BREXX/370 User's Guide

Release: V2R5M1

Peter Jacob (pej), Mike Großmann (mig)

Jan 06, 2023

Table of Contents

Chapter 1

Installation Guide

1.1 Introduction

This document covers the installation process of BREXX/370.

BREXX/370 is provided as-is, please test carefully in test systems only!

BREXX/370 is not the same as IBM's REXX; there are many similarities, but also differences, especially when using MVS-specific functions.

The next TK4- Update 9 release contains BREXX/370.

1.2 Prerequisites

1.2.1 MVS TK4- / MVS/CE

This version of BREXX/370 has been developed and tested within Jürgen Winkelmann's TK4- (https://wotho.ethz.ch/tk4-/). It also comes pre-installed in MVS/CE (https://github.com/MVS-sysgen/sysgen). It may work in other versions of MVS, such as https://www.jaymoseley.com/hercules/installMVS/iSYSGENv7.htm but can't be guaranteed.

1.2.2 Non MVS TK4- Installation

Users who run a non TK4- MVS installation should pay attention to the following differences:

XMIT RECEIVE STEPLIB DD Statement

It might be necessary to add a STEPLIB DD statement to locate the library containing the RECV370:

```
//RECV370
//STEPLIB
EXEC PGM=RECV370
DD
DSN=library
```

Please add it to the Jobs where needed.

IN TK4- RECV370 is contained in a system library; therefore, a STEPLIB DD statement is not needed!

REGION SIZE

For non-TK4- MVS versions it might be necessary to reduce the REGION size parameter to 4 MB or 6MB, as MVS may reject the REGION=8192Kparameter with the message: "REGION UNAVAILABLE, ERROR CODE=20". TK4- supports 8MB regions size:

```
//stepname EXEC PGM=xxxxx,REGION=6144K
ISPF support (optional)
```

BREXX/370 also supports Wally McLaughlin's version of ISPF and its contained SPF panels.

1.2.3 Recommendations

We recommend testing BREXX/370 in an isolated test system to avoid any impact on your current system. To achieve this, you can easily copy the entire Hercules/MVS directory to another location and install BREXX/370 there.

1.3 Preparation of your target MVS38J System

1.3.1 BREXX Catalogue

Make sure that your MVS system has a BREXX Alias pointing to a user catalogue defined in the master catalogue. To determine it, run the command:

```
listcat entries('brexx') all
```

The result must look like this:

```
ALIAS ---------- BREXX
     IN-CAT --- SYS1.VSAM.MASTER.CATALOG
     HISTORY
       RELEASE----------------2
     ASSOCIATIONS
       USERCAT--UCPUB001
```

If the BREXX Alias is not defined, add it:

```
//ADDBREXX EXEC PGM=IDCAMS
//SYSPRINT DD SYSOUT=*
//SYSIN    DD *
  DEFINE ALIAS (NAME(BREXX) RELATE(your-user-catalog))
```

If the submitted job is not running, it might be necessary to enter the password of the master-catalogue in the MVS console (in TK4- not needed).

If you omit this step, all BREXX data sets are catalogued in the Master Catalog. In this case, it may require the use of the Master Catalog password during the catalogue process. If you are running TK4- you do not see such requests as RAKF is providing the access authorisation of the Master Catalog, which therefore is not password protected. In the default TK4- configuration, only users HERC01 and HERC02 are authorised to update the master catalogue.

Important All JCLs in the installation and sample library contain now a *NOTIFY=&SYSUID* parameter in the JOB card. If the patch, to resolve it during the Submit process by the current user-id, is not applied, you need to change &SYSUID to your userid, or remove it from the JOB card!

The patch can be found on: http://prycroft6.com.au/vs2mods/index.html#zp60034

Make sure that dataset *BREXX.V2R5M1.INSTALL* is not already catalogued from a previous run. It is the recommended dataset name and will be created during the receiving process of RECV370.

Important If a previous version of this dataset name is still catalogued, the new version ends up as not catalogued: with a *NOT CATLG 2* message! The Job output does not reveal by a ccod. Any later job which is accessing *BREXX.V2R5M1.INSTALL* will use the old version of the dataset.

1.4 Installation

1.4.1 Step 0 - Unzip BREXX/370 Installation File

The ZIP installation file consists of several files:

- BREXX370_Users_guide.pdf - This user guide
- BREXX370_V2R5M1.XMIT - XMIT File containing BREXX modules and Installation JCL

1.4.2 Step 1 - Upload XMIT File

Use the appropriate upload facility of your terminal emulation. Such as IND$FILE or using *rdrprep* and inline JCL.

The file created during upload must have *RECFM FB* and *LRECL 80*. If the DCB does not match, the subsequent unpacking process fails.

1.4.3 Step 2 - Unpack XMIT File

Unpack the XMIT file with an appropriate JCL. If you don't have one you can use the following sample, just cut and paste it in one of your JCL libraries.

```
//BRXXREC JOB 'XMIT RECEIVE',CLASS=A,MSGCLASS=H
//* ------------------------------------------------------------
//* RECEIVE XMIT FILE AND CREATE DSN OR PDS
//* ------------------------------------------------------------
//RECV370  EXEC PGM=RECV370,REGION=8192K
//RECVLOG  DD SYSOUT=*
//XMITIN   DD DSN=HERC01.BREXX.version.XMIT,DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSUT1   DD DSN=&&XMIT2,
//         UNIT=3390,
//         SPACE=(TRK,(300,60)),
//         DISP=(NEW,DELETE,DELETE)
//SYSUT2   DD DSN=BREXX.version.INSTALL,
//         UNIT=3390,
//         SPACE=(TRK,(300,60,20)),
//         DISP=(NEW,CATLG,CATLG)
//SYSIN DD DUMMY
```

- **HERC01.UPLOAD.XMIT** represents the uploaded XMIT File - please change it accordingly to the name you have chosen during the upload process.
- **BREXX.V2R5M1.INSTALL** is the name of the unpacked library (created during the UNPACK process). It is recommendable to remain with this DSN as it is used in later processes. **Make sure there is no previous version of this PDS catalogued.**

Important If you use a different JCL to unpack the XMIT file, use *UNIT=3390* in the JCL. The unit type 3390 was the only reliably UNIT that ran in all tested TK4- environments. Other units may sometimes lead to various errors during the unpacking process.

Once the submitted job has successfully unpacked the XMIT file into the target PDS, you can proceed with STEP 3. The created library *BREXX.version.INSTALL* contains all JCL to pursue with unpacking and installing.

The next steps make usage of the unpacked library (in this example *BREXX.V2R5M1.INSTALL*)

Please run the JCL in the given order (refer to the **Step x** reference in the table). Submit Step 3 as the first JCL of the installation sequence. Entries without a Step reference are used from the JCLs as input datasets.

Filename	Description	Used in Step
$CLEANUP	Cleanup: Remove unnecessary installation files	-> Step 6
$INSTALL	Install BREXX/370	-> Step 4
$README	Read me file	
$TESTRX	Test job to verify the BREXX/370 installation	-> Step 5
$UNPACK	Unpack subsequent libraries	-> Step 3
BUILD	Contains BREXX/370 Version and date and XMIT date	
CMDLIB	xmit packed command proc	
SAMPLES	xmit packed BREXX commands	
JCL	xmit packed example JCL	
LINKLIB	xmit packed BREXX Load library	
PROCLIB	xmit packed BREXX JCL procedures	
RXINSTDL	Internal CLIST used during Installation	
RXLIB	xmit packed include library	

Activating the new BREXX V2R5M1 Release

The next steps describe how to enable your new BREXX Release. In summary, you must run the following jobs out of the above library in the listed sequence:

- **$UNPACK** - mandatory
- **$INSTALL** - mandatory
- **$TESTRX** - optional, recommended
- **$CLEANUP** - optional

See details in the step descriptions below.

1.4.4 Step 3 - Submit $UNPACK JCL of the unpacked Library

In the unpacking process, the contained installation files will be expanded into different partitioned datasets.

Important Before submitting the *$UNPACK* JCL, the XMITLIB parameter must match the dataset name used in the expand JCL of Step2.

If you followed the dataset naming recommendations it is: **BREXX.V2R5M1.INSTALL** and no change is required.

```
//BRXXUNP JOB 'XMIT UNPACK',CLASS=A,MSGCLASS=H,NOTIFY=&SYSUID
//*
//*RELEASE   SET 'V2R5M1'
//* ... BREXX          Version V2R5M1 Build Date 6. May 2022
//* ... INSTALLER DATE 06/05/2022 16:31:35
//* ------------------------------------------------------------------
//* UNPACK XMIT FILES INTO INSTALL LIBRARIES
//*   *** CHANGE XMITLIB= TO THE EXPANDED XMIT LIBRARY OF INSTALLATION
//* ------------------------------------------------------------------
//*           ----->   CHANGE XMITLIB TO YOUR UNPACKED XMIT FILE  <-----
//*                          XXXXXXXXXXX
//*                        X     X     X
//*                       X      X      X
//*                      X       X       X
```

```
//*                      X        X        X
//XMITLOAD PROC XMITLIB='BREXX.V2R5M1.INSTALL',
//         HLQ='BREXX.V2R5M1',     <-- DO NOT CHANGE HLQ ----
//         MEMBER=
```

Important If the job does not run and waits, check with option 3.8 the status. It is most likely "WAITING FOR DATASETS". The simplest method to resolve this is to LOGOFF and re-LOGON to your TSO session.

After completion of the *$UNPACK* JCL the following new Libraries are available:

Dataset	Description
BREXX.V2R5M1.CMDLIB	REXX commands are directly executable
BREXX.V2R5M1.SAMPLE	REXX Samples scripts
BREXX.V2R5M1.JCL	REXX Job Control
BREXX.V2R5M1.LINKLIB	BREXX Load Modules
BREXX.V2R5M1.APFLLIB	BREXX authorised Load Modules
BREXX.V2R5M1.PROCLIB	BREXX JCL Procedures
BREXX.V2R5M1.RXLIB	BREXX include Libraries

The unpacking process removes any old version of the above libraries, before the creation of the new version. If no old version of these libraries is available, the delete steps end with *RC=4*, as well as the job ends with *RC=4*. **Ignore these errors**, if the individual unpack steps return with *RC=0*. Therefore please carefully check the output of this job.

Important Before you install BREXX, you must decide either on the normal BREXX installation or the authorised BREXX installation.

With the authorised version you can call from BREXX utilities as IEBGENER, IEBCOPY, NJE38, etc. which run in authorised mode. This requires that the environment in which you start BREXX is authorised, meaning Wally Mclaughlin's ISPF, or RFE must be authorised. Plain TSO is already authorised.

Both installations are copied into the same partitioned datasets; they are, therefore, mutually exclusive!

If the standard installation is sufficient, continue with Step 4 If you plan to use the authorised, continue with Step 4A. In this case, the MVS authorisation table needs to be updated as well.

1.4.5 Step 4 - Submit $INSTALL JCL for the Standard Installation

The **$INSTALL** JCL copies all member from the following two partitioned datasets into the appropriate SYS2 datasets.

- BREXX.LINKLIB -> SYS2.LINKLIB
- BREXX.PROCLIB -> SYS2.PROCLIB

All these members are BREXX/370 specific and do not conflict with existing members. Members of the system libraries remain untouched.

Important Please log off and re-login to your TSO session before performing any online testing; this enforces the new loading of modules used during the testing, else you might see an 0C4. In rare situations, the installation of the BREXX Linklib members may create a new dataset extent in SYS2.LINKLIB. In this case, you must also restart your TK4- MVS session.

Continue with STEP 5

1.4.6 Step 4A- Submit $INSTAPF JCL for the Authorised Installation

The *$INSTPAPF* JCL copies all member from the following two partitioned datasets into the appropriate SYS2 datasets.

- BREXX.LINKLIB -> SYS2.LINKLIB
- BREXX.PROCLIB -> SYS2.PROCLIB

All these members are BREXX/370 specific and do not conflict with existing members. Members of the system libraries remain untouched.

To authorise the Modules to change the following Modules:

```
SYS1.UMODSRC(IKJEFTE2)
SYS1.UMODSRC(IKJEFTE8)
```

Add the BREXX modules to the sources:

```
     DC    C'BREXX   '             BREXX/370
     DC    C'REXX    '             BREXX/370
     DC    C'RX      '             BREXX/370
```

To activate the changes submit the Jobs:

- SYS1.UMODCNTL(ZUM0001)
- SYS1.UMODCNTL(ZUM0014)

Aftewards you **must** restart your MVS:

- Shut down your MVS
- Re-IPL your job with the CLPA option
- Shut Down MVS again
- Perform normal IPL

Important If you run Wally McLaughlin's ISPF the ISPF libraries must be authorised, otherwise calling a rexx from within ISPF will abend (usually S306).

1.4.7 Step 5 - Submit $TESTRX JCL of the unpacked Library

Submit *$TESTRX* start a test to verify the installation of BREXX/370. All steps should return with RC=0

1.4.8 Step 6 - Submit $CLEANUP JCL of the unpacked Library

The $CLEANUP job removes all unnecessary installation files they are no longer needed, as they were merged into the appropriate SYS2.xxx library.

- BREXX.V2R5M1.LINKLIB
- BREXX.V2R5M1.PROCLIB

You may also wish to remove the uploaded XMIT File, which was used for the first unpack process.

1.4.9 Step 7 - ADD BREXX Libraries into TSO Logon

To run BREXX with its shortcut RX, REXX, BREXX you must allocate the BREXX libraries into your Logon procedure. There are several ways to achieve this. The easiest and recommended method for TK4 users is to add lines into *SYS1.CMDPROC(USRLOGON)*. Non TK4 installation may use different libraries. MVS/CE and Jay Moseley sysgen use *SYS1.CMDPROC(TSOLOGON)*.

```
/* ALLOCATE RXLIB IF PRESENT */
IF &SYSDSN('BREXX.V2R5M1.RXLIB') EQ &STR(OK) THEN DO
  FREE FILE(RXLIB)
  ALLOC FILE(RXLIB) +
    DSN('BREXX.V2R5M1.RXLIB') SHR

/* ALLOCATE SYSEXEC TO SYS2 EXEC */
IF &SYSDSN('SYS2.EXEC') EQ &STR(OK) THEN DO
 FREE FILE(SYSEXEC)
 ALLOC FILE(SYSEXEC) DSN('SYS2.EXEC') SHR
END

/* ALLOCATE SYSUEXEC TO USER EXECS */
IF &SYSDSN('&SYSUID..EXEC') EQ &STR(OK) THEN DO
 FREE FILE(SYSUEXEC)
 ALLOC FILE(SYSUEXEC) DSN('&SYSUID..EXEC') SHR
END
```

insert the clist above before the line *%STDLOGON* in *SYS1.CMDPROC(USRLOGON)*.

The update of the TSO Logon CLIST is an entirely manual process! Please take a backup of *USRLOGON* CLIST first to allow a recovery in case of errors!

Important Users who upgrade from a previous release of BREXX need to update the logon clist and replace the RXLIB allocation with the current dataset name: BREXX.V2R5M1.RXLIB.

1.4.10 Step 8 - Your Tests

It is advised to LOGOFF and LOGON again to your system to make sure that the newly installed modules become active.

Now it's your turn to test BREXX/370! Please be advised BREXX/370 is not z/OS REXX, so you might miss some functions but find also functions not available in the "original".

1.4.11 Step 9 - Remove old BREXX Libraries (optional)

If you had a previous BREXX/370 version installed and your tests ran successfully, you can remove the libraries of the earlier BREXX version, for example, V2R2M0.

Dataset	Description
BREXX.V2R2M0.CMDLIB	REXX commands
BREXX.V2R2M0.SAMPLE	REXX Samples scripts
BREXX.V2R2M0.JCL	REXX Job Control
BREXX.V2R2M0.LINKLIB	BREXX Load Modules
BREXX.V2R2M0.PROCLIB	BREXX JCL Procedures
BREXX.V2R2M0.RXLIB	BREXX include Libraries

If you upgraded from the very first BREXX/370 version, you can remove the following libraries:

Dataset	Description
BREXX.CMDLIB	REXX commands
BREXX.SAMPLE	REXX Samples scripts
BREXX.JCL	REXX Job Control
BREXX.LINKLIB	BREXX Load Modules
BREXX.PROCLIB	BREXX JCL Procedures
BREXX.RXLIB	BREXX include Libraries

1.5 Additional Settings (optional)

If you want to communicate with the control program of the host system (either Hercules or VM) you can do so, by running:

```
ADDRESS COMMAND 'CP cp-parameter ...'
```

For VM you need to use a valid CP command. Example:

```
ADDRESS COMMAND 'CP QUERY TIME'
```

If your system is running within Hercules your CP commands are routed to Hercules and need to be Hercules commands. Example:

```
ADDRESS COMMAND 'CP DEVLIST'
```

To communicate with Hercules you need to enable the DIAG8 commands *DIAG8CMD ENABLE* in the Hercules console. In TK4- systems it is already enabled. If it is not enabled and you run an *ADDRESS COMMAND "CP command"* BREXX will abend typically with an 0C6.

BREXX Usage

There are JCL Procedures delivered, which facilitate the test and execution of REXX scripts. The installation process merges them into *SYS2.PROCLIB*.

The delivered RXLIB PDS contains several REXX functions, which are usable as if they were a BREXX internal function.

The delivered JCL procedures allocate the RXLIB library, and it is recommended to add it also into the TSO Logon procedures (Step 8).

2.1 TSO online

Executing rexx scripts in TSO uses either *RX* or *REXX*. You can either call scripts from dataset libraries or fully qualified dataset names.

To call a script from a library:

```
RX rexx-script-name
REXX rexx-script-name
```

BREXX performs all necessary allocations. It is advised to add a user-specific REXX library, naming convention: *&SYSUID.EXEC* (RECFM=VB, LRECL255). If available, the REXX-script searches path starts from there. The REXX library search sequence is:

1. SYSUEXEC - typically &SYSUID.EXEC
2. SYSUPROC - (optional)
3. SYSEXEC - (optional)
4. SYSPROC - (optional)

At least one of these libraries needs to be pre-allocated during the TSO logon process. It is not mandatory to have all of them allocated. It depends on your planned REXX development environment. The allocations may consist of concatenated datasets. If you followed the instructions above then SYSEXEC is assigned to *SYS2.EXEC* and SYSUEXEC is assigned to *&SYSUID.EXEC*.

Alternatively, you can specify a fully qualified dataset-name and member name (if the dataset is a PDS):

```
RX 'dataset-name(rexx-script-name)'
REXX 'dataset(rexx-script-name)'
```

2.2 TSO Batch (start REXX JCL Procedure)

There is a JCL Procedure defined that allows you to run REXX Scripts in a TSO Batch environment. The Procedure performs all necessary BREXX and TSO allocations.

Some ADDRESS TSO commands as ALLOC/FREE are supported.

```
//DATETEST JOB CLASS=A,MSGCLASS=H,REGION=8192K,NOTIFY=&SYSUID
//*
//* ------------------------------------------------------------------*
//* TEST REXX DATE AS TSO BATCH
//* ------------------------------------------------------------------*
//REXX EXEC RXTSO,EXEC='DATE#T',SLIB='BREXX.SAMPLES'
```

- **EXEC=** defines the rexx script to run
- **SLIB=** defines the library/partitioned dataset containing the rexx script defined in **EXEC**

Additionally, you can add a *P='input-parameters'* JCL Parameter field, if your rexx receives input parameters.

2.2.1 Plain Batch (start REXX JCL Procedure)

There is a JCL Procedure defined that allows you to run REXX Scripts in a plain Batch environment. The Procedure performs all necessary BREXX allocations

Warning ADDRESS TSO commands are not supported here!

```
//DATETEST JOB CLASS=A,MSGCLASS=H,REGION=8192K,NOTIFY=&SYSUID
//*
//* ------------------------------------------------------------------*
//* TEST REXX DATE AS TSO BATCH
//* ------------------------------------------------------------------*
//REXX EXEC RXBATCH,EXEC='ETIME#T',SLIB='BREXX.SAMPLES'
```

- **EXEC=** defines the rexx script to run
- **SLIB=** defines the library/partitioned dataset containing the rexx script defined in **EXEC**

Additionally, you can add a *P='input-parameters'* JCL Parameter field, if your rexx receives input parameters.

2.2.2 BREXX/370 Sample Library

The Library *BREXX.version.SAMPLES* contains a variety of REXX scripts that cover the following areas:

- Basic functionality in Members starting with '$'
- FSS samples, starting with '#'
- VSAM samples beginning with '@'
- All other scripts are original samples delivered with Vasilis Vlachoudis BREXX installation.

2.2.3 BREXX/370 Hints

Please make sure that your REXX files do not contain line numbering! They are not wiped out by BREXX/370 and therefore treated as the content of the script. This lead to errors during interpretation, sometimes even system abends! Use UNNUM as a primary command in the RFE editor to switch line

numbering off and remove existing numbers.

If the BREXX/370 call leads to an S106 Abend, the most likely reason is the creation of a new extent in *SYS2.LINKLIB* during the installation process. Its size and number of extents are loaded during IPL and kept while MVS is up and running. The creation of new extents will therefore not be discovered.

You can either re-IPL your system or better REORG SYS2.LINKLIB with IEBCOPY.

3 Tokens and Terms

REXX expressions and instructions may contain the following items:

3.1 comment

A comment is a sequence of characters (on one or more lines) delimited by */** and ***/**. Nested comments are also valid, as */* hello /* joe */ */*

3.2 string

A string is a sequence of characters delimited by a single quote or double quote. Use two quotes to obtain one quote inside a string. A string may be specified in binary or hexadecimal if the final quote is followed by a B or X. If it followed by an H then is treated as a hexadecimal number. Some valid strings are:

```
"Marmita"
'0100 0001'b
'He''s here'
'2ed3'x
'10'h (=16)
```

3.3 number

A number is a string of decimal digits with or without a decimal point. A number may be prefixed with a plus or minus sign, and/or written in exponential notation. Some valid numbers are:

```
23
12.07
141
12.2e6
+5
'-3.14'
```

3.4 symbol

A symbol refers to any group of characters from the following selection: *A-Z, a-z, 0-9, @ # $ _ . ? !*

Symbols are always translated to uppercase. Variables are symbols but the first character must not be a digit 0-9 or a dot '.'. Each symbol may consist of up to 250 characters.

3.5 function-call

A function-call invokes an internal, external, or built-in routine with 0 to 10 argument strings. The called routine returns a character string. A function-call has the format:

function-name([expr][,[expr]...)

function-name must be adjacent to the left parenthesis, and may be a symbol or a string.

All procedures can be called as functions or procedures. If a function is called as a procedure CALL left 'Hello',4 then the return string will be returned in the variable RESULT (where in this example will contain the string 'Hell')

```
copies('ab',3)    /* will return 'abababab' */
```

4 Expressions

Most REXX instructions permit the use of expressions, following normal algebraic style. Expressions can consists of strings, symbols, or functions calls. Expressions are evaluated from left to right, modified by parentheses and the priority of the operators (as ordered below). Parentheses may be used to change the order of evaluation.

All operations (except prefix operations) act on two items and result in a character string.

4.1 Prefix + - ^ \

Prefix operations: Plus; Minus; and Not. (For + and -, the item must evaluate to a number; for ^ and \, the item must evaluate to "1" or "0".)

```
-4    /* -4 */
```

4.2 **

Exponentiate. (Both items must evaluate to numbers, and the right-hand item must be a whole number.)

```
2 ** 3  /* 8 */
2 ** -3         /* 0.125 */
```

4.3 * / % //

Multiply; Divide; Integer Divide; Divide and return the remainder. (Both items must evaluate to numbers.)

```
4 * 3   /* 12 */
4 / 3   /* 1.333.. */
5 % 3   /* 1 */
5 // 3  /* 2 */
```

4.4 + -

Add; Subtract. (Both items must evaluate to numbers.)

```
2 + 3.02        /* 5.02 */
```

4.5 (blank) ||

Concatenate: with or without a blank. Abuttal of items causes direct concatenation.

```
a = 'One'
a 'two'         /* "One two" */
a || 'two'      /* "Onetwo" */
a'two'  /* "Onetwo" */
```

4.6 = > < >= <= ^= /= = ^> ^< >< <> == >> << >>= <<= ^== /== == ^>> ^<<

Comparisons (arithmetic compare if both items evaluate to a number.) The ==, >>, << etc. operators checks for an exact match.

```
'marmita' = ' marmita '         /* 1 (spaces are striped) */
'marmita' == ' marmita '        /* 0 */
'marmita' ^= ' marmita '        /* 0 (spaces are striped) */
'marmita' ^==' marmita '        /* 1 */
'2' = ' 2 '     /* 1 (arithmetic comparison) */
'2' == ' 2 '    /* 0 (string comparison) */
'2' >> ' 2 '    /* 1 (string comparison) */
```

4.7 &

Logical And. (Both items must evaluate to “0” or “1”.)

```
'a'='b' & 'c'='c'       /* 0 */
```

4.8 / &&

Logical Or; Logical Exclusive Or. (Both items must evaluate to “0” or “1”.)

```
'a'='b' | 'c'='c'       /* 1 */
'A'='b' && 'c'='c'      /* 1 */
```

In Ansi REXX the results of arithmetic operations are rounded according the setting to *NUMERIC DIGITS* (default is 9). Here all arithmetic operations follow C arithmetics. For a more detail description look at the Implementation Restrictions document.

5 Instructions

Each REXX instruction is one ore more clauses, the first clause is the one that identifies the instruction. Instructions end with a semicolon or with a new line. One instruction may be continued from one line to the next by using a comma at the end of the line. Open strings or comments are not affected by line ends.

5.1 General Guidelines

name;
: refers to a variable, which can be assigned any value. name is a symbol with the following exception: the first character may not be a digit or a period. The value of name is translated to uppercase before use, and forms the initial value of the value of the variable. Some valid names are:

  - Fred
  - COST?
  - next
  - index
  - A.j

name:
: is a form of labels for CALL instructions, SIGNAL instructions, and internal function calls. The colon acts as a clause separator.

template;
: is a parsing template, described in a later section.

instr;
: is any one of the listed instructions.

5.2 Instructions

expression;
: the value of expression is issued as a command, normally to the command interpreter or to the specified environment specified by the ADDRESS instruction. Look also the section “Issuing Commands to Host System.”

name = [expr];
: is an assignment: the variable name is set to the value of expr.

```
fred = 'sunset'
a = 1 + 2 * 3
a = /* a contains '' */
```

ADDRESS [<symbol | string> [expr]] | VALUE expr | (env);
: redirect commands or a single command to a new environment. ADDRESS VALUE expr may be used for an evaluated environment name.

```
address int2e 'dir' /* executes through int2e a dir cmd */
```

```
address system /* all the following command will be addressed to system */
env = 'dos'
address value env /* change address to dos */
address (env) /* change address to dos */
```

ARG <template>;

parse argument string(s) given to program or in an internal routine into variables according to template. Arguments are translated into uppercase before the parsing. Short for PARSE UPPER ARG.

```
/* program is called with args "autoexec.bat auto.old" */
arg src dest
/* src = "AUTOEXEC.BAT", dest="AUTO.OLD" */
/* a function is called MARMITA('Bill',3) */
marmita:
arg firstarg, secondarg
/* firstarg = "BILL", secondarg = "3" */
```

CALL [symbol | string] [<expr>] [,<expr>]... ;
[ON|OFF <condition> [NAME label]];

call an internal routine, an external routine or program, or a built-in function. Depending on the type of routine called, the variable RESULT contains the result of the routine. RESULT is uninitialized if no result is returned.

```
CALL SUBSTR 'makedonia',2,3
/* now. variable result = 'ake' */
/* the same can be obtained with */
result = SUBSTR('makedonia',2,3)
```

In the following sections there is a description of all the built-in rexx functions.

Internal functions are sequence of instructions inside the same program starting at the label that matches the name in the CALL instruction.

If the function is not found in the current program, then REXX will search for a file that matches the name in the CALL instruction and the same extension like the current program, and will load it as an external rexx function.

External routines are like internal but written in a separate module that can be used as a library. Rexx libraries are rexx files with many external routines which must be loaded with the built-in function LOAD before they are used (see below).

As external routines can be used any DOS command or program that uses standard input and output.

```
/* external programs can be called as routines */
/* and the output of the program (to stdout) will */
/* be returned as the result string of the function */
CALL "dir" "*.exe","/w" /* or */
files = "dir"('*.exe',"/w")
current_directory = 'cd'()
```

For CALL ON/OFF condition look below at the SIGNAL instruction.

DO [name=expri [TO exprt] [BY exprb] [FOR exprf]] | [FOREVER | exprr]
[UNTIL expru | WHILE exprw] ;
[instr]... ;
END[symbol] ;

DO is used to group many instructions together and optionally executes them repetively.

Simple DO loop are used to execute a block of instructions often used with IF-THEN statements.

Note Simple DO loops are not affected with ITERATE or LEAVE instructions (see below)

```
IF name = 'Vivi' THEN DO
        i = i + 1
        SAY 'Hello Vivi'
    END
```

Simple repetitive loops.

Note in DO expr, expr must evaluate to an integer number.

```
DO 3    /* would display 3 'hello' */
    SAY 'hello'
END
```

Inifinite loops

```
DO FOREVER    /* infinite loop, display always */
    SAY 'lupe forever'    /* 'hello' */
END
```

Loops with control variable. name is stepped from expri to exprt in steps of exprb, for a maximum of exprf iterations.

```
DO i = 1 TO 10 BY 3     /* would display the numbers */
    SAY i     /* 1, 4, 7, 10 */
END
```

Note all the expressions are evaluated before the loop is executed and may result to any kind of number, integer or real.

Conditional loops

```
a = 2        /* would display */
DO WHILE a < 5     /* 2 */
    SAY a     /* 4 */
    a = a + 2
END
```

Note exprw and expru are evaluated in each iteration and must result to 0 or 1. WHILE expression is evaluated before each iteration, where UNTIL expression is evaluated at the end of each iteration.

You can combine them like:

```
a = 1            /* would display */
DO FOR 3 WHILE a < 5     /* 1 */
    SAY a        /* 2 */
    a = a + 1        /* 3 */
END
```

DROP <name | (nameind)> [<name | (nameind)>]... ;

DROP (reset) the named variables or group of variables by freeing their memory. It returns them in their original uninitialized state. If a variable is enclosed in parenthesis then DROP resets all the variables that nameind contains as separate words. If an exposed variable is named, the variable itself in the older generation will be dropped! If a stem is specified all variables starting with that stem will be dropped.

```
j = 2
vars="j b stem."
```

```
DROP a x.1 y.j        /* resets variables A X.1 and Y.2 */
DROP z.        /* resets all variables with names
    starting with Z. */
DROP (name)        /* resets variables j b and stem. */
```

EXIT [expr] ;

leave the program (with return data, expr). EXIT is the same as RETURN except that all internal routines are terminated.

```
EXIT 12*3    /* will exit the program with RC=36 */
```

IF expr [;] THEN [;] instr ;
[ELSE [;] instr];

if expr evaluates to "1", executes the instruction following the THEN. Otherwise, when expr evaluates to "0", the instruction after ELSE is executed, if ELSE is present.

```
IF name="Vivi"    THEN SAY "Hello Vivian"
    ELSE SAY "Hello stranger"
```

INTERPRET expr ;

expr is evaluated and then is processed, as it was a part of the program.

```
cmd = "SAY 'Hello'"
INTERPRET cmd     /* displayes "Hello" */
```

ITERATE [name] ;

start next iteration of the innermost repetitive loop (or loop with control variable name).

```
DO    i = 1 TO 5        /* would display:     1 */
    IF i=3 THEN ITERATE        /*     2 */
    SAY i        /*     4 */
END         /*     5 */
```

LEAVE [name] ;

terminate innermost repetitive loop (or loop with control variable name).

```
DO    i = 1 TO 5        /* would display:     1 */
    IF i=3 THEN LEAVE        /*     2 */
    SAY i
END
```

LOWER name [name]...

translate the values of the specified individual variables to lowercase.

```
name = 'ViVi'
LOWER name        /* now, name = 'vivi' */
```

NOP ;

dummy instruction, has no effect.

```
IF name^='Vivi' THEN NOP; ELSE SAY 'Hello Vivi.'
```

NUMERIC DIGITS [expr] | FORM [SCIENTIFIC | ENGINEERING] | FUZZ [expr] ;

Set the number of significant digits used for all arithmetic operations.

Note In BRexx all numerical operations are performed either with the 32bit integer type or 64

double precission, so the numeric digits is limited for floating point operations to maximum 22 digits.

PARSE [type] + ARG + [template] ;

Parse is used to assign data from various sources to one or more variables according to the template (see below for template patterns) where the optional type is one of:

- ARG, parses the argument string(s) passed to the program, subroutine, or function. UPPER first translates the strings to uppercase. See also the ARG instruction.
- AUTHOR parse the author string.
- EXTERNAL, prompts for input and parses the input string
- LINEIN, same as EXTERNAL
- NUMERIC, parse the current NUMERIC settings.
- PULL, read and parse the next string from REXX stack if not empty otherwise prompts for input. See the PULL instruction.
- SOURCE, parse the program source description e.g. “MSDOS COMMAND prog.r C:REXX.EXE C:DOSCOMMAND.COM”
- VALUE, parse the value of expr.
- VAR, parse the value of name.
- VERSION, parse the version string of the interpreter.

PROCEDURE [EXPOSE name|(varind) [name|(varind)]...] ;

start a new generation of variables within an internal routine. Optionally named variables or groups of variables from an earlier generation may be exposed. If a stem is specified (variable ending in ‘.’ dot, ie ‘A.’) then every variable starting with this stem will be exposed. Indirect exposure is also posible by enclosing inside parenthesis the variable varind which contains contains as separate words all variables to be exposed

```
i = 1; j = 2
ind = "i j"
CALL myproc
CALL myproc2
EXIT
myproc: PROCEDURE EXPOSE i     /* would display */
SAY i j        /* 1 J */
RETURN
myproc2: PROCEDURE EXPOSE (ind)     /* would display */
say i j        /* 1 2 */
RETURN
```

PULL [template] ;

pops the next string from rexx internal stack. If stack is empty then it prompts for input. Translates it to uppercase and then parses it according to template. Short for PARSE UPPER PULL.

```
PUSH 'Vassilis Vlachoudis'
/* --- many instrs ---- */
PULL name surname     /* now: name='BILL', */
    /* surname='VLACHOUDIS' */
```

PUSH [expr] ;

push expr onto head of the rexx queue (stack LIFO)

QUEUE [expr] ;

add expr to the tail of the rexx queue (stack FIFO)

RETURN [expr] ;

return control from a procedure to the point of its invocation. if expr exits, then it is returned as the result of the procedure.

```
num = 6
SAY num || '! = ' fact(num)
EXIT
fact: PROCEDURE        /* calculate factorial with */
IF arg(1) = 0 THEN RETURN 1        /* recursion */
RETURN fact(ARG(1)-1) * ARG(1)        /* displayes: 6! = 720 */
```

SAY [expr];

evaluate expr and then writes the result to standard output (normally user's console) followed by a newline.

SELECT ;
WHEN expr [;] THEN [;] instr;
[WHEN expr [;] THEN [;] instr;]
[OTHERWISE [;] [instr]...];
END ;

SELECT is used to conditionally process one of several alternatives. Each WHEN expression is evaluated in sequence until one results in "1". instr, immediately following it, is executed and control leaves the block. If no expr evaluated to "1", control passes to the instructions following the OTHERWISE expression that must then be present.

```
num = 10
SELECT
    WHEN num > 0 THEN SAY num 'is positive'
    WHEN num < 0 THEN SAY num 'is negative'
    OTHERWISE SAY num 'is zero'
END
```

SIGNAL [name] |
[VALUE] expr |
<ON | OFF> + condition + [NAME label];

Parameters condition – Can be one of *ERROR HALT NOTREADY NOVALUE SYNTAX*

- name, jump to the label name specified. Any pending instructions, *DO … END*, *IF*, *SELECT*, and *INTERPRET* are terminated.
- VALUE, may be used for an evaluated label name.
- ON|OFF, enable or disable exception traps.
- Condition must be ERROR, HALT, NOTREADY, NOVALUE, or SYNTAX. Control passes to the label of the condition name if the event occurs while ON or to label if NAME label is specified.

```
SIGNAL vivi
...
vivi:
SAY 'Hi!'
```

A condition example:

```
SIGNAL ON SYNTAX NAME syntax_error;
SAY 1/0     /* Control passes to label syntax_error */
...
syntax_error:
SAY 'Syntax error in line:' SIGL
```

TRACE option | VALUE expr;

Trace according to following option. Only first letter of option is significant.

- A (All) trace all clauses.
- C (Commands) trace all commands.
- E (Error) trace commands with non-zero return codes after execution.
- I (Intermediates) trace intermediate evaluation results and name substitutions also.
- L (Labels) trace only labels.
- N (Negative or Normal) trace commands with negative return codes after execution (default setting).
- O (Off) no trace.
- R (Results) trace all clauses and expressions.
- S (Scan) display rest of program without any execution (shows control nesting).
- ? turn interactive debug (pause after trace) on or off, and trace according to next character. null restores the default tracing actions.

TRACE VALUE expr may be used for an evaluated trace setting.

UPPER name [name]...

translate the values of the specified individual variables to uppercase.

```
name = 'Vivi'
UPPER name        /* now: name = 'VIVI' */
```

Templates for ARG, PULL, and PARSE

The *PULL*, *ARG* and *PARSE* instructions use a template to parse a string.

The simplest template is a list of variables where each of them is assigned one word from the string, except the last variable in the list which will contain the rest of the string.

```
PARSE VALUE "one two three four " WITH a b c
now a="one"; b="two"; c="three four"
PARSE VALUE "one two three four " WITH a b c d e
now a="one"; b="two"; c="three"; d="four" and e=""
```

A dot '.' can be in the place of one or more variables, it is used as a place-holder.

```
PARSE VALUE "one two three four " WITH a . . d
now a="one"; d="four"
```

A more complex parsing is to use patterns for triggering:

- **number** which specifies an absolute position in string 1 - is the first character in string
- **=(name)** as a position may be a variable enclosed in parenthesis after an equal symbol
- **[+|-]number** signed numbers are used as a relative positioning

```
PARSE VALUE "one two three four " WITH 2 a 6 b
now a="ne t"; b="wo three four " pos=6;
PARSE VALUE "one two three four " WITH 2 a =(pos) b
now a="ne t"; b="wo three four " PARSE VALUE "one two three four " WITH
2 a +2 b
now a="ne"; b=" two three four "
```

- **string** - may be used as a target position.

```
 PARSE VALUE "marmita/bill/vivi' WITH a '/' b '/' c
 now a="marmita"; b="bill"; c="vivi"
```

- **(name)** - also as a target may be used a variable encolsed in parenthesis

```
 t = "%%"
 PARSE VALUE "aabbcc%%ddeeff%%gg%%" WITH . (t) middle (t) . now
middle="ddeeff"
```

A **comma** can be used as a "trigger" to move to the next string when there is more than one to be parsed (e.g. when there is more than one argument string to a routine).

```
CALL MyProc 'Hi',3,4
EXIT
MyProc:
```

```
PARSE ARG first, second, third      /* now first="Hi" */
...          /* second=3 */
/* third=4 */
```

Compound Variable Names

name may be “compound” in that it may be composed of several parts (separated by periods) some of which may have variable values. The parts are then substituted independently, to generate a fully resolved name. In general,

```
s0.s1.s2.---.sn     /* is substituted to form */
d0.v1.v2.---.vn     /* where d0 is uppercase of s0, and
    v1-vn are values of s1-sn */
```

This facility may be used for traditional arrays, content-addressable arrays, and other indirect addressing modes. As an example, the sequence:

```
J = 5; a.j = "fred";
```

would assign fred to the variable A.5.

The stem of name (i.e. that part up to and including the first “.”) may be specified on the *DROP* and *PROCEDURE EXPOSE* instructions and affect all variables starting with that stem. An assignment to a stem assigns the new value to all possible variables with that stem.

8 Special Variables

There are three special variables:

8.1 SIGL

holds the line number of the instruction that was last executed before control of program was transferred to another place. This can be caused by a SIGNAL instruction, a CALL instruction or a trapped error condition.

8.2 RC

is set to the errorlevel (return-code) after execution of every command (to host).

8.3 RESULT

is set by a RETURN instruction in a CALLed procedure.

Interactive Debugging

You can enter the interactive debuging either by executing a TRACE instruction with a prefix '?' or when calling REXX from command line issuing as a first argument the trace option:

rexx '?A' 'HLQ.DATASET(MEMBER)'

In interactive debug, interpreter pauses before the execution of the instructions that are to be traced and prompts for input. You may do one of following things:

- Enter a null line to continue execution.
- Enter a list of REXX instructions, which are interpreted immediately (*DO-END* instructions must be complete, etc.).

During the execution of the string, no tracing takes place, except that non-zero return codes from host commands are displayed. Execution of a TRACE instruction with the "?" prefix turns off interactive debug mode. Other TRACE instructions affect the tracing that occurs when normal execution continues.

10 Built-in Functions

The following are the built-in REXX functions. Which are divided into the following categories:

- Rexx
- String
- Word
- Math
- Data Convert
- File Functions

10.1 Rexx Functions

ADDR (*symbol[,[option][,pool]]*)

returns the physical address of symbol contents. Option can be 'Data' (default) variables data 'Lstring' lstring structure pointer 'Variable' variable structure.
If pool exist, the specific rexx pool is searched for the symbol. Valid pools are numbers from 0 up to current procedure nesting. (The result is normalized for MSDOS, ie *seg:ofs = seg*16+ofs*)

```
i = 5;
SAY addr('i')           /* something like 432009 decimal */
SAY addr('i','L')       /* something like 433000 */
SAY addr('i','V')       /* something like 403004 */
SAY addr('i','V',0)     /* something like 403004 */
SAY addr('j')           /* -1, is J variable doesn't exist */
```

ADDRESS ( )

return the current environment for commands.

```
SAY address() /* would display: SYSTEM */
```

ARG (**[**, *n* **[**, *option* **]** **]**)

- ARG() returns the number of arguments
- ARG(n) return then nth argument
- ARG(n,option) option may be **Exist** or **Omitted** (only the first letter is significant) test whether the nth argument Exists or is Omitted.

Returns "0" or "1"

```
call myproc 'a',,2
...
myproc:
SAY arg()               /* 3 */
SAY arg(1)              /* 'a' */
SAY arg(2,'O')          /* 1 */
SAY arg(2,'E')          /* 0 */
```

DATATYPE (*string* **[**, *type* **]**)

DATATYPE(string) - returns "NUM" is string is a valid REXX number, otherwise returns "CHAR".
DATATYPE(string,type) - returns "0" or "1" if string is of the specific type:

- Alphanumeric: characters A-Z, a-z and 0-9
- Binary: a valid BINARY number
- Lowercase: characters a-z
- Mixed: characters A-Z, a-z
- Number: is a valid REXX number
- Symbol: characters A-Z, a-z, 0-9, @%_.!#
- Uppercase: characters A-Z
- Whole-number: a valid REXX whole number
- X (heXadecimal): a valid HEX number

(only the first letter of type is required)
The special type 'Type' returns the either INT, REAL, or STRING the way the variable is hold into memory. Usefull when you combine that with INTR function.

```
SAY datatype('123')              /* NUM */
SAY datatype('21a')              /* CHAR */
SAY datatype(01001,'B')          /* 1 */
SAY datatype(i,'T')              /* maybe STRING */
```

DATE ([, *option*])

return current date in the format: dd Mmm yyyy

```
SAY date() /* 14 Feb 1993 */
```

or formats the output according to option

- Days returns number of days since 1-Jan as an integer
- European returns date in format dd/mm/yy
- Month returns the name of current month, ie. March
- Normal returns the date in the default format dd Mmm yyyy
- Ordered returns the date in the format yy/mm/dd
- (useful for sorting)
- Sorted returns the date in the format yyyymmdd
- (suitable for sorting)
- USA returns the date in the format mm/dd/yy
- Weekday returns the name of current day of week ie. Monday

DESBUF ( )

destroys the all system stacks, and returns the number of lines in system stacks.

```
PUSH 'hello'            /* now stack has one item */
CALL desbuf             /* stack is empty, and RESULT=1 */
```

DROPBUF ([, *num*])

destroys num top stacks, and returns the number of lines destroyed.

```
PUSH 'in stack1'                /* first stack has one item */
CALL makebuf            /* create a new buffer */
PUSH 'in stack2'                /* new stack has one item */
CALL dropbuf            /* one stack remains */
```

DIGITS ( )

returns the current setting of NUMERIC DIGITS.

ERRORTEXT (*n*)

returns the error message for error number n.

```
SAY errortext(8)        /* "Unexpected THEN or ELSE" */
```

FORM ( )

returns the current setting of NUMERIC FORM.

FUZZ ( )

returns the current setting of NUMERIC FUZZ.

GETENV (*varname*)

returns the environment variable varname

```
SAY getenv("PATH")
```

HASHVALUE (*string*)

return an integer hashvalue of the string like Java *hash = s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1]*

```
SAY hashvalue("monday")         /* -1068502768 */
```

IMPORT (*file*)

import a shared library file using dynamic linking with rexx routines. If it fails, then try to load a rexx file so it can be used as a library. import first searches the current directory, if not found it searches the directories pointed by the environment variable RXLIB.

returns

- "-1" if already imported
- "0" on success
- "1" on error opening the file

```
CALL IMPORT FSSAPI
call import "veclib"
```

MAKEBUF ( )

create a new system stack, and returns the number of system stacks created until now (plus the initial one).

```
PUSH 'hello'; SAY queued() queued(T)            /* display 1 1 */
CALL makebuf            /* create a new buffer */
PUSH 'aloha; SAY queued() queued(T)             /* display again 2 1 */
```

QUEUED (**[**, *option* **]**)

return the number of lines in the rexx stack (all stacks or the topmost) or the number of stacks. Option can be (only first letter is significant):

- All lines in All stacks (default)
- Buffers number of buffers created with MAKEBUF
- Topstack lines in top most stack

```
PUSH 'hi'
SAY queued(A) queued(B) queued(T)              /* 1 1 1 */
CALL makebuf
SAY queued(A) queued(B) queued(T)              /* 1 2 0 */
PUSH 'hello'
SAY queued(A) queued(B) queued(T)              /* 2 2 1 */
CALL desbuf
SAY queued(A) queued(B) queued(T)              /* 0 1 0 */
```

SOUNDEX (*word*)

return a 4 character soundex code of word in the format "ANNN" (used for phonetic comparison of words)

```
SAY soundex("monday")           /* M530 */
SAY soundex("Mandei")           /* M530 */
```

SOURCELINE (**[**, *n* **]**)

return the number of lines in the program, or the nth line.

```
SAY sourceline()        /* maybe 100 */
SAY sourceline(1)       /* maybe "/**/" */
```

STORAGE (*[address[,[length][,data]]]*)

returns the current free memory size expressed as a decimal string if no arguments are specified. Otherwise, returns length bytes from the user's memory starting at address. The length is in decimal; the default value is 1 byte. The address is a decimal number (Normalized address for MSDOS ie. seg:ofs = seg*16+ofs) If data is specified, after the "old" value has been retrieved, storage starting at address is overwritten with data (the length argument has no effect on this).

```
SAY storage()           /* maybe 31287 */
SAY storage(1000,3)             /* maybe "MZa" */
a = "Hello"
SAY storage(addr('a'),5,'aaa')  /* "Hello" */
SAY a           /* aaalo */
```

SYMBOL (*name*)

return "BAD" if name is not a valid REXX variable name, "VAR" if name has been used as a variable, or "LIT" if it has not.

```
i = 5
SAY symbol('i')         /* VAR */
SAY symbol(i)           /* LIT */
SAY symbol(':asd')              /* BAD */
```

TIME (**[**, *option* **]**)

return the local time in the format: *hh:mm:ss* if option is specified time is formated as:

- Civil returns time in format hh:mmxx ie. 10:32am
- Elapsed returns elapsed time since rexx timer was reset or from begging of program in format ssssss.uuuuuu
- Hours returns number of hours since midnight
- Long returns time and milliseconds hh:mm:ss.uu
- Minutes returns number of minutes since midnight
- Normal returns time in format hh:mm:ss

- Reset returns elapsed time in format ssssss.uuuuuu (like Elapsed) and resets rexx internal timer.
- Seconds returns number of seconds since midnight

TRACE (**[**, *option* **]**)

returns current tracing option. If option is specified then sets to new tracing option. Look up instruction TRACE.

```
SAY trace() /* normally 'N' */
```

VALUE (*name[,[newvalue][,pool]]*)

returns the value of the variable name. If newvalue is specified then after the retrieval of the old value the newvalue will be set to the variable. If pool is specified then the operation takes place at the specific pool. Pool initially exist in this version of Rexx are:

- 0 up to the current PROCEDURE nesting specifing the pool of each PROCEDURE
- Negative values from -1 to minus current PROCEDURE nesting, shows relative values from current procedure.
- SYSTEM is the system pool (like GETENV,PUTENV)
- User can create his own POOLs, Look Programing Rexx

```
i = 5; j = "i"
SAY value(j)            /* 5 */
SAY value('j',10)               /* 'i' */
SAY j           /* 10 */
CALL Procedure
...
Procedure: PROCEDURE
i = "I-var"
SAY value('i')          /* I-var */
SAY value('i',,0)               /* 5 */
SAY value('i',,1)               /* I-var */
SAY value('i',,-1)              /* 5 */
...
```

VARDUMP (*[symbol][,option]*)

returns the binary tree of the variables in the format

```
var = "value" \n
```

option can be “Depth” which prints out the binary tree in the format

```
depth var = "value" \n (used for balancing of variables )
```

symbol may be nothing for main bin-tree or a stem for an array bin-tree ie. “B.”
VARDUMP is an easy way to store the variables in a file or in stack and restores them later.

```
CALL write "vars.$$$", vardump() /* stores all variables */ /* in the file
"vars.$$$" */
```

on a later run you can do

```
DO UNTIL eof("vars.$$$")                /* this will read all variables */
        INTERPRET read("vars.$$$")              /* from file, and restore
them */
END
```

Warning VARDUMP is not fully implemented and may not work when variables have non-printable characters.

10.2 String Functions

ABBREV (*information*, *info* **[**, *length* **]**)

tests whether info is an abbreviation of information. returns "1" on true, else returns "0". If length is specified then searching takes place only for the first length characters.

```
abbrev("billy","bill")          /* 1 */
abbrev("billy","bila")          /* 0 */
abbrev("billy","bila",3)        /* 1 */
```

CENTRE (*string*, *length* **[**, *pad* **]**)

returns string centered in a padded string of length length.

```
center("rexx",2)                /* 'ex' */
center("rexx",8)                /* ' rexx ' */
center("rexx",8,'-')            /* '--rexx--' */
```

CHANGESTR (*target*, *string*, *replace*)

replaces all occurrences of the target in string, replacing them with the replace.

```
changestr("aa","aabbccaabbccaa","--")   /* --bbcc--bbcc-- */
```

COMPARE (*string1*, *string2* **[**, *pad* **]**)

returns "0" if string1==string2, else it returns the index of the first nonmatching character. Shorter string is padded with pad if necessary

```
compare('bill','bill')          /* 0 */
compare('bill','big')           /* 3 */
compare('bi ','bi')             /* 0 */
compare('bi--*','bi','-')       /* 5 */
```

COUNTSTR (*target*, *string*)

counts all the appearances of target in string

```
countstr("aa","aabbccaabbccaa")         /* 3 */
```

COPIES (*string*, *n*)

returns n concatenated copies of string.

```
copies('Vivi',3)        /* 'ViviViviVivi' */
```

DELSTR (*string*, *n* **[**, *length* **]**)

delete substring of string starting at the nth character and of length length.

```
delstr('bill',3)                /* 'bi' */
delstr('bill',2,2)              /* 'bl' */
```

INDEX (*haystack*, *needle* **[**, *start* **]**)

return the position of needle in haystack, beginning at start.

```
index('bilil','il')             /* 2 */
index('bilil','il',3)           /* 4 */
```

INSERT (*new,target[,[n][,[length][,pad]]]*)

insert the string new of length length into the string target, after the nth character (n can be 0)

```
insert('.','BNV',2)             /* 'BN.V' */
insert('.','BNV',2,2)           /* 'BN. V' */
insert('','BNV',2,2,'.')        /* 'BN..V' */
```

LASTPOS (*needle, haystack* **[**, *start* **]**)

return the position of the last occurrence of needle in haystack, beginning at start.

```
lastpos('il','bilil')           /* 4 */
lastpos('il','bilil',4)         /* 2 */
```

LEFT (*string, length* **[**, *pad* **]**)

return a string of length length with string left justified in it.

```
left('Hello',2)         /* 'He' */
left('Hello,10,'.')             /* 'Hello.....' */
```

LENGTH (*string*)

return the length of string

```
length('Hello')         /* 5 */
```

OVERLAY (*new,target[,[n][,[length][,pad]]]*)

overlay the string new of length length onto string target, beginning at the nth character.

```
overlay('.','abcd',2)           /* 'a.cd' */
overlay('.','abcd')             /* '.bcd' */
overlay('.','abcd',6,3,'+')             /* 'abcd+.++' */
```

POS (*needle, haystack* **[**, *start* **]**)

return the position of needle in haystack, beginning at start.

```
pos('ll','Bill')                /* 3 */
```

REVERSE (*string*)

swap string, end-to-end.

```
reverse('Bill')         /* 'lliB' */
```

RIGHT (*string, length* **[**, *pad* **]**)

returns length righmost characters of string.

```
right('abcde',2)                /* 'de' */
```

SUBSTR (*string,n[,[length][,pad]]*)

return the substring of string that begins at the nth character and is of length length. Default pad is space.

```
substr('abcde',2,2)             /* 'bc' */
```

```
substr('abcde',2)               /* 'bcde' */
substr('abcde',4,3,'-')         /* 'de-' */
```

STRIP (*string[,[<"L"|"T"|"B">][,char]]*)

```

```

returns string stripped of Leading, Trailing, or Both sets of blanks or other chars. Default is “B”.

```
strip(' abc ')          /* 'abc' */
strip(' abc ','t')              /* ' abc' */
strip('-abc--',,'-')            /* 'abc' */
```

TRANSLATE (*string[,[tableo][,[tablei][,pad]]]*)

translate characters in tablei to associated characters in tableo. If neither table is specified, convert to uppercase.

```
translate('abc')                /* 'ABC' */
translate('aabc','-','a')               /* '--bc' */
translate('aabc','-+','ab')             /* '--+c' */
```

VERIFY (*string,reference[,[option][,start]]*)

return the index of the first character in string that is not also in reference. if “Match” is given, then return the result index of the first character in string that is in reference.

```
verify('abc','abcdef')          /* 0 */
verify('a0c','abcdef')          /* 2 */
verify('12a','abcdef','m')              /* 3 */
```

XRANGE (*[start][,end]*)

return all characters in the range start through end.

```
xrange('a','e')         /* 'abcde' */
xrange('fe'x,'02'x)             /* 'feff000102'x */
```

10.3 Word Functions

DELWORD (*string, n* **[**, *length* **]**)

delete substring of string starting at the nth word and of length length words.

```
delword('one day in the year',3)                /* 'one day' */
delword('one day in the year',3,2)              /* 'one day year' */
```

FIND (*string, phrase* **[**, *start* **]**)

returns the word number of the first word of phrase in string. Returns “0” if phrase is not found. if start exists then search start from start word.

```
find('one day in the year','in the')    /* 3 */
```

JUSTIFY (*string, length* **[**, *pad* **]**)

justify string to both margins (the width of margins equals length), by adding pads between words.

```
justify('one day in the year',22)       /*'one day in the year'
```

SUBWORD (*string, n* **[**, *length* **]**)

return the substring of string that begins at the nth word and is of length length words.

```
subword('one day in the year',2,2)      /* 'day in' */
```

SPACE (*string[,[n][,pad]]*)

formats the blank-delimited words in string with n pad characters between each word.

```
space('one day in the year',2)  /*'one day in the year' */
```

WORDS (*string*)

return the number of words in string

```
words('One day in the year')            /* 5 */
```

WORD (*string, n*)

return the nth word in string

```
word('one day in the year',2)           /* 'day' */
```

WORDINDEX (*string, n*)

return the position of the nth word in string

```
wordindex('one day in the year',2)      /* 5 */
```

WORDLENGTH (*string, i*)

return the length of the nth word in string

```
wordlength('one day in the year',2)     /* 3 */
```

WORDPOS (*phrase, string* **[**, *start* **]**)

returns the word number of the first word of phrase in string. Returns “0” if phrase is not found

```
wordpos('day in','one day in the year')         /* 2 */
```

10.4 Math Functions

ABS (*number*)

return absolute value of number

```
abs(-2.3) /* 2.3 */
```

FORMAT (*number[,[before][,[after][,[expp][,expt]]]]*)

rounds and formats number with before integer digits and after decimal places. expp accepts the values 1 or 2 (WARNING Totally differen't from the Ansi-REXX spec) where 1 means to use the “G” (General) format of C, and 2 the “E” exponential format of C. Where the place of the total-width specifier in C is replaced by before+after+1. (expt is ignored!)

```
format(2.66)            /* 3 */
format(2.66,1,1)                /* 2.7 */
format(26.6,1,1,1)              /* 3.E+01 */
format(26.6,1,1,2)              /* 2.7E+01 */
```

IAND (*n, m*)

return bitwise AND of the integers n and m

```
iand(2,3) /* 2 */
```

INOT (*n*)

return bitwise complement of integers n

```
inot(2) /* -3 */
```

IOR (*n, m*)

return bitwise OR of the integers n and m

```
ior(2,3) /* 3 */
```

IXOR (*n, m*)

return bitwise XOR of the integers n and m

```
ixor(2,3) /* 1 */
```

MAX (*number* **[**, *number]..* **]**)

returns the largest of given numbers.

```
max(2,3,1,5) /* 5 */
```

MIN (*number* **[**, *number]..* **]**)

returns the smallest of given numbers.

```
min(2,3,1,5) /* 1 */
```

RANDOM (*[min][,[max][,seed]]*)

returns a pseudorandom nonnegative whole number in the range min to max inclusive.

SIGN (*number*)

return the sign of number (“-1”,”0” or “1”).

```
sign(-5.2)              /* -1 */
sign( 0.0)              /* 0 */
sign( 5.2)              /* 1 */
```

TRUNC (*number* **[**, *n* **]**)

returns the integer part of number, and n decimal places. The default n is zero.

```
trunc(2.6) /* 2 */
```

ACOS (*num*)

Arc-cosine

ASIN (*num*)

Arc-sine

ATAN (*num*)

Arc-tangent

COS (*num*)
Cosine

COSH (*num*)
Hyperbolic cosine

EXP (*num*)
Exponiate

LOG (*num*)
Natural logarithm

LOG10 (*num*)
Logarithm of base 10

POW10 (*num*)
Power with base 10

SIN (*num*)
Sine function

SINH (*num*)
Hyperbolic sine

SQRT (*num*)
Square root

TAN (*num*)
Tangent

TANH (*num*)
Hyperbolic tangent

POW (*a, b*)
Raises a to power b

10.5 Data Convert Functions

B2X (*string*)
Binary to Hexadecimal

```
b2x('01100001')         /* 'a' */
```

BITAND (*string1[,[string2][,pad]]*)
logically AND the strings, bit by bit

```
bitand('61'x,'52'x)             /* '40'x */
bitand('6162'x,'5253'x)         /* '4042'x */
bitand('6162'x,,'FE'x)          /* '6062'x */
```

BITOR (*string1[,[string2][,pad]]*)
logically OR the strings, bit by bit

BITXOR (*string1[,[string2][,pad]]*)
logically XOR the strings, bit by bit

C2D (*string* **[**, *n* **]**)

Character to Decimal. The binary representation of string is converted to a number (unsigned unless the length n is specified).

```
c2d('09'x)              /* 9 */
c2d('ff40')             /* 65344 */
c2d('81'x,1)            /* -127 */
c2d('81'x,2)            /* 129 */
```

C2X (*string*)

Character to Hexadecimal

```
c2x('abc')              /* '616263' */
c2x('0506'x)            /* '0506' */
```

D2C (*wholenumber* **[**, *n* **]**)

Decimal to Character. Return a string of length n, which is the binary representation of the number.

```
d2c(5)          /* '5'x */
d2c(97)         /* 'a' */
```

D2X (*wholenumber* **[**, *n* **]**)

Decimal to Hexadecimal. Return a string of length n, which is the hexadecimal representation of the number.

```
d2x(5)          /* '05' */
d2x(97)         /* '61' */
```

B2X (*string*)

Hexadecimal to Binary

```
x2b('a')                /* '01100001' */
```

X2C (*string*)

Hexadecimal to Character

```
x2c('616263')           /* 'abc' */
```

X2D (*hex-string* **[**, *n* **]**)

Hexadecimal to Decimal. hex-string is converted to a number (unsigned unless the length n is specified)

```
x2d('61')               /* 97 */
```

10.6 File Functions

General

There are two sets of I/O functions, the REXX-STEAM functions and the BREXX I/O routines.

Files can be referenced as a string containing the name of the file ie “TEST.DAT” or the file handle that is returned from OPEN function. (Normally the second way if prefered when you want to open 2 or more files with the same name).

There are always 3 special files:

Handle	FileName	Description
0	<STDIN>	Standard input
1	<STDOUT>	Standard output
2	<STDERR>	Standard error

All open files are closed at the end of the program from REXX interpreter except in the case of an error.

CHARIN (*[stream[,[start][,[length]]]]*)

reads length bytes (default=1) from stream (default="<STDIN>") starting at position start

```
ch = charin("new.dat")          /* read one byte */
ch = charin("new.dat",3,2)      /* read two bytes from position in file 3 */
```

CHAROUT (*[stream[,[string][,[start]]]]*)

write string to stream (default="<STDOUT>") starting at position start

```
CALL charout "new.dat","hello"  /* writes "hello" to file */
CALL charout "new.dat","hi",2)  /* writes "hi" at position 2 */
```

CHARS (**[**, *stream* **]**)

returns the number of characters remaining in stream.

```
CHARS("new.dat")        /* maybe 100 */
```

CLOSE (*file*)

closes an opened file. file may be string or the handle number

```
CALL close 'new.dat'    /* these two cmds are exactly the same */
CALL close hnd  /* where hnd=open('new.dat','w') */
```

EOF (*file*)

returns 1 at eof, -1 when file is not opened, 0 otherwise

```
DO      UNTIL eof(hnd)=1
        SAY read(hnd)   /* type file */
END
```

FLUSH (*file*)

flush file stream to disk

```
CALL flush 'new.dat'
```

LINEIN (*[stream[,[start][,[lines]]]]*)

reads lines lines (default=1) from stream (default="<STDIN>") starting at line position start

```
line = linein("new.dat")                /* read one line */
line = linein("new.dat",3,2)    /* read two lines from new.dat starting at
line 3 */
```

LINEOUT (*[stream[,[string][,[start]]]]*)

write string with newline appended at the end to stream (default="<STDOUT>") starting at line position start

```
CALL lineout "new.dat","hello"  /* writes line "hello" to file */
CALL lineout "new.dat","hi",2)  /* writes line "hi" at line position 2 */
```

LINES (**[**, *stream* **]**)

returns the number of lines remaining in stream. start

```
LINES("new.dat")        /* maybe 10 */
```

OPEN (*file*, *mode*)

opens a file. mode follows C prototypes:

“r”	for read	“w”	for write
“t”	for text,	“b”	for binary
“a”	for append	“+”	for read/write

and returns the handle number for that file. -1 if file is not found!

```
hnd     = open('new.dat','w')
IF      hnd = -1 THEN DO
        SAY 'Error: opening file "new.dat".'
        ...
END
irda = open('com3:115200,8,N,1,128','rw')
```

READ (*[file][,<length | "Char" | "Line" | "File">]*)

reads one line from file. If the second argument exists and it is a number it reads length bytes from file otherwise reads a Char, Line or the entire File. If file is not opened, it will be opened automatically in “r” mode. If file is ommited, it is assumed to be <STDIN>

```
kbdin = READ()  /* reads one line from stdin */
keypressed = read(,1)   /* -//- char -//- */
linein = read('new.dat')        /* reads one line from file */
linein = read(hnd)      /* -//- */
ch = read('new.dat',"C")        /* if file 'data' is not opened
        then it will be opened in "r" mode */
CALL write "new",read("old","F")        /* copy file */
```

SEEK (*file* **[**, *offset* **[**, *<"TOF" | "CUR" | "EOF">* **]** **]**)

move file pointer to offset relative from TOF Top Of File (default), CUR Current position, EOF End Of File and return new file pointer. This is an easy way to determine the filesize, by seeking at the end,

```
filesize = seek(file,0,"EOF")   /* return file size */
CALL seek 'data',0,"TOF"        /* sets the pointer to the start of the file
*/
filepos = seek('data',-5,"CUR")         /* moves pointer 5 bytes backwards */
```

STREAM (*stream[,[option][,command]]*)

STREAM returns a description of the state, or the result of an operation upon the stream named by the first argument.

option can be “Command”, “Description”, “Status”

When option is “Command” the third argument must exist and can take on of the following values:

Command	Description
READ	open in read-only mode ASCII

READBINARY	open in read-only mode BINARY
WRITE	open in write-only mode ASCII
WRITEBINARY	open in write-only mode BINARY
APPEND	open in read/write-append mode ASCII
APPENDBINARY	open in read/write-append mode BINARY
UPDATE	open in read/write mode (file must exist) ASCII
UPDATEBINARY	open in read/write mode BINARY

When option is “Status”, STREAM returns the current status of the stream can be on of the followings: “READY”, “ERROR”, “UNKNOWN”
When option is “Description”, STREAM returns a description of the last error.

```
 CALL stream "new.dat","C","WRITE"
 CALL stream "new.dat","C","CLOSE"
 CALL stream "new.dat","S"
```

WRITE (*[file][, string[,newline]]*)

writes the string to file. returns the number of bytes written. If string doesn’t exist WRITE will write a newline to file. If a third argument exists a newline will be added at the end of the string. If file is not opened, it will be opened automatically with “w” mode. If file is ommited, it is assumed to be <STDOUT>

```
 CALL write 'data','First line',nl;
 CALL write ,'a'          /* writes 'a' to stdout */
 CALL write '','one line',nl     /* write 'one line' to stdout */
 CALL write 'output.dat','blah blah'     /* writes 'blah blah' to 'output.dat'
file*/
```

Calling external REXX Scripts or Functions

Due to the extended calling functionality in the new version, importing of required REXX scripts is no longer necessary. You can now call any external REXX script directly.

11.1 Primary REXX Script location via fully qualified DSN

If you call a REXX script using a fully qualified partitioned dataset (PDS) member name, it must be present in the specified PDS. You can also use a fully qualified sequential dataset name that holds your script. If it is not available, an error message terminates the call. In TSO you can invoke your script using the REXX or RX commands. Example:

1. *RX 'MY.EXEC(MYREX)'* if the script resides in a PDS, alternatively:
2. *RX 'MY.SAMPLE.REXX'* if it is a sequential dataset

11.2 Location of the Main REXX script via PDS search (TSO environments)

In TSO environments the main script can be called with the RX or REXX command. The search path for finding your script is SYSUEXEC, SYSUPROC, SYSEXEC, SYSPROC. At least one of these need to be pre-allocated during the TSO logon. It is not mandatory to have all of them allocated. It depends on your planned REXX development environment. The allocations may consist of concatenated datasets.

11.3 Running scripts in batch

In batch, you can use the delivered RXTSO or RXBATCH JCL procedure and specify the REXX script and its location to execute it. There is no additional search path used to locate it.

11.4 Calling external REXX scripts

It is now possible to call external REXX scripts, either by: *CALL your-script parm1,parm2…* or by function call: *value=your-script(parm1,parm2,…)* The call might take place from within your main REXX, or from a called subroutine. The search of the called script is performed in the following sequence:

- Internal sub-procedure or label (contained in the running REXX script)
- current PDS (where the calling REXX is originated) [1]

1 only from the 1st library within a concatenation (this limitation may be lifted in a forthcoming release)

- from the delivered BREXX.RXLIB library, which then needs to be allocated with the DD-name RXLIB

11.5 Variable Scope of external REXX scripts

If the called external REXX does not contain a procedure definition, all variables of the calling REXX are accessible (read and update). If the called REXX creates new variables, they are available in the calling REXX after control is returned.

BREXX MVS Functions

12.1 Host Environment Commands

12.1.1 *ADDRESS MVS*

Interface to certain REXX environments as VSAM and EXECIO

12.1.2 *ADDRESS TSO*

Interface to the TSO commands, e.g. LISTCAT, ALLOC, FREE, etc.

12.1.3 *ADDRESS COMMAND 'CP host-command'*

Interface to the Host system in which your MVS3.8 is running. Typically it is Hercules or VM370. The result of the command is displayed on screen, but can be trapped in a stem by the OUTTRAP command:

```
call outtrap('myresult.')
ADDRESS COMMAND 'CP help'
call outtrap('off')
/* result is stored in stem myresult. */
do i=1 to myresult.0
Say mayresult.i
end
```

Some Hercules commands:

ADDRESS COMMAND 'CP HELP' to get a list of Hercules commands:

```
HHC01603I
HHC01602I Command           Description
HHC01602I ----------------  ---------------------------------
HHC01602I !message          *SCP priority message
HHC01602I #                 Silent comment
HHC01602I *                 Loud comment
HHC01602I .reply            *SCP command
HHC01602I ?                 alias for help
HHC01602I abs               *Display or alter absolute storage
HHC01602I aea               Display AEA tables
HHC01602I aia               Display AIA fields
...
```

ADDRESS COMMAND 'CP DEVLIST' shows a list of all active devices:

```
HHC02279I 0:0009 3215 *syscons cmdpref(/) IO[1541] open
HHC02279I 0:000C 3505 0.0.0.0:3505 sockdev ascii autopad trunc eof IO[3]
HHC02279I      (no one currently connected)
HHC02279I 0:000D 3525 /punchcards/pch00d.txt ebcdic IO[2] open
HHC02279I 0:000E 1403 /printers/prt00e.txt IO[6] open
HHC02279I 0:000F 3211 /printers/prt00f.txt IO[2] open
HHC02279I 0:0010 3270 GROUP=CONSOLE IO[3]
HHC02279I 0:0015 1403 /logs/mvslog.txt IO[2106] open
HHC02279I 0:001A 3505 0.0.0.0:3506 sockdev ebcdic autopad eof IO[3]
HHC02279I      (no one currently connected)
```

And many others: *ADDRESS COMMAND 'CP clocks'*:

```
HHC02274I tod = DC8F485DBB377093    2022.349 21:07:20.582007
HHC02274I h/w = DC8F485DBB377093    2022.349 21:07:20.582007
HHC02274I off = 0000000000000000       0.000 00:00:00.000000
HHC02274I ckc = DC8F485DC0400000    2022.349 21:07:20.602624
HHC02274I cpt = 7FFFFF7A01C85F00
HHC02274I itm = 7C0E1623                    07:31:40.233415
```

If you run under control of VM370 you can run VM commands: *ADDRESS COMMAND 'CP vm-command'*

12.1.4 *ADDRESS FSS*

Interface to the Formatted Screen Services. Please refer to formatted_screens.rst contained in the installation zip file.

Note The following host environments enable you to call external programs. The difference is the linkage conventions, and how input parameters are treated.

12.1.5 *ADDRESS LINK/LINKMVS/LINKPGM*

Call external an external program. The linkage convention of the called program can be found here: The LINK and ATTACH host command environments

12.1.6 *ADDRESS LINKMVS*

Call an external program. The linkage convention of the called program can be found here: The LINKMVS and ATTCHMVS host command environments

Example:

```
/* REXX - INVOKE IEBGENER WITH ALTERNATE DDNAMES. */
PROG = 'IEBGENER'
PARM = ''                    /* STANDARD PARM, AS FROM JCL */
DDLIST = COPIES('00'X,8) ||, /* DDNAME 1 OVERRIDE: SYSLIN   */
COPIES('00'X,8) ||,          /* DDNAME 2 OVERRIDE: N/A      */
COPIES('00'X,8) ||,          /* DDNAME 3 OVERRIDE: SYSLMOD  */
COPIES('00'X,8) ||,          /* DDNAME 4 OVERRIDE: SYSLIB   */
LEFT('CTL', 8) ||,           /* DDNAME 5 OVERRIDE: SYSIN    */
LEFT('REP', 8) ||,           /* DDNAME 6 OVERRIDE: SYSPRINT */
COPIES('00'X,8) ||,          /* DDNAME 7 OVERRIDE: SYSPUNCH */
LEFT('INP', 8) ||,           /* DDNAME 8 OVERRIDE: SYSUT1   */
LEFT('OUT', 8) ||,           /* DDNAME 9 OVERRIDE: SYSUT2   */
COPIES('00'X,8) ||,          /* DDNAME 10 OVERRIDE: SYSUT3  */
COPIES('00'X,8) ||,          /* DDNAME 11 OVERRIDE: SYSUT4  */
COPIES('00'X,8) ||,          /* DDNAME 12 OVERRIDE: SYSTERM */
COPIES('00'X,8) ||,          /* DDNAME 13 OVERRIDE: N/A     */
```

```
COPIES('00'X,8)              /* DDNAME 14 OVERRIDE: SYSCIN  */
ADDRESS 'LINKMVS' PROG 'PARM DDLIST'
```

12.1.7 *ADDRESS LINKPGM*

Call an external program. The linkage convention of the called program can be found here:

The LINKPGM and ATTCHPGM host command environments

12.1.8 *ADDRESS ISPEXEC*

Support calls functions to **Wally Mclaughlin ISPF** for MVS on Hercules. The functions supported depend on the functionality implemented in his API. Example:

```
ADDRESS ISPEXEC
"CONTROL ERRORS RETURN"
"DISPLAY PANEL(PANEL1)"
```

12.1.9 *OUTTRAP*

If the commands writes output to terminal you can trap the output using the *OUTTRAP* command. This will redirect it to a stem variable of your choice. Output produced by TSO full-screen macros cannot be trapped:

```
call outtrap('lcat.')
ADDRESS TSO 'LISTCAT LEVEL “BREXX”'
call outtrap('off')
/* listcat result is stored in stem lcat. */
do i=1 to lcat.0
Say lcat.i
end
```

13 Added BREXX Kernel functions and Commands

These are MVS-specific BREXX functions implemented and integrated into the BREXX kernel code. For the standard BREXX functions take a look into the BREXX User's Guide.

Note When reading the function descriptions between parentheses the argument/parameter is required unless surrounded by *[]* brackets.

13.1 Functions

ABEND (*user-abend-code*)

ABEND Terminates the program with specified User-Abend-Code. Valid values for the user evening abend-code are values between 0 and 4095.

Parameters **user-abend-code** – specified User-Abend-Code

Return type n/a

AFTER (*search-string*, *string*)

The remaining portion of the string that follows the first occurrence of the search-string within the string. If search-string is not part of string an empty string is returned.

Parameters
- **search-string** – search string
- **string** – string to search

Return type string

A2E (*ascii-string*)

Translates an ASCII string into EBCDIC. Caveat: not all character translations are biunique!

Parameters **ascii-string** – string to translate

Return type string

E2A (*ebcdic-string*)

Translates an EBCDIC string into ASCII. Caveat: not all character translations are biunique!

Parameters **ebcdic-string** – string to translate

Return type string

BEFORE (*search-string*, *string*)

The portion of the string that precedes the first occurrence of search-string within the string. If search-string is not part of string an empty string is returned.

Parameters
- **search-string** – search string
- **string** – string to search

Return type string
Example:

```
string='The quick brown fox jumps over the lazy dog'
say 'String                 'string
say 'Before Fox             'before('fox',string)
say 'After Fox              'after('fox',string)
```

Result:

```
STRING                THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG
BEFORE FOX            THE QUICK BROWN
AFTER FOX              JUMPS OVER THE LAZY DOG
```

BLDL (*program-name*)

Reports 1 if the program is callable via the active program library assignments (STEPLIB, JOBLIB, etc. DD statements). If it is not found, 0 is returned.

Parameters **program-name** – program name

Return type int

BASE64ENC (*string*)

Encodes a string or a binary string into a Base 64 encoded string. It is not an encryption process; it is, therefore, not usable for storing passwords.

Parameters **string** – string to encode

Return type string

BASE64DEC (*base64-string*)

Decodes a base64 string into a string or binary string.

Parameters **base64-string** – string to decode

Return type string
Example:

```
str='The quick brown fox jumps over the lazy dog'
stre=base64Enc(str)
say 'Encoded 'stre
strd=base64Dec(stre)
say 'Original "'strd'"'
say 'Decoded "'strd'"'
```

Result:

```
Encoded 44iFQJikiYOSQIKZlqaVQIaWp0CRpJSXokCWpYWZQKOIhUCTgamoQISWhw==
Original "The quick brown fox jumps over the lazy dog"
Decoded "The quick brown fox jumps over the lazy dog"
```

B2C (*bit-string*)

Converts bit string into a Character string

Parameters **bit-string** – string to decode

Return type string

Examples:

```
say B2C('1111000111110000') -> 10
say B2c('1100000111000010') -> AB
```

C2B (*character-string*)

Converts a character string into a bit string
Example:

```
say c2x('64'x) c2B('64'x) -> 64 01100100
say c2x(10) c2B(10)       -> F1F0 1111000111110000
say c2x('AB') c2B('AB')   -> C1C2 1100000111000010
```

C2U (*character-string*)

Converts a character string into an unsigned Integer string
Example:

```
say c2d(' B5918B39'x) -1248752839
say c2u(' B5918B39'x) 3046214457
```

D2P (*number*, *length* **[**, *fraction-digit* **]**)

D2P converts a number (integer or float) into a decimal packed field. The created field is in binary format. The fraction digit parameter is non-essential, as the created decimal does not contain any fraction information, for symmetry reasons to the P2D function it has been added.

P2D (*number*, *length*, *fraction-digit*)

P2D converts a decimal packed field (binary format) into a number.

CEIL (*decimal-number*)

CEIL returns the smallest integer greater or equal than the decimal number.

CONSOLE (*operator-command*)

Performs an operator command, but does not return any output. If you need the output for checking the result, please use the RXCONSOL function.

ENCRYPT (*string*, *password*)

Encrypts a string via a password. The encryption/decryption method is merely XOR-ing the string with the password in several rounds. This means the process is not foolproof and has not the quality of an RSA encryption.

DECRYPT (*string*, *password*)

Decrypts an encrypted string via a password. The encryption/decryption method is merely XOR-ing the string with the password in several rounds. This means the process is not foolproof and has not the quality of an RSA encryption.

Example:

```
a10='The quick brown fox jumps over the lazy dog'
a11=encrypt(a10,"myPassword")
a12=decrypt(a11,"myPassword")
say "original "a10
say "encrypted "c2x(a11)
say "decrypted "a12
```

Result:

```
original The quick brown fox jumps over the lazy dog
encrypted
E361A8D7F001D537D0D6CDCAF9EFD83CCA00F984897FBD538AAF964CA80E2806D4310205CEFAC709C9EACB43
decrypted The quick brown fox jumps over the lazy dog
```

DEFINED (*'variable-name'*)

Tests if variable or STEM exists, to avoid variable substitution, the variable-name must be enclosed in quotes. return values:

Return Value	Description
-1	not defined, but would be an invalid variable name
0	variable-name is not a defined variable
1	variable-name is defined it contains a string
2	variable-name is defined it contains a numeric value

To test whether a variable is defined, you can use: *If defined('myvar')> 0 then ..*

DUMPIT (*address*, *dump-length*)

DUMPIT displays the content at a given address of a specified length in hex format. The address must be provided in hex format; therefore, a conversion with the D2X function is required.
Example:

```
call mvscbs
/* load MVS CB functions */
call dumpit d2x(tcb()),256
```

Result:

```
009B8148 (+00000000) | 009AE448 00000000 009AD99C 009BF020 | ..U.......R...0.
009B8158 (+00000010) | 00000000 00000000 009B2578 80000000 | ................
009B8168 (+00000020) | 0000FFFF 009C7C88 0013B908 00000000 | ......@h........
009B8178 (+00000030) | 40D871C0 009DA1E0 002C13C0 002C1434 |  Q.{...\...{....
009B8188 (+00000040) | 002C1434 002C30A8 00000085 009ADA3C | .......y...e....
009B8198 (+00000050) | 00000002 00158000 00262F88 4025EBD0 | ...........h ..}
009B81A8 (+00000060) | 00BF52C0 0027F268 4026306E 00000000 | ...{..2. ..>....
009B81B8 (+00000070) | 001D4FB8 00000000 00000000 009DC330 | ..|...........C.
009B81C8 (+00000080) | 00000000 009B8730 00000000 00000000 | ......g.........
009B81D8 (+00000090) | 001D3048 00000000 009DF548 00000000 | ..........5.....
009B81E8 (+000000A0) | 009B23C4 809D5F88 00000000 00000000 | ...D..¬h........
009B81F8 (+000000B0) | 00000000 009DC6EC 00000000 00000000 | ......F.........
009B8208 (+000000C0) | 00000000 00000000 00000000 00000000 | ................
009B8218 (+000000D0) | 009B8270 00000000 00000000 009B8730 | ..b...........g.
009B8228 (+000000E0) | 00000000 00000000 00000000 00000000 | ................
009B8238 (+000000F0) | 80000040 00000000 009B2E58 00000000 | ... ............
```

DUMPVAR (*'variable-name'*)

DUMPVAR displays the content of a variable or stem-variable in hex format; the displayed length is variable-length +16 bytes. The variable name must be enclosed in quotes. If no variable is specified, all so far allocated variables are printed.
Example:

```
v21.1='Stem Variable, item 1'
v21.2='Stem Variable, item 2'
v21.3='Stem Variable, item 3'
call DumpVAR('v21.1')
```

Result:

```
002C2818 (+00000000) | E2A38594 40E58199 89818293 856B4089 | Stem Variable, i
002C2828 (+00000010) | A3859440 F1000000 00000000 00000000 | tem 1...........
```

DATE (**[**, *date-target-format* **]**)

The integrated DATE function replaces the RXDATE version stored in RXLIB. RXDATE will be available to guarantee consistency of existing REXX scripts. It may be removed in a future release.
The three arguments are options. *date* defaults to today,
Supported input formats:

Format	Description
Base	days since 01.01.0001
JDN	days since Monday 24. November 4714 BC
UNIX	days since 1. January 1970
DEC	01-JAN-20 DEC format (Digital Equipment Corporation)
XDEC	01-JAN-2020 extended DEC format (Digital Equipment Corporation)
Julian	yyyyddd e.g. 2018257

European	dd/mm/yyyy e.g. 11/11/18
xEuropean	dd/mm/yyyy e.g. 11/11/2018, extended European (4 digits year)
German	dd.mm.yyyy e.g. 20.09.2018
USA	mm/dd/yyyy e.g. 12.31.18
xUSA	mm/dd/yyyy e.g. 12.31.2018, extended USA (4 digits year)
STANDARD	yyyymmdd e.g. 20181219
ORDERED	yyyy/mm/dd e.g. 2018/12/19
LONG	dd month-name yyyy e.g. 12 March 2018, month is translated into month number (first 3 letters)
NORMAL	dd 3-letter-month yyyy e.g. 12 Mar 2018, month is translated into month number
QUALIFIED	Thursday, December 17, 2020
INTERNATIONAL	date format 2020-12-01
TIME	date since 1.1.1970 in seconds

Supported output formats:

Format	Description
Base	days since 01.01.0001
Days	ddd days in this year e.g. 257
Weekday	weekday of day e.g. Monday
Century	dddd days in this century
JDN	days since Monday 24. November 4714 BC
UNIX	days since 1. January 1970
DEC	01-JAN-20 DEC format (Digital Equipment Corporation)
XDEC	01-JAN-2020 extended DEC format (Digital Equipment Corporation)
Julian	yyyyddd e.g. 2018257
European	dd/mm/yyyy e.g. 11/11/18
xEuropean	dd/mm/yyyy e.g. 11/11/2018, extended European (4 digits year)
German	dd.mm.yyyy e.g. 20.09.2018
USA	mm/dd/yyyy e.g. 12.31.18
xUSA	mm/dd/yyyy e.g. 12.31.2018, extended USA (4 digits year)
STANDARD	yyyymmdd e.g. 20181219
ORDERED	yyyy/mm/dd e.g. 2018/12/19
LONG	dd month-name yyyy e.g. 12 March 2018, month is translated into month number (first 3 letters)
NORMAL	dd 3-letter-month yyyy e.g. 12 Mar 2018, month is translated into month number
QUALIFIED	Thursday, December 17, 2020
INTERNATIONAL	date format 2020-12-01
TIME	date since 1.1.1970 in seconds

DATETIME ([, *target-format*])

Formats a timestamp into various representations.

Parameters

- **target-format** – *optional* target-format defaults to Ordered
- **timestamp** – *optional* timestamp defaults to today current time
- **input-format** – *optional* input-format defaults to Timestamp

Return type string

Formats are:

Format	Description	Example
T	is timestamp in seconds	1615310123 (seconds since 1. January 1970)
E	timestamp European format	09/12/2020-11:41:13
U	timestamp US format	12.09.2020-11:41:13
O	Ordered Time stamp	2020/12/09-11:41:13
B	Base Time stamp	Wed Dec 09 07:40:45 2020

Time (*string*)

Time has gotten new input parameters. *String* can be one of:

- *MS* Time of today in seconds.milliseconds
- *US* Time of today in seconds.microseconds
- *CPU* Used CPU time in seconds.milliseconds

FILTER (*string, character-table* **[**, *filter-type* **]**)

The filter function removes all characters defined in the character table if 'drop' is used as filter-type. If 'keep' is specified, just those characters which are in the character table are kept. Filter-type defaults to drop.

Parameters
- **string** – string to filter
- **character-table** – filter table
- **filter-type** – *optional* either *drop* or *keep*

Return type string

For example, remove 'o' and 'blank':

```
say FILTER('The quick brown fox jumps over the lazy dog',' o')
Thequickbrwnfxjumpsverthelazydg
```

FLOOR (*decimal-number*)

FLOOR returns the smallest integer less or equal to the decimal number.

INT (*decimal-number*)

INT returns the integer value of a decimal number. Fraction digits are stripped off. There is no rounding in place. It's faster than saying *intValue=number%1*

JOBINFO ( )

Returns jobname and additional information about currently running job or TSO session in REXX variables, like JOB.NAME, JOB.NUMBER, STEP.NAME, PROGRAM.NAME

Example:

```
say jobinfo()
say job.name
say job.number
say job.step
say job.program
```

Result:

```
PEJ
PEJ
TSU02077
ISPFTSO.IS
```

JOIN (*string*, *target-string* **[**, *join-table* **]**)

Join merges a string into a target-string. The merge occurs byte by byte; if the byte in target-string is defined in the join-table. The join-table consists of one or more characters, which may be over-written. If it is in the target-string, it is replaced by the equivalent byte of the string. If it is not part of the join-table, it remains as it is. If the length of the string is greater than the target-string size is appending the target-string. The join-table is an optional parameter and defaults to blank.
Example:

```
SAY JOIN('     PETER        MUNICH','NAME=        CITY=          ')
NAME=PETER   CITY=MUNICH
```

LEVEL ( )

Level returns the current procedure level. The level information is increased by +1 for every CALL statement or function call.
Example:

```
say 'Entering MAIN 'Level()
call proc1
say 'Returning from proc1 'Level()
return

proc1:
   say 'Entering proc1 'Level()
   call proc2
   say 'Returning from proc2 'Level()
return 0

proc2: procedure
   if level()>5 then return 4
   say 'Entering proc2 'Level()
   prc=proc1()
   say 'Returning from proc1 'Level()
return 0
```

Result:

```
ENTERING MAIN 0
ENTERING PROC1 1
ENTERING PROC2 2
ENTERING PROC1 3
ENTERING PROC2 4
ENTERING PROC1 5
RETURNING FROM PROC2 5
RETURNING FROM PROC1 4
RETURNING FROM PROC2 3
RETURNING FROM PROC1 2
RETURNING FROM PROC2 1
RETURNING FROM PROC1 0
```

LINKMVS (*load-module*, *parms*)

Starts a load module. Parameters work according to standard conventions.

LINKPGM (*load-module*, *parms*)

Starts a load module. Parameters work according to standard conventions.

LISTIT (*'variable-prefix'*)

Returns the content of all variables and stem-variables starting with a specific prefix. The prefix must be enclosed in quotes. If no prefix is defined all variables are printed
Example:

```
v2='simple Variable'
v21.0=3
v21.1='Stem Variable, item 1'
v21.2='Stem Variable, item 2'
v21.3='Stem Variable, item 3'
call ListIt 'V2'
```

Result:

```
List Variables with Prefix 'V2'
-------------------------------
[0001]  "V2" => "SIMPLE VARIABLE"
[0002]  "V21." =>
>[0001] "|.0" => "3"
>[0002] "|.1" => "STEM VARIABLE, ITEM 1"
>[0003] "|.2" => "STEM VARIABLE, ITEM 2"
>[0004] "|.3" => "STEM VARIABLE, ITEM 3"
```

LOCK (*'lock-string',['lock-modes'][,timeout]*)

Locks a resource (could be any string, e.g. dataset-name) for usage by a concurrent program (which must request the same resource). Typically it is used to keep the integrity of several datasets.

Parameters

- **lock-string** – resource to lock
- **lock-modes** – *optional* One of TEST/SHARED/EXCLUSIVE. *TEST* tests whether the resource is available. *SHARED* shared access is wanted, other programs/-tasks are also shared access granted, but no exclusive lock can be granted, while a shared lock is active, *EXCLUSIVE* no other program/task can use the resource at this point.
- **timeout** – *optional* defines a maximum wait time in milliseconds to acquire the resource. If no timeout is defined the LOCK ends immediately if it couldn't be acquired.

Returns **0** if resource was locked, **4** resource could not be acquired in the requested time interval

UNLOCK (*'lock-string'*)

Unlocks a previously locked resource.

Returns 0 unlock was successful

MEMORY ( )

Determines and print the available storage junks:

```
MVS Free Storage Map
----------------------------
AT ADDR  3121152    5796 KB
Total               5796 KB
----------------------------
5935104
```

MTT (**[**, *'REFRESH'* **]**)

Returns the content of the Master Trace Table in the stem variable *_LINE.*, *_LINE.0* contains the number ofreturned trace table entries. The return code contains the number of trace table entries fetched.If **-1** is returned the Master Trace Table has not been changed since the last call, _LINE. remains unchanged.

If the optional *'REFRESH'* option is used, the Trace Table will be recreated even it it has not

changed.
Example:

```
RC = MTT()
SAY RC
IF RC > -1 THEN DO
  DO I=1 TO _LINE.0
    SAY _LINE.I
  END
END
```

Result:

```
...
0000 10.05.00           S ZTIMER
0200 10.05.00 STC  706  $HASP100 ZTIMER   ON STCINRDR
4000 10.05.00 STC  706  $HASP373 ZTIMER   STARTED
4000 10.05.00 STC  706  IEF403I ZTIMER - STARTED - TIME=10.05.00
0000 10.05.00 STC  706  $TA99,T=12.05,'$VS,''S ZTIMER'''
8000 10.05.00           $HASP000 ID 99   T=12.05 I=   0 $VS,'S ZTIMER'
4000 10.05.00 STC  706  IEF404I ZTIMER - ENDED - TIME=10.05.00
4000 10.05.00 STC  706  $HASP395 ZTIMER   ENDED
0000 12.05.00           S ZTIMER
0200 12.05.00 STC  707  $HASP100 ZTIMER   ON STCINRDR
4000 12.05.00 STC  707  $HASP373 ZTIMER   STARTED
4000 12.05.00 STC  707  IEF403I ZTIMER - STARTED - TIME=12.05.00
0000 12.05.00 STC  707  $TA99,T=14.05,'$VS,''S ZTIMER'''
8000 12.05.00           $HASP000 ID 99   T=14.05 I=   0 $VS,'S ZTIMER'
4000 12.05.00 STC  707  IEF404I ZTIMER - ENDED - TIME=12.05.00
4000 12.05.00 STC  707  $HASP395 ZTIMER   ENDED
0000 14.05.01           S ZTIMER
0200 14.05.01 STC  708  $HASP100 ZTIMER   ON STCINRDR
4000 14.05.01 STC  708  $HASP373 ZTIMER   STARTED
4000 14.05.01 STC  708  IEF403I ZTIMER - STARTED - TIME=14.05.01
0000 14.05.01 STC  708  $TA99,T=16.05,'$VS,''S ZTIMER'''
8000 14.05.01           $HASP000 ID 99   T=16.05 I=   0 $VS,'S ZTIMER'
4000 14.05.01 STC  708  IEF404I ZTIMER - ENDED - TIME=14.05.01
4000 14.05.01 STC  708  $HASP395 ZTIMER   ENDED
0000 16.05.00           S ZTIMER
0200 16.05.00 STC  709  $HASP100 ZTIMER   ON STCINRDR
4000 16.05.00 STC  709  $HASP373 ZTIMER   STARTED
4000 16.05.00 STC  709  IEF403I ZTIMER - STARTED - TIME=16.05.00
0000 16.05.00 STC  709  $TA99,T=18.05,'$VS,''S ZTIMER'''
8000 16.05.00           $HASP000 ID 99   T=18.05 I=   0 $VS,'S ZTIMER'
...
```

MTTSCAN ( )

MTTSCAN is an application that constantly analyses the Master Trace Table and passes control to the user's procedures for a registered function to perform user actions.
Example in *BREXX.V2R5M1.SAMPLE(MTTSCANT)*
In this example, the trace entries *$HASP373 (LOGON)* and *$HASP395 (LOGOFF)* are registered, and the associated call-back procedures will be called to initiate further actions.
Example:

```
/* ----------------------------------------------------------------
 * Scan Master Trace Table for LOGON/LOGOFF actions
 * ----------------------------------------------------------------
 */
/* ----------------------------------------------------------------*/
/*           + ---  REGISTER requestesd action                     */
```

```
/*            |           + --- action keyword in trace table        */
/*            |           |          + --- associated call back proc */
/*            Y           Y          Y                               */
call mttscan 'REGISTER','$HASP373','hasp373'
call mttscan 'REGISTER','$HASP395','hasp395'

/*            + ---  Start scanning Trace Table                     */
/*            |     + --- scan frequency in millisedonds            */
/*            Y     Y          default is 5000                      */
call mttscan 'SCAN',2000
return
/* ------------------------------------------------------------------
 * Call Back to handle $HASP373 Entries of the Trace Table: user LOGON
 *    arg(1) contains the selected line of the Trace Table
 * ------------------------------------------------------------------
 */
hasp373:
  user=word(arg(1),6)
/* call console 'c u='user     You can for example cancel the user   */
  say user ' has logged on'
  say 'Trace Table entry: 'arg(1)
  say copies('-',72)
return
/* ------------------------------------------------------------------
 * Call Back to handle $HASP395 Entries of the Trace Table: user LOGOFF
 *   arg(1) contains the selected line of the Trace Table
 * ------------------------------------------------------------------
 */
hasp395:
  user=word(arg(1),6)
  say user ' has logged off'
  say 'Trace Table entry: 'arg(1)
  say copies('-',72)
return
```

RXCONSOL ( )

An application that returns the output of a requested Console command in the stem variable *CONSOLE.n*

Returns >0 the command output could not be identified in the Master Trace Table
Example in *BREXX.V2R5M1.SAMPLE(CONSOLE)*:

```
/* ------------------------------------------------------------
 *  RXCONSOL Sample: Show output of a Console command
 * ------------------------------------------------------------
 */
call rxconsol('D A,L')
say copies('-',72)
say center('Console Output of D A,L',72)
say copies('-',72)
do i=1 to console.0
   say console.i
end
```

Warning The result of an operator command is not synchronously returned, but asynchronously assigned via the activity number. In certain situations, this may fail, then an exact match of operator command and its output is impossible. You will then see more output than expected.

NJE38CMD ( )

An application that returns the output of a requested NJE38 command in the stem variable *NJE38.n*

Returns >0 means the NJE38 command output could not be identified in the Master Trace Table
Example in *BREXX.V2R5M1.SAMPLE(NJECMD)*

```
/* ------------------------------------------------------------
 *  NJE38 Sample: Show available files in NJE38 inbox
 *    pass command to NJE38CMD and retrieve output
 * ------------------------------------------------------------
 */
rc=nje38CMD('NJE38 D fILes')
if rc>0 then do
   say 'Unable to pickup NJE38 results'
   return 8
end
say copies('-',72)
say center('NJE38 Spool Queue',72)
say copies('-',72)
do i=1 to nje38.0
   say nje38.i
end
```

Result:

```
------------------------------------------------------------------------
                          NJE38 SPOOL QUEUE
------------------------------------------------------------------------
NJE014I  File status for node DRNBRX3A
File  Origin   Origin    Dest     Dest
 ID   Node     Userid    Node     Userid    CL  Records
0006  DRNBRX3A PEJ1      DRNBRX3A PEJ       A   50
0010  CZHETH3C FIX0MIG   DRNBRX3A MIG       A   119
Spool 00% full
```

VLIST (*pattern* [, *"VALUES"/"NOVALUES"*])

VLIST scans all defined REXX-variable names for a specific pattern. This is mainly for stem-variables useful, where they can have various compound components. The pattern must be coded in the form *p1.p2.p3.p4.p5*, p1, p2, p3,p4,p5 are subpatterns that must match the stem variable name. There are up to 5 subpatterns allowed. You may use * as a subpattern for any variable in this position.
Example:

```
ADDRESS.PEJ.CITY='Munich'
ADDRESS.MIG.CITY='Berlin'
ADDRESS.pej.pub='Hofbrauhaus'
ADDRESS.mig.pub='Steakhaus'
ADDRESS='set'
call xlist('*.*.CITY')
call xlist('ADDRESS')
call xlist('ADDRESS.*.CITY')
call xlist('ADDRESS.PEJ')
call xlist('ADDRESS.MIG')
call xlist()
exit
xlist:
say '>>> 'arg(1)
say vlist(arg(1))
return
```

Result:

```
>>> *.*.CITY
ADDRESS.MIG.CITY="BERLIN"
ADDRESS.PEJ.CITY="MUNICH"

>>> ADDRESS
ADDRESS="SET"
ADDRESS.MIG.CITY="BERLIN"
ADDRESS.MIG.PUB="STEAKHAUS"
ADDRESS.PEJ.CITY="MUNICH"
ADDRESS.PEJ.PUB="HOFBRAUHAUS"

>>> ADDRESS.*.CITY
ADDRESS.MIG.CITY="BERLIN"
ADDRESS.PEJ.CITY="MUNICH"

>>> ADDRESS.PEJ
ADDRESS.PEJ.CITY="MUNICH"
ADDRESS.PEJ.PUB="HOFBRAUHAUS"

>>> ADDRESS.MIG
ADDRESS.MIG.CITY="BERLIN"
ADDRESS.MIG.PUB="STEAKHAUS"

>>>
ADDRESS="SET"
ADDRESS.MIG.CITY="BERLIN"
ADDRESS.MIG.PUB="STEAKHAUS"
ADDRESS.PEJ.CITY="MUNICH"
ADDRESS.PEJ.PUB="HOFBRAUHAUS"
SIGL="11"
VLIST.0="2"
```

LASTWORD (*string*)

Returns the last word of the provided string.

PEEKS (*decimal-address*, *length*)

PEEKS returns the content (typically a string) of a main-storage address in a given length. The address must be in decimal format.

PEEKS is a shortcut of *STORAGE(d2x(decimal-address),length)*.

PEEKA (*decimal-address*)

PEEKA returns an address (4 bytes) stored at a given address. The address must be in decimal format.

PEEKA is a shortcut of *STORAGE(d2x(decimal-address),4)*.

PEEKU (*decimal-address*)

PEEKU returns an unsigned integer stored at the given decimal address (4 bytes). The address must be in decimal format.

RACAUTH (*userid*, *password*)

The RACFAUTH function validates the userid and password against the RAKF definitions. If both pieces of information are valid, a one is returned.

RHASH (*string* **[**, *slots* **]**)

The function returns a numeric hash value of the provided string. The optional slots parameter defines the highest hash number before it restarts with 0. Slots default to 2,147,483,647 Even before reaching the maximum slot, the returned number is not necessarily unique; it may repeat

(collide) for various strings. The calculation is based on a polynomial rolling hash function

ROUND (*decimal-number*, *fraction-digits*)

The function rounds a decimal number to the precision defined by fraction-digits. If the decimal number does not contain the number of fraction digits requested, it is padded with 0s.

ROTATE (*string*, *position* **[**, *length* **]**)

The function is a rotating substring if the requested length for the substring is not available, it takes the remaining characters from the beginning of the string. If the optional length parameter is not coded, the length of the string is used.
Example:

```
Rotate("1234567890ABCDEF",10,10)
Rotate("1234567890ABCDEF",1)
Rotate("1234567890ABCDEF",5)
```

Result:

```
'0ABCDEF123'
'1234567890ABCDEF'
'567890ABCDEF1234'
```

PUTSMF (*smf-record-type*, *smf-message*)

Writes an SMF message of type smf-record-type. If you use a defined type with a certain structure, it must be reflected in smf-message. If necessary you can use den BREXX conversion functions (D2C, D2P, etc.) to create binary data.

SUBMIT (*options*)

Submits a job via the internal reader to your MVS system. Options are:

- fully qualified dataset name containing the JCL
- stem variable containing the JCL
- stack containing the JCL

Example:

```
submit("'pds-name(member-name)'") submit a DSN or a member in a PDS
submit('stem-variable.')          submit JCL stored in stem-variable
submit('*')                       submit JCL stored in a stack (queue)
```

Warning The internal reader has no knowledge of your userid, therefore the *&SYSUID* variable will not be resolved with your userid. It also does not return any “SUBMIT” message, this can easily be achieved by a small rexx script analysing the master trace table.

SPLIT (*string*, *stem-variable* **[**, *delimiter* **]**)

SPLIT splits a string into its words and store them in a stem variable. The optional delimiter table defines the split character(s), which shall be used to separate the words. SPLIT returns the number of found words. Also, *stem-variable.0* contains the number of words. The words are stored in the *stem-variable.1*, *stem-variable.2*, etc. It is recommended to enclose the receiving stem-variable-name in quotes.
Example:

```
Say Split('The quick brown fox jumps over the lazy dog','myStem.')
Call LISTIT
```

Result:

```
9
```

```
List all Variables
------------------
[0001]  "MYSTEM." =>
>[0001] "|.0" => "9"
>[0002] "|.1" => "THE"
>[0003] "|.2" => "QUICK"
>[0004] "|.3" => "BROWN"
>[0005] "|.4" => "FOX"
>[0006] "|.5" => "JUMPS"
>[0007] "|.6" => "OVER"
>[0008] "|.7" => "THE"
>[0009] "|.8" => "LAZY"
>[0010] "|.9" => "DOG"
```

Example with list of word delimiters:

```
say split('City=London,Address=Picadelly Circus 24(7th
floor)','mystem.','()=,')
call listit
```

Result:

```
5
List all Variables
------------------
[0001]  "MYSTEM." =>
>[0001] "|.0" => "5"
>[0002] "|.1" => "CITY"
>[0003] "|.2" => "LONDON"
>[0004] "|.3" => "ADDRESS"
>[0005] "|.4" => "PICADELLY CIRCUS 24"
>[0006] "|.5" => "7TH FLOOR"
[0002]  "X" => "CITY=LONDON,ADDRESS=PICADELLY CIRCUS 24(7TH FLOOR)"
```

SPLITBS (*string*, *stem-variable* **[**, *split-string* **]**)

SPLIT splits a string into its words and store them in a stem variable. The split-string defines the string which shall be used to separate the words. SPLIT returns the number found words. Also, *stem-variable.0* contains the number of words. The words are stored in the *stem-variable.1*, *stem-variable.2*, etc. It is recommended to enclose the receiving stem-variable-name in quotes.

Example:

```
say splitbs('today</N>tomorrow</N>yesterday','mystem.','</N>')
call listit 'mystem.'
```

result:

```
3
List Variables with Prefix 'MYSTEM.'
-----------------------------------
[0001]  "MYSTEM." =>
>[0001] "|.0" => "3"
>[0002] "|.1" => "TODAY"
>[0003] "|.2" => "TOMORROW"
>[0004] "|.3" => "YESTERDAY"
```

EPOCHTIME (**[**, *daymonthyear* **]**)

EPOCHTIME returns the Unix (epoch) time of a given date. It's the seconds since 1. January 1970. You can easily extend the date by adding the seconds of the day.

As calculation internally is done on integer fields, the maximum date which is supported is 19 January 2038 04:14:07. If no parameters are specified, the current date/time will be returned.

Example:

```
time= EPOCHTIME(1,1,2000)+3600*hours+60*minutes+seconds
```

EPOCH2DATE (*unix-epochtime*)

EPOCH2DATE translates a Unix (epoch) time-stamp into a readable date/time format. Internally the date conversion is done by the RXDATE module of RXLIB

Example:

```
tstamp=EPOCHTIME()
say tstamp
SAY EPOCH2DATE(tstamp)
```

Result:

```
1600630022
20/09/2020 19:27:02
```

STIME ( )

Time since midnight in hundreds of a second

USERID ( )

USERID returns the identifier of the currently logged-on user. (available in Batch and Online)

UPPER (*string*)

UPPER returns the provided string in upper cases.

LOWER (*string*)

LOWER returns the provided string in lower cases.

MOD (*number*, *divisor*)

MOD divides and returns the remainder, equivalent to the // operation.

VERSION (**[**, *'FULL'* **]**)

Returns BREXX/370 version information, if FULL is specified the Build Date of BREXX is added and returned.

Example:

```
SAY VERSION()       -> V2R5M1
SAY VERSION('FULL') -> Version V2R5M1 Build Date 15. Jan 2021
```

WAIT (*wait-time*)

Stops REXX script for some time, wait-time is in thousands of a second

WORDDEL (*string*, *word-to-delete*)

WORDDEL removes a specific word from the string. If the specified word does not exist, the full string is returned.

Example:

```
say worddel('I really love Brexx',1)
say worddel('I really love Brexx',2)
say worddel('I really love Brexx',3)
say worddel('I really love Brexx',4)
say worddel('I really love Brexx',5)
```

Result:

```
REALLY LOVE BREXX
I LOVE BREXX
```

```
I REALLY BREXX
I REALLY LOVE
I REALLY LOVE BREXX
```

WORDINS (*new-word*, *string*, *after-word-number*)

WORDINS inserts a new word after the specified word number. If 0 is used as wobaserd number it is inserted at the beginning of the string.
Example:

```
say wordins('really','I love BREXX',1)
say wordins('really','I love BREXX',2)
say wordins('really','I love BREXX',3)
say wordins('really','I love BREXX',0)
```

Result:

```
I REALLY LOVE BREXX
I LOVE REALLY BREXX
I LOVE BREXX REALLY
REALLY I LOVE BREXX
```

WORDREP (*new-word*, *string*, *word-to-replace*)

WORDREP replace a word value by a new value.
Example:

```
say wordrep('!!!','I love Brexx',1)
say wordrep('!!!','I love Brexx',2)
say wordrep('!!!','I love Brexx',3)
```

Result:

```
!!! LOVE BREXX
I !!! BREXX
I LOVE !!!
```

WTO (*console-message*)

Write a message to the operator's console. It also appears in the JES Output of the Job.

XPULL ( )

PULL function which returns the stack content casesensitive.

14 GLOBAL Variables

You can define global variables which can be accessed from within the rexx whatever the current procedure variable scope is. STEMS are not supported.

SETG (*'variable-name'*, *'content'*)

SETG sets or updates a variable with the given content.

GETG (*'variable-name'*)

GETG returns the current content of the global variable.
Example:

```
call setg('ctime',time('l'))
call setg('city','Munich')
call testproc
exit 0

testproc: procedure
  /* normal variable scope can't access variables from the calling rexx */
  say 'Global Variables from the calling REXX'
  say getg('ctime')
  say getg('city')
return 0
```

Result:

```
GLOBAL VARIABLES FROM THE CALLING REXX
19:45:12.538474
MUNICH
```

Dataset Functions

CREATE (*dataset-name*, *allocation-information*)

The CREATE function creates and catalogues a new dataset (if the user has the required authorisation level). If dataset-name is not fully qualified, it will be prefixed by the user name.
Fully qualified DSN is: *BREXX.TEST.SEQ*
Not fully qualified: *TEST.SEQ* will be prefixed by user name (e.g. HERC01) "HERC01.TEST.SQ"

Parameters **allocation-information** – can be: *DSORG*, *RECFM*, *BLKSIZE*, *LRECL*, *PRI*, *SEC*, *DIRBLKS*, *UNIT* (not all are mandatory):.

The space allocations for PRI (primary space) and SEC (secondary space) is the number of tracks.

Returns If the create is successful, the return code will be zero; else a negative value will be returned. The CREATE function does not open the dataset.

Return codes:

- 0 Create was successful
- -1 Dataset cannot be created (various reasons as, space limitations, authorisation, etc.)
- -2 Dataset is already catalogued

Example:

```
CREATE('TEST','recfm=fb,lrecl=80,blksize=3120,unit=sysda,pri=5,DIRBLKS=5')
```

DIR (*partitioned-dataset-name*)

The DIR command returns the directory of a partitioned dataset. If the partitioned-dataset is not fully qualified, it will be prefixed by the user name. The directory is provided in the stem variable *DIRENTRY.*.

Table showing the structure of the returned stem. **n is the number of the member entry**.

STEM Name	Description
DIRENTRY.0	contains the number of directory members
DIRENTRY.n.CDATE	creation date of the member, e.g. => "19-04-18"
DIRENTRY.n.INIT	initial size of member
DIRENTRY.n.MOD	mod level
DIRENTRY.n NAME	member name
DIRENTRY.n.SIZE	current size of member
DIRENTRY.n.TTR	TTR of member
DIRENTRY.n.UDATE	last update date, e.g. " 20-06-09"
DIRENTRY.n.UID	last updated by user- id
DIRENTRY.n.UTIME	last updated time
DIRENTRY.n.CDATE	creation date

EXISTS (*dataset-name/partitioned-dataset(member)*)

The EXISTS function checks the existence of a dataset or the presence of a member in a partitioned dataset. EXISTS returns 1 if the dataset or the member in a partitioned dataset is available. It returns 0 if it does not exist. If the dataset-name is not fully qualified, it will be prefixed by the user name.

REMOVE (*dataset-name/partitioned-dataset(member)*)

The REMOVE function un-catalogues and removes the specified dataset (if the user has the required authorisation level). If dataset-name is not fully qualified, it will be prefixed by the user name.If the removal is successful, the return code will be zero; else a negative value will be returned. Return codes:

- 0 Create was successful
- -1 Dataset cannot be created (various reasons as, space limitations, authorisation, etc.)
- -2 Dataset is already catalogued

The REMOVE function on members of a partitioned dataset removes the specified member (if the user has the required authorisation level). If dataset-name is not fully qualified, it will be prefixed by the user name. If the removal is successful, the return code will be zero; else a negative value will be returned.

RENAME (*old-dataset-name*, *new-dataset-name*)

The RENAME function renames the specified dataset. The user requires the authorisation for the dataset to rename as well as the new dataset. If dataset-name is not fully qualified, it will be prefixed by the user name. If the rename is successful, the return code will be zero; else a negative value will be returned.

The RENAME function on members renames the specified member into a new one. The user requires the authorisation for the dataset. The RENAME must be performed in the same partitioned dataset. If the rename is successful, the return code will be zero; else a negative value will be returned.

ALLOCATE (*ddname*, *dataset-name/partitioned-dataset(member-name)*)

The ALLOCATE function links an existing dataset or a member of a partitioned dataset to a dd-name, which then can be used in services requiring a dd-name. If dataset-name is not fully qualified, it will be prefixed by the user name.

If the allocation is successful, the return code will be zero; else a negative value will be returned.

FREE (*ddname*)

The FREE function de-allocates an existing allocation of a dd-name. If the de-allocation is successful, the return code will be zero; else a negative value will be returned.

OPEN (*dataset-name*, *open-option*, *allocation-information*)

The OPEN function has now a third parameter, which allows creating new datasets with appropriate DCB and system definitions. If the dataset already exists, the existing definition is used, the DCB is not updated. If the dataset-name is not fully qualified, it will be prefixed by the user name. The dataset-name may contain a member name, which must be enclosed within parenthesis. e.g. *OPEN(""myPDS(mymember)"")*

If the open is performed with the read-option, the member name must be present, else the open fails. If the write-option is used, you can refer to a member-name that does not yet exist and will be created by following write commands. If the member name exists, the current content will be overwritten. The open-options have not changed, please refer to the official BREXX documentation.

Parameters **allocation-information** – can be: *DSORG*, *RECFM*, *BLKSIZE*, *LRECL*, *PRI*, *SEC*, *DIRBLKS*, *UNIT* (not all are mandatory).

The space allocations for PRI (primary space) and SEC (secondary space) is the number of tracks.

If the open is successful, a file handle (greater zero) will be returned; it will be less or equal zero if the open is not successful.

Warning Important notice: opening a member of a partitioned dataset in write mode requires full control of the entire dataset (not just the member), if you edit or browse the member concurrently the open will fail.

'EXECIO'

The EXECIO is a **host** command; therefore, it is enclosed in apostrophes.
EXECIO performs data set I/O operations either on the stack or stem variables, it supports only dataset containing text records. For records containing binary data you can use There is just a subset of the known EXECIO functions implemented: Full read/write from a dd-name. The ddname must be allocated either by TSO ALLOC command, or DD statement in the JCL. Specifying a Dataset-Name (DSN) is not supported!
Syntax: *EXECIO <lines-to-read/*> <DISKR/DISKW/LIFOR/LIFOW/FIFOR/FIFOW> (<STEM stem-variable-name/LIFO/FIFO> [SKIP skip-lines] [START first-stem-entry] [KEEP keep-string] [DROP drop-string] [SUBSTR(offset,length)]*

EXECIO Param	Description
Lines-to	read is the number of records which shall be read from the file, * means read all records
DISKR	read from dataset
DISKW	write into dataset
LIFOR/FIFOR	read from stack, stack structure can't be changed, it is fixed by the ways it was created
LIFOW/FIFOW	write to stack inLIFO or FIFO way
STEM	read into a stem/write from a stem variable
first-stem-entry	start adding entries at given stem.number, only available on DISKR with STEM parameter
LIFO	read from / write into a lifo stack
FIFO	read from / write into a fifo stack
skip-lines	skip number of lines before processing dataset/stack
keep-string	process just records containing the string
drop-string	process just records which do not contain the string
SUBSTR	process a substring of the given record

Example:

```
/* Read entire File into Stem-Variable*/
"EXECIO * DISKR dd-name (STEM stem-name."

/* Write Stem-Variable into File */
"EXECIO * DISKW dd-name (STEM stem-name."

/* Append File by Stem-Variable */
"EXECIO * DISKA dd-name (STEM stem-name."

/* ---- Read into REXX FIFO Stack ------- */
"EXECIO * DISKR dd-name (FIFO "
do i=1 to queued()
  parse pull line
  say line
end

/* ---- Read into REXX LIFO Stack ------- */
"EXECIO * DISKR dd-name (LIFO "
```

```
do i=1 to queued()
parse pull line
say line
end
```

After completing the Read stem-name.0 contains the number of records read The number of lines to become written to the file is defined in stem-variable.0

TCP Functions

TCP Functions are only usable in TK4- and MVS/CE, or an equivalent MVS3.8j installation running on SDL Hyperion with activated TCP support. For non TK4- or MVS/CE installation it might be necessary to start the TCP functionality in the Hercules console before the IPL of MVS is performed:

```
facility enable HERC_TCPIP_EXTENSION
facility enable HERC_TCPIP_PROB_STATE
```

for details you look up the following document: https://github.com/SDL-Hercules-390/hyperion/blob/-master/readme/README.TCPIP.md

Warning If TCP support is not enabled, the TCP environment is in an undefined state, and all subsequent TCP functions will end up with indeterminate results or even cause an ABEND.

Warning In case of errors or ABENDs an automatic cleanup of open TCP sockets takes place. If in rare cases the cleanup cannot resolve it a reconnect will be rejected. You can then reset all sockets by the TSO command RESET.

`TCPINIT` ( )

TCPINIT initialises the TCP functionality. It is a mandatory call before using any other TCP function.

`TCPSERVE` (*port-number*)

TCPSERVE opens a TCP Server on the defined port-number for all its assigned IP-addresses.

The function returns zero if it is performed successfully, else an error occurred.

`TCPOPEN` (*host-ip*, *port-number* **[**, *time-out-secs* **]**)

Client function to open a connection to a server. Host-ip can be an ip-address or a host-name, which translates into an ip-address. Port-number is the port in which the server listens for incoming requests. The timeout parameter defines how long the function will wait for a confirmation of the open request; the default is 5 seconds.

If rc= 0 the open was successful if less than zero an error occurred during the open process.

The BREXX variable _FD contains the unique token for the connection. It must be used in various TCP function calls to address the appropriate socket.

`TCPWAIT` (**[**, *time-out-secs* **]**)

TCPWAIT is a Server function; it waits for incoming requests from a client. The optional timeout parameter defines an interval in seconds after the control is returned to the server, to perform for example some cleanup activities, before going again in a wait. TCPWAIT returns several return codes which allow checking which action has ended the wait:

Return	Description
#receive	an incoming message from a client has been received

#connect	a new client requests a connect
#timeout	a time-out occurred
#close	a close request from a client occurred
#stop	a socket returned stop; typically the socket connection has been lost.
#error	an unknown error occurred in the socket processing

Example of a server TCPWAIT and how it is processed:
Example:

```
/* rexx */
do forever
  event = tcpwait(20)
  if event <= 0 then call eventerror event
  select
    when event = #receive then do
      rc=receive()
      if rc=0 then iterate /* proceed */
      if rc=4 then leave /* close client socket */
      if rc=8 then leave /* shut down server */
    end
    when event = #connect then call connect
    when event = #timeout then call timeout
    when event = #close then call close
    when event = #stop then call close /* is /F console cmd */
    when event = #error then call eventError
    otherwise call eventError
  end
end
```

TCPSEND (*clientToken*, *message* **[**, *timeout-secs* **]**)

Sends a message to a client. ClientToken specifies the unique socket of the client. The optional timeout parameter allows the maximum wait time in seconds to wait for confirmation from the client, that it has received it. The default timeout is 5 seconds.
If sendLength is less than zero, an error occurred during the sending process:

- >0 message has been sent and received by the client, number of bytes transferred
- -1 socket error
- -2 client is not ready to receive a message

Example: *SendLength=TCPSEND(clientToken, message[,time-out-secs])*

TCPReceive (*clientToken* **[**, *time-out-secs* **]**)

The message length is returned by the TCPRECEIVE Function, The message itself is provided in the variable *_Data*.
If messageLength is less than zero, an error occurred during the receiving process:

- >0 message has been received from, number of bytes received
- -1 client is not ready to receive a message
- -2 socket error

Example: *MessageLength=TCPReceive(clientToken,[time-out-secs])*

TCPTERM ( )

Closes all client sockets and removes the TCP functionality

TCPSF (*port* **[**, *timeout* **]**)

TCPSF is a generic TCP Server Facility. It opens a TCP server and controls all events. Call-back

labels in the calling rexx support the event handling. Therefore the calling REXX-script must contain the following labels:

Label	Description
TCPCONNECT	There was a client connect request. The connect will be performed by the TCPSF. If you want, you can do some logging of the incoming requests. ARG(1)) client token Return codes from user procedure control the continuation: return: - 0 proceed - 4 immediately close client - 8 shut down server
TCPTIMEOUT	There was a time-out, no user requests occurred. Typically it is used to allow some maintenance. Doing nothing (plain return 0) is also possible. If the user procedure wants to set a new time-out value, it must be set in the rexx variable NEWTIMEOUT. It is set in seconds. There are no arguments passed. return: - 0 proceed - 8 shut down server
TCPDATA	client has sent a message ARG(1) client token ARG(2) contains the original message ARG(3) contains the message translated from ASCII to EBCDIC Return codes from user procedure control the continuation: - 0 proceed - 4 immediately close client
TCPCLOSE	client has closed the connection. TCPCLOSE can be used for housekeeping. ARG(1) client token Return codes from user procedure control the continuation: -0 proceed -8 shut down server
TCPSTOP	client will be stopped. ARG(1) client token There is no special return code treatment

The following commands sent from a client are processed from the TCP Server:

- */CANCEL* shut down the TCP server
- */QUIT* log off the client from the TCP Server

An example of a TCP Server is defined in *BREXX.V2R5M1.SAMPLE($TCPSERV)*

TSO REXX Functions

TSO REXX functions are only available in TSO environments (online or batch) not in plain batch.

SYSDSN (*dataset-name[(member-name)]*)

Returns a message indicating whether a dataset exists or not.

A fully qualified dataset-name must be enclosed in apostrophes (single quotes) they must be delivered to the MVS function, it is, therefore, necessary to put double quotes around the dataset-name. If the dataset-name does not contain an apostrophe, it is completed by the user-name as the prefix.

Return message	Description
OK	dataset or member is available
DATASET NOT FOUND	dataset or member is not available
INVALID DATASET	NAME dataset name is not valid
MISSING DATASET	NAME no dataset name given

Example:

```
x=SYSDSN("'HERC01.TEST.DATA'")
IF x = 'OK' THEN
  do something
ELSE
  do something other
```

SYSVAR (*request-type*)

TSO-only function to retrieve certain TSO runtime information. Available request-types:

Type	Description
SYSUID	UserID
SYSPREF	system prefix of current TSO session (typically hlq of userid)
SYSENV	FORE/BACK/BATCH forground/background TSO execution, or plain batch
SYSISPF	ISPF active 1, not active 0
SYSTSO	TSO active 1, not active 0
SYSAUTH	script runs in authorised mode (1), 0 not authorised
SYSCP	returns the host-system which runs MVS38j. It is either MVS or VM/370
SYSCPLVL	shows the release of the host-system
SYSHEAP	allocated heap storage
SYSSTACK	allocated stack storage
RXINSTRC	BREXX Instruction Counter

Example:

```
say sysvar('SYSISPF')
say sysvar('SYSUID')
say sysvar('SYSPREF')
say sysvar('SYSENV')
say sysvar('SYSAUTH')
say sysvar('SYSCP')
say sysvar('SYSCPLVL')
say sysvar('RXINSTRC')
```

Result:

```
NOT ACTIVE
IBMUSER
IBMUSER
FORE
1
Hercules
Hercules version 4.4.1.10647-SDL-gd0ccfbc9
16
```

MVSVAR (*request-type*)

Return certain MVS information.

Type	Description
SYSNAME	system name
SYSOPSYS	MVS release
CPUS	number of CPUs
CPU	CPU type
NJE	1 = NJE38 is running, 0 = NJE38 is not running/installed
NJEDSN	Dataset name of the NJE38 spool queue
SYSNETID	Netid of MVS (if any)
SYSNJVER	Version of NJE38
JOBNUMBER	current job number

Example:

```
Say MVSVAR('SYSNAME')
SAY MVSVAR('SYSOPSYS')
SAY MVSVAR('CPU')
SAY MVSVAR('CPUS')
SAY MVSVAR('NJE')
SAY MVSVAR('NJEDSN')
SAY MVSVAR(SYSNETID)
SAY MVSVAR(SYSNJVER)
SAY MVSVAR('MVSUP')
SAY sec2time(MVSVAR('MVSUP'),'DAYS')
```

Results:

```
MVSC
MVS 03.8
148
0002
1
NJE38.NETSPOOL
DRNBRX3A
V2.2.0 01/14/21 07.11
```

```
1339432
15 day(s) 12:03:52
```

LISTDSI (*dataset*)

Returns information of non-VSAM datasets in REXX variables.

Parameters **dataset** – Either *""dataset-name""* or *'dd-name FILE'*

A fully qualified dataset-name must be enclosed in apostrophes (single quotes) they must be delivered to the MVS function, it is, therefore, necessary to put double-quotes around the dataset-name. If the dataset-name does not contain an apostrophe, it is prefixed by the user-name

Variable	Description
SYSDSNAME	Dataset name
SYSVOLUME	Volume location
SYSDSORG	PS for sequential, PO for partitioned datasets
SYSRECFM	record format, F,FB,V,VB, ...
SYSLRECL	record length
SYSBLKSIZE	block size
SYSSIZE	file size, For partitioned it is 0

18 Matrix and Integer Array functions

ICREATE (*elements*, *mode*)

Creates an integer array with the size elements. Returned is the array number to be used to address the array with ISET and IGET. You can have up to 64 integer arrays. Depending on the virtual storage they may contain 1 million elements and more. Accessing integer arrays is very fast as there is no overhead compared to STEM variables.

Parameters

- **elements** – Number entries available
- **mode** – The initialization type. If mode is not set the array remains uninitialized.

Mode	Description
Element-Number	index of element
NULL	elements are set to 0
DESCENT	index of the element in reverse order
SUNDARAM	prime numbers (Sundaram algorithm)
PRIME	prime numbers (sieve of Erasthones)

ISET (*array-number*, *element-number*, *integer-value*)

Sets a certain element of an array with an integer value.

IGET (*array-number*, *element-number*)

Gets (returns) a certain element of an array with an integer value.

MCREATE (*rows*, *columns*)

Creates a (Float) matrix with size *[rows x columns]*. Returned is the Matrix number to be used in various matrix operations. You can have up to 128 matrixes, depending on the virtual storage available. Accessing a matrix is very fast as there is no overhead compared to STEM variables.

MSET (*matrix-number*, *row*, *column*, *float-value*)

Sets a certain element of the matrix with a float value.

MGET (*matrix-number*, *row*, *column*)

Gets (returns) a certain element of the matrix.

MMULTIPLY (*matrix-number-1*, *matrix-number-2*)

Multiplies 2 matrices and creates a new matrix, which is returned. Input matrices remain untouched. The format of matrix-1 is *[rows x columns]*, therefore the format of matrix-2 must be *[columns x rows]*. The format of the result matrix is rows x rows.

MINVERT (*matrix-number*)

Inverts the given matrix and creates a new matrix, which is returned. The input matrix must be squared and remains untouched. The format of the result matrix remains the same as the input matrix.

MTRANSPOSE (*matrix-number*)

Transposes the given matrix and creates a new matrix, which is returned. The input matrix remains untouched. If the format of the input matrix is [rows x columns] then the result matrix is columns x rows.

MCOPY (*matrix-number*)

Copies the given matrix and creates a new matrix, which is returned. The input matrix remains untouched. Formats of both matrices are equal.

MNORMALISE (*matrix-number*, *mode*)

Normalises the given matrix and creates a new matrix, which is returned. The input matrix remains untouched. Formats of both matrices are equal.

Mode	Description
STANDARD	row is normalized to mean=0 variance=1
ROWS	row value is divided by number of rows
MEAN	row value is normalized to mean=0, variance remains unchanged

MDELROW (*matrix-number*, *row-number* **[**, *row-number* **[**, *row-number...* **]** **]**)

Copies the given matrix without the specified rows-to-delete as a new matrix, which is returned. The input matrix remains untouched.

MDELCOL (*matrix-number*, *col-number* **[**, *col-number* **[**, *col-number...* **]** **]**)

Copies the given matrix without the specified columns-to-delete as a new matrix, which is returned. The input matrix remains untouched.

MPROPERTY (*matrix-number* **[**, *"FULL"/"BASIC"* **]**)

Returns the properties of the given matrix in BREXX variables:

_rows	number of rows of matrix
_cols	number of columns of matrix

If **FULL** is specified additionally the the following stem variables are returned:

Stem	Description
_rowmean.column-i	mean of rows of column-i
_rowvariance.column-i	variance of rows of column-i
_rowlow.column-i	lowest row value of column-i
_rowhigh.column-i	highest row value of column-i
_rowsum.column-i	sum of row value of column-i
_rowsqr.column-i	sum of squared row value of column-i
_colsum.row-i	sum of column values of row-i

MSCALAR (*matrix-number*, *number*)

Multiplies each element of a matrix with a number (float). The result is stored in a new matrix, which is returned. The input matrix remains untouched.

MADD (*matrix-number-1*, *matrix-number-2*)

Adds each element of a matrix-1 with the same element of matrix-2. The result is stored in a new matrix, which is returned. The input matrix remains untouched. Matrix-1 and matrix-2 must have the same dimensions.

MSUBTRACT (*matrix-number-1*, *matrix-number-2*)

Subtracts each element of a matrix-2 from the same element of matrix-1. The result is stored in a new matrix, which is returned. The input matrix remains untouched. Matrix-1 and matrix-2 must have the same dimensions.

MPROD (*matrix-number-1*, *matrix-number-2*)

Multiplies each element of a matrix-1 with the same element of matrix-2. The result is stored in a new matrix, which is returned. The input matrix remains untouched. Matrix-1 and matrix-2 must have the same dimensions.

MSQR (*matrix-number*)

Squares each element of the matrix. The result is stored in a new matrix, which is returned. The input matrix remains untouched.

MINSCOL (*matrix-number*)

Inserts a new column as the first column. The initial first column becomes the second column, etc. The result is stored in a new matrix, which is returned. The input matrix remains untouched.

MFREE (**[**, *matrix-number/integer-array-number"MATRIX"/"INTEGER-ARRAY"* **]**)

Frees the storage of allocated matrices and/or integer arrays. If no parameter is specified all allocations are freed. To release a specific matrix or integer-array the matrix-number or integer-array-number must be used as the first parameter, followed by the type to release.

RXLIB functions

BREXX can implement new functions or commands in REXX. They are transparent and are called in the same way as basic BREXX functions. They are stored in the library BREXX.RXLIB and are automatically allocated (via *DD RXLIB*) in RXBATCH and RXTSO (Batch). In this release, BREXX delivers the following functions.

RXMSG (*msg-number*, *'msg-level'*, *'message'*)

Standard message module to display a message in a formatted way

Parameters

- **msg-number** – message number to be displayed
- **msg-level** – One of: I, W, E, C

message level can be:

Message Level	Description
I	for an information message
W	for a warning message
E	for an error message
C	for a critical message

Example:

```
rc=rxmsg( 10,'I','Program started')
rc=rxmsg(200,'W','Value missing')
rc=rxmsg(100,'E','Value not Numeric')
rc=rxmsg(999,'C','Divisor is zero')
```

Results:

```
RX0010I    PROGRAM STARTED
RX0200W    VALUE MISSING
RX0100E    VALUE NOT NUMERIC
RX0999C    DIVISOR IS ZERO
```

Additionally, the following REXX variables are maintained and can be used in the calling REXX script.
Return code from call **RXMSG**:

Return Code	Description
0	an information message was written
4	a warning message was written
8	an error message was written
12	a critical message was written

MSLV contains the written message level

Message Level	Description
I	for an information message
W	for a warning message
E	for an error message
C	for a critical message

MSTX contains the written message text part
MSLN includes the complete message with the message number, message level and text
MAXRC contains the highest return code so far; this can be used to exit the top level REXX. If you used nested procedures, it is required to expose MAXRC, to make it available in the calling procedures.

DCL (*'field-name'*, *length* **[**, *offset* **]**)

Defines a structure of fields which maps typically to an I/O record. The function returns the next available offset in the structure.
Initialize the function with *DCL('$DEFINE','structure-name')* where:

- *$DEFINE* initialises the structure definition
- *structure-name* all following field definitions are associated with the structure-name.

Parameters

- **field-name** – name of the rexx variable containing/receiving the field content of the record
- **offset** – offset of the field in the record. This definition is optional if left out the next offset from the previous *DCL(field…)* definition is used, or 1 if there was none.
- **length** – length if the field in the record
- **type** – field-type either **CHAR** no translation takes place, CHAR is default or **PACKED** decimal Packed field. Translation into/from Decimal packed into Numeric REXX value takes place

call SPLITRECORD 'structure_name,record-to-split splits record-to-split in the defined field-names (aka REXX variables). The variable containing the record to split is typically read from a dataset.
Record=SETRECORD('student') combines the content of all defined fields (aka REXX variables) at the defined position and the defined length to a new record.
Example:

```
n=DCL('$DEFINE','student')
n=DCL('Name',1,32,'CHAR')
n=DCL('FirstName',1,16,'CHAR')
n=DCL('LastName',,16,'CHAR')
n=DCL('Address',,32,'CHAR')
recin='Fred            Flintstone      Bedrock'
/*    '12345678901234567890123456789012345678901234567890 */
call splitRecord 'student',recin
say Name
say FirstName
say LastName
say Address
firstName='Barney'
LastName='Rubble'
address='Bedrock'
say setRecord('student')
```

Results:

```
FRED            FLINTSTONE
```

```
FRED
FLINTSTONE
BEDROCK
BARNEY          RUBBLE          BEDROCK
```

DAYSBETW (*date1*, *date-2* **[**, **[**, *format-date1* **]** **[**, *format-date2* **]** **]**)

Return days between 2 dates of a given format.

Parameters

- **format-date1** – date format of date1 defaults to European
- **format-date2** – date format of date2 defaults to European

the format-dates reflect the Input-Format of DATE and can be found in details there.

DUMP (*string* **[**, *hdr* **]**)

Displays string as a Hex value, useful to check if a received a string contains unprintable characters. One can specify hdr as an optional title.
Example:

```
CALL DUMP 'THIS IS THE NEW VERSION OF BREXX/370 V2R1M0','DUMP LINE'
```

Results:

```
DUMP LINE
0000(0000)  THIS  IS  THE  NEW    VERS ION  OF B REXX
0000(0000)  ECCE 4CE4 ECC4 DCE4   ECDE CDD4 DC4C DCEE
0000(0000)  3892 0920 3850 5560   5592 9650 6602 9577

0032(0020)  /370  V2R 1M0
0032(0020)  6FFF 4EFD FDF
0032(0020)  1370 0529 140
```

LISTALC ( )

Lists all allocated Datasets in this session or region.
Example:

```
CALL LISTALC
```

Results:

```
STDOUT    *terminal
STDIN     *terminal
SYSPROC   SYS1.CMDPROC
SYSHELP   SYS1.HELP
          SYS2.HELP
SYS00002  UCPUB001
RXLIB     BREXX.V2R5M0.RXLIB
SYSEXEC   SYS2.EXEC
SYS00005  UCPUB000
ISPPROF   IBMUSER.ISP.PROF
ISPMLIB   SYSGEN.ISPF.MLIB
STDERR    *terminal
ISPSLIB   SYSGEN.ISPF.SLIB
ISPCLIB   SYSGEN.ISPF.CLIB
          SYSGEN.REVIEW.CLIST
ISPLLIB   SYSGEN.ISPF.LLIB
          SYSGEN.REVIEW.LOAD
ISPTABL   SYSGEN.ISPF.TLIB
ISPPLIB   SYSGEN.ISPF.PLIB
          SYSGEN.ISPF.RFEPLIB
```

```
ISPTLIB   SYSGEN.ISPF.TLIB
REVPROF   IBMUSER.ISP.PROF
SYS00012  SYSGEN.ISPF.LLIB
SYS00013  IBMUSER.CLIST
```

LISTCAT (**[**, *list-cat-parameter* **]**)

Returns listcat output in the stem LISTCAT.

MVSCBS ( )

Allows addressing of some MVS control blocks. There are several dependent control blocks combined. To use them, MVSCBS must be imported first. After that, they can be used.
Currently integrated control blocks are: - CVT() - TCB() - ASCB() - TIOT() - JSCB() - RMCT() - ASXB() - ACEE() - ECT() - SMCA()
The definition and the content of the MVS control blocks can be found in the appropriate IBM manuals: MVS Data Areas, Volume 1 to 5.
IMPORT command is described in Vassilis N. Vlachoudis BREXX documentation: http://home-.cern.ch/~bnv

QUOTE (*string*, *qtype*)

Enclose string in quotes, double quotes, or parenthesis,

Parameters `qtype` – can be:

- ‘ single quote (default),
- “ double quote
- *(* bracket, the closing character is ‘)’
- *[* square bracket, the closing character is ‘]’

Example:

```
 Mystring='string to be quoted'
 Say QUOTE(mystring,'"')
 Say QUOTE(mystring,"'")
 Say QUOTE(mystring,'(')
 Say QUOTE(mystring,'[')
```

Results:

```
'STRING TO BE QUOTED'
'STRING TO BE QUOTED'
(STRING TO BE QUOTED)
[STRING TO BE QUOTED]
```

PDSRESET (*pds-name*)

Removes all members of a PDS and runs a compress. After execution, the PDS is empty.

READALL (*file*, *variable* **[**, *'DSN'/'DDN'* **]**)

Reads the entire file into a stem variable. The file can be either a dd-name or a ds-name. After successful completion, the stem *variable.0* contains the number of lines read into the stem. The file name can either represent an allocated dd name or a fully qualified DSN. The third parameter defines the file type and is either DSN or DDN. If it is missing DDN is the default.

PERFORM (*pds-name*, *process-member-rexx*)

Reads member list of a PDS and runs the process-member-rexx against each member. The REXX to be called receives the parameters:

- Pds-name
- Member-name

RXSORT (*sort-type* **[**, *ASCENDING/DESCENDING* **]**)

Sorts the stem variable SORTIN. SORTIN.0 must contain the number of entries of SORTIN. The sort algorithms supported are: QUICKSORT, SHELLSORT, HEAPSORT, BUBBLESORT. After Completion of RXSORT the stem variable SORTIN. is sorted. If you requested ASCENDING (also default) it is in ascending order, for DESCENDING in descending order.

Sorting with REXX is only recommended for a small number of stem entries. Up to 1000 entries, RXSORT works in a reasonable time.

If the stem you want to sort is not in SORTIN, you can use the SORTCOPY function to copy it over to SORTIN.

SEC2TIME (*seconds* **[**, *'DAYS'* **]**)

Converts a number of seconds into the format hh:mm:ss, or days hh:mm:ss if the 'DAYS' parameter is specified.

Example:

```
say sec2Time(345000)
say sec2Time(345000,'DAYS')
```

Results:

```
95:50:00
3 day(s) 23:50:00
```

SORTCOPY (*stem-variable*)

Copies any stem variable into the stem *SORTIN.*, which then can be used by RXSORT. *Stem-variable.0* must contain the number of entries of the stem.

STEMCOPY (*source-stem-variable*, *target-stem-variable*)

Copies any stem variable into another stem variable. *source-stem-variable.0* must contain the number of entries of the stem. Stem-variables must end with a trailing '.', e.g. *mystem.*

STEMCLEN (*stem-variable*)

Cleansing of a stem variable, it removes empty and unset stem items and adjusts the stem numbering. *Stem-variable.0* must contain the number of entries of the stem and will after the cleansing the modified number of entries. Stem-variables must end with a trailing '.', e.g. *mystem.*

STEMGET (*dataset-name*)

Reads the saved content of one or more stem variables and re-apply the stem. Stem names are save in the dataset.

STEMINS (*stem-to-insert*, *insert-into-stem*, *position*)

Inserts stem-to-insert into insert-into-stem beginning at position. The content of the original stem at the position is shifted down n positions, whereby n is the size of the stem to be inserted. *Stem-variable(s).0* must contain the number of entries of the stem. Stem-variables must end with a trailing '.', e.g. *mystem.*

STEMPUT (*dataset-name,stem1[,stem2{,stem3]...*)

Saves the content of one or more stems in a fully qualified dataset-name Stem-variable.0 must contain the number of entries of the stem. Stem-variables must end with a trailing '.', e.g. *mystem.*

STEMREOR (*stem-variable*)

reorders stem variable from top to bottom.

1. element becomes last,
2. next to last, etc.

Stem-variable.0 must contain the number of entries of the stem. Stem-variables must end with

a trailing '.', e.g. *mystem.*

TODAY (*[output_date_format[,date[,input_date_format]]) [date-format]*)

Returns today's date based on the requested format. You can also use a date which is in the past or the future. Details of date-formats can be found in the DATE output-format description.

UNQUOTE (*string*)

Remove from string leading and trailing quotes, double quotes, parenthesis and '<' and '>' signs. Example:

```
 Say UNQUOTE(“ 'quoted-string' “)
 Say UNQUOTE(“<entry 1>“)
 Say UNQUOTE(“(entry 2)“)
 Say UNQUOTE(“[entry 3]“)
```

Results:

```
'QUOTED-STRING'
ENTRY 1
ENTRY 2
ENTRY 3
```

WRITEALL (*file*, *variable* **[**, *'DSN'/'DDN'* **]**)

Writes a stem variable into a file. The file can be either a dd-name or a ds-name. The stem variable.0 must contain the number of entries of the stem. The file name can either represent an allocated dd name or a fully qualified DSN. The third parameter defines the file type and is either DSN or DDN. If it is missing DDN is the default.

Building TSO Commands

A BREXX function can be converted to work as a TSO command by creating a clist and call the BREXX script. To perform the new clist, it must be stored in one of the pre-allocated clists libraries which are active in your TSO session; alternatively, you can use *SYS2.CMDPROC*. Once this is done, you can call it from TSO directly.

20.1 LA List all allocated Libraries

The clist calls the BREXX LISTALC script with a BREXX CALL statement. A minus sign immediately following the REXX command tells BREXX to interpret a BREXX statement. The statement(s) must be coded in one line. To place more than one BREXX statement in a line, separate them by using a semi-colon ';'.

```
REXX -
CALL LISTALC('PRINT')
```

20.2 WHOAMI Display current User Id

This one-liner outputs the *userid()* function by a say statement.

```
REXX -
SAY USERID()
```

20.3 TODAY

Display today's Date

```
REXX -
SAY DATE(); SAY TIME()
```

20.4 USERS

List active Users. The clist calls the BREXX WHO script directly, therefore no minus sign is necessary:

```
REXX WHO
```

20.5 REPL

Interactive REXX Processor.

The clist calls the BREXX REPL which opens the interactive REXX processor. It allows you to enter and execute rexx statements.

```
RX REPL NOSTAE
```

Chapter 21

Callable External Functions

With the new External Function feature, you can call compiled programs written in conventional language, as PL1, Assembler, and maybe more.

We closely adapted IBM's TSO/E REXX programming services: https://www.ibm.com/support/knowledgecenter/SSLTBW_2.2.0/com.ibm.zos.v2r2.ikja300/progsrv.htm

How it works:

21.1 BREXX Call an external Program

To call an external program, you call it in the same way as a normal BREXX function:

```
say load-module(argument-1,argument-2,...,argument-15)
```

you can pass up to 15 arguments to the external function. The size of the return value can be up to 1024 bytes.

Example:

```
Say RXPI()
```

RXPI is a load module that must be accessible within the link list chain. It does not have any arguments.

21.2 BREXX Programming Services

BREXX provides control blocks containing the arguments and a 1024 bytes return buffer.

21.3 Called Program

The program needs to match the BREXX calling conventions to manage the argument and return value handling. To ease it, we have isolated communication control blocks and internal functions in a copybook. Once included, it will transparently provide the functionality to the program.

Example, PI calculation:

```
RXPI:  PROCEDURE(EFPL_PTR) OPTIONS(MAIN);
  %INCLUDE RXCOMM;
...
```

21.4 Benefits

The performance of a compiled program is much higher than in BREXX. So if you have complex mathematical calculations, they will be significantly faster than code implemented in BREXX. In our testing, we implemented an algorithm for calculating PI with 500 digits. In comparison, it was over 600 times faster than the same algorithm implemented in BREXX.

VSAM User's Guide

The VSAM User's Guide contains the BREXX functions to access VSAM KSDS files.

The VSAM Interface is based on Steve Scott's VSAM API: https://sourceforge.net/projects/rxvsam/ .

We gratefully thank Steve for allowing us the integration in BREXX and his support to achieve it.

The underlying VSAM API allows full support for KSDS, RRDS and ESDS, but we focused just on the KSDS functionality, so there is no support for RRDS and ESDS. If this limitation is lifted in the future depends on user requests.

22.1 Integration of the VSAM Interface in BREXX

We decided to integrate the interface as host commands rather than BREXX functions. It is now similar to the EXECIO host command for sequential datasets. The host command name is VSAMIO. Host commands are typically enclosed in quotes or double-quotes.

Example:

"VSAMIO OPEN VSIN (UPDATE"

22.1.1 Limitations/Restrictions

The implementation has only tested with UNIQUE cluster definitions, not with type SUBALLOCATION (which requires in MVS 3.8 a DEFINE SPACE and DEFINE VSAM catalogue definition). The UNIQUE specification does not allow the REUSE CLUSTER definition, which would be necessary for the initial loading of an empty KSDS dataset.

22.1.2 Initialising empty VSAM Files

A program cannot directly process an empty VSAM file it must be initialised first. The procedure to achieve this is to use IDCAMS REPRO to write a "null"-record into it. After this exercise, the VSAM file can be updated by BREXX/370 with normal VSAM Write commands from BREXX.

22.1.3 Key of Records

The key must be part of the record, and nevertheless, you must additionally specify the key in the following commands:

- *"VSAMIO READ ddname (KEY key ... "*
- *"VSAMIO LOCATE ddname (KEY key ... "*
- *"VSAMIO WRITE ddname (KEY key ... "*
- *"VSAMIO DELETE ddname (KEY key ... "*

The key of a record must consist of a sequence of contiguous non-space characters; this means

blanks are not allowed being part of a key. This limitation might be lifted in one of the forthcoming releases.

You can easily convert spaces in a key with the TRANSLATE function: *key=translate(key,'_',' ')*

22.1.4 Return Codes

Each VSAMIO command call returns two return codes:

- **RC** the usual return code, containing:

Return Code	Description
0	call was successful
4	call was not successful and ended with warnings, typically in record-not-found Situations
8	call ended with errors

- **RCX** The extended VSAM Return code, and it consists of a 9 character field with the following format: *rrr-vvvvv* **rrr** is the function return code, **vvvvv** is the VSAM return code

You can look up the details of the extended VSAM return code in IBM's MVS System Messages under message IDC3351I.

22.1.5 System Abend A03

The RXVSAM API runs as independent subtask within the address space. By the end of the REXX Script, an automatic shutdown of the subtask is performed. If the REXX script unexpectedly terminates, you possibly see a SYSTEM ABEND A03, which means the main task (BREXX) has been terminated and there is still a subtask in the background active. MVS forces the ABEND of the subtask with A03. There are no further actions required; there is no impact on the system or the VSAM datasets.

22.1.6 Random and Sequential Access

The used VSAM IO module distinguishes two access methods:

- Random Access always requires a key to read/write/delete a record
- Sequential Access allows to position to a particular record and reads/write/delete records from there sequentially

Both methods can be used concurrently, but it is essential to understand that they do not mutual interfere. Having read a record with random access does not allow to read from this record sequentially the next records, as this is sequential access. But you can perform a LOCATE command with a key and continue the read from there sequentially.

22.1.7 VSAM Dataset reference

Each VSAMIO command uses the DDNAME as a reference to the VSAM dataset. It must be pre-allocated via a JCL DD Statement or a TSO ALLOCATE command.

There are no plans to allow a dataset name (DSN) instead of the DDNAME!

22.1.8 REXX VSAM Debugging

By using BREXXDBG as the BREXX interpreter you can produce additional log entries in the operator's console, as well as in the spool output of a batch job:

Example:

```
//BRXVSMKY JOB CLASS=A,MSGCLASS=H,REGION=8192K,
//         NOTIFY=&SYSUID
//*
```

```
//* ---------------------------------------------------------------
//* READ STUDENT VSAM FILE VIA KEY
//* ---------------------------------------------------------------
//*
//BATCH EXEC RXTSO,BREXX='BREXXDBG',
//         EXEC='$STUDENK',
//         SLIB='BREXX.V2R5M1.SAMPLES'
//SYSPRINT DD SYSOUT=*,
//         DCB=(RECFM=FBA,LRECL=133,BLKSIZE=133)
//SYSUDUMP DD SYSOUT=*
//
```

Results:

```
07.35.01 JOB 1466 $HASP373 PEJRXKEY STARTED - INIT 1 - CLASS A - SYS
TK4-
07.35.01 JOB 1466 IEF403I PEJRXKEY - STARTED - TIME=07.35.01
07.35.02 JOB 1466 +VSAMIO - STUDENTM ACCESS TRACE, REQUEST = OPEN
07.35.02 JOB 1466 +VSAMIO - KEY=NONE
07.35.02 JOB 1466 +VSAMIO - STUDENTM ACCESS TRACE, REQUEST = READU
07.35.02 JOB 1466 +VSAMIO - KEY=X"C1D5C4C5D9E2D6D55EC2C5D56D6D6D6D6D6D6D6D6D6
07.35.02 JOB 1466 +VSAMIO - STUDENTM ACCESS TRACE, REQUEST = READU
07.35.02 JOB 1466 +VSAMIO - KEY=X"C1D5C4C5D9E2D6D55EC7C1C2D9C9C5D36D6D6D6D6D6
07.35.02 JOB 1466 +VSAMIO - STUDENTM ACCESS TRACE, REQUEST = READU
07.35.02 JOB 1466 +VSAMIO - KEY=X"C2C1D3C4E6C9D55EC1D9D3C5D5C56D6D6D6D6D6D6D6
07.35.02 JOB 1466 +VSAMIO - STUDENTM ACCESS TRACE, REQUEST = READU
07.35.02 JOB 1466 +VSAMIO - KEY=X"E2E3C5D7C8C5D5E2D6D55ED7C1E3D9C9C3C9C16D6D6D6
07.35.02 JOB 1466 +VSAMIO - STUDENTM ACCESS TRACE, REQUEST = CLOSE
07.35.02 JOB 1466 +VSAMIO - KEY=NONE
07.35.02 JOB 1466 IEFACTRT - Stepname Procstep Program  Retcode
07.35.02 JOB 1466 PEJRXKEY   BATCH    EXEC     IKJEFT01 RC= 0000
07.35.02 JOB 1466 IEF404I PEJRXKEY - ENDED - TIME=07.35.02
07.35.02 JOB 1466 $HASP395 PEJRXKEY ENDED
```

22.2 VSAM Commands in BREXX

22.2.1 OPEN VSAM Dataset

"VSAMIO OPEN ddname ([READ/UPDATE] "

Example:

```
"VSAMIO OPEN VSIN1 (READ"
"VSAMIO OPEN VSIN2 (UPDATE"
```

VSIN1 is opened in reading mode, VSIN2 in UPDATE mode.

22.2.2 READ with KEY

Access-Type: Random:

```
"VSAMIO READ ddname (KEY key-to-read VAR rexx-variable"
```

If you want to update the record, you must prepare for it by adding the UPDATE keyword:

```
"VSAMIO READ ddname (KEY key-to-read UPDATE VAR rexx-variable"
```

The UPDATE keyword requires a File OPEN with UPDATE

Example:

```
"VSAMIO READ VSIN1 (KEY "key1" VAR record1"
"VSAMIO READ VSIN2 (KEY "key2" UPDATE VAR record2"
```

Read a record with key1/key2 (contained in a rexx variable) into the rexx variable record1/record2

22.2.3 READ NEXT

Access-Type: Sequential

After positioning with LOCATE to a particular record, you can read the next records sequentially. If no LOCATE has been previously performed, the first record is read.:

```
"VSAMIO READ ddname (NEXT VAR rexx-variable"
```

If you want to update the record, you must prepare for it by adding the UPDATE keyword:

```
"VSAMIO READ ddname (NEXT UPDATE VAR rexx-variable"
```

The UPDATE keyword requires a File OPEN with UPDATE

Example:

```
"VSAMIO LOCATE VSIN (KEY "key
Do until rc>0
    "VSAMIO READ VSIN (NEXT VAR record"
    Say record
End
```

Position to record key (contained in a rexx variable) and read all records from there into rexx variable record

22.2.4 LOCATE position to a certain record

Access-Type: Sequential

Position the record pointer in front of the provided key or a key prefix:

```
"VSAMIO LOCATE ddname (KEY [key-to-position/key-prefix]"
```

To subsequentially read the next records a READ NEXT is required. After a successful read, the position is shifted to the next record position.

Example refer to READ NEXT.

22.2.5 WRITE KEY

Access-Type: Random

To update a record, it must be priorly read with a READ KEY, regardless whether the record exists. If the record doesn't exist, it is inserted:

```
"VSAMIO WRITE ddname (KEY key-to-write VAR rexx-variable"
```

Example:

```
"VSAMIO READ VSIN (KEY "key" UPDATE VAR CURRENT"
say 'READ 'rc' Extended RC 'rcx
"VSAMIO WRITE VSIN (KEY "key" VAR RECORD"
if rc<>0 then say key' Error during Insert'
else say inkey' Record inserted'
say 'WRITE 'rc' Extended RC 'rcx
```

To insert a new record; the READ is mandatory to verify if a record is already defined.

22.2.6 WRITE NEXT

Access-Type: Sequential To update a record, it must be priorly read with a READ NEXT.:

```
"VSAMIO WRITE ddname (NEXT VAR rexx-variable"
```

22.2.7 DELETE KEY

Access-Type: Random

To delete an existing record.:

```
"VSAMIO DELETE ddname (KEY key-to-delete "
```

Example:

```
"VSAMIO OPEN VSERR (UPDATE"
say 'OPEN 'rc' Extended RC 'rcx
"VSAMIO DELETE VSERR (KEY 0000000"
say 'Delete Dummy Record 'rc' Extended RC 'rcx
```

22.2.8 DELETE NEXT

Access-Type: Sequential

To delete an existing record, it must be priorly read with a READ NEXT.:

```
"VSAMIO DELETE ddname (NEXT "
```

Example:

```
"VSAMIO LOCATE VSIN (KEY "prefix
say "LOCATE "rc
say "Extended RC "rcx
do forever
    "VSAMIO READ VSIN (NEXT UPDATE VAR INREC"
    if rc<>0 then leave
    say "record='"INREC"' RC "rc" Extended RC "rcx
    key=substr(inrec,1,8)
    "VSAMIO DELETE VSIN (NEXT "
    if rc=0 then reci=reci+1
    say 'DELETE RC 'rc' Extended RC 'rcx
end
```

22.2.9 CLOSE

"VSAMIO CLOSE ddname "

To close all open VSAM datasets you can also use *"VSAMIO CLOSE ALL "*

Example:

```
"VSAMIO CLOSE VSERR"
```

22.3 BREXX VSAM Example

The installation file contains in the dataset *BREXX.V2R5M1.JCL* a working example of a student database using fictitious student entries, containing first name, family name, birth date, the field of study, address.

You can submit the REXX scripts in batch out of *BREXX.V2R5M1.JCL*

Member	Description
STUDENTC	Creates the VSAM Cluster definition
STUDENTI	Inserts the student records into the VSAM dataset
STUDENTK	Read the VSAM dataset with KEYs
STUDENTN	Read the VSAM dataset sequentially

The REXX scripts are stored in *BREXX.V2R5M1.SAMPLES*

Member	Description
@STUDENI	insert student records
@STUDENK	read student records by key
@STUDENL	Query student records by using formatted screens
@STUDENN	read student records sequentially

The following example illustrates the definition and population of a VSAM dataset using BREXX.

1. Define a VSAM Cluster: Define a new VSAM Cluster and import a “Null”-Record, submit the job *STUDENTC*
2. Sample BREXX Program to update the VSAM Dataset: run the script *@STUDENI*
3. JCL Upate VSAM Dataset: The BREXX Program is updating the new VSAM Dataset. Submit the job *STUDENTI*
4. Using a Formatted Screen Application to Query the Student File: *TSO RX “BREXX.V2R5M1.SAMPLE(@SUTDENTL)”*

Formatted screens

The following document is a brief description of the new Formatted Screen (FSS) feature. It allows to set up simple screen definitions within a BREXX script.

For detail take a closer look at the FSS samples in the delivered Installation library *BREXX.V2R5M1.SAMPLES*

23.1 Delivered Samples

The relevant FSS samples are prefixed with the #-sign:

Member	Description
#TSOAPPL	Shows in a detailed usage of all FSS functions how to set up a menu and “paint” a TK4 like design
#BROWSE	A pre-packed FSS application to display data in a List Buffer instead of using SAYs
#FSS1COL	A pre-packed FSS application to generate input requests (in one column)
#FSS2COL	A pre-packed FSS application to generate input requests (distributes in two columns)
#FSS3COL	A pre-packed FSS application to generate input requests (distributes in three columns)
#FSS4COL	A pre-packed FSS application to generate input requests (distributes in four columns)
#FSS4CLX	A pre-packed FSS application to generate input requests (distributes in four columns)

23.2 FSS Limitation

The FSS screen limitation has been dropped. Now large screen widths and heights are supported.

FSS supports just one FSS Screen definition at a time. If you need to display more than one FSS Screen in your REXX application, you must close the first and set up and display the next FSS definition. Using this method, you can easily switch between different FSS Screens. It is a good idea to separate the FSS definitions in different sub-procedures; this allows their display by calling it.

23.3 FSS Function Overview

To use FSS functions in BREXX, you must import the FSS API library from BREXX.RXLIB, address and initialise it by a call to FSSINIT, be aware that FSS is a host command application that requires an ADRESS FSS command, it is sufficient to use it once at the beginning. From this time on all host, commands are directed to FSS. If it happens to be and you have to switch to another host API (e.g. ADDRESS TSO or ADDRESS SYSTEM), you can do so, but you must make sure to switch back to the FSS API by re-issuing an ADDRESS FSS command:

```
/* IMPORT THE API LIBRARY */
CALL IMPORT FSSAPI
/* ADDRESS THE FSS SUBSYSTEM */
ADDRESS FSS
/* SWITCH TO FULL-SCREEN MODE */
CALL FSSINIT
```

23.3.1 FSSINIT Inits the FSS subsystem

Initialise the FSS environment; this must be performed before any other FSS call: *CALL FSSINIT*

23.3.2 Principles of Defining Formatted Screens

You can define your formatted screen by using a series of FSSTEXT and FSSFIELD and/or some wrapped FSS functions as FSSMESSAGE, FSSCOMMAND, etc. in your REXX script. Essential parameters are, in all cases, the ROW and COLUMN positions. Be aware that consistency validations are very basic and not bulletproof at all. It is, for example, possible to accidentally re-use occupied ranges, which may lead to unwanted behaviour or results. Performing just necessary validations increases the performance of the screen handling. It is, therefore, essential that you carefully design your Formatted Screens.

23.3.3 FSSTEXT

`CALL FSSTEXT 'text',row,column,[text-length],attributes`

Display a text field

Parameters

- **`text`** – text to be displayed in the screen
- **`row`** – row where text should be placed
- **`column`** – column where text should be placed.
- **`text-length`** – length occupied by the text, this is an optional parameter; it defaults to the text length.
- **`attributes`** – screen attributes, like colours, protected, high-lighted etc. For details refer to the attributes section

23.3.4 FSSFIELD

`CALL FSSFIELD 'field',row,column,[length],attributes[,init-value]`

Display an input field and associate it with a BREXX Variable

Parameters

- **`field`** – field-name of an input area to be displayed on the screen
- **`row`** – row where text should be placed
- **`column`** – column where text should be placed.
- **`length`** – length occupied by the text, this is an optional parameter; it defaults to the text length.
- **`attributes`** – screen attributes, like colours, protected, high-lighted etc. For details refer to the attributes section
- **`init-value`** – what should be displayed as content of the input field. It defaults to blank.

Note Important Notice on the Column Position

Each text or field definition starts with the defined attribute byte, which itself is invisible but tells how the text or field appears on the screen. Therefore the original text or field-definition start at

column+1.

Note Important Notice on Screen Definitions

Be aware that all definitions provided by FSSTEXT and FSSFIELD are stacked internally. They do not create a formatted screen on the fly.

23.3.5 Attribute Definition

The attribute definitions trigger the behaviour or colours of the Formatted Screen text or input elements.

Attribute	Description
#PROT	Definition is protected (default for fsstext)
#NUM	input field must be numeric
#HI	text is displayed high-lighted
#NON	text/field-input is invisible
#BLINK	text/field blinks
#REVERSE	background is set with defined colour text appears white
#USCORE	Underscore field

Colors:

Attribute	Description
#BLUE	text or input field is of blue colour
#RED	text or input field is of red colour
#PINK	text or input field is of pink colour
#GREEN	text or input field is of green colour
#TURQ	text or input field is of turquoise colour
#YELLOW	text or input field is of yellow colour
#WHITE	text or input field is of white colour

You can combine several attribute bytes by adding them. e.g. *#PROT+#BLUE* combining several colours is not allowed and may lead to unexpected errors.

23.3.6 FSSTITLE

Displays a centred Title in Screen line 1

`CALL FSSTITLE title-text[,attributes]`

Besides the title definition the right hand 25 bytes may contain a short message in case of errors, it overwrites the title part in error situations and automatically resets it, if the enter key is used. The error field is named ZERRSM and maybe set also by your program.

23.3.7 FSSOPTION

`CALL FSSOPTION [row[,option-length[,attribute1,[attribute2]]]`

Creates an OPTIONs line, typically used in a menu to select a menu option:

```
OPTION ===> _____________________________________________
```

Parameters

- **`row`** – defaults to 2
- **`option-length`** – defines the line length to proved the option input, default is length of the remaining line
- **`attribute1`** – Attribute of “OPTION”, default is #PROT+#WHITE

- **attribute2** – Attribute of the option line,default is #HI+#RED+#USCORE

23.3.8 FSSCOMMAND

```
CALL FSSCOMMAND [row[,option-length[,attribute1,[attribute2]]]
```

Creates an input line for entering menu options or commands, it appears with the "COMMAND ===>" prefix and is typically located in row 2.:

```
COMMAND ===> _____________________________________________
```

Parameters

- **row** – defaults to 2
- **option-length** – defines the line length to provide the command input, default is length of the remaining line
- **attribute1** – Attribute of "COMMAND", default is #PROT+#WHITE
- **attribute2** – Attribute of the command line,default is #HI+#RED+#USCORE

23.3.9 FSSTOPLINE

```
CALL FSSTOPLINE prefix,[row[,option-length[,attribute1,[attribute2]]]
```

Create an Option/Command Line. FSSTOPLINE is a variation of FSSCOMMAND which allows the free definition of the input line prefix. It is typically located in row 2.:

```
MY-OPTION ===> _____________________________________________
```

Prefix String which should appear in front of the input line. In the example above it is "MY-OPTION"

Row defaults to 2

Option-length defines the line length to provide the command input; default is the length of the remaining line

Attribute1 Attribute of "COMMAND", default is #PROT+#WHITE

Attribute2 Attribute of the command line, default is #HI+#RED+#USCORE

23.3.10 FSSMESSAGE

CALL FSSMESSAGE [row[,attribute]]

Creates a message line to display messages. The message line occupies a full-screen line.

param row defaults to 3

param attribute attribute of message line, default is #PROT+#HI+#RED

A call to FSSZERRLM sets the Message

23.3.11 FSSZERRSM

Set Error/Warning/Info Short Message. The message is set in Field ZERRSM. ZERRSM is automatically created by using an FSSTITLE definition; otherwise, it must be defined explicitly. If implicitly used with the FSSTITLE definitions, it starts on the right-hand side after the end of the message; its length is dependant on the length of the title.

```
CALL FSSZERRSM 'message'
```

23.3.12 FSSZERRLM

Set Error/Warning/Info Long Message. The message is set in Field ZERRLM, which has been defined

on the screen by a CALL FSSMESSAGE.

```
CALL FSSZERRLM 'message'
```

23.3.13 FSSFSET

Set Field Content

```
CALL FSSFSET 'field',content
```

Make sure the field-name is enclosed in quotes; otherwise, there is a chance of unwanted substitution by its value!

23.3.14 FSSFGET

Get current Field Content .. function:: Value=FSSFGET('field')

Make sure the field-name is enclosed in quotes; otherwise, there is a chance of unwanted substitution by its value!

23.3.15 FSSFGETALL

Get Contents of all Fields

```
Number=FSSFGETALL()
```

All field contents of the screen are fetched and stored in the associated BREXX fields defined by FSSFIELD(...)

23.3.16 FSSCURSOR

Set Cursor to a Field .. function:: CALL FSSCURSOR 'field'

23.3.17 FSSCOLOUR

Change Colour of a Field

```
CALL FSSCOLOUR 'field',colour-attribute alternatively
```

```
CALL FSSCOLOR 'field' ,colour-attribute
```

23.3.18 FSSKEY

Return Key entered. When the user presses an action-key on a screen the used key value to return control can be accessed by FSSKEY. The optional parameter CHAR returns it in a translated readable form if not set the value returned is the decimal value assigned to the action key.

```
key=FSSKEY([CHAR])
```

By FSS supported keys:

REXX Variable	Numeric Value	Translated Value
#ENTER	125	ENTER
#PFK01	241	PF01
#PFK02	242	PF02
#PFK03	243	PF03
#PFK04	244	PF03
#PFK05	245	PF05
#PFK06	246	PF06
#PFK07	247	PF07
#PFK08	248	PF08

#PFK09	249	PF09
#PFK10	122	PF10
#PFK11	123	PF11
#PFK12	124	PF12
#PFK13	193	PF13
#PFK14	194	PF14
#PFK15	195	PF15
#PFK16	196	PF16
#PFK17	197	PF17
#PFK18	198	PF18
#PFK19	199	PF19
#PFK20	200	PF20
#PFK21	201	PF21
#PFK22	74	PF22
#PFK23	75	PF23
#PFK24	76	PF24
#CLEAR	109	CLEAR
#RESHOW	110	RESHOW

23.3.19 FSSDISPLAY

Displays or Re-Displays the active screen.

```
CALL FSSDISPLAY
```

```
CALL FSSREFRESH
```

23.3.20 Get Screen Dimensions

```
width=FSSWidth()
```

Returns number of available columns defined by Emulation

```
height=FSSHeight()
```

Returns number of available rows defined by Emulation

23.3.21 Close FSS Environment

Once the Screen Handling is finished it is recommended to terminate the FSS environment with one of:

```
CALL FSSTERM
```

```
CALL FSSTERMINATE
```

```
CALL FSSCLOSE
```

23.4 Creating a Dialog Manager

To handle user's action-keys, you can set up a simple Dialog Manager, as shown in this example:

```
/* ---------------------------------------------------------------------
* Display screen in primitive Dialog Manager and handle User's Input
```

```
* ------------------------------------------------------------------
*/
do forever
    fsreturn=fssDisplay()           /* Display Screen */
    if fsreturn='PFK03' then leave  /* QUIT requested */
    if fsreturn='PFK04' then leave  /* CANCEL requested */
    if fsreturn='PFK15' then leave  /* QUIT requested */
    if fsreturn='PFK16' then leave  /* CANCEL requested */
    if fsreturn<>'ENTER' then iterate
    call fSSgetD()                  /* Read Input Data */
    /* Add input checking if needed */
end
call fssclose /* Terminate Screen Environment */
```

23.5 Simple Screen Applications

There is a simple way to create formatted screens using preformatted rexx scripts, and this allows an easy screen setup without coding all the screen definitions manually.

23.5.1 Screen with Attributes in one Column

From *BREXX.V2R5M0.SAMPLES(#FSS1COL)*

```
 /*          + ------------------- screen with 1 column
  *          !
  *          !    + -------------- Title line of screen
  *          !    !     */
frc=FMTCOLUM(1,'One Columned Formatted Screen',
     ,'1. Input ===>',
     ,'2. Input ===>',
     ,'3. Input ===>',
     ,'4. Input ===>',
     ,'5. Input ===>',
     ,'6. Input ===>',
     ,'7. Input ===>',
     ,'8. Input ===>',
     ,'9. Input ===>',
     )
do i=1 to _screen.input.0
   say "User's Input "i'. Input Field: '_screen.input.i
end
return
```

The above definition creates and displays this screen:

```
------------------------ One Columned Formatted Screen ------------------------

1. Input ===> ____________________________________________________________________
2. Input ===> ____________________________________________________________________
3. Input ===> ____________________________________________________________________
4. Input ===> ____________________________________________________________________
5. Input ===> ____________________________________________________________________
6. Input ===> ____________________________________________________________________
7. Input ===> ____________________________________________________________________
8. Input ===> ____________________________________________________________________
9. Input ===> ____________________________________________________________________
```

After entering input and pressing enter, you receive the provided input:

```
------------------------ One Columned Formatted Screen ------------------------

1. Input ===> Brexx
2. Input ===> is
3. Input ===> the
4. Input ===> Best
5. Input ===>
6. Input ===>
7. Input ===>
8. Input ===>
9. Input ===>
```

The provided input is stored in SCREEN.INPUT.xx an can be used or printed as in this REXX script:

```
User's Input 1. Input Field: Brexx
User's Input 2. Input Field: is
User's Input 3. Input Field: the
User's Input 4. Input Field: Best
User's Input 5. Input Field:
User's Input 6. Input Field:
User's Input 7. Input Field:
User's Input 8. Input Field:
User's Input 9. Input Field:
```

23.5.2 Screen with Attributes in two Columns

From *BREXX.V2R5M0.SAMPLES(#FSS2COL)*

```
 /*          + ------------------- screen with 2 columns
  *          !
  *          !    + -------------- Title line of screen
  *          !    !     */
frc=FMTCOLUM(2,'Two Columned Formatted Screen',
     ,'1. Input ===>',
     ,'2. Input ===>',
     ,'3. Input ===>',
     ,'4. Input ===>',
     ,'5. Input ===>',
     ,'6. Input ===>',
     ,'7. Input ===>',
     ,'8. Input ===>',
     ,'9. Input ===>',
     )
do i=1 to _screen.input.0
   say "User's Input "i'. Input Field: '_screen.input.i
end
return
```

you get the attributes in two columns:

```
------------------------ Two Columned Formatted Screen ------------------------

1. Input ===> ___________________________ 2. Input ===> ___________________________
3. Input ===> ___________________________ 4. Input ===> ___________________________
5. Input ===> ___________________________ 6. Input ===> ___________________________
7. Input ===> ___________________________ 8. Input ===> ___________________________
9. Input ===> ___________________________
```

Entered input is provided in the same way as in the one column screen example.

23.5.3 Screen with Attributes in three Columns

Just change the number of columns to 3: *frc=FMTCOLUM(3,'Three Columned Formatted Screen',*

23.5.4 Screen with Attributes in four Columns

Last option is to place the attributes in four columns: *frc=FMTCOLUM(4,'Three Columned Formatted Screen',*

23.5.5 Screen special Attributes

You can tailor the appearance of formatted column screens, by setting **_screen.xxxx** variables:

Presetting Screen input fields

Use _SCREEN.INIT.n='input-value-as-default', n is the reference to the field in the FMTCOLUMN definition. 1 is first, 2 second, etc.

Example:

```
_SCREEN.INIT.1='FRED'
_SCREEN.INIT.3='Flintstone'
_SCREEN.INIT.4='FL2311'
_SCREEN.INIT.5='Quarry'
```

Calling the formatted screen, you get a pre-set Screen:

```
------------------------ One Columned Formatted Screen ----------------------

1. First Name  ===> Fred_____________________________________________
1. Family Name ===> Flintstone_______________________________________
2. UserId      ===> FL2311___________________________________________
3. Department  ===> Quarry___________________________________________
```

Input field appearance

If not changed, the input fields appear with an underscore in the available length. You can change it by setting _screen.preset. If you set _screen.preset='+' (one character) the input field filled by the character you defined. If you use more than one character *_screen.preset='_ '* only the given string is displayed.

Input field length

The field length is, by default, delimited by the following field definition in the row, or by the end of the line.

If you want to limit it to a certain length by: *_SCREEN.LENGTH.n=field-length* n is the field number you want to set. It is sufficient to set just the field length you want to limit.

Input Field CallBack Function

Normally, if you press enter, the screen control is giving back to your rexx, and the variable content is returned. If you prefer to check the entered input while your formatted screen is still active, for example, to validate user's input, you can define a callback function:

```
_screen.ActionKey='internal-subprocedure'
```

The internal sub-procedure must be coded without a PROCEDURE statement; else you cannot use the screen input variables

```
_screen.ActionKey='checkInput'
frc=FMTCOLUM(2,'Two Columned Formatted Screen',
...
```

```
return
/* ---------------------------------------------------------------------
* Call Back Routine from FMTCOLUMN to check provided Input
* ---------------------------------------------------------------------
*/
checkInput:
  if _screen.input.1 = '' then do
    call FSSzerrsm 'Field 1 ist mandatory'
    call FSSzerrlm 'Please enter valid content in Field 1'
    return 1
end
if _screen.input.2 = '' then do
    call FSSzerrsm 'Field 2 is mandatory'
    call FSSzerrlm 'Please enter valid content in Field 2'
    return 1
end
...
```

In case of an error, your call back function can use the FSSzerrsm function, which displays a short message in the formatted screen's title line and/or the FSSzerrlm function to display a long message. The error message is displayed in the last line of Formatted Screen. Your callback sub-procedure signals with its return code how to proceed:

Return	Description
return 0	everything ok, leave screen an pass control back to calling rexx
return 128	something is wrong, re-display the screen
return 256	something is wrong, leave the screen
return n:	field n contains wrong input, re-display screen n >0 and n<128 represents the field number in error

23.6 FSSMENU Supporting Menu Screens

To ease the creation of menu screens, you can use the FSSMENU definition. It creates the screen layout as well as the dialogue handling part.

23.6.1 Defining a Menu Screen

```
CALL FSSMENU 'option','note','description','action',[startRow],[startCol]
```

Parameters

The FSS menu definitions can be included within a typical FSS Screen definition to add additional fields or text parts to the formatted screen. These parts can be dynamically updated if you specify a callback procedure in the FSSMENU Display call.

The FSSMENU definition relies on the existence of the following fields (FSSMENU does not automatically generate them); they must be defined separately, either implicitly or explicitly:

- ZCMD is defined by FSSTOPLINE or FSSCOMMAND
- ZERRSM is defined by FSSTITLE

Example defined in a REXX script:

```
...
CALL FSSMENU 1,"RFE",    'SPF like" productivity tool',
              ,"TSO CALL 'SYS2.CMDLIB(RFE)"
CALL FSSMENU 2,"RPF",    'SPF like" productivity tool','TSO RPF'
CALL FSSMENU 3,"IM",     'IMON/370 system monitor','TSO IM'
CALL FSSMENU 4,"QUEUE",  'spool browser','TSO Q'
```

```
CALL FSSMENU 5,"HELP",   'general TSO help','TSO HELP'
CALL FSSMENU 6,"UTILS",
     ,'information on utilities and commands available','TSO HELP UTILS'
CALL FSSMENU 7,"TERMTEST" ,'verify 3270 terminal capabilities',
     ,'TSO TERMTEST'
...
```

23.6.2 FSSMENU Displaying a Menu Screen

To display the menu and handle the selected actions, FSSMENU must be called with the $DISPLAY parameter:

returnkey=FSSMENU('$DISPLAY',[callback-procedure],[actionkey-procedure])

Parameters

Example: Simple Display without any exits

```
rckey=FSSMENU('$DISPLAY')
say 'End Key 'rckey
...
```

Example: Before Display update some variables via a callback procedure

```
rckey=FSSMENU('$DISPLAY','UPDVAR')
say 'End Key 'rckey
...
/* ---------------------------------------------------------------------
* Update some Variables before displaying the Menu
* ---------------------------------------------------------------------
*/
Updvar:
MDate=date() /* assuming MDATE/MTIME are defined in the MENU */
MTime=time('L')
Return
...
```

Example: Before Display update some variables via a callback procedure, and check command line input via an enter-exit

```
rckey=FSSMENU('$DISPLAY','UPDVAR','CHECKKEY')
say 'End Key 'rckey
...
...
/* ---------------------------------------------------------------------
* Update some Variables before displaying the Menu
* ---------------------------------------------------------------------
*/
Updvar:
MDate=date() /* assuming MDATE/MTIME are defined in the MENU */
MTime=time('L')
Return
/* ---------------------------------------------------------------------
* Check user's Input in command Line
* Return code handling:
*  0 input has been handled by exit, re-display Menu
*  4 input has not been handled, continue with internal checks
*  8 exit Menu immediately
* ---------------------------------------------------------------------
*/
CheckKey:
Parse arg actionkey,usercommmand
If length(usercommand)>2 then do
```

```
  Say usercommand' is not an Option'
  Return 0 /* continue, command already checked */
End
Return 4
/* maybe an Option, continue to option check */
```

23.6.3 FMTMENU Fully Defined Menu Screens

Using FSSMENU, you can define the menu lines and generate the menu handling, but it must be incorporated in a normal REXX script containing the other parts of the screen definition and handling. **FMTMENU** allows you the definition of a menu screen in one step, but there are additional screen definitions in the menu possible.

Definition of the Menu

```
CALL FMTMENU 'option','note','description','rexx-script'
```

Parameters

- **option** – option code which leads to performing the associated action. The option can be a numeric or alphanumeric string.
- **note** – the short description of the action to perform
- **description** – long description of the action to perform
- **rexx-script** – REXX script which performs the action when the option is selected. Note the difference, to FSSMENU, here it must be a REXX script, but it may also contain calls to TSO, etc.

An FMTMENU always contains a title line (first row) an option line (second row) a message line (last row -1) and a footer line (last row).

Displaying the FMTMENU Screen

To display the menu and handle the selected actions, FMTMENU must be called with the $DISPLAY parameter:

```
returnkey=FMTMENU('$DISPLAY','menu-title')
```

Parameters

- **returnkey** – key which was pressed to end the dialogue handling, it is either PF03, PF04, PF15, or PF16
- **$DISPLAY** – Display the menu defined before
- **menu-title** – defining the menu title

Menu Tailoring

There are some settings, which allow you to tailor the menu layout. The usage of the stem *_screen* defines all settings .xxx. These settings are supported in FSSMENU as well as in FMTMENU.

Setting	Description
_screen.MenuRow	starting row of first Menu entry (default is 4)
_screen.MenuCol	Column of Option parameter (default is 6)
_screen.Menucol2	Column of note parameter (default is _screen.MenuCol+3)
_screen.Menucol3	Column of note parameter (default is _screen.MenuCol+14)

Note for FSSMENU:

there are separate parameters startrow and startcol in the menu definition

```
CALL FSSMENU 'option','note','description','action',[startRow],[startCol]
```

If they are defined, they take precedence over the screen.MenuRow and screen.MenuCol definition.

_screen.MenuFooter	defines the contents of a footer line (placed on the last row)

Setting just for FSSMENU (in FMTMENU they are managed automatically)

Setting	Description
_screen.MenuOption 1	adds an Option line, else it must be defined manually
_screen. MenuMessage 1	adds a message line (last row-1)
_screen. Menutitle' 1	adds a title line

Formatted List Output

The usage of SAY statements displays the standard output of a REXX script. The disadvantage you can not scroll in it. Alternatively, you can write it in a sequential file and view it after the script has ended. By using the FMTLIST command and passing a result buffer in a stem variable, you can browse in the output while your REXX script is still running.

Example REXX reads entire RXDATE Member and displays it:

```
/* REXX */
ADDRESS TSO
"ALLOC FILE(INDD) DSN('BREXX.CURRENT.RXLIB(RXDATE)')"
"EXECIO * DISKR INDD (STEM BUFFER."
"FREE FILE(INDD)"
CALL FMTLIST
RETURN
```

Results:

```
CMD ==>                                                  ROWS 00001/00199 COL 001 B01
***** ***************************** Top of Data ******************************
00001 /* REXX */
00002 /* ------------------------------------------------------------------
00003  *  should not be used anymore, all date functions are integrated in
00004  *    DATE(<output-format>,<date>,<input-format>)
00005  * ------------------------------------------------------------------
00006  *  RXDATE Transforms Dates in various types
00007  *  ............................. Created by PeterJ on 21. November 2018
00008  *  RXDATE(<output-format>,<date>,<input-format>)
00009  *  date is formatted as defined in input-format
00010  *    it defaults to today's date
00011  *  Input Format represents the input date format
00012  *    it defaults to 'EUROPEAN'
00013  *     Base      is days since 01.01.0001
00014  *     JDN       is days since 24. November 4714 BC
00015  *     UNIX      is days since 1. January 1970
00016  *     Julian    is yyyyddd    e.g. 2018257
00017  *     European  is dd/mm/yyyy e.g. 11/11/2018
00018  *     German    is dd.mm.yyyy e.g. 20.09.2018
00019  *     USA       is mm/dd/yyyy e.g. 12.31.2018
00020  *     STANDARD  is yyyymmdd   e.g. 20181219
...
```

Using the PF7 and PF8 you scroll upward and forward, PF10 and PF11 scroll left and right. M in the CMD line and PF7 moves buffer to the top, M and PF8 to the bottom. A number and PF7 or PF8 moves the buffer the specified lines up or down.

FMTLIST Prerequisites

FMTLIST always displays the content of the stem variable BUFFER. The buffer must have the general structure:

BUFFER.0	contains the number of entries in BUFFER
BUFFER.1	contains the first line
BUFFER.2	second line
...	
BUFFER.n	last line

As the name is fixed, it does not need to be passed to FMTLIST.

FMTLIST calling Syntax

```
FMTLIST
[length-line-area],[line-area-character],[header-1],[header-2],[applicationID]
```

Parameters

- **`length-line-area`** – length of displayed line-area, default is 5
- **`line-area-character`** – character which should be displayed in the line area, default is none, then the line area contains the line number
- **`header-1`** – this is an optional header line which is shown as first-line the displayed buffer
- **`header-2`** – optional second header, only if header-1 is also defined
- **`applicationID`** – If you specify an application ID, the FMTLIST screen supports line commands. The Line commands must be defined and coded in the calling REXX script as a callback label: applicationID_linecommand.

If you use PF7/PF8 to scroll up and down, the two header lines are always displayed as the buffer top lines.

FMTLIST supported PF Keys and Scrolling commands

PF3/PF4	exit FMTLIST screen
PF7	scroll one page up
PF8	scroll one page down
PF10	shift buffer 50 columns left
PF11	shift buffer 50 columns right
PF12	Display last command

If you use a combination of a number in the command line and PF7 or PF8, the buffer scrolls the number of lines up or down.

Command-line functions:

TOP	displays the first line of the buffer
M and PF7	displays the first line of the buffer
BOTTOM	displays the last line of the buffer
BOT	displays the last line of the buffer
M and PF8	displays the last line of the buffer

FMTLIST Customising Options

By setting _SCREEN.xxxx, you can manipulate the appearance of FMTLIST in various ways:

Variable Name	Default	Allowed Values	Note
_screen.cmdchar	blank		Command Line character building the command line. Default is blank and creates an empty command line which is displayed with the 3270 attribute #USCORE

_screen.color.Cmd	#red	Attribute Definitions	Colour of Command Line
_screen.color.Stats	#white	Attribute Definitions	Colour of Statistics (line and buffer numbering)
_screen.color.Top1	#red	Attribute Definitions	Colour of line area first line
_screen.color.Top2	#blue	Attribute Definitions	Colour of line conten first line (Top of Data)
_screen.color.Bot1	#red	Attribute Definitions	Colour of line area last line
_screen.color.Bot2	#blue	Attribute Definitions	Colour of line content last line (End of Data)
_screen.color.List1	#white	Attribute Definitions	Colour of line area (content part)
_screen.color.List2	#green	Attribute Definitions	Colour of line content part
_screen.footer	undefined	Content of footer (PF1 ...)	Fixed Footer Line (at screen height)
_screen.color.footer	#white	Attribute Definitions	Colour of line content part
_screen.Message	undefined	1 for defining message	Fixed Message Line (screen height-1)
_screen.TopRow	1	1 up to Screen height-3	Begin row of fmtlist, if it is 2 or more there are empty lines above FMTLIST
_screen.TopRow.proc	Undefined		Is a call-back proc name in the rexx calling FMTLIST. There you can define the line above the FMTLIST screen. They can be set with FSSfield or FSSText commands. The number of added rows must not exceed _screen.TopRow-1
_screen.BotLines	Lines reserve at bottom of FMTLIST	1 up to Screen height-3	As screen height is dynamic depending on the 3270 definitions.
_screen.BotLines.proc	Undefined		Is a call-back proc name in the rexx calling FMTLIST. There you can define the lines at the end of the FMTLIST screen. They can be set with FSSfield or FSSText commands. The first line number which can be set is passed as arg(1) parameter. For consistency reasons of call back parameters, it is enclosed in quotes. This means you must strip them off: *first=strip(translate(arg(1),",""))*

FMTLIST calling other REXX scripts from the command line

If you want to play another REXX script from within the FMTLIST buffer you can do so, by entering: *rexx-script-name* in the command command line.

Simple REXX scripts

A simple Rexx script does not contain any call to an FSS Screen. A sequence of say statements may provide the result, or you can place it in a buffer.x stem. If you do so, the result displayed in the current

FMTLIST buffer. Which means the existing content is overwritten.

```
Buffer.1='first line'
Buffer.2='second line'
Buffer.0=2
```

If you want to keep the contents of the current buffer, use the prefix command *LOOKASIDE rexx-script-name*, and a new stacked buffer is created residing on top of the previous buffer. The previous buffer can be re-activated by pressing the PF3 key; it destroys the current buffer and returns to the last buffer.

If the called rexx-script contains an FMTLIST, FSSMENU, or FMTMENU itself a new buffer is created automatically.

Formatted List Line and Primary Commands

The FMTLIST Buffer supports Line Commands if it is called with an applicationID. The line command is coded within the calling procedure (performing the FMTLIST) as a callback label, to keep the scope of the variables there must not be a PROCEDURE statement used. The callback label must be coded as: applicationID_linecommand. In the following example there is a line command S, U and D defined :

```
/* REXX */
ADDRESS TSO
"ALLOC FILE(INDD) DSN('BREXX.RXLIB(RXDATE)')"
"EXECIO * DISKR INDD (STEM Buffer."
"FREE FILE(INDD)"
call fmtlist ,,,,MYLIST /* MYLIST is application ID */
return

/* ---------------------------------------------------------------------
* Line commands are organised as "call-back' labels to the calling REXX
* Format is REXX name_linecmd
* ---------------------------------------------------------------------
*/

mylist_s: /* line command S, just output selected line */
  say Arg(1)
  return 0 /* tell FMTLIST to proceed normally */

mylist_u: /* line command U, allow editing line */
  newLine=lineedit(,arg(1))
  return 4 /* tell FMTLIST, you changed line */

mylist_e: /* line command E, automatically change line */
  newLine='new Line set'
  zerrsm='update'
  zerrlm='Line has been updated'
  return 4 /* tell FMTLIST, line is changed line */

mylist_d: /* Delete Line */
  return 5 /* tell FMTLIST to delete selected line */
```

RC Code actions

RC=0	means the line command was processed
RC=4	means the line command was processed; if the REXX variable NEWLINE contains a value, the selected line will be overwritten by this value.
RC=5	delete this line
RC=6	a completely new buffer.n stem has been provided and should be displayed immediately. The old buffer content will be removed. If you set a ZERRSM or ZERRLM message the message will be kept and displayed.

RC=7	a new buffer.n stem has been provided and should be displayed in a new FMTLIST buffer, which is stacked on top of the previous one. Once you return with PF3 you will see the old buffer content. If you set a ZERRSM or ZERRLM message the message will be kept and displayed.
RC=8	invalid line command

Additionally, you can change the colour of the line in the buffer; you have to set:

- SETCOLOR1 sets the colour of the selected line of the line area, e.g. *setcolor1=#green*
- SETCOLOR2 sets the colour of the selected buffer content line, e.g. *setcolor2=#red*

If none or just one of the colours have been set, the other field colour remains unchanged

Formatted List Samples

There are several scripts in *BREXX.V2R5M1.SAMPLES* illustrating the usage of FMTLIST.

FMTOPBOT	has an embedded FMTLIST with user-defined header and footer lines.
@STUDENTL	the front end of the VSAM student database example
#BROWSE	Displays the LISTALC command

Debugging Simple Screen Applications

If you need to debug the behaviour of simple screen applications, you can switch on a trace feature in the calling REXX script:

```
_screen.FTRACE=1
```

You get a trace of the performed step within the screen application.

```
/* REXX */
do i=1 to 35
    buffer.i='Buffer Line 'i
end
buffer.0=i-1
/*
_screen.color.top2=#yellow
_screen.color.mylist=#red
_screen.color.cmd =#blue
_screen.color.stats=#white
*/
_screen.footer='PF1 Help PF3 Return PF4 Return'
_screen.Message=1
CALL FMTLIST ,,'','','TEST'
```

Displaying Trace in TSO:

```
09:45:27.09 Entering FMTLIST
09:45:27.18 Display Screen
***
The screen is displayed, waiting for the next user action
09:45:56.65 User Action PF08
09:45:56.69 Command Line ''
09:45:56.71 Display Screen
***
The screen is displayed, waiting for the next user action
09:46:42.13 User Action PF07
09:46:42.17 Command Line '10'
09:46:42.20 Display Screen
***
```

```
The screen is displayed, waiting for the next user action
09:47:10.09 User Action PF03
09:47:10.09 Command Line ''
***
```

Formatted List Monitor FMTMON

By setting up a formatted list monitor you can monitor certain events on a timely basis. You can for example continuously view updated entries of the Master Trace Table

```
CALL IMPORT FSSAPI
/* ---------------------------------------------------------------------------
* FMTMON is an FSS application that refreshes itself every xxx milliseconds
* the refresh takes place in the call-back procedure MonTimeOut it must
* provide a new buffer or just return
* There is also an enter-key call-back procedure MonEnter where you can
* execute commands, e.g. CONSOLE and modify the buffer if wanted
* ---------------------------------------------------------------------------
*/
call fmtmon "MVS Trace Table",1000
return 0
```

FMTMON calling Syntax

FMTMON header,[refresh-frequency]

Parameters

- **header** – is displayed as title in the FMTMON screen
- **refresh-frequency** – refresh timer in milliseconds

FMTMON Call-Back Procedures

FMTMON requires two call-back procedures, which must be implemented in the calling REXX procedure.

1. MONENTER: is called when has entered input and presses the enter-key

```
/* ---------------------------------------------------------------
* MONENTER Call Back PROC of FMTMON Enter key pressed, do something
* return 0 continue normally
*  4 continue normally, buffer is not touched
*  8 end monitor (as PF3)
*  12 end monitor (as PF4)
* ---------------------------------------------------------------
*/
MonEnter:
call CONSOLE arg(1)
/* action requested console command */
return 0
```

2. MONTIMEOUT: is called when the frequence-time-out has been reached

```
/* ---------------------------------------------------------------
* MONTIMEOUT Call Back PROC of FMTMON Enter key pressed, do something
* Timeout in FSS, you can provide new content in
* BUFFER.i i=1 to number of lines
* BUFFER.0 must contain number of lines
* return 0 continue buffer is unchanged
* 1 continue new buffer provided
* ---------------------------------------------------------------
```

```
*/
MonTimeout:
/* arg(1) entry count */
/* create new contents of FMTMON Buffer.
return
```

FMTMON provide data to display

FMTMON displays the content of the stem variable BUFFER, typically it is updated in the MONTimeout call-back procedure.

The buffer must have the general structure:

BUFFER.0	contains the number of entries in BUFFER
BUFFER.1	contains the first line
BUFFER.2	second line
...	
BUFFER.n	last line

As the name is fixed, it does not need to be passed to FMTMON.

FMTMON predefined Action Keys

- Help key: PF1
- Scrolling keys: PF7/PF8
- Commands: TOP/BOT/UP n(-lines)/DOWN n(-lines)

FMTMON Application display Master Trace Table

This example is stored in: *BREXX.V2R5M1.SAMPLES(MTT)*

23.7 FSS Functions as Host Commands

Alternatively to the FSS functions described in ”FSS Function Overview” you can use the FSS Host command API directly. In this case, all definitions, calculations, validations, etc. must be handled by your REXX script directly.

23.7.1 INIT FSS Environment

Initialise the FSS environment; this must be performed before any other FSS call:

```
ADDRESS FSS
'INIT'
```

23.7.2 Defining a Text Entry

```
ADDRESS FSS
'TEXT 'row column attributes text'
```

- **text**: text to be displayed on the screen
- **row**: row where text should be placed
- **column**: column where text should be placed.
- **attributes**: screen attributes, like colours, protected, high-lighted etc. For details refer to the attributes section

23.7.3 Defining a Field Entry

```
ADDRESS FSS
'FIELD 'row column attributes field flen [preset]'
```

- **text**: text to be displayed on the screen
- **row**: row where text should be placed
- **column**: column where text should be placed.
- **attributes**: screen attributes, like colours, protected, high-lighted etc. For details refer to the attributes section
- **field**: Screen field name
- **flen**: length of input area representing field name
- **preset**: content initially displayed (optional), defaults to blank

23.7.4 Getting Field Content

```
ADDRESS FSS
'GET FIELD field rexx-variable'
```

- **field**: Screen field name
- **rexx-variable**: variable receiving the field content

23.7.5 Setting Field Content

```
ADDRESS FSS
'SET FIELD field value'
```

or

```
ADDRESS FSS
'SET FIELD field 'rexx-variable'
```

- **field**: Screen field name
- **value**: new field content
- **rexx-variable**: variable containing the field content

23.7.6 Setting Cursor to a field

Sets the cursor to the beginning of the Screen Field

```
ADDRESS FSS
'SET CURSOR field'
```

- **field**: Screen field name

23.7.7 Setting Colour

Sets the Colour of a Screen Field

```
ADDRESS FSS
'SET COLOR field/text colour'
```

- **field**: Screen field name
- **colour**: Color definition,for details refer to the attributes section

23.7.8 Getting action Key

When the user presses an action-key on a screen, the key value can be fetched in a rexx-variable:

```
ADDRESS FSS
'GET AID rexx-variable'
```

- **rexx-variable**: variable receiving the action key

23.7.9 Display or Refresh Formatted Screen

Used to display the Formatted Screen the first time, or to refresh an active screen:

```
ADDRESS FSS
'REFRESH'
```

23.7.10 End or Terminates FSS Environment

Ends the Formatted Screen environment and releases all used main storage:

```
ADDRESS FSS
'TERM'
```

23.7.11 Get Terminal Width

```
ADDRESS FSS
'GET WIDTH rexx-variable'
```

- **rexx-variable**: variable receiving the action key

23.7.12 Get Terminal Height

```
ADDRESS FSS
'GET HEIGHT rexx-variable'
```

- **rexx-variable**: variable receiving the action key

24 Implementation Restrictions

The name of a variable or label, and the length of a literal string may not exceed 250 bytes. More characters than 250 will be truncated. (Can be changed from rexx.h)

Numbers follows C restrictions, thus integers are long and real numbers are held as double.

The FOR and simple counts on a DO instruction, and the right-hand term of an exponentiation may not exceed maximum long number.

The control stack (for DO, IF, CALL, etc.) is limited to a nesting level of 256 and from the internal stack of the Operating system.

Functions and subroutines cannot be called with more than 15 arguments (Can be changed from rexx.h).

Input and Output cannot be redirected for commands executed through INT2E.

24.1 Variables

Variables are held in a binary tree, where the tree is balanced when one branch starts to become very big. Even though the variables are stored as a bintree there is an internal cache system for the faster access.

Each variable in rexx is a length prefix string, and it is kept in memory in 3 different types, long, real, string according the last operation that affect that variable.

```
ie. a = 2   /* will be kept as string (Length-prefixed) */
    a = 2 + 1/* will be kept as integer (long) */
    a = 2 + 0.1/* will be kept as real (double) */
```

The advantage of the above scheme is that numerical operations are performed much faster than the other algorithms. The main disadvantage is on the integer operations. 32 bit integers have a maximum of 2billion, so if you try something like this

```
    factorial = 1
    do i = 1 to 50
            factorial = factorial * i
    end
```

will result to 0 instead of the factorial of 50! To find the correct result you have to fool the interpreter to think that factorial is real and not integer, this can be done if you write factorial = 1.0 ....

You can easilly translate a variable to any format you like with the following instructions

```
    a = a + 0.0   /* will translate a to real */
    a = trunc(a)  /* will translate a to integer */
    a = a || ''   /* will translate a to string */
```

Sometimes it is very important to know how a variable is kept in memory (usually for the INTR function) so there is an extra option in DATATYPE function “TYPE” that returns the way one variable is hold.

```
    DATATYPE(2,"TYPE")      -> "STRING"
    DATATYPE(2+0.0,"TYPE") -> "REAL"
    DATATYPE(2+0,"TYPE")    -> "INT"
```

C routines are used for the translation of string to number, so a string like ‘- 2’ will be reported by DATATYPE as a NUMber when rexx tries to evaluate it as a number it will return a value of 0 instead of -2, because of the spaces between the sign and the number.

24.2 Stems

substitution to stems may be anything including strings with any character. No translation to uppercase is done to subscripts

```
     lower = 'ma'  ; stem.lower   -> 'STEM.ma'
     upper = 'MA'  ; stem.upper   -> 'STEM.MA'
```

Stems can be initialized with a command like stem. = 'Initial value'

24.3 Functions

TRANSLATE sometimes wont work properly for strings with characters above ASCII 127. Works OK for Greek character set.

VARTREE wont work properly with variables with non-printable characters

25 Migration and Upgrade Notices

This section covers the changes in the new version, migration instruction to upgrade from the previous Release. The installation process is separately described in the BREXX/370 Installation section.

25.1 Upgrade from a previous BREXX/370 Version

Before upgrading backup your system. The easiest way is creating a copy of your TK4- directory containing all of your settings DASD volumes. In the cases of errors or unwanted behaviour, you can easily recover to the backup version.

25.2 BREXX V2R1M0

25.2.1 Important Changes

Due to the extended calling functionality in the new version of BREXX/370 an import of required REXX scripts is no longer necessary. For this reason, all pre-defined import libraries have been removed from the JCL Procedures RXTSO and RXBATCH. The installation will update them in SYS2.PROCLIB. For similar reasons the CLISTs RX and REXX are no longer necessary and will be therefore removed from SYS2.CMDPROC, there will be an RX and REXX member in SYS2.LINKLIB which replaces the CLIST version.

Important If you made changes or extensions in the PROCLIB Member RXTSO and/or RXBATCH save the changes to re-introduce them in the newly installed version.

If you made changes or extensions in the CMDPROC Members RX and/or REXX save them to incorporate them in a newly created function of your own.

25.2.2 Libraries

The following BREXX libraries are necessary for running REXX:

- BREXX.JCL
- BREXX.SAMPLES
- BREXX.SAMPLIB **new with this version**
- BREXX.RXLIB

They will be delivered and created during the installation process, existing libraries will be overwritten!

Warning If you made changes or added your own entries in one of these libraries, save them before beginning with the installation process!

25.2.3 Calling external REXX Scripts or Functions

It is now possible to call external REXX scripts, either by:

```
CALL your-script parm1,parm2...
```

or a function call:

```
Value=your-script(parm1,parm2,...)
```

The called script will be sought by the following sequence:

- Internal subprocedure or label (contained in the running script)
- current library (where the calling REXX is originated)
- BREXX.RXLIB

Important The variable scope slightly differs from IBM's REXX implementation. In IBM's implementation a call to an external REXX script, the called REXX is always treated as a procedure without access to the caller's variable pool. BREXX can access and update the caller's pool. If a PROCEDURE is used in the called BREXX script, variables can be made available by the EXPOSE statement.

25.2.4 Software Changes requiring actions

The STORAGE function has been changed to become compatible with IBM's z/OS REXX STORAGE version.

In IBM's REXX, the storage address must be in hex format, in BREXX it was decimal. With this version, we match with IBM's specification and allow only hex notation.

Important If you have created REXX scripts using the STORAGE function and dislike to update all of them, you can replace them by the BSTORAGE function, which still works with decimal addresses. BSTORAGE is a REXX function being part of the BREXX: RXLIB library.

25.2.5 New Functionality

BREXX functions coded in REXX

ABEND (*abend-code*)
: Terminate program with Abend-Code, produces an SYSUDUMP

USERID ( )
: Signed in UserId (available in Batch and Online)

WAIT (*wait-time*)
: Stops REXX script for some time, wait-time is in hundreds of a second

WTO (*console-message*)
: Write a message to the operator's console

SYSVAR (*request-type*)
: a TSO-only function to retrieve certain TSO runtime information request-type:

Request Type	Description
SYSENV	FORE/BACK - Foreground TSO / Batch TSO
SYSISPF	ACTIVE/NOT ACTIVE
SYSPREF	TSO Prefix (only available in Foreground TSO)
SYSUID	Userid

Brexx has the capability code new functions or command in REXX. They are transparent and will be called in the same way as basic BREXX function.

Overview:

BSTORAGE (...)
Storage command in the original BREXX decimal implementation

LISTALC ( )
Lists all allocated Datasets in this session or region

BRXMSG (...)
Standard message module to display a message in a formatted way, examples:

```
rc=brxmsg( 10,'I','Program has been started')
rc=brxmsg(100,'E','Value is not Numeric')
rc=brxmsg(200,'W','Value missing, default used')
rc=brxmsg(999,'C','Division will fail as divisor is zero')
```

will return::
BRX0010I PROGRAM HAS BEEN STARTED BRX0100E VALUE IS NOT NUMERIC BRX0200W VALUE MISSING, DEFAULT USED BRX0999C DIVISION WILL FAIL AS DIVISOR IS ZERO

DAYSBETW (*date1*, *date-2*, ...)
Return days between 2 dates

JOBINFO (*request-type*)
Return information about currently running job or TSO session.
Request-type:

- JOBNAME - returns job name
- JOBNUMBER - returns job number
- STEPNAME - returns step name
- PROGRAMNAME - returns running program name

LINKMVS (*load-module*, *parms*)
Starts a load module. Parameters (if any) will send in the format JCL would pass it.

MVSCBS ( )
Allows addressing of some MVS control blocks. This function must be imported, then the following functions can be used: Cvt(), Tcb(), Ascb(), Tiot(), Jscb(), Rmct(), Asxb(), Acee(), Ecvt(), Smca()

PDSDIR (*dsn*)
Return directory entries in a stem variable

RXDATE (...)
Return and optionally convert dates in certain formats

RXDSINFO (*dsn/dd-name*, *options*)
Return dsn or dd-name attributes

3RXDYNALC (...)
Allows dynamic allocations of datasets or output

RXSORT (...)
Sorts a stemvariable with different sort algorithms

SEC2TIME (*seconds*)
Converts an amount of seconds into the format [days]hh:mm:ss

SORTCOPY (*stem-variable*)
Copies any stem variable into the stem SORTIN, which can be used by RXSORT

STEMCOPY (*source-stem-variable*, *target-stem-variable*)
Copies any stem variable into another stem variable

TODAY ( )
Returns todays date in certain formats

25.3 BREXX V2R2M0

25.3.1 Software Changes requiring actions

OPEN ( )
In the *OPEN(ds-name„ 'DSN')* function the third parameter DSN has been removed to achieve closer compatibility to z/OS REXX.
Change Required: Replace OPEN(ds-name„'DSN') by OPEN('ds-name',) putting the ds-name in quotes or double-quotes signals BREXX that an open for a ds-name instead of a dd-name. If you use a BREXX variable instead of a fixed ds-name the quotes must be coded like this:

```
file='BREXX.RXLIB'
OPEN("'"file"'")
```

QUALIFY ( )
The QUALIFY(ds-name) function added in TSO environments the user-prefix. This function has been removed as its functionality has been integrated into the OPEN function.

BSTORAGE ( )
The BSTORAGE function has been removed as it was a temporary solution for users of the very first BREXX/370 version. If you plan to keep it, take a copy from the previous RXLIB library.

25.3.2 Reduction of Console Messages

In previous releases, console messages have been displayed during the search of a called REXX script in the BREXX search path. It reported the library name if it was not located in the specific library. This messaging has been significantly brought down. These messages only appear if the called member could not found anywhere in the search path.

25.3.3 Known Problems

Reading Lines fromsequential Dataset

Reading lines of sequential datasets always truncate trailing spaces. This may be an unwanted behaviour for fixed-length datasets. To circumvent this problem you can use the following method:

If the dataset is allocated via a DD statement:

```
X=LISTDSI('INFILE FILE')
fhandle=OPEN(infile,'RB')
Record=READ(fhandle,SYSLRECL)
```

If the dataset is used directly:

```
dsn='HERC01.TEMP'
X=LISTDSI("'"dsn"'")
fhandle=OPEN("'"dsn"'",'RB')
```

```
Record=READ(fhandle,SYSLRECL)
```

LISTDSI returns the necessary DCB information (SYSLRECL). The OPEN must be performed with OPTION 'RB' which means READ, BINARY. Read uses the record length to create the record.

BREXX FORMAT Function

The BREXX FORMAT function differs from the standard behaviour of REXX FORMAT:

> FORMAT rounds and formats number with before integer digits and after decimal places. expp accepts the values 1 or 2 (WARNING Totally different from the Ansi-REXX spec) where 1 means to use the "G" (General) format of C, and 2 the "E" exponential format of C. Where the place of the total width specifier in C is replaced by before+after+1. (expt is ignored!)

After determining the code we discovered that a complete re-write would be necessary. As the effort does not stand in proportion to the benefit, we decided to leave it as it is.

25.3.4 New Functionality

25.3.5 BREXX functions

- **Support of Formatted Screens** Refer to the section on formatted screens for more information.
- **Integration of VSAM I/O** Refer to the section on VSAM files for more information.
- **EXECIO Command** Allows accessing sequential datasets either fully or line by line.

CLRSCRN ( )
: Clears the TSO screen by removing all lines from the TSO Buffer.

CEIL ( )
: Returns the smallest integer greater or equal then the decimal number

FLOOR ( )
: Returns the greatest Integer less or equal then the decimal number

DCL ( )
: Enables definition of copybook like definitions of REXX Variables, including conversion from and to decimal packed fields.

P2D ( )
: Converts Decimal Packed Field into REXX Numeric value.

D2P ( )
: Converts REXX Numeric value into Decimal Packed Field.

25.4 BREXX V2R3M0

25.4.1 Authorised BREXX Version available

With this release, we ship a standard installation of BREXX as well as an authorised version, which allows to system programs as IEBCOPY, NJE38, etc. The decision about what to install must be made before installation.

25.4.2 BREXXSTD load module removed

We have straightened the load module structure and removed the BREXXSTD load module from the installation library. If you use JCL with an explicit BREXXSTD call, replace it by BREXX. During the installation process, any existing BREXXSTD module will be removed from SYS2.LINKLIB.

25.4.3 Call PLI Functions

Example compile jobs for callable PLI Functions can be found in BREXX.V2R3M0.JCL:

- RXPI calculate PI with 500 digits
- RXCUT Return every n.th character of a string

25.4.4 New and amended functionality

25.4.5 BREXX functions

CEIL (*decimal-number*)
CEIL returns the smallest integer greater or equal than the decimal number.

D2P (*number*, *length*, *fraction-digit*)
Converts a number (integer or float) into a decimal packed field. The created field is in binary format

P2D (*number*, *length*, *fraction-digit*)
Converts a decimal packed field into a number.

ENCRYPT (*string*, *password* <, *rounds*>)

DECRYPT (*string*, *password* <, *rounds*>)
Encrypts/Decrypts a string

DUMPIT (*address*, *dump-length*)
DUMPIT displays the content at a given address of a specified length in hex format. The address must be provided in hex format; therefore, a conversion with the D2X function is required.

DUMPVAR (*'variable-name'*)
DUMPVAR displays the content of a variable or stem in Hex format-

FILTER (*string*, *character-table* <, *drop/keep*>)
The filter function removes all characters defined in the character-table

FLOOR (*decimal-number*)
FLOOR returns the smallest integer less or equal than the decimal number.

LISTIT (*'variable-prefix'*)
Returns the content of all variables and stem-variables starting with a specific prefix

RHASH (*string*, <*slots*>)
The function returns a numeric hash value of the provided string.

ROUND (*decimal-number*, *fraction-digits*)
The function rounds a decimal number to the precision defined by fraction-digits

UPPER (*string*)

LOWER (*string*)
UPPER returns the provided string in upper cases. LOWER in lower cases.

ROTATE (*string*, *position*<, *length*>)
The function returns a rotating substring

TIMESTAMP (**[,** *daymonthyear* **]**)
TIMESTAMP returns the unix (epoch) time, seconds since 1. January 1970.

Dataset Functions

CREATE (*dataset-name*, *allocation-information*)
The CREATE function creates and catalogues a new dataset

DIR (*partitioned-dataset-name*)
The DIR command returns the directory of a partitioned-dataset

EXISTS (*dataset-name*)

EXISTS (*partitioned-dataset(member)*)
The EXISTS function checks the existence of a dataset or the presence of a member in a partitioned dataset.

REMOVE (*dataset-name*)
The REMOVE function un-catalogues and removes the specified dataset

REMOVE (*partitioned-dataset(member)*)
The REMOVE function on members of a partitioned dataset removes the specified member.

RENAME (*old-dataset-name*, *new-dataset-name*)
The RENAME function renames the specified dataset.

RENAME (*partitioned-dataset(old-member)*, *partitioned-name(new-member)*)
The RENAME function on members renames the specified member into a new one.

ALLOCATE (*ddname*, *dataset-name*)

ALLOCATE (*ddname*, *partitioned-dataset(member-name)*)
The ALLOCATE function links an existing dataset or a member of a partitioned dataset to a dd-name.

FREE (*ddname*)
The FREE function de-allocates an existing allocation of a dd-name.

OPEN (*dataset-name*, *open-option*, *allocation-information*)
The OPEN function has now a third parameter, which allows creating ew datasets with appropriate DCB and system definitions.

TCP Functions

TCPINIT ( )
TCPINIT initialises the TCP functionality.

TCPSERVE (*port-number*)
Opens a TCP Server on the defined port-number for all its assigned IP-addresses.

TCPOPEN (*host-ip*, *port-number* **[,** *time-out-secs* **]**)
TCPOPEN opens a client session to a server.

TCPWAIT (**[,** *time-out-secs* **]**)
TCPWAIT is a Server function; it waits for incoming requests from a client.

TCPSEND (*clientToken*, *message* **[,** *timeout-secs* **]**)
SendLength=TCPSEND(clientToken, message[,time-out-secs]) sends a message to a client.

TCPReceive (*clientToken* **[**, *time-out-secs* **]**)
Receives a message from another client or server.

TCPTERM ( )
Closes all client sockets and removes the TCP functionality

New BREXX functions coded in REXX

GETTOKEN ( )
returns a token which is unique within a running MVS System or in this century

BAS64ENC ( )
Encodes a string or binary string with Base64.

BAS64DEC ( )
Decodes a base64 encoded string into a string or binary string Returns the hash number of a string

STIME ( )
Time since midnight in hundreds of a second

25.5 BREXX V2R4M0

25.5.1 Functions with changed functionality

There is a major change in every time functions. We have increased the precision of the time format from hundreds of a second to milliseconds in some cases to microseconds. If you use them or rely on the format, please change your REXX scripts accordingly:

```
say TIME('L') /* 16:38:03.112765 */
call wait 100 /* now waits 0.1 seconds */
call wait 5000 /* waits 5 seconds */
```

25.5.2 New Functions

This sections contains all new or changed BREXX V2R4M0 functions

DATE (*target-date-format*, *date*, *input-date-format*)
The new date function has now the “used” formats provided by the original REXX.

DATETIME (*target-format*, *timestamp*, *input-format*)
Formats are:

```
T is timestamp in seconds 1615310123
E timestamp European format 09/12/2020-11:41:13
U timestamp US format 12.09.2020-11:41:13
O Ordered Time stamp 2020/12/09-11:41:13
B Base Time stamp Wed Dec 09 07:40:45 2020
```

Time (*'MS'/'US'/'CPU'*)
Time has gotten new input parameters:

- MS Time of today in seconds.milliseconds
- US Time of today in seconds.microseconds
- CPU used CPU time in seconds.milliseconds

LINKMVS (*load-module*, *parms*)

LINKPGM (*load-module*, *parms*)

Start a load module. Parameters work according to standard conventions.

LOCK (*'lock-string'*, *<TEST/SHARED/EXCLUSIVE><*, *timeout>*)

UNLOCK (*'lock-string'*)

Locks/unlocks a resource to avoid concurrent access to it

TIMESTAMP (**[**, *daymonthyear* **]**)

TIMESTAMP returns the unix (epoch) time, seconds since 1. January 1970.

25.6 BREXX V2R4M1

25.6.1 Important Changes

RAKF restrictions lifted

We have removed the rigid RAKF checking during the BREXX startup, which caused unnecessary ABENDS for non-authorized users (e.g. HERC03, HERC04). Some of the BREXX functions which require access to system resources (SVC244, DIAGCMD) are no longer available to non-authorized users, they will be reported as unknown functions.

Matrix and Integer Arrays

Added are mathematical Matrix functions and integer arrays. Both allow high-performance access and large-sized matrices and integer arrays outside the standard stem notation.

About

BREXX has been developed and supported by Vasilis.Vlachoudis@cern.ch

26.1 BREXX/370

BREXX/370 is a ported version of BREXX to IBM's operation system MVS 3.8j. Jason Winter and Jürgen Winkelmann ported BREXX initially to MVS3.8j. The BREXX/370 releases have been created and are supported by the BREXX/370 team: Mike Grossmann and Peter Jacob.

26.2 License

BREXX is licensed under the GNU General Public License v2.0. See https://github.com/vlachoudis-/brexx/blob/master/COPYING All rules mentioned above apply to BREXX/370!

26.3 BREXX/370 documentation

The essential BREXX documentation applies to BREXX/370: https://ftp.gwdg.de/pub/languages/rexx/brexx/html/rx.html

Please install according to the Installation Guide.

26.4 DISCLAIMER

THE SOFTWARE REFERENCED IS MADE AVAILABLE AS - IS. THE AUTHOR MAKES NO WARRANTY ABOUT THE SOFTWARE AND ITS CONFORMITY TO ANY APPLICATION. THE AUTHOR IS NOT RESPONSIBLE FOR ANY DAMAGE, LOSS OF DATA, OR LOSS OF MONEY CAUSED BY THIS PROGRAM.

- genindex
- modindex
- search

This user's guide documents the BREXX standard functions from https://ftp.gwdg.de/pub/languages/rexx/brexx/html/rx.html as well as the changes and amendments to BREXX to be used on MVS 3.8j.

27 Credits

- BREXX has been developed by Vasilis Vlachoudis, who made it publicly available as freeware for non-commercial purposes.
- Jason Winter's JCC Compiler for compiled BREXX
- JCC and the JCC-Library are owned and maintained by him. While not being freeware, Jason allows non-commercial usage and distribution of Software created using JCC through a relaxed license, as long as the complete source code always accompanies those distributions.
- Vasilis and Jason explicitly consented to make the JCC based version of BREXX available on TK4-. Thanks to both for their significant valuable contribution to the TK4- MVS 3.8j Tur(n)key system.
- The VSAM Interface is based on Steve Scott's VSAM API.
- The FSS Part is based on Tommy sprinkle's FSS - TSO Full-Screen Services
- Daniel Gaeta contributed his EXECIO implementation.
- The NJE38DIR load module was extracted out of Bob Polmanter's NJE38 V2 modules

We wish to thank the following persons for patiently answering our questions and for their support and advice:

- Vasilis Vlachoudis
- Jürgen Winkelmann
- Jason Winter
- Wally Mclaughlin
- Greg Price
- Bob Polmanter
- Steve Scott

and many others!

Chapter 28

BREXX/370 Source Code

The BREXX/370 Source Code can be found and downloaded at: https://github.com/mvslovers/brexx370/

29 Some Notes on BREXX Arithmetic Operations

BREXX stores numeric values in the appropriate type format. The benefit compared to save it as strings is a significant performance improvement during calculations. As the expensive string to numeric conversion before and vice versa after arithmetic operations is omitted; this allows speedy calculations without the required conversion overhead.

BREXX supports two numeric types:

- **Integer** Integers are stored in 4-bytes a full word (LONG), this means their range is from -2,147,483,648 to +2,147,483,647
- **Decimal Numbers** Decimal Numbers (decimal numbers with a fractional part) are represented in the double-precision floating-point format (doubleword), the length is 8-bytes consisting of an exponent and the significand (fraction). It consists of 56 bits for the fraction part, 7-bit exponent and one-bit for the sign. This representation is IBM specific and differs slightly from the IIEE 754 floating-point standard.
  The precision of floating-point numbers is not as good as decimal packed numbers which are not supported in BREXX (nor in REXX). This means, for example, 2.0 might be stored as 19999999999999999e-17, or for 5.0 you will is stored as 50000000000000003e-17; this is not an error, but the usual behaviour for floating-point numbers. It is caused by the conversion between the numbers of base 10 to base two a bit-exact reversibility is not always given. This effect may build up during arithmetic calculations.

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

T

U

V

W

X