type OBJECT. EAGL is located in the data set *prefix*.SEAGPRC. "Link-Editing of Object Modules" on page 207 describes the following code.

```
//*****************************************************************
//*
//* REXXL   Link edit a REXX program of OBJ type.
//*
//*    Licensed Materials - Property of IBM
//*    5695-014 IBM REXX Library
//*    (C) Copyright IBM Corp. 1989, 2003
//*
//* Change Activity:
//*   03-05-28       Release 4.0
//*
//*****************************************************************
//*
//* Parameters:
//*
//*   STUB        Stub type: MVS, CPPL, CALLCMD, EFPL, CPPLEFPL.
//*               Default: EFPL.
//*
//*   LIBDSN      DSN of IBM REXX Library load library for Stubs.
//*
//*   LIBXDSN     DSN of IBM REXX Library exec library.
//*
//* Required:
//*
//*   PLKED.SYSIN DDNAME, REXX program of OBJ type to be link
//*               edited.
//*
//* Example:
//*
//*   To link MYREXX.OBJ(MYPROG), a compiled REXX program of OBJECT
//*   type, together with a stub suitable for invocation in MVS
//*   batch, and to place the resulting load module in
//*   MYREXX.LOAD(MYPROG), use the following invocation:
//*
//*   //S1 EXEC REXXL,STUB=MVS
//*   //PLKED.SYSIN  DD DSN=MYREXX.OBJ(MYPROG),DISP=SHR
//*   //LKED.SYSLMOD DD DSN=MYREXX.LOAD(MYPROG),DISP=SHR
//*
//* Modifications:
//*   Change #HLQREXX to the appropriate high-level qualifier of
//*   your installation.
//*
//*****************************************************************
//*
//REXXL   PROC STUB=EFPL,                     Type of stub
//             LIBDSN='#HLQREXX.SEAGLMD',     REXX Library stub load
//             LIBXDSN='#HLQREXX.SEAGCMD'     REXX Library exec lib
//*
//*---------------------------------------------------------------
//* Prepare SYSLIN data set for subsequent link step.
//*---------------------------------------------------------------
//*
//PLKED   EXEC PGM=IRXJCL,PARM='REXXL &STUB'
//*
//SYSEXEC   DD DSN=&LIBXDSN,DISP=SHR
//SYSTSPRT  DD SYSOUT=*
//SYSOUT    DD DSN=&&SYSOUT,DISP=(MOD,PASS),UNIT=SYSDA,
//             DCB=(RECFM=FB,LRECL=80,BLKSIZE=800),
//             SPACE=(800,(800,100))
//*
//*---------------------------------------------------------------
```

```
//* Link together stub and program.
//*-------------------------------------------------------------------
//*
//LKED    EXEC PGM=HEWL,PARM='LIST,AMODE=31,RMODE=ANY,RENT,MAP',
//             COND=(0,NE,PLKED)
//*
//SYSLIN    DD DSN=&&SYSOUT,DISP=(OLD,DELETE)
//SYSLIB    DD DSN=&LIBDSN,DISP=SHR
//SYSUT1    DD UNIT=SYSDA,SPACE=(1024,(200,20))
//SYSPRINT  DD SYSOUT=*
//SYSLMOD   DD DSN=&&GOSET(GO),DISP=(MOD,PASS),UNIT=SYSDA,
//             SPACE=(1024,(50,20,1))
```

MVS2OE (Only Hardcopy Sample)

The following REXX program is a simple example of an interactive procedure for copying a sequential data set, such as a CEXEC (compiled EXEC) to OpenEdition. This example is provided in hardcopy only.

```
/* ------------------------------ REXX ------------------------------ */
/*                               MVS2OE                                */
/*               Copy a z/OS data set to OpenEdition                   */
/*                                                                     */
/* MVS2OE:  This EXEC will copy a sequential data set or a member in  */
/* a library to OpenEdition (OE).  It will run in a TSO environment.  */
/* You may find it helpful when you copy a compiled REXX program      */
/* to OE for execution there.  However, it has intentionally been     */
/* kept simple but you can adapt it to your own purposes.  You can    */
/* improve plausibility checking, for example by using the sysdsn()   */
/* function to see if the data set to be copied exists and is         */
/* available.  You can read in the DSNAME from the invocation line    */
/* (with ARG or PARSE ARG) and only prompt the user if no arguments   */
/* have been given. For your convenience, debugging routines for      */
/* NOVALUE and SYNTAX have been included in case you do want to
/* modify this program.                                               */
/*                                                                    */
/* This exec uses 3 values: 1) the DSNAME of the sequential data set  */
/* to be written, 2) the path name under OE to be written to, 3) an   */
/* indication if the data set is binary (for example, a load module   */
/* or compiled exec, a CEXEC). These values are saved at the end of   */
/* this exec, the saved values are retrieved at the start of this     */
/* exec.                                                              */
/*                                                                    */
/* This exec is invoked by:                                           */
/*        EXEC  lib(MVS2OE)                                           */
/* from the TSO prompt, usually selection 6 from the ISPF primary     */
/* option menu, where 'lib' is the name of the library containing     */
/* this exec. If the name of the library does not start with the      */
/* prefix specified in your profile, you must enclose lib(MVS2OE)     */
/* within single quotes.  This exec does not expect any arguments     */
/* from the invocation line.                                          */
/**********************************************************************/

signal on novalue; signal on syntax

/* try to retrieve previous values                                    */
address ISPEXEC "VGET (OEDSN,OEPATH,OEBIN)"
if (rc = 0) then do                      /* vget o.k., confirm values */
  say 'MVS data set name';    oedsn  = check(oedsn)
  say 'OE path name';         oepath = check(oepath, 'lower')
  say 'Binary file (Y or N)'; oebin  = check(oebin)
end
else do                                 /* vget not o.k., read in values */
  say 'please key in the complete DSNAME with High Level Qualifier'
  pull oedsn
  say 'please key in the OE path'
```

```
  parse pull oepath
  say 'is it an executable (binary) program (Y or N)?'
  pull oebin
end

say 'Abort run? "Y" aborts, anything else performs copy'
say 'from' oedsn 'to' oepath
pull answer
if (answer = 'Y') then exit

if (oebin = 'Y') then DO   /* set up some of the file's OE attributes */
  mode = 'SIXUSR'
  bin = 'BINARY'
end
else do
  mode = ''
  bin = 'TEXT'
end

msg_status = msg('OFF')               /* suppress msgs from FREE etc. */
"FREE  DDNAME(OEIN)"             /* make sure OEIN and OEOUT are free */
"FREE  DDNAME(OEOUT)"
msg_status = msg(msg_status)             /* restore to previous value */

"ALLOC DDNAME(OEIN)  DSN('"oedsn"') SHR"
"ALLOC DDNAME(OEOUT) PATH('"oepath"') PATHDISP(KEEP KEEP)" ,
              "PATHOPTS(ORDWR OCREAT) PATHMODE(SIRUSR SIWUSR" mode")"

"OCOPY INDD(OEIN) OUTDD(OEOUT)" bin         /* perform copy operation */
if (rc <> 0) then say 'RC from OCOPY=' rc        /* check return code */
"FREE  DDNAME(OEIN)"
"FREE  DDNAME(OEOUT)"

/* save values for next invocation                                    */
address ISPEXEC "VPUT (OEDSN,OEPATH,OEBIN) PROFILE"
exit 0                                          /* leave this exec */

/* subprogram to request user to confirm or overwrite a value         */
/* -------------------------------------------------------------      */
check:
 say 'Use <ENTER> to use' arg(1) 'or key in new value'
 if (arg(2) = 'lower') then do
   parse pull answer                        /* keep case as typed in */
 end
 else do
   parse upper pull answer                        /* uppercase input */
 end
 if (answer = '') then return arg(1); else return answer

/* Debugging routines for NOVALUE and SYNTAX                          */
/* ---------------------------------------                            */
novalue: say ' '
say '    Novalue condition from line' sigl
say sourceline(sigl)
say '    variable:' condition('D'); trace ?r; nop; exit

syntax: say ' '
say '    Syntax error no.' rc 'from line' sigl
say '   'errortext(rc)
say sourceline(sigl)
say '    description:'condition('D'); trace ?r ; nop
```

Appendix E. The VSE/ESA Cataloged Procedures Supplied by IBM

This appendix contains the following cataloged procedures supplied by IBM:

REXXPLNK

For more information about the REXXPLNK format refer to "REXXPLNK Cataloged Procedure (VSE/ESA)" on page 78.

```
// PROC STUBLIB='PRD1.BASE',STUBNAM='EFPL'
 GOTO SKIPCOM
 * ********************************************************************
 *
 * REXXPLNK Combine a program of OBJ type with the appropriate stub.
 *
 *      Licensed Materials - Property of IBM
 *      5695-014 IBM REXX Library
 *      (C) Copyright IBM Corp. 1989, 2003
 *
 *   Change Activity:
 *     03-05-28       Release 4.0
 *
 * ********************************************************************
 *
 * Parameters:
 *
 *   STUBLIB      is the name of the sublibrary where the stub resides.
 *                Default: PRD1.BASE
 *
 *   STUBNAM      is the member name of the stub residing in STUBLIB
 *                or one of the predefined stub names: VSE, EFPL.
 *                Default: EFPL
 *
 *   INLIB        is the name of the sublibrary where the input object
 *                module resides.
 *
 *   INNAME       is the member name of the input object module
 *                residing in INLIB.
 *
 *   OUTLIB       is the name of the sublibrary where the output object
 *                module will be stored.
 *
 *   OUTNAME      is the member name of the output object module that
 *                will be stored in OUTLIB.
 *
 *
 * Example:
 *
 *   To combine the program MYAPPL.OBJ residing in the sublibrary
 *   MYLIB.TEST with the EFPL stub, which is appropriate if the
 *   program will be invoked as a REXX external routine, and to
 *   store the resulting object module under the name CMYAPPL.OBJ
 *   residing in the sublibrary MYLIB.TEST, use the following
 *   invocation:
 *
 *  // EXEC PROC=REXXPLNK,INLIB='MYLIB.TEST',INNAME=MYAPPL,
```

```
*                OUTLIB='MYLIB.TEST',OUTNAME=CMYAPPL
*
* ********************************************************************
*
/. SKIPCOM
// EXEC REXX=REXXL,PARM='&STUBLIB &STUBNAM &INLIB &INNAME &OUTLIB &OUTNC
              AME'
/+
```

REXXLINK

For more information about the REXXLINK format refer to "REXXLINK Cataloged Procedure (VSE/ESA)" on page 79.

```
// PROC STUBLIB='PRD1.BASE',STUBNAM='EFPL',PHASNAM=''
 GOTO SKIPCOM
 * ********************************************************************
 *
 * REXXLINK Link-edit a program of OBJ type and catalog the resulting
 *          phase in a VSE/ESA library.
 *
 *       Licensed Materials - Property of IBM
 *       5695-014 IBM REXX Library
 *       (C) Copyright IBM Corp. 1989, 2003
 *
 *    Change Activity:
 *      03-05-28       Release 4.0
 *
 * ********************************************************************
 *
 * Parameters:
 *
 *   STUBLIB      is the name of the sublibrary where the stub resides.
 *                Default: PRD1.BASE
 *
 *   STUBNAM      is the member name of the stub residing in STUBLIB
 *                or one of the predefined stub names: VSE, EFPL.
 *                Default: EFPL
 *
 *   INLIB        is the name of the sublibrary where the input object
 *                module resides.
 *
 *   INNAME       is the member name of the input object module
 *                residing in INLIB.
 *
 *   OUTLIB       is the name of the sublibrary where the output object
 *                module will be stored.
 *
 *   OUTNAME      is the member name of the output object module that
 *                will be stored in OUTLIB.
 *
 *   PHASNAM      is the member name of the phase that will be
 *                cataloged in the sublibrary specified by a
 *                LIBDEF PHASE,CATALOG=lib.sublib statement.
 *                Default: OUTNAME
 *
 *
 * Example:
 *
 *   To link-edit the program MYAPPL.OBJ residing in the sublibrary
 *   MYLIB.TEST with the VSE stub, which is appropriate if the program
 *   will be invoked as a VSE program, and to catalog the resulting
 *   phase under the name MYAPPL.PHASE in the sublibrary MYLIB.TEST,
 *   you have to specify as well the name of the resulting object
 *   module serving as input for the linkage editor: for example
 *   CMYAPPL.OBJ in the sublibrary MYLIB.TEST.
 *   To perform this task use the following invocation:
```

```
*
*  // LIBDEF PHASE,CATALOG=MYLIB.TEST
*  // EXEC PROC=REXXLINK,STUBNAM=VSE,INLIB='MYLIB.TEST',INNAME=MYAPPL,
*                OUTLIB='MYLIB.TEST',OUTNAME=CMYAPPL,PHASNAM=MYAPPL
*
* ********************************************************************
*
/. SKIPCOM
IF PHASNAM='' THEN
// SETPARM PHASNAM=&OUTNAME
// EXEC REXX=REXXL,PARM='&STUBLIB &STUBNAM &INLIB &INNAME &OUTLIB &OUTNC
             AME'
IF $RC NE 0 THEN
GOTO $EOJ
// LIBDEF OBJ,SEARCH=&OUTLIB
// OPTION CATAL
   PHASE &PHASNAM,*,SVA
   INCLUDE &OUTNAME
// EXEC LNKEDT
/+
```

REXXL

For more information about the REXXL format refer to "REXXL Cataloged Procedure (VSE/ESA)" on page 80.

```
/*REXX ************************************************************
*
* REXXL - Combine a stub with the input object module and build an
*         object module which serves as input for the linkage editor.
*
*      Licensed Materials - Property of IBM
*      5695-014 IBM REXX Library
*      (C) Copyright IBM Corp. 1989, 2003
*
*   Change Activity:
*     03-05-28       Release 4.0
*
*********************************************************************/
/*********************************************************************
* Syntax:
*
* // EXEC REXX=REXXL,PARM='stublib stubnam inlib inname outlib outname'
*  or
* Call rexxl 'stublib stubnam inlib inname outlib outname'
*
* where:
* stublib   Is the name of the sublibrary where the stub resides in the
*           form lib.sublib.
*
* stubnam   Is the member name of the stub residing in stublib in the
*           form mn. Member type is always OBJ. You can also use one of
*           the predefined stub names:
*           VSE   The program will be invoked by VSE JCL as a program.
*           EFPL  The program will be invoked as a REXX external
*                 routine.
*
* inlib     Is the name of the sublibrary where the input object module
*           resides in the form lib.sublib.
*
* inname    Is the member name of the input object module residing in
*           inlib in the form mn. Member type is always OBJ.
*
* outlib    Is the name of the sublibrary where the output object
*           module will be stored in the form lib.sublib.
*
* outname   Is the member name of the output object module that will be
```

```
*           stored in outlib in the form mn. Member type is always OBJ.
*
* Semant:
*
* The REXXL EXEC builds as output an object module that contains the
* stub combined with the input object module. The resulting object
* module can be link-edited with other object modules to create a
* phase.
*
*********************************************************************/

/*
** Initialize all global variables, catch variables used before set,
** catch syntax errors and handle Attention / HI
*/

Signal On Novalue
Signal On Syntax
Signal On Halt
g. = ''

/*
** Customization Section:
**
**    stubmembs  Member names of predefined stubs.
**    stubnames  Name under which the stub is known to REXXL,
**               i.e. the stub name passed as argument to REXXL.
*/

stubmembs='EAGSDVSE EAGSDEFP'
stubnames='VSE   EFPL'

/*
** Messages issued by this procedure
*/

g.0m.1 = 'Unexpected rc=&1 from "&2"'
g.0m.3 = 'One or more parameters missing'
g.0m.4 = 'Extraneous parameters "&1"'
g.0m.7 = 'The &1 "&2" specified for parameter &3 is longer than'!!,
' 8 characters'
g.0m.8 = 'No sublibrary specified for parameter &1: "&2"'
g.0m.9 = 'The &1 "&2" specified for parameter &3 is longer than 7'!!,
' characters'
g.0m.11 = 'The &1 "&2" specified for parameter &3 consists of'!!,
' non-alphanumeric characters'
g.0m.12 = 'The first character of the library name "&1" specified'!!,
' for parameter &2 is not alphabetic'
g.0m.13 = 'Invoke REXXL in the following format:'
g.0m.14 = "// EXEC REXX=REXXL,PARM='stublib stubnam inlib inname"!!,
" outlib outname' or"
g.0m.15 = "CALL REXXL 'stublib stubnam inlib inname outlib outname'"

/*
** Get the arguments and give help if necessary.
** If a predefined stubname is used, assign the appropriate member name
** to the variable stubnam.
*/

Parse Upper Arg stublib stubnam inlib inname outlib outname rest
If (stublib = '') ! (outname = '') Then Do
  Call msg 3
  Call msg 13
  Call msg 14
  Call msg 15
  Exit 20
  End
```

```
ind = Wordpos(stubnam,stubnames)
If ind > 0 Then
  stubnam = Word(stubmembs,ind)

/*
** Check the arguments.
** If errors occurred, issue a message and exit with rc=20.
*/

message_written = 'no'
If rest <> '' Then
  Call msg 4,rest
Call testsyn 'STUB',stublib,stubnam
Call testsyn 'IN',inlib,inname
Call testsyn 'OUT',outlib,outname
If message_written = 'yes' Then
  Exit 20

/*
** Set the full names for the members stubnam, inname and outname
** in the variables stub, input and output.
*/

stub = stublib'.'stubnam'.OBJ'
input = inlib'.'inname'.OBJ'
output = outlib'.'outname'.OBJ'

/*
** Read from stub and input.
** Some variables are set:
** inline1 will contain the contents of the stub in object module form
** inline2 will contain the contents of the input object module
*/

cmd = 'EXECIO * DISKR' stub '(STEM INLINE1. FINIS'
Call excmd cmd
cmd = 'EXECIO * DISKR' input '(STEM INLINE2. FINIS'
Call excmd cmd

/*
** Prepare the input for the linkage editor in the variable inline1:
** Get the csect name, that is the name of the object module in the ESD
** record. For the stub and the input object module make the following
** changes: Change the external name of the stub to the csect name, and
** the name of the compiled REXX program to a temporary name, which is
** the csect name, prefixed with a '$' sign, and turncated to eight
** characters. Append the contents of the input object module to the
** variable inline1.
** Finally the variable inline1 contains the stub combined with the
** compiled REXX program, ready to be link-edited.
*/

csect = Substr(inline2.1,17,8)
unin='$'Left(csect,7)
inline1.1 = Left(inline1.1,16)!!csect!!Substr(inline1.1,25,8)!!,
            unin!!Right(inline1.1,40)
index = inline1.0 + 1
inline1.index = Left(inline2.1,16)!!unin!!Substr(inline2.1,25)
Do i = 2 To inline2.0
  index = index + 1
  inline1.index = inline2.i
  End

/*
** Write the input for the linkage editor to the member outname
** and exit with rc=0, if no errors occurred.
*/
```

```
cmd = 'EXECIO' index 'DISKW' output '(STEM INLINE1. FINIS'
Call excmd cmd
Exit 0

excmd:Procedure Expose g. inline1. inline2.
/*
** Execute an EXECIO command
** and exit with rc=12, if an error occurred.
** arg: the EXECIO command
** set: message_written to 'yes', if an error occurred
** ref: inline1 contains data that are read or are written
**      inline2 contains data that are read
** out: result of the executed command
*/

Arg cmd
cmd
cmdrc = rc
If cmdrc <> 0 Then Do
  Call msg 1,cmdrc,cmd
  Exit 12
  End
Return

testsyn:Procedure Expose g. message_written
/*
** Test the syntax of the arguments.
** arg: the type of the argument; it can be STUB, IN or OUT
**      the library name in the form lib.sublib
**      the member name
** set: message_written to 'yes', if an error occurred
*/

Arg type,lib,mn
Parse Var lib lib '.' sublib

/*
** Test the library name: 1-7 characters, alphanumeric, the first
** character has to be alphabetic.
*/

If Length(lib) > 7 Then
  Call msg 9,'library name',lib,type'LIB'
If Datatype(lib,'A') <> 1 Then
  Call msg 11,'library name',lib,type'LIB'
If Datatype(Left(lib,1),'U') <> 1 Then
  Call msg 12,lib,type'LIB'

/*
** Test the sublibrary name: 1-8 characters, alphanumeric.
** Test it only if lib is specified.
*/

If sublib = '' Then
  Call msg 8,type'LIB',lib
Else Do
  If Length(sublib) > 8 Then
    Call msg 7,'sublibrary name',sublib,type'LIB'
  If Datatype(sublib,'A') <> 1 Then
    Call msg 11,'sublibrary name',sublib,type'LIB'
  End

/*
** Test the member name: 1-8, alphanumeric.
*/
```

```
If Length(mn) > 8 Then
  Call msg 7,'member name',mn,Left(type'NAME',7)
If Datatype(mn,'A') <> 1 Then
  Call msg 11,'member name',mn,Left(type'NAME',7)

Return

msg:Procedure Expose g. message_written
/*
** Issue a message (after substituting the inserts).
** arg: the message number
**      the message inserts
** set: message_written is set to 'yes'
** out: display the message with inserts on the terminal
*/

Parse Arg msgnum,i1,i2,i3
msgtxt = g.0m.msgnum
Do ii = 1 To 3
  mi = '&'ii
  If Pos(mi,msgtxt) > 0 Then Do
    Parse Var msgtxt ma (mi) mb
    msgtxt = ma!!Value('I'ii)!!mb
    End
  End
Say msgtxt
message_written = 'yes'
Return

halt:
/*
** HALT condition was raised.
** arg: none
** out: display message
*/

Say 'REXXL has been halted'
Exit 20

syntax:
/*
** SYNTAX condition was raised.
** arg: none
** out: display invalid line
*/

zsigl = sigl
Say 'Syntax error' rc 'at line' zsigl':' Errortext(rc)
If Sourceline() <> 0 Then
  Say 'Line' zsigl':' Sourceline(zsigl)
Exit 20

novalue:
/*
** NOVALUE condition was raised, undefined variable referenced.
** arg: none
** out: display invalid line
*/

zsigl = sigl
Say 'Undefined variable referenced:' Condition('D')
If Sourceline() <> 0 Then
  Say 'Line' zsigl':' Sourceline(zsigl)
Exit 20
```

Appendix F. Interlanguage Job Samples

This appendix contains interlanguage job samples that show how to call a REXX exec from within another program written in a different programming language, such as Assembler, C, Cobol, or PL/I under z/OS. It provides sample jobs that show how to pass parameters back and forth between REXX programs and an Assembler, C, Cobol, or PL/I program.

You must use the IRXJCL and IRXEXEC interfaces to call the REXX program from an Assembler, C, Cobol, or PL/I program. These interfaces are available in any address space and are applicable to interpreted and compiled REXX execs.

Note: Under z/OS these samples are located in the data set *prefix*.SEAGSAM.

The following jobs show the use of interpreted programs. Each job consists of:

- A step which copies the REXX exec as a member into a temporary partitioned data set.
- An invocation of a cataloged procedure to compile, link, and execute the Assembler, C, Cobol, or PL/I program for calling the REXX exec. The execute step is assigned additional DD statements that are required by the REXX exec.

The IRXJCL interface is easier to use, but it is not as flexible as the IRXEXEC interface. If you use the IRXJCL:

- You can only set the parameters to call IRXJCL.
- You can only pass one argument string to the REXX program, however, this string can contain any number of tokens or words.
- The REXX exec can only return a numeric return code, limited to a maximum number of 4095 (decimal).

If you use the IRXEXEC interface:

- You must set up the EXEC block. the evaluation block, and the arguments to IRXEXEC. The EXEC block is a control block that describes the REXX exec to be loaded.
- You can use multiple argument strings. Each argument string can consist of multiple tokens or words.
- The REXX exec can return to its caller, by using the RETURN or EXIT clause, which is a variable length character string that consists of any number of tokens or words. This character string and its length is returned in a control block called the evaluation block.

  Note: In the following example the character string is restricted to a maximum length of 256 bytes.

For more information on IRXJCL and IRXEXEC routines, and the control blocks used in the examples, refer to *z/OS TSO/E REXX Reference*.

Calling REXX from Assembler

The following samples show how to call a REXX exec from an Assembler program under z/OS.

EAGGJASM for Calling IRXJCL

This is the EAGGJASM sample for calling IRXJCL from an Assembler program:

```
//* -->uidIEC JOB - Specify your Job card here
//*-------------------------------------------------------------*
//*
//*    Interlanguage Communication in z/OS
//*       Calling REXX from ASSEMBLER
//*
//*-------------------------------------------------------------*
//*
//*  Licensed Materials - Property of IBM
//*  5695-013
//*  (C) Copyright IBM Corp. 1989, 2003
//*
//*-------------------------------------------------------------*
//* Sample JCL for calling IRXJCL from an Assembler program.
//* For a description also refer to the REXX Compiler guide
//* SH19-8160.
//* You may modify this sample for your needs by including
//* a REXX of your own. The ARGUMENT for the REXX procedure
//* may be taylored for your needs.
//*
//* Change Activity: 030708 - new for Release 4
//*-------------------------------------------------------------*
//* JCLLIB accesses the CLG procs, modify to your needs
//*-------------------------------------------------------------*
//*MYLIB   JCLLIB ORDER=RXT.INTLANG.CNTL
//*-------------------------------------------------------------*
//* Create REXX procedure HELLO into a temporary Library &&REXX
//* SYSUT1 is setup for card input, modify to a DSN if desired
//*-------------------------------------------------------------*
//CREATE  EXEC PGM=IEBGENER
//SYSPRINT DD SYSOUT=*
//SYSIN    DD DUMMY
//SYSUT2   DD DSN=&&REXX(HELLO),DISP=(NEW,PASS),
//            UNIT=SYSDA,SPACE=(TRK,(1,1,1)),
//            DCB=(RECFM=FB,LRECL=80,BLKSIZE=800,DSORG=PO)
//SYSUT1   DD *,DLM=$$
/* REXX - This simple REXX EXEC is called by an Assembler */
Parse source src
Arg n
  Say 'I am' src
  Say 'Received parm:' n
  If n='' Then n=1
  Do i=1 TO n
    Say 'Hello World the' i'. time ...'
  End
Return n                              /* Set Return Code to n */
$$
//*-------------------------------------------------------------*
//*
//*  Compile, link and execute an Assembler program.
//*  The cataloged procedure below is for HLASM.
//*
//*-------------------------------------------------------------*
//ASMACLG EXEC ASMACLG
//C.SYSIN DD *
*-------------------------------------------------------------*
* Invoke REXX procedure HELLO with TSO IRXJCL
* HELLO is to be called with number 3 as a parameter
*-------------------------------------------------------------*
ASMPROG  CSECT
*        RMODE 24 is because the DCB must be below 16M
ASMPROG  RMODE 24
ASMPROG  AMODE 31
                                                        SPACE 1
```

```
*-------------------------------------------------------------*
* Standard starting (housholding) sequence
*-------------------------------------------------------------*
         SAVE  (14,12)
         BASR  11,0                establish ...
         USING *,11                ...addressability
         ST    13,SA+4             chain...
         LR    12,13               ...the...
         LA    13,SA                  ...save...
         ST    13,8(12)                  ...areas
                                                              SPACE 1
*-------------------------------------------------------------*
* Put out starting message
*-------------------------------------------------------------*
         OPEN  (ASMOUT,(OUTPUT))
         PUT   ASMOUT,MESSAGE1
                                                              SPACE 1
*-------------------------------------------------------------*
* Set up input parameters, in our case: HELLO 3
*-------------------------------------------------------------*
         LA    3,L'ARGUMENT        get length of argument...
         STH   3,ARGLEN            ...bring to halfword
                                                              SPACE 1
*-------------------------------------------------------------*
* Invoke IRXJCL
*-------------------------------------------------------------*
         LINK  EP=IRXJCL,PARAM=ARGLEN
         LR    5,15                save return code for later
                                                              SPACE 1
*-------------------------------------------------------------*
* Print out the returncode
*-------------------------------------------------------------*
         CVD   5,DWORD             start editing return code
         ED    DIGITS,DWORD+4      last 7 digits from return code
         PUT   ASMOUT,MESSAGE2
         CLOSE (ASMOUT)
                                                              SPACE 1
*-------------------------------------------------------------*
* Standard exiting sequence
*-------------------------------------------------------------*
         L     13,SA+4
         LR    15,5                return code to reg 15 again
         RETURN (14,12),RC=(15)
                                                              SPACE 1
*-------------------------------------------------------------*
* Definiton area.
*-------------------------------------------------------------*
         DS    0F                  align on fullword
ARGLEN   DC    H'0'                length of exec name+argument...
*-------------------------------------------------------------*
* You may try different arguments for REXX ------------------*
*-------------------------------------------------------------*
ARGUMENT DC    C'HELLO  3'         ... string for REXX
                                                              SPACE 1
DWORD    DC    D'0'                doubleword scratch area
                                                              SPACE 1
MESSAGE1 DC    CL40'Starting Assembler prog'
                                                              SPACE 1
MESSAGE2 DC    CL40' '
         ORG   MESSAGE2            redefine the message       SPACE 1
         DC    C'Return code from REXX ='
*        stencil for ED instruction to print out a numeric value
DIGITS   DC    X'4020202020202120'
         DC    C' (dec.)'
         ORG   ,                   return to high water (address) mark
                                                              SPACE 1
SA       DC    18F'0'              save area
```

```
                                                                SPACE 1
ASMOUT   DCB   DDNAME=ASMOUT,RECFM=FB,LRECL=40,MACRF=PM,DSORG=PS
         END
/*
//*-------------------------------------------------------------*
//* Define the library containing the REXX exec
//*-------------------------------------------------------------*
//G.SYSEXEC DD DISP=(SHR,PASS),DSN=&&REXX
//*-------------------------------------------------------------*
//* Next DD is the data set equivalent to terminal input
//*-------------------------------------------------------------*
//G.SYSTSIN  DD    DUMMY
//*-------------------------------------------------------------*
//* Next DD is the data set equivalent to terminal output
//*-------------------------------------------------------------*
//G.SYSTSPRT DD    SYSOUT=*
//*-------------------------------------------------------------*
//* Next DD is for the Assembler program's output
//*-------------------------------------------------------------*
//G.ASMOUT   DD    SYSOUT=*
//*-------------------------------------------------------------*
//
```

EAGGXASM for Calling IRXEXEC

This is the EAGGXASM sample for calling IRXEXEC from an Assembler program:

```
//* -->uidIEC JOB - Specify your Job card here
//*-------------------------------------------------------------*
//*
//*    Interlanguage Communication in z/OS
//*       Calling REXX from ASSEMBLER
//*
//*-------------------------------------------------------------*
//*
//*  Licensed Materials - Property of IBM
//*  5695-013
//*  (C) Copyright IBM Corp. 1989, 2003
//*
//*-------------------------------------------------------------*
//* Sample JCL for calling IRXEXEC from an Assembler program.
//* For a description also refer to the REXX Compiler guide
//* SH19-8160.
//* You may modify this sample for your needs by including
//* a REXX of your own. The ARGUMENT for the REXX procedure
//* may be taylored for your needs.
//* The use of IRXEXEC is more complex than the use of IRXJCL.
//* Refer to the TSO guide SC28-1883 for using these services.
//*
//* Change Activity: 030708 - new for Release 4
//*-------------------------------------------------------------*
//* JCLLIB accesses the CLG procs, modify to your needs
//*-------------------------------------------------------------*
//*MYLIB   JCLLIB ORDER=RXT.INTLANG.CNTL
//*-------------------------------------------------------------*
//* Create REXX procedure HELLO into a temporary Library &&REXX
//* SYSUT1 is setup for card input, modify to a DSN if desired
//*-------------------------------------------------------------*
//CREATE  EXEC PGM=IEBGENER
//SYSPRINT DD SYSOUT=*
//SYSIN    DD DUMMY
//SYSUT2   DD DSN=&&REXX(HELLO),DISP=(NEW,PASS),
//            UNIT=SYSDA,SPACE=(TRK,(1,1,1)),
//            DCB=(RECFM=FB,LRECL=80,BLKSIZE=800,DSORG=PO)
//SYSUT1   DD *,DLM=$$
/* REXX - This simple REXX EXEC is called by an Assembler */
Parse source src
Arg n
```

```
  Say 'I am' src
  Say 'Received parm:' n
  If n='' Then n=1
  Do i=1 TO n
    Say 'Hello World the' i'. time ...'
  End
Return n                            /* Set Return Code to n */
$$
//*-------------------------------------------------------------*
//*
//*  Compile, link and execute an Assembler program.
//*  The cataloged procedure below is for HLASM.
//*
//*-------------------------------------------------------------*
//ASMACLG EXEC ASMACLG
//C.SYSIN DD *
                                                                SPACE 1
*-------------------------------------------------------------*
* Invoke REXX procedure HELLO with parameter 3 using IRXEXEC.
* Note that use is made of two mapping macros, so not all the
* symbolic addresses used here are defined in this source. You
* should assemble this program for clarity.
* Procedure HELLO is to be called with number 3 as a parameter
*-------------------------------------------------------------*
ASMPROG  CSECT
*        RMODE 24 is because the DCB must be below 16M
ASMPROG  RMODE 24
ASMPROG  AMODE 31
                                                                SPACE 1
*-------------------------------------------------------------*
* Standard starting (housholding) sequence
*-------------------------------------------------------------*
         SAVE  (14,12)
         BASR  11,0                establish ...
         USING *,11                ...addressability
         ST    13,SA+4             chain...
         LR    12,13               ...the...
         LA    13,SA                  ...save...
         ST    13,8(12)                  ...areas
                                                                SPACE 1
*-------------------------------------------------------------*
* Put out starting message
*-------------------------------------------------------------*
         OPEN  (ASMOUT,(OUTPUT))
         PUT   ASMOUT,MESSAGE1
                                                                SPACE 1
*-------------------------------------------------------------*
* Mainline section
* Set up input parameters and call IRXEXEC,
* Print out the return code from REXX exec
*-------------------------------------------------------------*
* Make EXECBLOCK addressable, fill in fields
* The setup here is for calling member HELLO.
* You may use either a member name or a DD name
* Be careful when modifying EXECBLOCK
*-------------------------------------------------------------*
         LA    4,EXECBLK#          point to storage
         USING EXECBLK,4
         MVC   EXEC_BLK_ACRYN,EXECB_ID  move acronym (identifier)
         LA    5,EXECBLEN          length of block in bytes...
         ST    5,EXEC_BLK_LENGTH   ...to length field
         MVC   EXEC_MEMBER,=CL8'HELLO' exec (member) name
         MVC   EXEC_DDNAME,=CL8' '  blank out ddname
         MVC   EXEC_SUBCOM,=CL8' '  blank out subcom field
         SR    5,5                 clear to zeroes, bring to...
         ST    5,EXEC_BLK_LENGTH+4  ...reserved field
         ST    5,EXEC_DSNPTR       ...pointer to DSN
```

```
         ST    5,EXEC_DSNLEN        ...length of DSN field
         DROP  4                    do not need base reg any more
                                                                SPACE 1
*-------------------------------------------------------------*
* EVALBLOCK already addressable, fill in fields
*-------------------------------------------------------------*
         ST    5,EVALBLOCK_EVPAD1   reg5 is still zero, bring...
         ST    5,EVALBLOCK_EVPAD2   ...zeros to padding fields
         LA    5,EVALBLEN           length of block in bytes...
         SRA   5,3                  ...now in double words
         ST    5,EVALBLOCK_EVSIZE   size in double words to evalblock
                                                                SPACE 1
*-------------------------------------------------------------*
* Invoke the IRXEXEC service routine
* This setup is fairly basic,  you probably do not need
* to change it
*-------------------------------------------------------------*
         LINK  EP=IRXEXEC,                                      *
               VL=1,                    variable length switch *
               PARAM=(EXECBLK_PTR,                              *
               ARGTABLE_PTR,                                    *
               FLAGS,                                           *
               INSTBLK_PTR,                                     *
               RES_PARM5,                                       *
               EVALBLK_PTR,                                     *
               RESERVED_WORKAREA_PTR,                           *
               RESERVED_USERFIELD_PTR,                          *
               RESERVED_ENVBLOCK_PTR,                           *
               REXX_RETURN_CODE_PTR)
                                                                SPACE 1
*-------------------------------------------------------------*
* Put out the results, i.e. return code and REXX string
* in reserved REXX variable result
*        (contents of REXX_RETURN_CODE are the same as in reg 15)
*-------------------------------------------------------------*
         CVD   15,DWORD             start editing return code
         ED    DIGITS,DWORD+4       last 7 digits from return code
         PUT   ASMOUT,MESSAGE2      put out message
         PUT   ASMOUT,MESSAGE3      put out another message
         L     15,EVALBLOCK_EVLEN   get length of string from REXX
         LTR   15,15                check if length is 0
         BZ    NORESULT             nothing returned from REXX
         LA    6,L'MESSAGE4         check length of message area...
         CR    15,6                 ...against length of REXX result
         BNH   LENOK                branch if result fits in area
         LR    15,6                 result too long, truncate
LENOK    EQU   *                    when here, string fits in area
         BCTR  15,0                 reduce length by one for MVC
         EX    15,MOVEIT            move result to message area
         PUT   ASMOUT,MESSAGE4      put out string returned by REXX
NORESULT EQU   *
         PUT   ASMOUT,MESSAGE5      put out closing message

         CLOSE (ASMOUT)
                                                                SPACE 1
*-------------------------------------------------------------*
* Standard exiting sequence, pass on REXX retcode
*-------------------------------------------------------------*
         L     13,SA+4
         L     15,REXX_RETURN_CODE  REXX return code to reg 15
         RETURN (14,12),RC=(15)
                                                                SPACE 1
*-------------------------------------------------------------*
* Data areas
*-------------------------------------------------------------*
DWORD    DC    D'0'                 doubleword scratch area
                                                                SPACE 1
```

```
MESSAGE1 DC    CL40'Starting Assembler prog'
                                                                SPACE 1
MESSAGE2 DC    CL40' '              space for next message
         ORG   MESSAGE2             redefine the message
         DC    C'Ret code from IRXEXEC ='
*        Stencil for ED instruction to print out a numeric value
DIGITS   DC    X'4020202020202120'
         DC    C' (dec.)'
         ORG   ,                    return to high water (address) mark
                                                                SPACE 1
MESSAGE3 DC    CL40'Result from REXX exec in next line:'
MESSAGE4 DC    CL40' '              work area for REXX output string
MESSAGE5 DC    CL40'End of Assembler prog'
                                                                SPACE 1
SA       DC    18F'0'               save area
                                                                SPACE 1
*-------------------------------------------------------------*
* Map the EXEC block, define storage for it, init to blanks
* Note that the IRXECEXB macro contains a DSECT.
*-------------------------------------------------------------*
         IRXEXECB
*        revert to CSECT as IRXEXECB contains a DSECT
ASMPROG  CSECT
EXECBLK# DC    CL(EXECBLEN)' '
                                                                SPACE 1
*-------------------------------------------------------------*
* Define storage for the EVAL block, map it.  Note that
* this invocation of IRXEVALB does not contain a DSECT
* because of DECLARE=YES.
*-------------------------------------------------------------*
EVALBLK# DC    5F'0'
         DC    CL256' '             for result from REXX exec
EVALBLEN EQU   *-EVALBLK#           length of block
         ORG   EVALBLK#             go back to map it
         IRXEVALB DECLARE=YES       map the EVAL block
         ORG      ,                 back to high water mark
                                                                SPACE 1
*-------------------------------------------------------------*
* The parameters to be passed to IRXEXEC
*-------------------------------------------------------------*
EXECBLK_PTR  DC A(EXECBLK#)          pointer to EXECBLOCK
ARGTABLE_PTR DC A(ARGTABLE)         pointer to table of arguments
FLAGS        DC   X'40000000'
INSTBLK_PTR  DC A(0)
RES_PARM5    DC A(0)
EVALBLK_PTR  DC A(EVALBLK#)
RESERVED_WORKAREA_PTR  DC A(0)
RESERVED_USERFIELD_PTR DC A(0)
RESERVED_ENVBLOCK_PTR  DC A(0)
REXX_RETURN_CODE_PTR   DC A(REXX_RETURN_CODE)
                                                                SPACE 1
REXX_RETURN_CODE       DC A(0)
                                                                SPACE 1
*-------------------------------------------------------------*
* The REXX argument string
*-------------------------------------------------------------*
ARG1     DC    C'3'                 argument for REXX exec
                                                                SPACE 1
*-------------------------------------------------------------*
ARGTABLE         DS  0F             align
ARGSTRING_PTR    DC  A(ARG1)        pointer to first argument
ARGSTRING_LENGTH DC A(L'ARG1)       length of first argument
ARGTABLE_LAST    DC XL8'FFFFFFFFFFFFFFFF'  end of argument fence
                                                                SPACE 1
MOVEIT   MVC   MESSAGE4,EVALBLOCK_EVDATA
                                                                SPACE 1
ASMOUT   DCB   DDNAME=ASMOUT,RECFM=FB,LRECL=40,MACRF=PM,DSORG=PS
```

```
        END
/*
//*-------------------------------------------------------------*
//* Define the library containing the REXX exec
//*-------------------------------------------------------------*
//G.SYSEXEC DD DISP=(SHR,PASS),DSN=&&REXX
//*-------------------------------------------------------------*
//* Next DD is the data set equivalent to terminal input
//*-------------------------------------------------------------*
//G.SYSTSIN  DD    DUMMY
//*-------------------------------------------------------------*
//* Next DD is the data set equivalent to terminal output
//*-------------------------------------------------------------*
//G.SYSTSPRT DD    SYSOUT=*
//*-------------------------------------------------------------*
//* Next DD is for the Assembler program's output
//*-------------------------------------------------------------*
//G.ASMOUT   DD    SYSOUT=*
//*-------------------------------------------------------------*
//
```

Calling REXX from C

The following samples show how to call a REXX exec from a C program under z/OS.

EAGGJC for Calling IRXJCL

This is the EAGGJC sample for calling IRXJCL from a C program:

```
//* -->uidIEC JOB - Specify your Job card here
//*--------------------------------------------------------------------*
//*                                                                    *
//*    Interlanguage Communication in z/OS                             *
//*       Calling REXX from C                                          *
//*                                                                    *
//*--------------------------------------------------------------------*
//*                                                                    *
//*  Licensed Materials - Property of IBM                              *
//*  5695-013                                                          *
//*  (C) Copyright IBM Corp. 1989, 2003                                *
//*                                                                    *
//*--------------------------------------------------------------------*
//* Sample JCL for calling IRXJCL from a C program.                    *
//* For a description also refer to the REXX Compiler guide            *
//* SH19-8160.                                                         *
//* You may modify this sample for your needs by including             *
//* a REXX of your own. The ARGUMENT for the REXX procedure            *
//* may be taylored for your needs.                                    *
//*                                                                    *
//* Change Activity: 030708 - new for Release 4                        *
//*--------------------------------------------------------------------*
//* JCLLIB accesses the CLG procs, modify to your needs                *
//*--------------------------------------------------------------------*
//*MYLIB   JCLLIB ORDER=RXT.INTLANG.CNTL
//*--------------------------------------------------------------------*
//* Create REXX procedure HELLO into a temporary Library &&REXX        *
//* SYSUT1 is setup for card input, modify to a DSN if desired         *
//*--------------------------------------------------------------------*
//CREATE  EXEC PGM=IEBGENER
//SYSPRINT DD SYSOUT=*
//SYSIN    DD DUMMY
//SYSUT2   DD DSN=&&REXX(HELLO),DISP=(NEW,PASS),
//            UNIT=SYSDA,SPACE=(TRK,(1,1,1)),
//            DCB=(RECFM=FB,LRECL=80,BLKSIZE=800,DSORG=PO)
//SYSUT1   DD *,DLM=$$
```

```
/* REXX ***************************************** START sample exec */
Parse source src                                     /* sample exec */
Arg n                                                /* sample exec */
  Say 'I am' src                                     /* sample exec */
  Say 'Received parm:' n                             /* sample exec */
  If n='' Then n=1                                   /* sample exec */
  Do i=1 TO n                                        /* sample exec */
    Say 'Hello World the' i'. time ...'              /* sample exec */
  End                                                /* sample exec */
Return n                  /* Set Return Code to n */ /* sample exec */
/********************************************** END sample exec */
$$
//*-------------------------------------------------------------------*
//*                                                                   *
//*  Compile, link and execute a 'C' program                          *
//*                                                                   *
//*-------------------------------------------------------------------*
//*  invocate cataloged procedure EDCCLG to compile, link and         *
//*  execute the below listed C Program.                              *
//*  the C program calls the above listed REXX sample exec stored     *
//*  in a temporary dataset using TSO service IRXEXEC.                *
//*-------------------------------------------------------------------*
//EDCC  EXEC EDCCLG,CPARM='XREF,SOURCE'
//COMPILE.SYSIN DD *,DLM=$$
/********************************************** START  C-program */

/* invoke REXX procedure HELLO with parameter 3 using IRXJCL         */

#include <stdlib.h> /* for fetch() prototype */

typedef int (*funcPtr) (); /* pointer to a function returning an int */
funcPtr fetched;

/*********************************************************************/
/* define IRXJCL parameter block                                     */
/*********************************************************************/
typedef struct IRXJCL_type
 {
  short int  arg_length;
  char argument[9];
 } IRXJCL_type;

/*********************************************************************/
/* define local variables                                            */
/*********************************************************************/
IRXJCL_type this_param;
IRXJCL_type* param_ptr;
int return_code;

main()
{
  printf("Start of C program\n");

  fetched = (funcPtr) fetch("IRXJCL");
  if (fetched == 0)
  {
    printf("ERROR: fetch() failed\n");
  }
  else
  {
    printf("now execute fetched module\n");
    /*****************************************************************/
    /* generate IRXJCL parameter block                               */
    /*****************************************************************/
    this_param.arg_length = 8;                    /* <============= */
    strcpy(this_param.argument,"HELLO  3");       /* <============= */
    param_ptr = &this_param;
```

```
    /*************************************************************/
    /* call the REXX Exec                                        */
    /*************************************************************/
    return_code = (*fetched)(param_ptr);
    printf("REXX return code= %d\n", return_code);
  }

  printf("End of C program\n");
}
/********************************************** END    C-program */
$$
//*-------------------------------------------------------------------*
//* Define the library containing the REXX exec                       *
//*-------------------------------------------------------------------*
//GO.SYSEXEC DD DISP=SHR,DSN=&&REXX
//*-------------------------------------------------------------------*
//* Next DD is the data set equivalent to terminal input              *
//*-------------------------------------------------------------------*
//GO.SYSTSIN  DD    DUMMY
//*-------------------------------------------------------------------*
//* Next DD is the data set equivalent to terminal output             *
//*-------------------------------------------------------------------*
//GO.SYSTSPRT DD    SYSOUT=*
//*-------------------------------------------------------------------*
//
```

EAGGXC for Calling IRXEXEC

This is the EAGGXC sample for calling IRXEXEC from a C program:

```
//* -->uidIEC JOB - Specify your Job card here
//*-------------------------------------------------------------------*
//*                                                                   *
//*    Interlanguage Communication in z/OS                            *
//*       Calling REXX from C                                         *
//*                                                                   *
//*-------------------------------------------------------------------*
//*                                                                   *
//*  Licensed Materials - Property of IBM                             *
//*  5695-013                                                         *
//*  (C) Copyright IBM Corp. 1989, 2003                               *
//*                                                                   *
//*-------------------------------------------------------------------*
//* Sample JCL for calling IRXEXEC from a C program.                  *
//* For a description also refer to the REXX Compiler guide           *
//* SH19-8160.                                                        *
//* You may modify this sample for your needs by including            *
//* a REXX of your own. The ARGUMENT for the REXX procedure           *
//* may be taylored for your needs.                                   *
//* The use of IRXEXEC is more complex than the use of IRXJCL.        *
//* Refer to the TSO guide SC28-1883 for using these services.        *
//*                                                                   *
//* Change Activity: 030708 - new for Release 4                       *
//*-------------------------------------------------------------------*
//* JCLLIB accesses the CLG procs, modify to your needs               *
//*-------------------------------------------------------------------*
//*MYLIB   JCLLIB ORDER=RXT.INTLANG.CNTL
//*-------------------------------------------------------------------*
//* Create REXX procedure HELLO into a temporary Library &&REXX       *
//* SYSUT1 is setup for card input, modify to a DSN if desired        *
//*-------------------------------------------------------------------*
//CREATE  EXEC PGM=IEBGENER
//SYSPRINT DD SYSOUT=*
//SYSIN    DD DUMMY
//SYSUT2   DD DSN=&&REXX(HELLO),DISP=(NEW,PASS),
//            UNIT=SYSDA,SPACE=(TRK,(1,1,1)),
//            DCB=(RECFM=FB,LRECL=80,BLKSIZE=800,DSORG=PO)
//SYSUT1   DD *,DLM=$$
```

```
/* REXX ***************************************** START sample exec */
Parse source src                                    /* sample exec */
Arg n                                               /* sample exec */
  Say 'I am' src                                    /* sample exec */
  Say 'Received parm:' n                            /* sample exec */
  If n='' Then n=1                                  /* sample exec */
  Do i=1 TO n                                       /* sample exec */
    Say 'Hello World the' i'. time ...'             /* sample exec */
  End                                               /* sample exec */
Return n                  /* Set Return Code to n */ /* sample exec */
/************************************************** END sample exec */
$$
//*-------------------------------------------------------------------*
//*                                                                   *
//*  Compile, link and execute a 'C' program                          *
//*                                                                   *
//*-------------------------------------------------------------------*
//*  invocate cataloged procedure EDCCLG to compile, link and         *
//*  execute the below listed C Program.                              *
//*  the C program calls the above listed REXX sample exec stored     *
//*  in a temporary dataset using TSO service IRXEXEC.                *
//*-------------------------------------------------------------------*
//EDCC  EXEC EDCCLG,CPARM='XREF,SOURCE'
//COMPILE.SYSIN DD *,DLM=$$
/************************************************* START  C-program */

/* invoke REXX procedure HELLO with parameter 3 using IRXEXEC       */

#include <stdlib.h>                         /* for fetch() prototype */

typedef int (*funcPtr) (); /* pointer to a function returning an int */
funcPtr fetched;

/*********************************************************************/
/* define EXECBLOCK control block                                   */
/*********************************************************************/
typedef struct EXECBLK_type
 {
  char EXECBLK_ACRYN[8];
                    /* The description of each bit is as follows:    */
                    /* An eight-character field that                 */
                    /* identifies the exec block. It                 */
                    /* must contain the character                    */
                    /* string 'IRXEXECB'.                            */
  int  EXECBLK_LENGTH;
                    /* Specifies the length of the                   */
                    /* exec block in bytes.                          */
  int  EXECBLK_reserved;
  char EXECBLK_MEMBER[8];
                    /* Specifies the member name of                  */
                    /* the exec if the exec is in a                  */
                    /* partitioned data set. If the                  */
                    /* exec is in a sequential data                  */
                    /* set, this field must be blank.                */
  char EXECBLK_DDNAME[8];
                    /* Specifies the name of the DD                  */
                    /* from which the exec is loaded.                */
                    /* An exec cannot be loaded from                 */
                    /* a DD that has not been                        */
                    /* allocated. The ddname you                     */
                    /* specify must be allocated to a                */
                    /* data set containing REXX execs                */
                    /* or to a sequential data set                   */
                    /* that contains an exec.                        */
                    /*                                               */
                    /* If this field is blank, the                   */
                    /* exec is loaded from the DD                    */
```

```
                    /* specified in the LOADDD field                */
                    /* of the module name table (see                */
                    /* topic 14.8). The default is                  */
                    /* SYSEXEC.                                     */
  char EXECBLK_SUBCOM[8];
                    /* Specifies the name of the                    */
                    /* initial host command                         */
                    /* environment when the exec                    */
                    /* starts running.                              */
                    /*                                              */
                    /* If this field is blank, the                  */
                    /* environment specified in the                 */
                    /* INITIAL field of the host                    */
                    /* command environment table is                 */
                    /* used. For TSO/E and ISPF, the                */
                    /* default is TSO. For a                        */
                    /* non-TSO/E address space, the                 */
                    /* default is MVS. The table is                 */
                    /* described in "Host Command                   */
                    /* Environment Table" in                        */
                    /* topic 14.9.                                  */
  void * EXECBLK_DSNPTR;
                    /* Specifies the address of a                   */
                    /* data set name that the PARSE                 */
                    /* SOURCE instruction returns.                  */
                    /* The name usually represents                  */
                    /* the name of the exec load data               */
                    /* set. The name can be up to 54                */
                    /* characters long (44 characters               */
                    /* for the fully qualified data                 */
                    /* set name, 8 characters for the               */
                    /* member name, and 2 characters                */
                    /* for the left and right                       */
                    /* parentheses).                                */
                    /*                                              */
                    /* If you do not want to specify                */
                    /* a data set name, specify an                  */
                    /* address of 0.                                */
  int  EXECBLK_DSNLEN;
                    /* Specifies the length of the                  */
                    /* data set name that is pointed                */
                    /* to by the address at offset                  */
                    /* +40. The length can be 0-54.                 */
                    /* If no data set name is                       */
                    /* specified, the length is 0.                  */
 } EXECBLK_type;

/*********************************************************************/
/* define ARGUMENT block                                             */
/*********************************************************************/
typedef struct one_parameter_type
 {
  void * ARGSTRING_PTR;
  int    ARGSTRING_LENGTH;
 } one_parameter_type;

/*********************************************************************/
/* define EVALBLOCK control block                                   */
/*********************************************************************/
typedef struct EVALBLK_type
 {
  int  EVALBLK_EVPAD1;
  int  EVALBLK_EVSIZE;
                    /* Specifies the total size of the              */
                    /* evaluation block in doublewords.             */
  int  EVALBLK_EVLEN;
  int  EVALBLK_EVPAD2;
```

```
  char EVALBLK_EVDATA[256];
 } EVALBLK_type;
                    /*

/*********************************************************************/
/* define IRXEXEC argument block                                     */
/*********************************************************************/
typedef struct IRXEXEC_type
 {
  EXECBLK_type ** execblk_ptr;
                    /* Specifies the address of the exec block       */
                    /* (EXECBLK). The exec block is a control        */
                    /* block that describes the exec to be           */
                    /* loaded. It contains information needed        */
                    /* to process the exec, such as the DD           */
                    /* from which the exec is to be loaded and       */
                    /* the name of the initial host command          */
                    /* environment when the exec starts running.     */

  one_parameter_type ** argtable_ptr;
                    /* Specifies the address of the arguments        */
                    /* for the exec. The arguments are               */
                    /* arranged as a vector of address/length        */
                    /* pairs followed by a fence                     */

  int * flags_ptr;  /************************************************/
                    /* The description of each bit is as follows:    */
                    /*                                               */
                    /* Bit 0 - This bit must be set on if the exec is*/
                    /* being invoked as a "command"; that is, the    */
                    /* exec is not being invoked from another exec as*/
                    /* an external function or subroutine. If you    */
                    /* pass more than one argument to the exec, do   */
                    /* not set bit 0 on.                             */
                    /*                                               */
                    /* Bit 1 - This bit must be set on if the exec is*/
                    /* being invoked as an external function         */
                    /* (a function call).                            */
                    /*                                               */
                    /* Bit 2 - This bit must be set on if the exec is*/
                    /* being invoked as a subroutine for example,    */
                    /* when the CALL keyword instruction is used.    */
                    /************************************************/
  int * instblk_ptr;
                    /* Specifies the address of the in-storage       */
                    /* control block (INSTBLK), which defines        */
                    /* the structure of a preloaded exec in storage. */
                    /* This parameter is required if the             */
                    /* caller of IRXEXEC has preloaded the           */
                    /* exec.  Otherwise, this parameter must be 0.   */
  int * reserved_parm5;
                    /* Specifies the address of the command          */
                    /* processor parameter list (CPPL) if you        */
                    /* call IRXEXEC from the TSO/E address           */
                    /* space. If you do not pass the address         */
                    /* of the CPPL (you specify an address of        */
                    /* 0), TSO/E builds the CPPL without a           */
                    /* command buffer.                               */
                    /* If you call IRXEXEC from a non-TSO/E          */
                    /* address space, specify an address of 0.       */
  EVALBLK_type ** evalblk_ptr;
                    /* Specifies the address of an evaluation        */
                    /* block (EVALBLOCK). IRXEXEC uses the           */
                    /* evaluation block to return the result         */
                    /* from the exec that was specified on           */
                    /* either the RETURN or EXIT instruction.        */
  int * reserved_workarea_ptr;
```

```
                    /* Specifies the address of an 8-byte           */
                    /* field that defines a work area for the       */
                    /* IRXEXEC routine. In the 8-byte field, the:   */
                    /*   First four bytes contain the               */
                    /*     address of the work area                 */
                    /*   Second four bytes contain the              */
                    /*     length of the work area.                 */
                    /* If you do not want to pass a work area,      */
                    /* specify an address of 0.                     */
  int * reserved_userfield_ptr;
                    /* Specifies the address of a user field.       */
                    /* If you do not want to use a user field,      */
                    /* specify an address of 0.                     */
  int * reserved_envblock_ptr;
                    /* The address of the environment block         */
                    /* that represents the environment in           */
                    /* which you want IRXEXEC to run.               */
  int * REXX_return_code_ptr;
                    /* A 4-byte field that IRXEXEC uses to          */
                    /* return the return code.                      */

 } IRXEXEC_type;

/*********************************************************************/
/* define local variables                                            */
/*********************************************************************/
IRXEXEC_type this_param;
EXECBLK_type this_EXECBLK;
EXECBLK_type * an_EXECBLK_ptr;
EVALBLK_type this_EVALBLK;
EVALBLK_type * an_EVALBLK_ptr;
one_parameter_type this_argument[2];
one_parameter_type * an_argtable_ptr;
char arg1;
int flags;
int REXX_return_code;
int dummy_zero;

main()
{
  printf("Start of CPROG\n");

  fetched = (funcPtr) fetch("IRXEXEC");
  if (fetched == 0)
  {
    printf("ERROR: fetch() failed\n");
  }
  else
  {
    /*****************************************************************/
    /* generate REXX Exec parameter block, finished with fence       */
    /*****************************************************************/
    arg1 = '3';
    this_argument[0].ARGSTRING_PTR = &arg1;
    this_argument[0].ARGSTRING_LENGTH = 1;
    this_argument[1].ARGSTRING_PTR = (void *)0xFFFFFFFF;
    this_argument[1].ARGSTRING_LENGTH = 0xFFFFFFFF;
    an_argtable_ptr = &this_argument[0];
    /*****************************************************************/
    /* generate EXECBLOCK                                            */
    /*****************************************************************/
    an_EXECBLK_ptr = &this_EXECBLK;
    strcpy(this_EXECBLK.EXECBLK_ACRYN,"IRXEXECB");
    this_EXECBLK.EXECBLK_LENGTH = 48;
    this_EXECBLK.EXECBLK_reserved = 0;
    strcpy(this_EXECBLK.EXECBLK_MEMBER,"HELLO"); /* <================*/
    strcpy(this_EXECBLK.EXECBLK_SUBCOM,"        ");
```

```
    this_EXECBLK.EXECBLK_DSNPTR = 0;
    this_EXECBLK.EXECBLK_DSNLEN = 0;
    /*****************************************************************/
    /* generate EVALBLOCK                                            */
    /*****************************************************************/
    an_EVALBLK_ptr = &this_EVALBLK;
    this_EVALBLK.EVALBLK_EVPAD1 = 0;
    this_EVALBLK.EVALBLK_EVSIZE = 34;
    this_EVALBLK.EVALBLK_EVLEN = 0;
    this_EVALBLK.EVALBLK_EVPAD2 = 0;
    /*****************************************************************/
    /* generate IRXEXEC parameter block                              */
    /*****************************************************************/
    this_param.execblk_ptr = &an_EXECBLK_ptr;
    this_param.argtable_ptr = &an_argtable_ptr;
    this_param.flags_ptr = &flags;
    this_param.instblk_ptr = &dummy_zero;
    this_param.reserved_parm5 = &dummy_zero;
    this_param.evalblk_ptr = &an_EVALBLK_ptr;
    this_param.reserved_workarea_ptr = &dummy_zero;
    this_param.reserved_userfield_ptr = &dummy_zero;
    this_param.reserved_envblock_ptr = &dummy_zero;
    this_param.REXX_return_code_ptr = &REXX_return_code;
    this_param.REXX_return_code_ptr =
      (int *)((int)this_param.REXX_return_code_ptr | 0x80000000);
    dummy_zero = 0;
    flags = 0x20000000;                /* exec invoked as subroutine */
    REXX_return_code = 0;
    /*****************************************************************/
    /* call the REXX Exec                                            */
    /*****************************************************************/
    REXX_return_code = (*fetched)(this_param);
    /*****************************************************************/
    /* handle return code and result                                 */
    /*****************************************************************/
    printf("REXX return code is: %d\n", REXX_return_code);
    printf("REXX result is: %-*.*s\n",
          this_EVALBLK.EVALBLK_EVLEN,
          this_EVALBLK.EVALBLK_EVLEN,
          this_EVALBLK.EVALBLK_EVDATA);
  }

  printf("End of CPROG\n");
}
/************************************************ END    C-program */
$$
//*-------------------------------------------------------------------*
//* Define the library containing the REXX exec                       *
//*-------------------------------------------------------------------*
//GO.SYSEXEC DD DISP=SHR,DSN=&&REXX
//*-------------------------------------------------------------------*
//* Next DD is the data set equivalent to terminal input              *
//*-------------------------------------------------------------------*
//GO.SYSTSIN  DD    DUMMY
//*-------------------------------------------------------------------*
//* Next DD is the data set equivalent to terminal output             *
//*-------------------------------------------------------------------*
//GO.SYSTSPRT DD    SYSOUT=*
//*-------------------------------------------------------------------*
//
```

Calling REXX from Cobol

The following samples show how to call a REXX exec from a Cobol program under z/OS.

EAGGJCOB for Calling IRXJCL

This is the EAGGJCOB sample for calling IRXJCL from a Cobol program:

```
//* -->uidIEC JOB - Specify your Job card here
//*-------------------------------------------------------------*
//*
//*    Interlanguage Communication in z/OS
//*       Calling REXX from COBOL
//*
//*-------------------------------------------------------------*
//*
//*  Licensed Materials - Property of IBM
//*  5695-013
//*  (C) Copyright IBM Corp. 1989, 2003
//*
//*-------------------------------------------------------------*
//* Sample JCL for calling IRXJCL from COBOL program.
//* For a description also refer to the REXX Compiler guide
//* SH19-8160.
//* You may modify this sample for your needs by including
//* a REXX of your own. The argument for the REXX procedure
//* may be taylored for your needs.
//*
//* Change Activity: 030708 - new for Release 4
//*-------------------------------------------------------------*
//* JCLLIB accesses the CLG procs, modify to your needs
//*-------------------------------------------------------------*
//*MYLIB   JCLLIB ORDER=RXT.INTLANG.CNTL
//*-------------------------------------------------------------*
//* Create REXX procedure HELLO into a temporary Library &&REXX
//* SYSUT1 is setup for card input, modify to a DSN if desired.
//*-------------------------------------------------------------*
//CREATE  EXEC PGM=IEBGENER
//SYSPRINT DD SYSOUT=*
//SYSIN    DD DUMMY
//SYSUT2   DD DSN=&&REXX(HELLO),DISP=(NEW,PASS),
//            UNIT=SYSDA,SPACE=(TRK,(1,1,1)),
//            DCB=(RECFM=FB,LRECL=80,BLKSIZE=800,DSORG=PO)
//SYSUT1   DD *,DLM=$$
/* REXX - A simple exec ***************** Start sample exec */
Parse source src                              /* sample exec */
Arg n                                         /* sample exec */
  Say 'I am' src                              /* sample exec */
  Say 'Received parm:' n                      /* sample exec */
  If n='' Then n=1                            /* sample exec */
  Do i=1 TO n                                 /* sample exec */
    Say 'Hello World the' i'. time ...'       /* sample exec */
  End                                         /* sample exec */
Return n           /* Set Return Code to n */ /* sample exec */
/****************************************** End sample exec */
$$
//*-------------------------------------------------------------*
//*  Compile, link and execute a COBOL program
//*-------------------------------------------------------------*
//* Invoke the cataloged procedure IGYWCLG to compile, link and
//* execute the below listed Cobol program.
//* The Cobol program below calls the above listed REXX sample
//* exec as stored in the temporary dataset using the
//* TSO service IRXJCL.
//*-------------------------------------------------------------*
//IGYWCLG  EXEC IGYWCLG
//COBOL.SYSIN DD *
CBL NOADV,NODYN,NONAME,NONUMBER,QUOTE,SEQ,XREF,VBREF,DUMP,LIST
       TITLE "COBOL TEST PROGRAM  ".
       Identification Division.

       Program-id.    COBPRG.
```

```
IA0040 Author.    ...
      ***                                                            **
      ***  Invoke REXX procedure HELLO with parameter 3 using IRXJCL **
      ***                                                            **
      ***  Expected display messages:                                **
      ***    "PROGRAM CONPRG   - BEGINNING"                          **
      ***    "PROGRAM COBPRG   - NORMAL END"                         **
      ***                                                            **
      ****************************************************************

       Environment division.
IA0970 Configuration section.
         Special-names.
       Input-output section.
         File-control.
           Select PRINT-FILE
               assign to SYS014-S-UPDPRNT
               file status is UPDPRINT-FILE-STATUS.
       Data division.
       File section.
IA1570 FD  PRINT-FILE
           recording mode F
           block 0 records
           record 121 characters
           label record standard.
IA1620   01 print-record                  pic x(121).
       Working-storage section.
         01 Working-storage-for-COBPRG     pic x.

         01 ARGUMENT.
            03 ARG-SIZE                   pic 9(2) comp.
            03 ARG-CHAR                   pic x(8).

         77 UPDPRINT-file-status          pic xx.

         77 PGM-NAME                      pic x(8).

      /****************************************************************
      ***                 D O   M A I N   L O G I C                  **
      ****************************************************************
       procedure division.
         000-do-main-logic.
           display "PROGRAM COBPRG   - Beginning".
           display "Return code before call is " RETURN-CODE.
      *
      *    Pass the procedure pame HELLO to IRXJCL.
      *    Pass 3 to REXX procedure 'HELLO'.
      *    Set the size of the argument.
      *
           move "HELLO  3" to ARG-CHAR.
           move 8 to arg-size.

      *    Call "IRXJCL" in order to execute the REXX procedure
           move "IRXJCL" to PGM-NAME.
           CALL PGM-NAME     USING ARGUMENT.

      *    Display the return code.
           display "Return code after  call is " RETURN-CODE.
           display "PROGRAM COBPRG   - Normal end".
           stop run.

IA9990 end program COBPRG.
/*
//*-------------------------------------------------------------*
//* Define the library containing the REXX exec
//*-------------------------------------------------------------*
//GO.SYSEXEC DD DISP=(SHR,PASS),DSN=&&REXX
```

```
//*-------------------------------------------------------------*
//* Next DD is the data set equivalent to terminal input
//*-------------------------------------------------------------*
//GO.SYSTSIN  DD    DUMMY
//*-------------------------------------------------------------*
//* Next DD is the data set equivalent to terminal output
//*-------------------------------------------------------------*
//GO.SYSTSPRT DD    SYSOUT=*
//GO.SYSOUT   DD    SYSOUT=*
//*-------------------------------------------------------------*
//
```

EAGGXCOB for Calling IRXEXEC

This is the EAGGXCOB sample for calling IRXEXEC from a Cobol program:

```
//* -->uidIEC JOB - Specify your Job card here
//*-------------------------------------------------------------*
//*
//*    Interlanguage Communication in z/OS
//*       Calling REXX from COBOL
//*
//*-------------------------------------------------------------*
//*
//*  Licensed Materials - Property of IBM
//*  5695-013
//*  (C) Copyright IBM Corp. 1989, 2003
//*
//*-------------------------------------------------------------*
//* Sample JCL for calling IRXEXEC from COBOL program.
//* For a description also refer to the REXX Compiler guide
//* SH19-8160.
//* You may modify this sample for your needs by including
//* a REXX of your own. The argument for the REXX procedure
//* may be taylored for your needs.
//*
//* Change Activity: 030708 - new for Release 4
//*-------------------------------------------------------------*
//* JCLLIB accesses the CLG procs, modify to your needs
//*-------------------------------------------------------------*
//*MYLIB   JCLLIB ORDER=RXT.INTLANG.CNTL
//*-------------------------------------------------------------*
//* Create REXX procedure HELLO into a temporary Library &&REXX
//* SYSUT1 is setup for card input, modify to a DSN if desired.
//*-------------------------------------------------------------*
//CREATE  EXEC PGM=IEBGENER
//SYSPRINT DD SYSOUT=*
//SYSIN    DD DUMMY
//SYSUT2   DD DSN=&&REXX(HELLO),DISP=(NEW,PASS),
//            UNIT=SYSDA,SPACE=(TRK,(1,1,1)),
//            DCB=(RECFM=FB,LRECL=80,BLKSIZE=800,DSORG=PO)
//SYSUT1   DD *,DLM=$$
/* REXX - A simple exec ***************** Start sample exec */
Parse source src                           /* sample exec */
Arg n                                      /* sample exec */
  Say 'I am' src                           /* sample exec */
  Say 'Received parm:' n                   /* sample exec */
  If n='' Then n=1                         /* sample exec */
  Do i=1 TO n                              /* sample exec */
    Say 'Hello World the' i'. time ...'    /* sample exec */
  End                                      /* sample exec */
Return n           /* Set Return Code to n */ /* sample exec */
/****************************************** End sample exec */
$$
//*-------------------------------------------------------------*
//*  Compile, link and execute a COBOL program
//*-------------------------------------------------------------*
//* Invoke the cataloged procedure IGYWCLG to compile, link and
```

```
//* execute the below listed Cobol program.
//* The Cobol program below calls the above listed REXX sample
//* exec as stored in the temporary dataset using the
//* TSO service IRXEXEC.
//*-------------------------------------------------------------*
//IGYWCLG  EXEC IGYWCLG
//COBOL.SYSIN DD *
CBL NOADV,NODYN,NONAME,NONUMBER,QUOTE,SEQ,XREF,VBREF,DUMP,LIST
CBL TRUNC(OPT)
       TITLE "COBOL TEST PROGRAM  ".
       Identification Division.

       Program-id.    COBPRG.
IA0040 Author. ...
      ***                                                           **
      ***  Invoke REXX procedure HELLO with parm 3 using IRXEXEC    **
      ***                                                           **
      ***  Expected display messages:                               **
      ***    "PROGRAM COBPRG   - BEGINNING"                         **
      ***    "PROGRAM COBPRG   - NORMAL END"                        **
      ***                                                           **
      ***************************************************************

       Environment division.
IA0970 Configuration section.
         Special-names.
       Input-output section.
         File-control.
           Select PRINT-FILE
               assign to SYS014-S-UPDPRNT
               file status is UPDPRINT-FILE-STATUS.
       Data division.
       File section.
IA1570 FD  PRINT-FILE
           recording mode F
           block 0 records
           record 121 characters
           label record standard.
IA1620   01 print-record                  pic x(121).
       Working-storage section.
         01 Working-storage-for-COBPRG    pic x.

         77 PGM-NAME                      pic X(8).

      *
      * Define the IRXEXEC argument blocks
      *
         01 EXECBLK.
            03 EXECBLK-ACRYN              pic X(8).
            03 EXECBLK-LENGTH             pic S9(8) binary.
            03 EXECBLK-reserved           pic S9(8) binary.
            03 EXECBLK-MEMBER             pic X(8).
            03 EXECBLK-DDNAME             pic X(8).
            03 EXECBLK-SUBCOM             pic X(8).
            03 EXECBLK-DSNPTR             POINTER.
            03 EXECBLK-DSNLEN             pic 9(4) comp.

         01 EVALBLK.
            03 EVALBLK-EVPAD1             pic S9(8) binary.
            03 EVALBLK-EVSIZE             pic S9(8) binary.
            03 EVALBLK-EVLEN              pic S9(8) binary.
            03 EVALBLK-EVPAD2             pic S9(8) binary.
            03 EVALBLK-EVDATA             pic x(256).

         77 flags                         pic S9(8) binary.
         77 REXX-return-code              pic S9(8) binary.
         77 dummy-zero                    pic S9(8) binary.
```

```
   01 ARGUMENT.
     02 ARGUMENT-1 OCCURS 1 TIMES.
      05 ARGSTRING-PTR              POINTER.
      05 ARGSTRING-LENGTH           pic S9(8) binary.
     02 ARGSTRING-LAST1             pic S9(8) binary.
     02 ARGSTRING-LAST2             pic S9(8) binary.
   77 arg1                         pic x(1).
   77 execblk-ptr                  POINTER.
   77 argtable-ptr                 POINTER.
   77 evalblk-ptr                  POINTER.

   77 UPDPRINT-file-status          pic xx.

/*****************************************************************
***                D O   M A I N   L O G I C                  **
*****************************************************************
 procedure division.
   000-do-main-logic.
     display "PROGRAM COBPRG   - Beginning".

*--- Pass 3 as argument to the REXX procedure 'HELLO'.
     move "3" to arg1.

     call "GET-ARG1-PTR" using arg1 ARGSTRING-PTR(1).
     move 1 to ARGSTRING-LENGTH(1).
     move -1 to ARGSTRING-LAST1.
     move -1 to ARGSTRING-LAST2.
     call "GET-ARGUMENT-PTR" using argument argtable-ptr.

     move "IRXEXECB" to EXECBLK-ACRYN.
     move 48 to EXECBLK-LENGTH.
     move 0 to EXECBLK-reserved.

*--- Pass the procedure name HELLO to IRXEXEC.
     move "HELLO" to EXECBLK-MEMBER.
     move "        " to EXECBLK-SUBCOM.
     move "        " to EXECBLK-DDNAME.
     set EXECBLK-DSNPTR to NULL.
     move 0 to EXECBLK-DSNLEN.
     call "GET-EXECBLK-PTR" using EXECBLK execblk-ptr.
     move 0 to EVALBLK-EVPAD1.
     move 34 to EVALBLK-EVSIZE.
     move 0 to EVALBLK-EVLEN.
     move 0 to EVALBLK-EVPAD2.
     call "GET-EVALBLK-PTR" using EVALBLK evalblk-ptr.
     move 0 to dummy-zero.

*--- Set flags to HEX 20000000
*    i.e. exec invoked as subroutine
     move 536870912 to flags.
     move 0 to REXX-return-code.

*--- Call the REXX exec ---
     move "IRXEXEC " to PGM-NAME.
     CALL PGM-NAME USING execblk-ptr
                         argtable-ptr
                         flags
                         dummy-zero
                         dummy-zero
                         evalblk-ptr
                         dummy-zero
                         dummy-zero
                         dummy-zero
                         REXX-return-code.
     CANCEL PGM-NAME.

*--- Display the return code.
```

```
     display "REXX return code is: " REXX-return-code.
     display "REXX result is: " EVALBLK-EVDATA.
     display "PROGRAM COBPRG   - Normal end".
     stop run.


*--- Addressing helper
 Identification Division.
 Program-id.    GET-ARG1-PTR.
 Environment division.
 Data division.
 Working-storage section.
 Linkage section.
   77 arg1                          pic x(1).
   77 arg-ptr                       POINTER.

 procedure division using arg1 arg-ptr.
     set arg-ptr to address of arg1.
     goback.
 end program GET-ARG1-PTR.


*--- Addressing helper
 Identification Division.
 Program-id.    GET-ARGUMENT-PTR.
 Environment division.
 Data division.
 Working-storage section.
 Linkage section.
   01 ARGUMENT.
     02 ARGUMENT-1 OCCURS 1 TIMES.
      05 ARGSTRING-PTR              POINTER.
      05 ARGSTRING-LENGTH           pic S9(8) binary.
     02 ARGSTRING-LAST1             pic S9(8) binary.
     02 ARGSTRING-LAST2             pic S9(8) binary.
   77 argtable-ptr                  POINTER.

 procedure division using ARGUMENT argtable-ptr.
     set argtable-ptr to address of ARGUMENT.
     goback.
 end program GET-ARGUMENT-PTR.


*--- Addressing helper
 Identification Division.
 Program-id.    GET-EXECBLK-PTR.
 Environment division.
 Data division.
 Working-storage section.
 Linkage section.
   01 EXECBLK.
      03 EXECBLK-ACRYN              pic X(8).
      03 EXECBLK-LENGTH             pic 9(4) comp.
      03 EXECBLK-reserved           pic 9(4) comp.
      03 EXECBLK-MEMBER             pic X(8).
      03 EXECBLK-DDNAME             pic X(8).
      03 EXECBLK-SUBCOM             pic X(8).
      03 EXECBLK-DSNPTR             POINTER.
      03 EXECBLK-DSNLEN             pic 9(4) comp.
   77 execblk-ptr                   POINTER.

 procedure division using EXECBLK execblk-ptr.
     set execblk-ptr to address of EXECBLK.
     goback.
 end program GET-EXECBLK-PTR.
```

```
      *--- Addressing helper
       Identification Division.
       Program-id.    GET-EVALBLK-PTR.
       Environment division.
       Data division.
       Working-storage section.
       Linkage section.
         01 EVALBLK.
            03 EVALBLK-EVPAD1             pic 9(4) comp.
            03 EVALBLK-EVSIZE             pic 9(4) comp.
            03 EVALBLK-EVLEN              pic 9(4) comp.
            03 EVALBLK-EVPAD2             pic 9(4) comp.
            03 EVALBLK-EVDATA             pic x(256).
         77 evalblk-ptr                   POINTER.

       procedure division using EVALBLK evalblk-ptr.
           set evalblk-ptr to address of EVALBLK.
           goback.
       end program GET-EVALBLK-PTR.


IA9990 end program COBPRG.
/*
//*-------------------------------------------------------------*
//* Define the library containing the REXX exec
//*-------------------------------------------------------------*
//GO.SYSEXEC DD DISP=SHR,DSN=&&REXX
//*-------------------------------------------------------------*
//* Next DD is the data set equivalent to terminal input
//*-------------------------------------------------------------*
//GO.SYSTSIN  DD    DUMMY
//*-------------------------------------------------------------*
//* Next DD is the data set equivalent to terminal output
//*-------------------------------------------------------------*
//GO.SYSTSPRT DD    SYSOUT=*
//GO.SYSOUT   DD    SYSOUT=*
//*-------------------------------------------------------------*
//
```

Calling REXX from PL/I

The following samples show how to call a REXX exec from a PL/I program under z/OS.

EAGGJPLI for Calling IRXJCL

This is the EAGGJPLI sample for calling IRXJCL from a PL/I program:

```
//* -->uidIEC JOB - Specify your Job card here
//*-------------------------------------------------------------*
//*
//*    Interlanguage Communication in z/OS
//*       Calling REXX from PLI
//*
//*-------------------------------------------------------------*
//*
//*  Licensed Materials - Property of IBM
//*  5695-013
//*  (C) Copyright IBM Corp. 1989, 2003
//*
//*-------------------------------------------------------------*
//* Sample JCL for calling IRXJCL from PL/I program.
//* For a description also refer to the REXX Compiler guide
//* SH19-8160.
//* You may modify this sample for your needs by including
//* a REXX of your own. The argument for the REXX procedure
```

```
//* may be taylored for your needs.
//*
//* Change Activity: 030708 - new for Release 4
//*-------------------------------------------------------------*
//* JCLLIB accesses the CLG procs, modify to your needs
//*-------------------------------------------------------------*
//*MYLIB   JCLLIB ORDER=RXT.INTLANG.CNTL
//*-------------------------------------------------------------*
//* Create REXX procedure HELLO into a temporary Library &&REXX
//* SYSUT1 is setup for card input, modify to a DSN if desired.
//*-------------------------------------------------------------*
//CREATE  EXEC PGM=IEBGENER
//SYSPRINT DD SYSOUT=*
//SYSIN    DD DUMMY
//SYSUT2   DD DSN=&&REXX(HELLO),DISP=(NEW,PASS),
//            UNIT=SYSDA,SPACE=(TRK,(1,1,1)),
//            DCB=(RECFM=FB,LRECL=80,BLKSIZE=800,DSORG=PO)
//SYSUT1   DD *,DLM=$$
/* REXX - A simple exec ***************** Start sample exec */
Parse source src                              /* sample exec */
Arg n                                         /* sample exec */
  Say 'I am' src                              /* sample exec */
  Say 'Received parm:' n                      /* sample exec */
  If n='' Then n=1                            /* sample exec */
  Do i=1 TO n                                 /* sample exec */
    Say 'Hello World the' i'. time ...'       /* sample exec */
  End                                         /* sample exec */
Return n           /* Set Return Code to n */ /* sample exec */
/****************************************** End sample exec */
$$
//*-------------------------------------------------------------*
//*  Compile, link and execute a PL/1 program
//*
//*  an alternative catlogued procedure is
//*  IBMZCBG for z/OS Enterprise PL/1
//*-------------------------------------------------------------*
//* Invoke the cataloged procedure to compile, link and
//* execute the below listed PL/1 program.
//* The PL/1 program below calls the above listed REXX sample
//* exec as stored in the temporary dataset using the
//* TSO service IRXJCL.
//*-------------------------------------------------------------*
//IEL1CLG  EXEC IEL1CLG
//PLI.SYSIN DD *
*PROCESS INCLUDE,SYSTEM(MVS),FLAG(I),XREF(SHORT),MAP,LIST;
*PROCESS LINECOUNT(100);

 PLIPROG: PROC OPTIONS(MAIN);

 /* invoke REXX procedure HELLO with parameter 3 using IRXJCL */

 DCL IRXJCL ENTRY EXTERNAL OPTIONS(ASSEMBLER RETCODE);
 DCL 1 IRXJCL_PARM,
       3 ARG_LENGTH     FIXED BINARY(15),
       3 ARGUMENT       CHAR(9);
 DCL PLIRETV BUILTIN;
 DCL RETURN_CODE        FIXED BINARY(31);


 PUT SKIP EDIT ('Start of PLIPROG') (A);
 ARG_LENGTH = 8;

 /* Pass the procedure name HELLO to IRXJCL. */
 /* Pass 3 as argument to the REXX procedure 'HELLO'. */
 ARGUMENT = 'HELLO  3';

 /* Call the REXX exec */
```

```
 FETCH IRXJCL;
 CALL IRXJCL(IRXJCL_PARM);

 /* Handle the return code. */
 RETURN_CODE = PLIRETV;
 PUT SKIP EDIT ('REXX RETURN CODE: ' , RETURN_CODE) (A, F(4));
 PUT SKIP EDIT ('End of PLIPROG') (A);
 RETURN;

 END PLIPROG;
/*
//*-------------------------------------------------------------*
//* Define the library containing the REXX exec
//*-------------------------------------------------------------*
//GO.SYSEXEC DD DISP=(SHR,PASS),DSN=&&REXX
//*-------------------------------------------------------------*
//* Next DD is the data set equivalent to terminal input
//*-------------------------------------------------------------*
//GO.SYSTSIN  DD    DUMMY
//*-------------------------------------------------------------*
//* Next DD is the data set equivalent to terminal output
//*-------------------------------------------------------------*
//GO.SYSTSPRT DD    SYSOUT=*
//*-------------------------------------------------------------*
//
```

EAGGXPLI for Calling IRXEXEC

This is the EAGGXPLI sample for calling IRXEXEC from a PL/I program:

```
//* -->uidIEC JOB - Specify your Job card here
//*-------------------------------------------------------------*
//*
//*    Interlanguage Communication in z/OS
//*       Calling REXX from PLI
//*
//*-------------------------------------------------------------*
//*
//*  Licensed Materials - Property of IBM
//*  5695-013
//*  (C) Copyright IBM Corp. 1989, 2003
//*
//*-------------------------------------------------------------*
//* Sample JCL for calling IRXEXEC from PL/I program.
//* For a description also refer to the REXX Compiler guide
//* SH19-8160.
//* You may modify this sample for your needs by including
//* a REXX of your own. The argument for the REXX procedure
//* may be taylored for your needs.
//*
//* Change Activity: 030708 - new for Release 4
//*-------------------------------------------------------------*
//* JCLLIB accesses the CLG procs, modify to your needs
//*-------------------------------------------------------------*
//*MYLIB   JCLLIB ORDER=RXT.INTLANG.CNTL
//*-------------------------------------------------------------*
//* Create REXX procedure HELLO into a temporary Library &&REXX
//* SYSUT1 is setup for card input, modify to a DSN if desired.
//*-------------------------------------------------------------*
//CREATE  EXEC PGM=IEBGENER
//SYSPRINT DD SYSOUT=*
//SYSIN    DD DUMMY
//SYSUT2   DD DSN=&&REXX(HELLO),DISP=(NEW,PASS),
//            UNIT=SYSDA,SPACE=(TRK,(1,1,1)),
//            DCB=(RECFM=FB,LRECL=80,BLKSIZE=800,DSORG=PO)
//SYSUT1   DD *,DLM=$$
/* REXX - A simple exec ***************** Start sample exec */
Parse source src                          /* sample exec */
```

```
Arg n                                          /* sample exec */
  Say 'I am' src                               /* sample exec */
  Say 'Received parm:' n                       /* sample exec */
  If n='' Then n=1                             /* sample exec */
  Do i=1 TO n                                  /* sample exec */
    Say 'Hello World the' i'. time ...'        /* sample exec */
  End                                          /* sample exec */
Return n           /* Set Return Code to n */ /* sample exec */
/*************************************** End sample exec */
$$
//*-------------------------------------------------------------*
//*  Compile, link and execute a PL/1 program
//*
//*  an alternative catlogued procedure is
//*  IBMZCBG for z/OS Enterprise PL/1
//*-------------------------------------------------------------*
//* Invoke the cataloged procedure to compile, link and
//* execute the below listed PL/1 program.
//* The PL/1 program below calls the above listed REXX sample
//* exec as stored in the temporary dataset using the
//* TSO service IRXEXEC.
//*-------------------------------------------------------------*
//IEL1CLG  EXEC IEL1CLG
//PLI.SYSIN DD *
*PROCESS INCLUDE,SYSTEM(MVS),FLAG(I),XREF(SHORT),MAP,LIST;
*PROCESS LINECOUNT(100);

 PLIPROG: PROCEDURE OPTIONS(MAIN);

 /* invoke REXX procedure HELLO with parameter 3 using IRXEXEC */
 DCL IRXEXEC ENTRY EXTERNAL OPTIONS(ASSEMBLER RETCODE);

 /* Declare the IRXEXEC argument blocks */
 DCL 1 EXECBLK,
       3 EXECBLK_ACRYN      CHAR(8),
       3 EXECBLK_LENGTH     FIXED BINARY(31),
       3 EXECBLK_reserved   FIXED BINARY(31),
       3 EXECBLK_MEMBER     CHAR(8),
       3 EXECBLK_DDNAME     CHAR(8),
       3 EXECBLK_SUBCOM     CHAR(8),
       3 EXECBLK_DSNPTR     PTR,
       3 EXECBLK_DSNLEN     FIXED BINARY(31);

 DCL 1 EVALBLK,
       3 EVALBLK_EVPAD1     FIXED BINARY(31),
       3 EVALBLK_EVSIZE     FIXED BINARY(31),
       3 EVALBLK_EVLEN      FIXED BINARY(31),
       3 EVALBLK_EVPAD2     FIXED BINARY(31),
       3 EVALBLK_EVDATA     CHAR(256);

 DCL 1 ARGTABLE,
       3 ARGUMENTS(1),
         5 ARGSTRING_PTR    PTR,
         5 ARGSTRING_LENGTH FIXED BINARY(31),
       3 ARGTABLE_LAST      CHAR(8);

 DCL EXECBLK_PTR            PTR;
 DCL ARGTABLE_PTR           PTR;
 DCL INSTBLK_PTR            PTR;
 DCL reserved_parm5         PTR;
 DCL EVALBLK_PTR            PTR;
 DCL reserved_workarea_ptr  PTR;
 DCL reserved_userfield_ptr PTR;
 DCL reserved_envblock_ptr  PTR;
 DCL REXX_return_code_ptr   PTR;
 DCL ARG1                   CHAR;
 DCL flags                  CHAR(4);
```

```
DCL REXX_return_code       FIXED BINARY(31);
DCL PLIRETV BUILTIN;
DCL SYSNULL BUILTIN;
DCL ADDR BUILTIN;
DCL SUBSTR BUILTIN;
DCL RETURN_CODE            FIXED BINARY(31);

PUT SKIP EDIT ('Start of PLIPROG') (A);

/* Pass 3 as argument to the REXX procedure 'HELLO'. */
ARG1 = '3';
ARGSTRING_PTR(1) = ADDR(ARG1);
ARGSTRING_LENGTH(1) = 1;
ARGTABLE_LAST = 'FFFFFFFFFFFFFFFF'X;
ARGTABLE_PTR = ADDR(ARGSTRING_PTR(1));
EXECBLK_PTR = ADDR(EXECBLK);
EXECBLK_ACRYN = 'IRXEXECB';
EXECBLK_LENGTH = 48;
EXECBLK_reserved = 0;

/* Pass the procedure name HELLO to IRXEXEC. */
EXECBLK_MEMBER = 'HELLO';
EXECBLK_SUBCOM = '        ';
EXECBLK_DSNPTR = SYSNULL;
EXECBLK_DSNLEN = 0;
EXECBLK_DDNAME = '        ';
EVALBLK_PTR = ADDR(EVALBLK);
EVALBLK_EVPAD1 = 0;
EVALBLK_EVSIZE = 34;
EVALBLK_EVLEN = 0;
EVALBLK_EVPAD2 = 0;

/* Set flags for exec invokation */
flags = '40000000'x;
REXX_return_code_ptr = ADDR(REXX_return_code);
REXX_return_code = 0;
INSTBLK_PTR = SYSNULL;
reserved_parm5 = SYSNULL;
reserved_workarea_ptr = SYSNULL;
reserved_userfield_ptr = SYSNULL;
reserved_envblock_ptr = SYSNULL;

/* Call the REXX exec */
FETCH IRXEXEC;
CALL  IRXEXEC(EXECBLK_PTR,
              ARGTABLE_PTR,
              flags,
              INSTBLK_PTR,
              reserved_parm5,
              EVALBLK_PTR,
              reserved_workarea_ptr,
              reserved_userfield_ptr,
              reserved_envblock_ptr,
              REXX_return_code_ptr);

/* Handle the return code. */
RETURN_CODE = PLIRETV;
PUT SKIP EDIT ('     RETURN CODE: ' , RETURN_CODE) (A, F(4));
PUT SKIP EDIT ('REXX RETURN CODE: ' , REXX_RETURN_CODE) (A, F(4));
PUT SKIP EDIT ('REXX RESULT IS: ' ||
         SUBSTR(EVALBLK_EVDATA,1,EVALBLK_EVLEN)) (A);
PUT SKIP EDIT ('End of PLIPROG') (A);
RETURN;
END PLIPROG;
/*
//*-------------------------------------------------------------*
//* Define the library containing the REXX exec
```

```
//*-------------------------------------------------------------*
//GO.SYSEXEC DD DISP=SHR,DSN=&&REXX
//*-------------------------------------------------------------*
//* Next DD is the data set equivalent to terminal input
//*-------------------------------------------------------------*
//GO.SYSTSIN  DD    DUMMY
//*-------------------------------------------------------------*
//* Next DD is the data set equivalent to terminal output
//*-------------------------------------------------------------*
//GO.SYSTSPRT DD    SYSOUT=*
//*-------------------------------------------------------------*
//
```

Appendix G. Notices

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

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Mail Station P300
522 South Road
Poughkeepsie New York 12601-5400
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

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

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

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurement may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

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

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

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

COPYRIGHT LICENSE:

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

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. _enter the year or years_. All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Programming Interface Information

This *User's Guide and Reference* documents intended Programming Interfaces that allow the customer to write programs to obtain services of the IBM Compiler and Library for REXX on System z.

Trademarks

The following terms are trademarks of the IBM Corporation in the United States, other countries, or both:

C/370
GDDM
IBM
MVS
MVS/ESA
NetView
OpenEdition
OS/390
SAA
Systems Application Architecture
SP
VM/ESA
VSE/ESA
z/Architecture
z/OS
z/VM

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, and service names may be trademarks or service marks of others.

Glossary of Terms and Abbreviations

This glossary defines terms as they are used in this book. If you cannot find the term you are looking for, refer to the *Dictionary of Computing* New York: McGraw-Hill, 1994.

A

allocate
To assign a resource, such as a disk file, to a specific task. Contrast with deallocate.

authorized program facility (APF)
Allows to identify system programs and user programs that can use sensitive system functions, and restricts the use of such functions to APF-authorized programs.

B

BIF Built-in function.

C

CEXEC output
Output produced by the IBM Compiler for REXX on System z licensed program when the CEXEC option is specified.

clause According to *SAA Common Programming Interface REXX Level 2 Reference*, a REXX program is built from a series of **clauses** that are composed of:
- Zero or more blanks (which are ignored)
- A sequence of tokens
- Zero or more blanks (again, ignored)
- A semicolon (;) delimiter that may be implied by line-end, certain keywords, or the colon (:)

CMS Conversational Monitor System.

compiled EXEC
A compiled REXX program file that has the same file type that the corresponding source file would have for interpretation.

Conversational Monitor System (CMS
A virtual machine operating system that provides general interactive time sharing, problem solving, and program development capabilities, and operates only under control of the z/VM control program.

CPPL TSO/E command processor parameter list.

CPPLEFPL
A stub that is a combination of the CPPL and EFPL stubs. It contains the logic to determine if the REXX program is being invoked as a TSO/E command or as a REXX external routine. Once this has been determined, the compiled REXX program is given control with the appropriate parameters.

cross-reference listing
The portion of the compiler listing that contains information on where symbols are referenced in a program.

D

data set
The major unit of data storage and retrieval, consisting of a collection of data in one of several prescribed arrangements and described by control information to which the system has access.

DBCS Double-byte character set.

DCSS Discontiguous saved segment. Also known as discontiguous shared segment.

ddname
Data definition name.

DD statement
Data definition statement.

discontiguous saved segment (DCSS)
An area of storage beyond the address of your virtual machine address space (not contiguous with your virtual storage) where segments are loaded as needed.

double-byte character set (DBCS)
A character set, such as Kanji, for languages that require 2 bytes to uniquely define each character.

E

EFPL External function parameter list.

ESD External symbol dictionary.

external symbol dictionary (ESD)
Control information associated with an

object or load module that identifies the external symbols in the module.

F

FMID Function modification identifier.

H

HFS data set
A hierarchical file system data set, which is used to store, and is essentially identified with, a file system.

I

IEXEC output
Output produced by the IBM Compiler for IBM Compiler for REXX on System z licensed program when the IEXEC option is specified.

Interactive System Productivity Facility (ISPF)
An IBM-licensed program that provides a common dialog management facility across operating system environments.

interpreter
A program that translates and executes each instruction of a high-level programming language before it translates and executes the next instruction.

ISPF Interactive System Productivity Facility.

K

KB Kilobyte; 1024 bytes.

keyword
A language-defined word which identifies a clause. Examples of keywords are: IF, THEN, SAY.

L

LPA Link pack area.

M

MB Megabyte; 1 048 576 bytes.

MMS MVS message service.

module
An object code file whose external references have been resolved.

MVS Multiple Virtual Storage.

MVS/ESA
Multiple Virtual Storage/Enterprise System Architecture.

N

NLS National Language Support.

O

OBJECT output
Output produced by the IBM Compiler for REXX on System z licensed program when the OBJECT option is specified.

object program
A target program suitable for execution. An object program may or may not require linking. Contrast with source program. Object program is used on only.

OpenEdition
Pertaining to the elements of OS/390 that incorporate the UNIX interfaces standardized in POSIX.

OS/390 Operating System
The IBM licensed program OS/390 includes and integrates functions previously provided by many IBM software products. OS/390 is made up of elements and features. The elements deliver essential operating system functions. When you order OS/390, you receive all of the elements. The features are orderable with OS/390 and provide additional operating system functions.

P

partitioned data set (PDS)
A data set in direct access storage that is divided into partitions, called members, each of which can contain a program, part of a program, or data. A partitioned data set has a directory that contains information about each member. Each member can be accessed individually by its unique 1- to 8-character name.

partitioned data set extended (PDSE)
A partitioned data set managed by the Storage Management Subsystem (SMS). Similar to PDS, but with a number of enhancements.

PDF Program Development Facility.

phase In VSE, the smallest complete unit of executable code that can be loaded into virtual storage. It is the output of the linkage editor.

phrase
A language construct associated with a sub-keyword. Examples of phrases are: TO-phrase, WHILE-phrase.

S

SBCS Single-byte character set.

sequential data set
A data set in which the contents are arranged in successive physical order and are stored as an entity. The data set can contain data, text, a program, or part of a program. Contrast with partitioned data set (PDS).

SFS Shared file system.

SI The shift-in character (X'0F') indicating the end of a double-byte character string.

SO The shift-out character (X'0E') indicating the start of a double-byte character string.

SPI System Product Interpreter.

stub A code segment that transforms parameter lists from one format into another.

sub-keyword
A language-defined word occurring in (but not identifying) a clause. Examples of sub-keywords are: TO, BY, FOR, VALUE.

supervisor call (SVC)
A request that serves as the interface into operating system functions, such as allocating storage. The SVC protects the operating system from inappropriate user entry. All operating system requests must be handled by SVCs.

supervisor call instruction
An instruction that interrupts a program being executed and passes control to the supervisor so that it can perform a specific service indicated by the instruction.

SVC Supervisor call.

System Product Interpreter (SPI)
The component of the VM/XA SP operating system that processes procedures, XEDIT macros, and programs written in the Restructured Extended Executor (REXX) language.

T

TEXT file
An object-code file whose external references have not been resolved. This term is used on z/VM only.

token According to *SAA Common Programming Interface REXX Level 2: Reference*, a token is the unit of low-level syntax from which clauses are built.

TPA Transient program area.

transient program area (TPA)
In CMS, the virtual storage area occupying locations X'E000' to X'10000'. Some CMS commands and user programs can be executed in this area of CMS storage.

TSO/E Time Sharing Option Extensions.

V

Virtual Machine/Enterprise Systems Architecture (VM/ESA)
An IBM licensed program that manages the resources of a single computer so that multiple computing systems appear to exist. Each virtual machine is the functional equivalent of a real machine.

Virtual Machine/Extended Architecture System Product (VM/XA SP)
An IBM-licensed program with extended architecture support that manages the resources of a single computing system so that multiple computing systems (virtual machines) appear to exist.

VM/ESA
Virtual Machine/Enterprise Systems Architecture.

VM/XA SP
Virtual Machine/Extended Architecture System Product.

VSE/ESA
Virtual Storage Extended/Enterprise Systems Architecture.

X

XA Extended architecture.

Z

z/OS The IBM-licensed program z/OS is a highly secure, scalable, high-performance enterprise operating system on which to build and deploy Internet and Java-enabled applications, providing a comprehensive and diverse application execution environment.

z/VM The IBM-licensed program z/VM is the newest VM operating system and is based on the new 64-bit z/Architecture®. It provides a highly flexible test and

production environment for enterprises deploying the latest e-business solutions. Built upon the solid VM/ESA base, z/VM exploits the z/Architecture and helps enterprises meet their growing demands for multi-user server solutions with a broad range of support for operating system environments such as z/OS, OS/390, TPF, VSE/ESA, CMS, or Linux on System z.

Related Publications

This section lists each book in the REXX library. There is also a list of publications for other IBM products that you might use with REXX.

IBM Compiler and Library for REXX on System z Publications

The following books are part of the IBM Compiler and Library for REXX on System z publications:

IBM Compiler and Library for REXX on System z: User's Guide and Reference, SH19-8160, describes how to compile and run programs written in the REXX language.

IBM Compiler and Library for REXX on System z: Diagnosis Guide, SH19-8179, provides information for system programmers and other data processing professionals responsible for maintaining the IBM Compiler and Library for REXX on System z. It explains how to diagnose suspected errors in the product and how to report them to the appropriate IBM personnel.

IBM Compiler and Library for REXX on System z: Licensed Program Specifications, GH19-8161, describes the software and hardware requirements of IBM Compiler and Library for REXX on System z.

IBM Compiler for REXX on z/OS: Program Directory, GI10-8170 describes the requirements and installation of IBM Compiler for REXX on z/OS.

IBM Library for REXX on z/OS: Program Directory, GI10-9910 describes the requirements and installation of IBM Library for REXX on z/OS.

IBM Alternate Library for REXX on z/OS: Program Directory, GI10-3243 describes the requirements and installation of the IBM Alternate Library for REXX on z/OS.

You can also find the library of the IBM Compiler and Library for REXX on System z on the home page at: `http://www.ibm.com/software/awdtools/rexx/`

The unlicensed REXX books with prefix SH are also available on the following collection kits:

- *IBM eServer System z Online Library VM Collection* CD-ROM, SK2T-2067
- *IBM eServer System z Online Library VSE Collection* CD-ROM, SK2T-0060
- *IBM eServer System z Online Library z/OS Software Products Collection* CD-ROM, SK3T-4270

Other IBM Publications

These books contain information related to REXX or its related products.

ISPF Publications

ISPF V4 R2.0 Dialog Developer's Guide and Reference, SC34-4486

ISPF V4 R2.0 Services Guide, SC34-4485

ISPF V4 R2.0 User's Guide, SC34-4484

ISPF/PDF Guide (ISPF 3.2 & ISPF/PDF 3.1) for VM, SC34-4299

ISPF/PDF Guide and Reference V3.4 for MVS, SC34-4258

ISPF/PDF Guide Version 3, Release 2 for VM, SC34-4306

Learning REXX

- *TSO/E Version 2 REXX/MVS: User's Guide*, SC28-1882
- *VM/SP System Product Interpreter: User's Guide*, SC24-5238
- *VM/XA SP Interpreter: User's Guide*, SC23-0375
- *VM/ESA REXX/VM: User's Guide*, SC24-5465

REXX Reference

- *TSO/E Version 2 Procedures Language MVS/REXX*, SC28-1883
- *VM/XA SP Interpreter: Reference*, SC23-0374
- *VM/ESA Release 2 REXX/VM: Reference*, SC24-5466
- *IBM VSE/ESA REXX/VSE Reference*, SC33-6529, is interesting for experienced programmers, particularly those who have used a structured high-level language. They list the REXX messages and describes instructions, functions, debugging aids, and parsing.
- *Systems Application Architecture Common Programming Interface: REXX Level 2 Reference*, SC24-5549, describes the SAA REXX interface.

TSO/E and MVS/ESA Publications

- *TSO/E Version 2: Primer*, GC28-1879
- *TSO/E Version 2: Customization*, SC28-1872
- *TSO/E Version 2 REXX/MVS: User's Guide*, SC28-1882
- *TSO/E Version 2 REXX/MVS: Reference*, SC28-1883
- *TSO/E Version 2: Command Reference*, SC28-1881
- *MVS/DFP 3.3: Linkage Editor and Loader*, SC26-4564
- *MVS/ESA SP V4 Planning: Operations*, GC28-1625
- *MVS/ESA SP V4 Assembler Programming Guide*, GC28-1644

OpenEdition Publication

- *OpenEdition MVS Command Reference*, SC23-3014

VM/SP Publications

- *VM/SP CMS: Primer*, SC24-5236
- *VM/SP CMS: Primer for Line-Oriented Terminals*, SC24-5242
- *VM/SP CMS: User's Guide*, SC19-6210
- *VM/SP CMS: Command Reference*, SC19-6209
- *VM/SP System Product Editor: User's Guide*, SC24-5220
- *VM/SP: Administration*, SC24-5285
- *VM/SP System Messages and Codes*, SC19-6204

VM/XA SP Publications

VM/XA SP CMS: Primer, SC23-0368
VM/XA SP CMS: User's Guide, SC23-0356
VM/XA SP CMS: Command Reference, SC23-0354
VM/XA SP System Product Editor: User's Guide, SC23-0373
VM/XA SP: Administration, SC23-0353
VM/XA SP Interpreter Reference, SC23-0374
VM/XA SP CP Command Reference, SC23-0358

VM/ESA Publications

- *VM/ESA R2.2 REXX/VM User's Guide*, SC24-5465
- *VM/ESA V2R3.0 Diagnosis Guide*, GC24-5854
- *VM/ESA V2R4.0 REXX/VM Reference*, SC24-5770
- *VM/ESA V2R4.0 CP Command and Utility Reference*, SC24-5773

VSE/ESA Publication

- *VSE/ESA V2R1.0 System Control Statements*, SC33-6613
- *VSE/ESA V2R1.0 System Utilities*, SC33-6617
- *VSE/ESA V2R4.0 Guide for Solving Problems*, SC33-6710

C Publication

- *IBM C/370 Programming Guide Version 2 Release 1*, SC09-1384

CMS Publications

- *VM/ESA CMS: Primer*, SC24-5458
- *VM/ESA CMS: User's Guide*, SC24-5775
- *VM/ESA CMS: Command Reference*, SC24-5776
- *VM/ESA XEDIT: User's Guide*, SC24-5779
- *z/VM V4R1.0 CMS User's Guide*, SC24-6009
- *z/VM V4R3.0 CMS Command and Utility Reference*, SC24-6010
- *z/VM V4R3.0 CMS Planning and Administration*, SC24-6042

z/VM Publications

- *z/VM V3R1.0 CP Command and Utility Reference*, SC24-5967
- *z/VM V4R3.0 CP Command and Utility Reference*, SC24-6008
- *z/VM V4R3.0 Saved Segments Planning and Administration*, SC24-6056

z/OS Publications

- *z/OS V1R2.0 TSO/E REXX User's Guide*, SA22-7791
- *z/OS V1R4.0 TSO/E REXX Reference*, SA22-7790
- *z/OS V1R4.0 TSO/E Command Reference*, SA22-7782
- *z/OS V1R3.0 MVS Planning: Operations*, SA22-7601

OS/390 Publications

- *OS/390 V2R9.0 TSO/E REXX User's Guide*, SC28-1974
- *OS/390 V2R10.0 TSO/E REXX Reference*, SC28-1975

- *OS/390 V2R9.0 TSO/E System Programming Command Reference*, SC28-1972
- *OS/390 V2R10.0 MVS Planning: Operations*, GC28-1760

Index

Special characters

Numerics

A

B

C

D

E

F

G

H

I

J

L

M

N

O

S

T

U

V

W

X

Z

Printed in USA

SH19-8160-06