INTERFACING WITH REXX

ANTHONY RUDD
DATEV

Interfacing with REXX

ABSTRACT

This aim of this paper is to give an overview of the interfaces available in REXX, and to show how these interfaces can be used. This paper deals only with the MVS environment – however, most other environments (e.g. OS/2) offer similar facilities.

Although REXX is a powerful language in its own right (especially now that REXX compilers are available), there are certain features missing (for example, processing of VSAM files, direct SQL processing). Furthermore, there are REXX features (e.g. parsing) that can simplify the processing of programs written in conventional languages (Assembler, PL/I, COBOL, etc).

REXX caters for both these situations by providing interfaces. There are two forms of interface:

- high level
- low level.

High-level interfaces are invoked directly from a REXX exec. *Low-level* interfaces are those routines (services) provided by the REXX processor.

There are three forms of high-level interface:

- function
- (address) environment
- program invocation.

A **function** can be written in either REXX or a conventional programming language. To improve performance functions can be physically grouped together as a **function package**. A function is invoked by its name, and serves to extend the standard functions provided with REXX (e.g. `WORD`, `WORDINDEX`). A function may be passed arguments, and may return a value (the **function return value**).
An **address environment** can only be written in a conventional programming language. High-level interfaces may (and normally will) make use of low-level REXX interfaces. REXX as an address environment processes any non-REXX statements. A user-address-environment extends the standard REXX environments (e.g. `MVS`, `TSO`).
A **program invocation** is made with the `LINK` or `ATTACH` command.

1. INTRODUCTION

REXX implementations offer many interfaces for using REXX services from programs written in conventional programming languages. This paper describes only those interfaces of interest to the applications developer – there are a number of other interfaces which can be used by systems specialists to customise the system.

The interfaces can be grouped into the following categories:

- program invocation of a REXX exec
- programs as REXX functions (and the grouping of such programs into function packages)
- program access to REXX variables
- stack operations
- general service routines.

1.1 High-level REXX interfaces

High-level REXX interfaces are invoked directly from REXX execs. Such interfaces can be regarded as being extensions to the REXX language.

Standard address environments:

- ISPEXEC (ISPF Dialog Manager)
- ISREDIT (ISPF/PDF Edit Macro)
- DB2 (program that runs in the DB2 environment)
- QMF.

Typical user environments:

• REXXDB2	process SQL query
• REXXVSAM	process VSAM dataset.

Representative examples of user functions:

- SHIFT function (perform bit-shift on REXX variable)
- SIN function (calculate trigonometric sine value).

1.2 Low-level REXX interfaces

The most useful low-level REXX interface routines:

• IRXEXCOM	access REXX variables
• IRXEXEC	invoke REXX exec
• IRXINIT	process REXX environment
• IRXJCL	invoke REXX exec (batch mode)
• IRXLOAD	load exec
• IRXRLT	get result
• IRXSTK	access REXX stack.

REXX programs (i.e. programs that make use of REXX services) can access certain REXX control blocks:

- Argument List (AL). The Argument List describes the input arguments passed to a function. Each argument passed to the function has one Argument List entry (consisting of two words) in the Argument List. The Argument List is terminated with two words each containing binary -1 (X'F...F').
- External Functions Parameter List (EFPL). The EFPL describes the external arguments for a function; the pointer to the input arguments and to the result field. The input arguments are defined in the Argument List. The result is defined in the Evaluation Block (EVALBLOCK).
- Environment Block (ENVBLOCK). The ENVBLOCK describes the REXX operating environment. An ENVBLOCK is automatically created when the REXX environment is initiated. The ENVBLOCK is principally used by the application developer to obtain error messages.
- Evaluation Block (EVALBLOCK). The EVALBLOCK describes the result passed back from a function.
- Execution Block (EXECBLK). The EXECBLK specifies the information necessary to locate an external exec.
- In-Storage Control Block (INSTBLK). The INSTBLK describes (address and length) the individual records (lines) of a REXX exec contained in main-storage. The IRXLOAD service can be used to build the INSTBLK.
- Shared Variable (Request) Block (SHVBLOCK). The SHVBLOCK describes the variable to be accessed from the variable pool. SHVBLOCKs can be chained together.

- Vector of External Entry Points (VEEP). The VEEP contains the addresses of the external REXX service routines.

Most of these control blocks are read-only, although some can be altered (INSTBLK, SHVBLOCK).

2. HIGH-LEVEL INTERFACES

2.1 MVS-TSO/E implementation

The MVS-TSO/E implementation allows a REXX exec to run in several environments, both dialogue and batch. From within this invoking environment the `ADDRESS` instruction can be used to select a sub-environment for non-REXX statements. This sub-environment is the interface to other components, for example, the `ISPEXEC` sub-environment for ISPF Dialog Manager services.

2.1.1 Invocation
A REXX exec can be invoked from:

- TSO/ISPF dialogue
- TSO batch
- MVS batch.

The REXX exec is stored as member of a partitioned dataset (library). The name of this dataset must be made available to the REXX interpreter.

2.1.2 Linkage to host (MVS-TSO/E) environment
A REXX exec can link to components from the host environment. The `ADDRESS` instruction is used to set the host environment.

Example:
```
      ADDRESS TSO "TIME";
```
invokes the TSO `TIME` command.

2.1.3 Linkage to programs
A REXX exec can pass control to a program written in a conventional programming language. The program is invoked with either the `ATTACH` or `LINK` host command. The `ATTACH` command invokes the program asynchronously (i.e. as a separate task), the `LINK` command invokes the program synchronously. The program is loaded from the program (load) library assigned to the environment.

The program may be passed a single parameter, which may contain subparameters. The invoked program receives two parameters on entry:

- the address of the parameter string;
- the length of the parameter string (full-word).

Note: This is not the standard MVS program linkage convention. TSO/E V2R3.1 offers new facilities: `LINKMVS`, `ATTCHMVS`, `LINKPGM`, `ATTCHPGM`. These pass multiple parameters according to MVS conventions.

2.1.4 Interface with ISPEXEC (ISPF Dialog Manager)

REXX execs invoked from the TSO/ISPF environment can use the ADDRESS ISPEXEC instruction to access ISPEXEC (ISPF Dialog Manager) services. The parameters for the ISPEXEC service are passed as a normal REXX string, i.e. may be a literal, symbol or mixture. However, ISPEXEC accepts only upper-case characters. The return code from the ISPEXEC service is set into the RC special variable.
REXX execs and ISPF Dialog Manager share the same function pool, with two restrictions:

- variable names longer than 8 characters cannot be used in ISPF;
- the VGET and VPUT services cannot be used with stem variables.

Example:

```
    panname = "PAN1";
    ADDRESS ISPEXEC "DISPLAY PANEL("panname")";
    SAY RC;
```

uses ISPEXEC to display panel PAN1, the return code from the service is displayed.

2.1.5 Interface with ISREDIT (ISPF/PDF Edit macro)

The ISPF/PDF Editor can invoke a procedure to perform processing on a dataset – this procedure is called an Edit macro and can be a REXX exec. The ADDRESS ISREDIT instruction invokes Edit macro services. The parameters for the ISREDIT service are passed as a normal REXX string, i.e. may be a literal, symbol or mixture. The return code from the ISREDIT service is set into the RC special variable.

Edit macros can make full use of REXX facilities. The powerful string processing features of REXX make it an ideal language for the implementation of Edit macros.

Example:

```
    /* REXX Edit macro */
    ADDRESS ISREDIT;
    "MACRO (STRING)"
    "FIND" string "NEXT"
    IF RC <> 0 THEN SAY "search argument not found";
    "END" /* terminate macro */
```

2.1.6 Interface with DB2 (Database 2)

The TSO DSN command is used in initiate the DB2 session. The DB2 RUN subcommand is used to invoke a program which is to run in the DB2 environment.

The DB2 subcommands to invoke the program, and to terminate the DB2 session, RUN and END, respectively, are set into the stack in the required order before the DB2 session is initiated.

Note: The subcommands cannot be passed directly, as is the case with CLISTs.

Example:

```
    QUEUE "RUN PROGRAM(TDB2PGM) PLAN(TDB2PLN) LIB('USER.RUNLIB.LOAD')";
    QUEUE "END";
    ADDRESS TSO "%DSN"; /* invoke DB2 */
```

or

```
    ADDRESS ISPEXEC "SELECT CMD(%DSN)"; /* invoke DB2 with ISPF services */
```

2.1.7 Interface with QMF (Query Management Facility)

With QMF Version 3 Release 1 the SAA Callable Interface (DSQCIx) is now available for REXX. This means that there are now two methods of invoking QMF:

- Callable Interface
- Command Interface.

The Callable Interface:

- ISPF not required
- QMF does not need to be active.

The Command Interface:

- requires ISPF
- requires QMF to be active.

The Command Interface invocation of QMF is more involved; two steps are required:

- initiate the QMF session (program DSQQMFE), and execute a QMF procedure;
- this QMF procedure passes control to a REXX exec, which in turn uses the QMF Command Interface (CI, program DSQCCI) to process a QMF command.

The following three QMF examples all perform the same function: run the QMF query Q1.

2.1.7.1 Callable Interface - Version 1

Example:

```
/* REXX - QMF Callable Interface */
ADDRESS "TSO";
/* allocate QMF files */
"ALLOC F(DSQDEBUG) DUMMY REUS"
"ALLOC F(DSQPNLE) DSN('qmf.test.dsqpnle') SHR REUS"
"ALLOC F(ADMGGMAP) DSN('qmf.test.dsqmape') SHR REUS"
CALL DSQCIX "START (DSQSSUBS=DB2T,DSQSMODE=INTERACTIVE"; /* start QMF */
CALL TESTRC;
CALL DSQCIX "RUN QUERY Q1"; /* run query */
CALL TESTRC;
CALL DSQCIX "EXIT"; /* terminate QMF */
CALL TESTRC;
EXIT; /* terminate exec */
TESTRC:
  IF DSQ_RETURN_CODE > 4 THEN DO;
    SAY "QMF RC:" DSQ_RETURN_CODE;
    SAY  DSQ_MESSAGE_TEXT;
  END;
RETURN;
```

2.1.7.2 Callable Interface - Version 2

Example:

```
/* REXX - QMF Callable Interface */
ADDRESS "TSO";
"ALLOC F(DSQDEBUG) DUMMY REUS"
"ALLOC F(DSQPNLE) DSN('qmf.test.dsqpnle') SHR REUS"
"ALLOC F(ADMGGMAP) DSN('qmf.test.dsqmape') SHR REUS"
CALL DSQCIX "START (DSQSSUBS=DB2T,DSQSMODE=INTERACTIVE"; /* start QMF */
CALL TESTRC;
ADDRESS "QRW"; /* QMF environment */
"RUN QUERY Q1" /* run query */
CALL TESTRC;
"EXIT" /* terminate QMF */
CALL TESTRC;
EXIT; /* terminate exec */
TESTRC:
  IF DSQ_RETURN_CODE > 4 THEN DO;
    SAY "QMF RC:" DSQ_RETURN_CODE;
    SAY  DSQ_MESSAGE_TEXT;
  END;
RETURN;
```

Version 2 is basically the same as version 1, except that the QMF environment QRW is used.

2.1.7.3 Command Interface

Example:

Phase 1 - Initiate QMF session (DSQQMFE program). The following exec allocates the (minumum) QMF files, initiates QMF session and invokes the QMF procedure QP1:

```
/* REXX - QMF COMMAND INTERFACE */
ADDRESS "TSO";
"ALLOC F(DSQDEBUG) DUMMY REUS"
"ALLOC F(DSQPNLE) DSN('qmf.test.dsqpnle') SHR REUS"
"ALLOC F(ADMGGMAP) DSN('qmf.test.dsqmape') SHR REUS"
ADDRESS "ISPEXEC";
"SELECT PGM(DSQQMFE) NEWAPPL(DSQE) PARM(S=DB2T,I=USER.QP1)"
```

Phase 2 - The QMF procedure QP1 passes control to the TSO procedure (REXX exec) QR2:

```
TSO %QR2
```

Phase 3 - The QR2 exec invokes the QMF Command Interface (DSQCCI program) to process the specified QMF commands (this REXX exec actually causes the QMF query (Q1) to be run):

```
/* REXX */
ADDRESS "ISPEXEC";
"SELECT PGM(DSQCCI) PARM(RUN Q1)"
"SELECT PGM(DSQCCI) PARM(INTERACT)"
"SELECT PGM(DSQCCI) PARM(EXIT)" /* terminate QMF */
```

Fig. 1 illustrates the use of the QMF Command Interface.

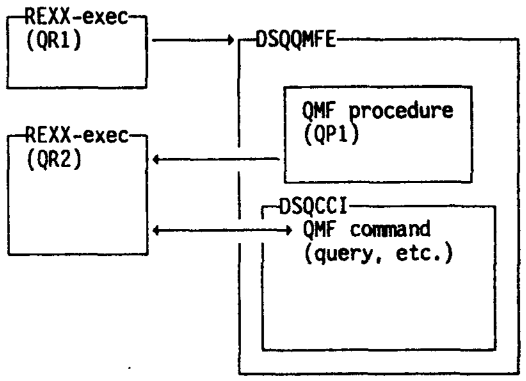


Fig. 1 – Schematic use of QMF Command Interface

2.2 User interfaces

User programs can be invoked as:

- function (e.g. `x = funct(p1,p2,...);`)
- host command (e.g. `ADDRESS userenv; "cmd p1 p2 ...";`)
- program (e.g. `LINK "pgm p1 p2 ...";`).

The most suitable interface depends on such aspects as:

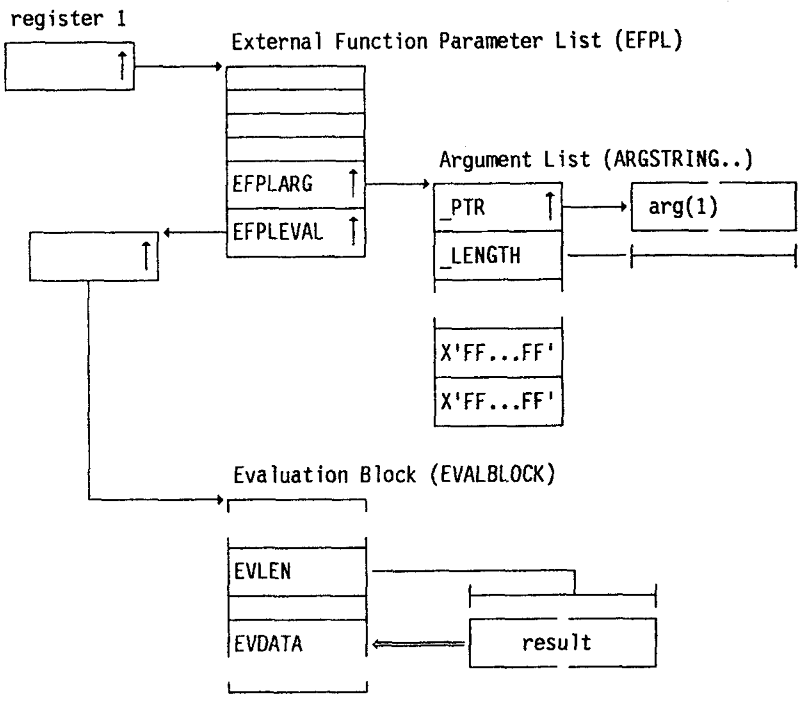
- the form of the arguments to be passed (a natural calling sequence);
- the form of the results to be returned;
- the programming language used.

2.2.1 Function interface

A user function receives zero or more parameters (parsed in the Argument List), and must return a function result (in the Evaluation Block). Fig. 2 illustrates the function interface.

Example:

```
    y = SIN(x);
```



register 1
External Function Parameter List (EFPL)
EFPLARG
EFPLEVAL
Argument List (ARGSTRING..)
_PTR
arg(1)
_LENGTH
X'FF...FF'
X'FF...FF'
Evaluation Block (EVALBLOCK)
EVLEN
EVDATA
result

Fig. 2 – Function interface

2.2.2 Host command interface

A host command is processed by the currently active environment, i.e. the environment activated with the ADDRESS command. All non-REXX commands are passed to the host command environment. A host command cannot directly return any data (other than a return code for the command) – data can be passed back in the stack or as (stem) variables. Fig. 3 illustrates the host command interface.

Many installations have a single router program that passes control to the appropriate processing program.

Example:

```
ADDRESS USER;
"REXXVSAM READ DDNAME GE ALPHA(STEM A.";
```

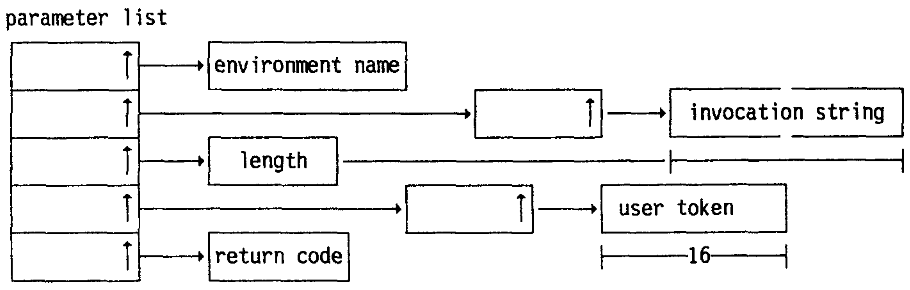


Fig. 3 – Host Command Environment Interface

2.2.3 Program invocation interface

A program can be directly invoked with the ATTACH (asynchronous) or LINK (synchronous) command. This is the only way of invoking a C/370 Version 1 program. *Note*: The parameters passed to a program do not conform to the MVS calling convention. Fig. 4 illustrates the program invocation interface.

Example:

```
ADDRESS LINK "ALPHA BETA GAMMA";
```

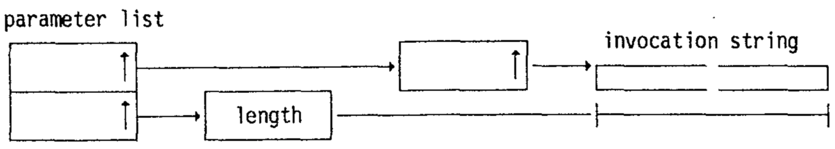


Fig. 4 – Program invocation (via LINK, ATTACH)

3. LOW-LEVEL INTERFACES

3.1 General conditions

The low-level interfaces are subject to the following conditions:

- Programs can be written in Assembler, COBOL, PL/I, and C/370 Version 2 (to a limited extent Version 1). Not all high-level programming languages provide full support for all the required facilities.
- Programs using REXX services must use 31-bit addressing (AMODE 31).
- Numeric fields are in binary format, either fullword (4 bytes) or halfword (2 bytes).
- Standard calling conventions are used:
  - register 15 – entry point address;
  - register 14 – return address;
  - register 13 – address of save-area.
- The return code is passed back in register 15 (PL/I: PLIRETV variable, COBOL: RETURN-CODE special register, C: function return value). Many routines also set an error message in the Environment Block.
- Parameter address lists passed in register 1 must have the high-order bit set in the last address word.
- Standard macros (in the SYS1.MACLIB system macro library) are available for use by Assembler programs to map the more important control blocks. Programs written in high-level programming languages (e.g. COBOL, PL/I) must themselves define the required control block structures – Fig. 5 shows the equivalent field types in various programming languages.

type	Assembler	PL/I	COBOL VS II	C
address	A	PTR	POINTER	*
character string	CLn	CHAR(n)	PIC X(n)	char [n+1]
fullword	F	FIXED BIN(31)	PIC S9(9) COMP	int
halfword	H	FIXED BIN(15)	PIC S9(4) COMP	short
hexadecimal	X	BIT(8)	X' ... '	0X

Fig. 5 – Equivalent field types

240

Notes:
1. Only the most important information for the interfaces is described in this paper – the appropriate manual should be consulted if a more detailed description is required.
2. The entry *symbol..* in diagrams denotes that *symbol* is used as prefix to the field names in the corresponding block. The diagrams show only the significant fields. Any fillers at the end of field layout figures are omitted.

Sample PL/I program:
```
BETA: PROC OPTIONS(MAIN);
DCL IRXSTK EXTERNAL OPTIONS(RETCODE,INTER,ASSEMBLER);
DCL PLIRETV BUILTIN;
DCL 1 FC CHAR(8);              /* function code */
DCL 1 ADDR_ELEM PTR;           /* pointer to data */
DCL 1 LEN_ELEM FIXED BIN(31); /* length of data */
DCL 1 FRC FIXED BIN(31);       /* function return code */
DCL 1 ELEM CHAR(256) BASED(ADDR_ELEM); /* data */
FC = 'PULL';                   /* function */
FETCH IRXSTK;                  /* load address of entry point */
CALL IRXSTK(FC,ADDR_ELEM,LEN_ELEM,FRC);
IF PLIRETV = 0 THEN PUT SKIP LIST (SUBSTR(ELEM,1,LEN_ELEM));
END;
```
This PL/I program retrieves and displays the next element from the data stack.

3.2 Invocation of a REXX exec

There are three ways of an application program to invoke a REXX exec:

- using the IRXJCL program;
- using the TSO Service Facility (IJKEFTSR program);
- using the IRXEXEC program.

These three methods are listed in order of ease of use. This is also the order of increasing flexibility, e.g. the IRXEXEC program interface offers more flexibility than the IRXJCL program interface but is more difficult to use.

3.2.1 Interface from programs to batch REXX (IRXJCL)

Programs written in a conventional language can use IRXJCL to invoke a REXX exec. Fig. 6 shows the form of the parameter as passed from the invoking program.

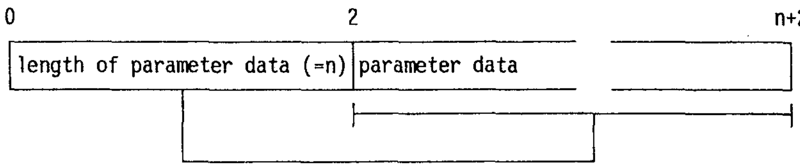


Fig. 6 – Format of parameter passed to IRXJCL

3.2.2 Invocation of a REXX exec using the TSO Service Facility (IJKEFTSR)

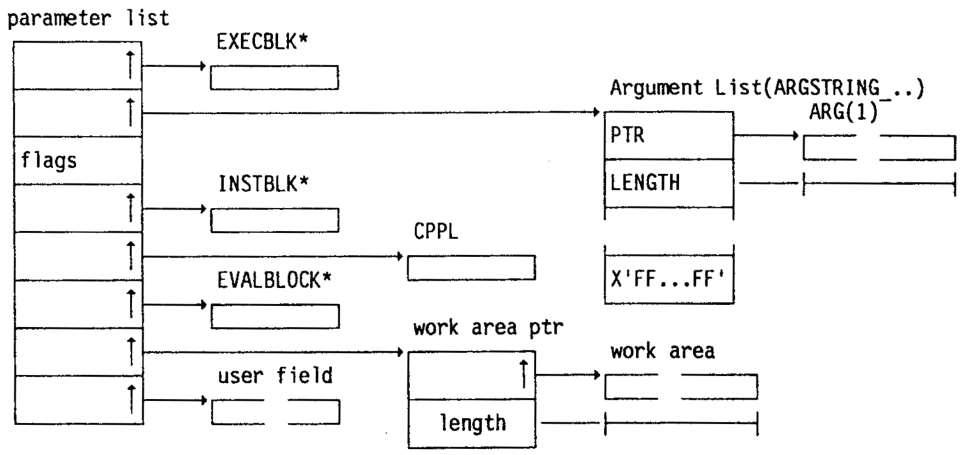
REXX execs can also be invoked from the TSO environment (either dialogue or batch) with the TSO Service Facility (IJKEFTSR program) – the TSO Service Facility has the alias TSOLNK.

3.2.3 Interface from program to REXX processor (IRXEXEC)

The IRXEXEC routine is the most flexible method of invoking a REXX exec:

- it can invoke either an internal or external exec;
- it can pass more than one parameter.

If the INSTBLK address is zero, an internal exec is invoked, otherwise an external exec is loaded using the information in the EXECBLK (EXEC_BLK_DDNAME – library ddname, EXEC_BLK_MEMBER – member name). Fig. 7 illustrates the IRXEXEC service.

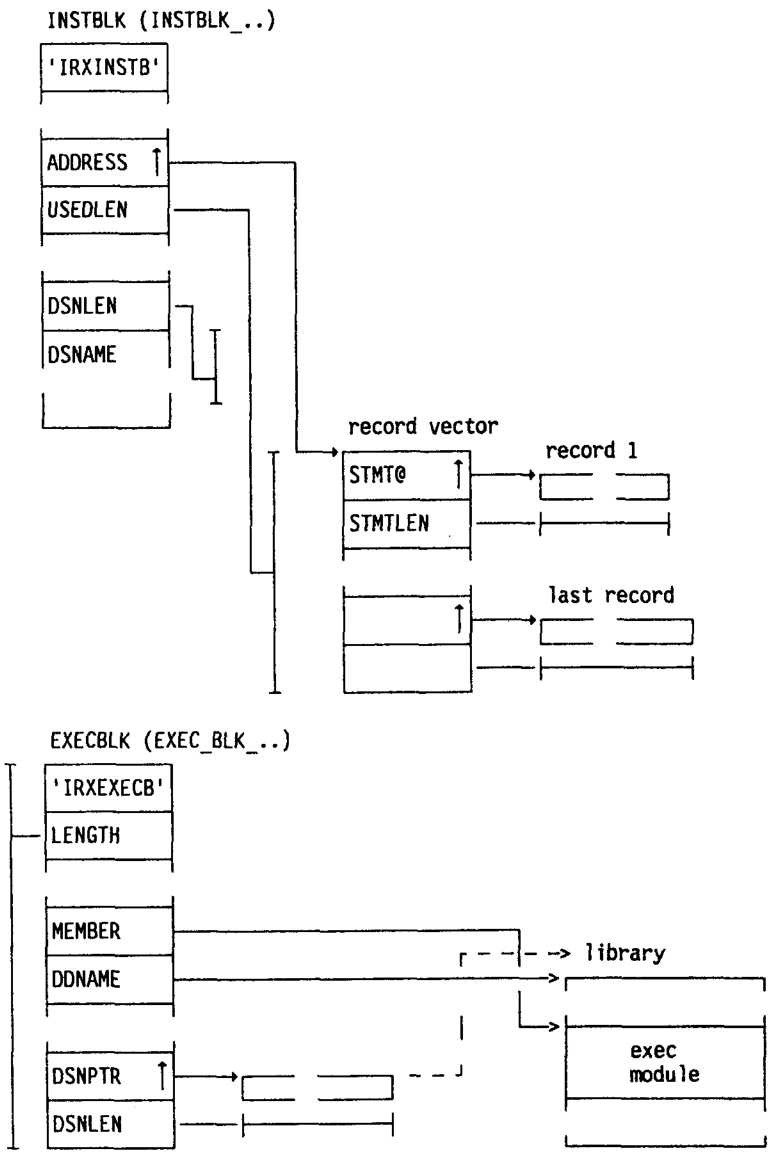


*•Detailed diagram follows (in part 2)
Fig. 7 – IRXEXEC interface (part 1 of 2)

3.3 Program access to REXX variables (IRXEXCOM service)

Programs running in a REXX environment can use the IRXEXCOM service to access variables in the environment pool. Fig. 8 illustrates the IRXEXCOM service. The following functions are available:

- copy value
- set variable
- drop variable
- retrieve symbolic name
- set symbolic name
- drop symbolic name
- fetch next variable
- fetch user data.



– - implicit (only informative)

Fig. 7 – IRXEXEC interface (part 2 of 2)

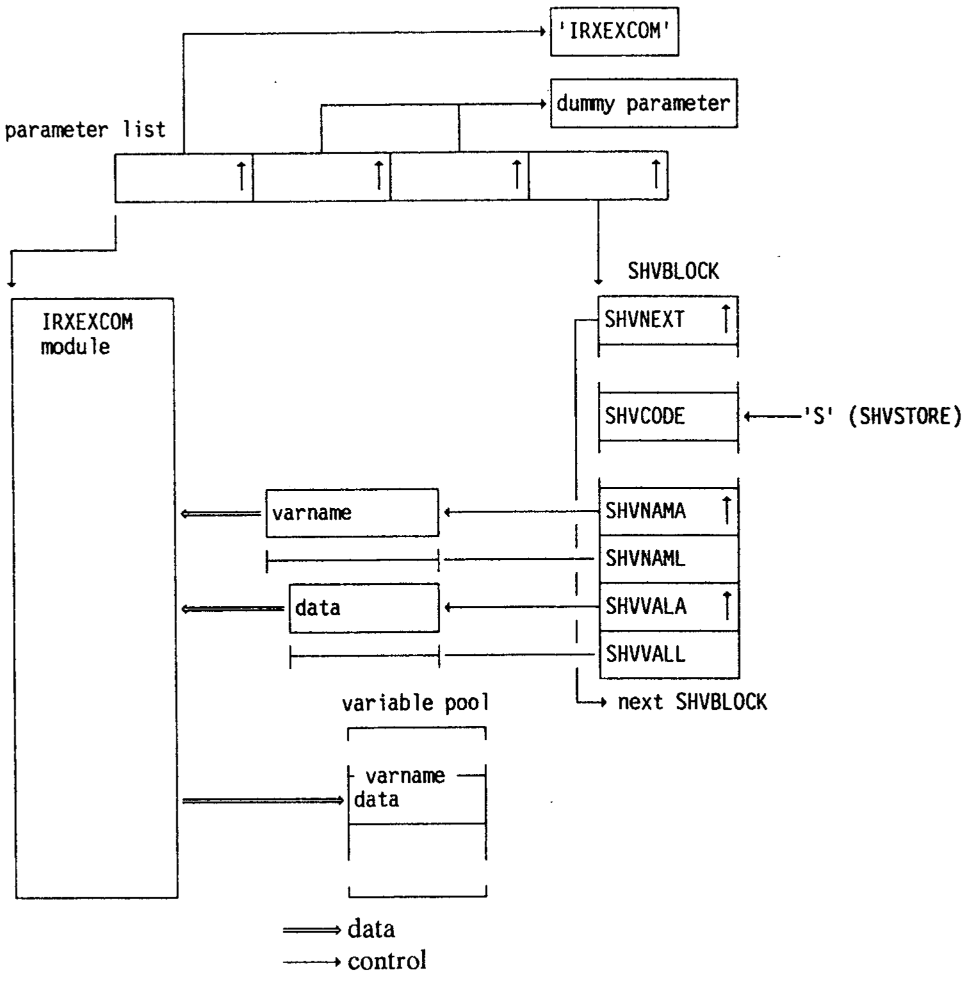


Fig. 8 – IRXEXCOM service to store a variable

3.4 Stack processing (IRXSTK service)

Programs can use the IRXSTK service to perform processing on the current stack. The operations:

- DELSTACK
- DROPBUF
- MAKEBUF
- NEWSTACK
- PULL
- PUSH
- QELEM
- QSTACK
- QUEUE
- QUEUED

have their standard function.

The two operations:

- `DROPTERM`
- `MAKETERM`

are used by system routines to coordinate stack access from TSO and ISPF. These operations should not be used by application programs.

3.5 Function interface

Programs written in a conventional programming language and stored as a load module in a library can be invoked as external REXX functions or subroutines. A function differs from a a subroutine in that it must return a value.

3.5.1 Function package

For reasons of efficiency, functions can be grouped together as a **function package** – function packages are searched before the other libraries. Three classes of function package can be defined:

- user function package
- local function package
- system function package.

The system support personnel will usually be responsible for the local and system function packages, and so they will not be discussed in this paper, although the general logic is the same as for the user function package.

A function package consists of a **function package directory** and functions. The function package directory is a load module contained in the load library – `IRXFUSER` is the standard name for the load module defining the user function package. Fig. 9 shows the diagrammatic representation of a function package.

The function package directory contains the names of the functions (subroutines) as invoked from a REXX exec and a pointer to the appropriate load module. This pointer can have one of two forms:

- The address of a load module which has been linkage edited together with the function package directory – such load modules must be serially reusable, as they are loaded only once.
- The name of a load module which will be loaded from the specified load library.

3.5.1.1 Function directory

The Function Directory defines the functions contained in a function package. The Function Directory consists of a header and one entry for each function contained in the Function Directory.

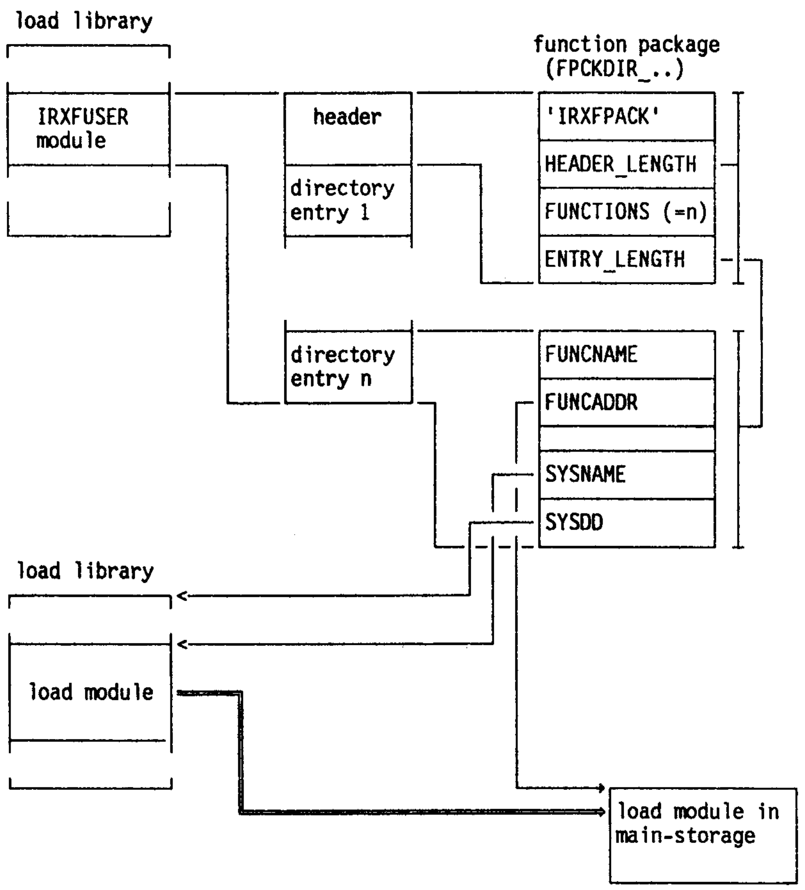


Fig. 9 – Diagrammatic representation of a function package

Sample Function Package Directory:

```
IRXFUSER CSECT
         DC    CL8'IRXFPACK'       identifier
         DC    AL4(SOD-IRXFUSER)   length of header
         DC    AL4(ND)             no. of entries in directory
         DC    FL4'0'              zero
         DC    AL4(LDE)            entry length
SOD      EQU   *                   start of directory (first entry)
         DC    CL8'FDIGIT'         function name
         DC    VL4(FDIGIT)         address, reserved
         DC    FL4'0'              reserved
         DC    CL8' '              name of entry point
         DC    CL8' '              DD-name of load library
LDE      EQU   *-SOD               length of directory entry
* next entry
         DC    CL8'FGEDATE'        function name
         DC    AL4(0)              address, 0 = load from library
         DC    FL4'0'              reserved
         DC    CL8'FGEDATE'        name of entry point
         DC    CL8'ISPLLIB'        DD-name of load library
EOD      EQU   *                   end of directory
ND       EQU   (EOD-SOD)/LDE       no. of directory entries
         END
```

This sample Function Package Directory contains two functions:

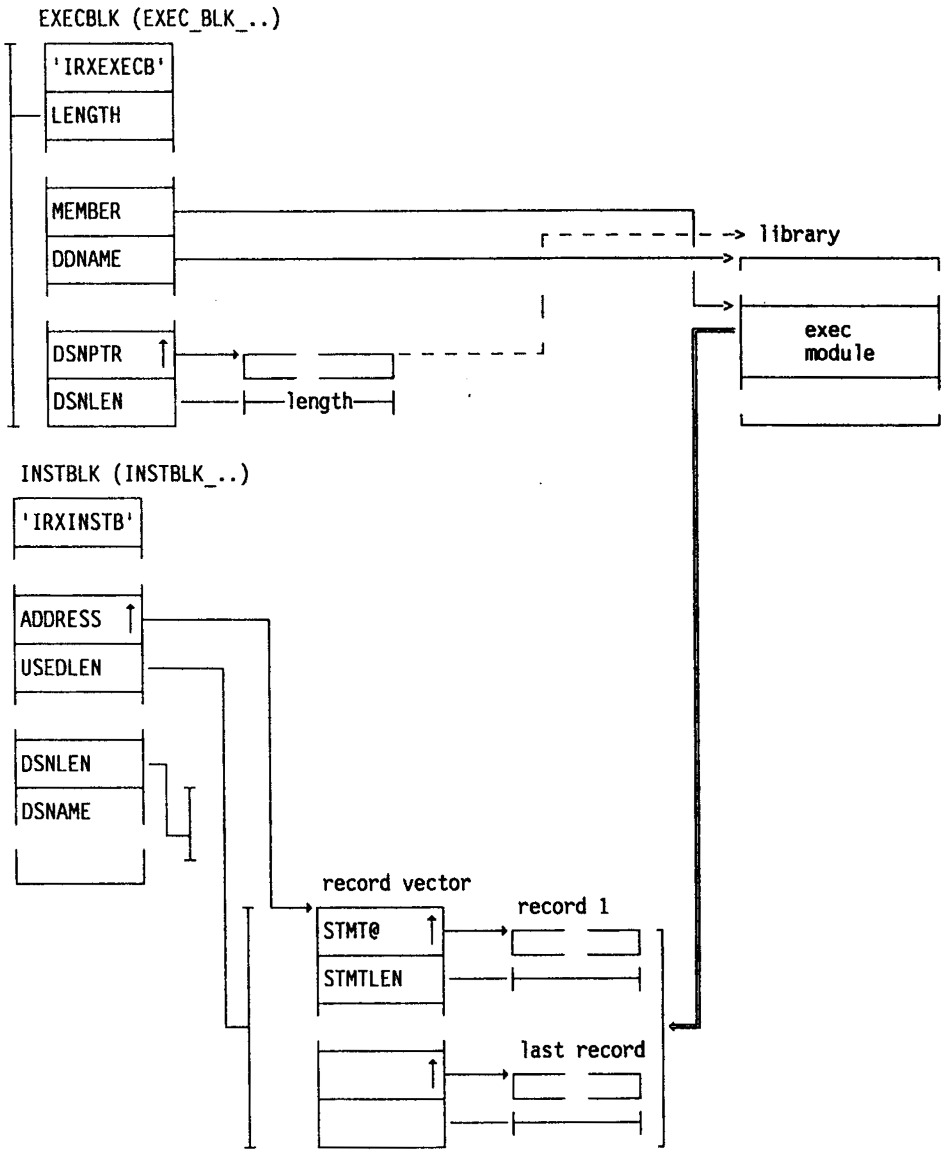
- FDIGIT – linkage edited with the Function Package Directory;
- FGEDATE – to be loaded from the ISPLLIB library.

3.6 Load routine – IRXLOAD service

The load routine (IRXLOAD) can be used in several ways:

- load an exec into main-storage – this creates the In-Storage Control Block for the exec;
- check whether an exec is currently loaded in main-storage;
- free an exec;
- close a file from which execs have been loaded.

IRXLOAD is also used when the language processor environment is initialised and terminated. Fig. 10 illustrates the IRXLOAD service (load function).



– - implicit (only informative)

Fig. 10 – IRXLOAD interface

3.7 Initialisation routine – IRXINIT service

The initialisation routine (IRXINIT) can be used in two ways:

- initialise a new environment;
- obtain the address of the current Environment Block.

The first function is normally only used by system specialists. The second function is used principally to access an error message which has been set by a service routine. Fig. 11 illustrates the ENVBLOCK.

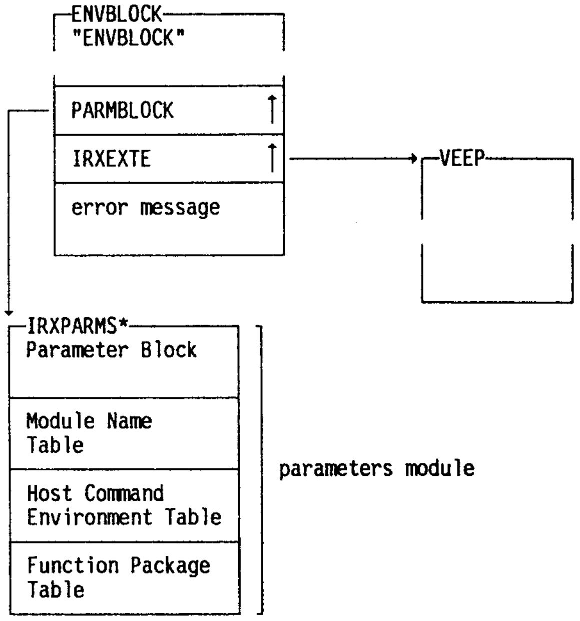


Fig. 11 – ENVBLOCK

3.8. Get result – IRXRLT service

The get result routine (IRXRLT) can be used in two ways:

- fetch result set by an exec invoked with the IRXEXEC service;
- allocate an Evaluation Block of the specified size.

This paper is adapted from my book:
Practical Usage of REXX
published in 1990 by Ellis Horwood Limited, Chichester.

Anthony Rudd, April 1992.