RXS Scripting Language

0. Scope of RXS

0a. RXS is a programming language: Programming the nodes

RXS has inherited its basic syntax from REXX. (REXX is a scripting language from IBM). That is, the structure of the 'if'-statements, the 'do'-loops etc. in RXS is specified in REXX syntax. Nothing special about that - and the peculiarities in the REXX syntax are also inherited in RXS:

- REXX uses '&' instead of 'and' and '|' instead of 'or'. So a typical statement in REXX could be "`if A=14 & C=15 then do`"
- REXX uses 'say' instead of 'write'

So an *REXX* program could be

```
if a = 14 & b = 15 then do
     say 'all is well'
end
else do
     say 'not ok'
end
```

REXX is a language with a strong function package, and this is also inherited by RXS.

RXS uses a subset of REXX: selections, iterations and sequences. Input and output is programmed at a meta-level, and internal procedure calls are not used. So: RXS is a simple programming language with a strong function package.

To program in RXS, you put some coding inside delimiters `)action` and `)endaction`. This means that RXS-programs can be situated inside coding in other languages, that is: inside textfiles. RXS programs are executable regardless of where they are situated. Every textfile is a RXS program: RXS will execute the text, that is copying all of its line to output and reacting specially only on any `)action )endaction` blocks in the text.

A RXS program could be

```
)action
     a=13
     b=15
     if a = 14 & b = 15 then do
          'all is well'
     end
     else do
          'not ok'
     end
)endaction
```

The example introduces a key feature of RXS: Any line in the coding describing a string (a quoted line, normally), or any line which evaluates to a string after execution, is a description of output: that is: a description of the lines to be collected in a file (REXX has the same feature, but in REXX output in this sense is never written to a file). This has some importance: The goal of any coding in any programming language is to produce output, by reacting on input. RXS has a stronger focus on this than other programming language because output in RXS is default. This means that when reading a RXS program you will see the output to be produced only slightly dimmed by the footprint of the coding:

```
)action
     if a = 14 then do
          "Alas my love, you do me wrong,"
          "To cast me off, discourteously."
     end
     else do
          "One bottle of beer on the wall."
```

```
                "Take it down and pass it around."
            end
        )endaction
```

This is essential when using RXS for generation of code. Code generation means that you have to see the logic of the generated code while building the logic to generate the code. Otherwise stated: You have to see a clear view of the output while looking at the coding to produce the output.

Below is a non-trivial example of RXS programming: Calculate the mathematical constant PI (3.1415....) with 70 decimals:

```
        )action
            numeric digits 70
            pi = 0
            s = 16
            r = 4
            v = 5
            vs = v * v
            g = 239
            gs = g * g
            do n = 1 by 2
               pi = pi + s / (n * v) - r / (n * g)
               if pi = old then leave
               s = -s
               r = -r
               v = v * vs
               g = g * gs
               old = pi
            end
            pi
        )endaction
```

The example shows that RXS can be used for testing algorithms. Just like REXX - the differences between RXS and REXX here are small: The RXS program can be written and executed everywhere in the ISPF environment on the mainframe, and specifying output is simpler than in REXX.

0b. RXS is a pipeline-language, using named pipes to connect nodes

The pipeline is a fundamental concept in UNIX. Example:

```
ls | grep key | more
```

is an example of an UNIX pipeline. *'ls'* creates a list of the files on the current directory, *'greb key'* filters strings containing the string *'key'* and *'more'* presents the resulting file on the screen. The smart thing about pipelines is that the output of the first command is input to the second command and so on. You specify 'the nodes' that is the workstations transforming input to output, and just connect them with the pipe symbol, which is the '`|`'.

This enables the programmer to specify a complex flow using very little writing. But of course this pipeline-syntax has its limitations:

1. What if one node creates two or several output-pipes?
2. What if one node uses more than one input-pipe?
3. What if you need to put some custom coding inside a node?
4. What if you need a more complex topology of the piping than the straight linear flow down the connected pipes?

Different approaches have been tried on these questions, trading between the simplicity of the original pipeline concept and the more complex syntax of a more flexible solution. The result of these efforts has been that the original pipeline concept is alive and well, and the more complex pipeline-topologies (like 'Hartmann pipes') are not much used: Simplicity is important.

RXS opens this discussion again, assuming that a pipeline language which addresses all the four bullets above might qualify as a *general* programming language: RXS is not trying to stand up as a better pipeline language than the original UNIX pipes, but trying to use the pipeline concept as the defining concept for a general programming language. To accomplish this, RXS opens for a topology with nodes inside nodes, and introduces 'named' pipes for connecting of the nodes.

```
)action address='unix'
)&        out='filelist'
     'ls'
)endaction
)action in='filelist'
     if pos('key', unit.1) > 0 then do
          )action outfile='withkey'
               unit.1
          )endaction
     end
     else do
          )action outfile='nokey'
               unit.1
          )endaction
     end
)endaction
```

This RXS program lists the current UNIX directory and splits the list in two: File-names containing the string *'key'* and the rest. The two created files are presented on the screen using ispf edit.

The `)action )endaction` blocks defines the nodes. The nodes are connected with named pipes, using the variable `out` for naming the output pipe and in for naming the input pipe, and thereby enabling any topology of a network of pipes and nodes.

The nodes contain coding. This coding might be addressed to any sub-system on the mainframe, for instance UNIX: the starting point of the pipes in both examples is the UNIX 'ls' command.

If the node has an input pipe, then the coding in the node is executed once for every 'unit' delivered from the input pipe. If the node has no input pipe, the node is executed once.

The above discussion might indicate that RXS has a tight connection to UNIX on mainframe. It has not. For RXS, UNIX is just a data source among other.

A minimalistic RXS program or 'RXS node' is a node containing one line

```
)action in='r2d2.c.txt(hovsa)'
  unit.1
)endaction
```

It may not seem obvious that this one word is a complete program written in a procedural language. But so it is:

- The coding in the node is triggered every time the RXS interpreter delivers one unit from the input pipe. This delivery could be the next record or a line in a file, depending on the kind of input pointed to in the)action-line.
- The unit of input is normally named `unit.1`, so `unit.1` is a symbol containing the record just read. The RXS interpreter will substitute any symbols in the program with their current content during execution.
- So `unit.1` is a line in the coding that evaluates to a string when executed. Default handling of strings in RXS is to write then to current output.

In all: The program will copy all lines from the input-file to the output-file. Since the output-file is not indicated, the file `rxs.data` will be created containing the output.

0c. RXS pipes may connect nodes to any data store

RXS programs executes under TSO/ISPF on the mainframe. RXS programs are able to read all the data stores the mainframe is able to address - a pipeline in RXS can be any file or any other data structure.

```
)action in='O:\AcmeCorp\HR\Employee of the Month\October'
)& ||     ' 2013\Nominees.doc'
)&        pc='r2d2'
     unit.1
)endaction
```

This RXS program reads the indicated local network file and lists it on the mainframe. The indicated PC (indicated by its symbolic name or IP-address) is used for transportation.

A RXS program may accordingly write files to the local network.

DB2 and MQSeries are other kinds of data stores which RXS is able to read (and write).

0d. RXS programs may contain any number of nodes and in any topology

An action block may contain another action block: An action block is syntactically like any other chunk of coding in the RXS program. That is, it may be conditioned by an *if*-statement etc. The example in section *0b* above demonstrates this. This means that any topology (any web of nodes connected by pipes) may be created in a RXS program: a hierarchy, a sequence and any combination of the two. Output from one action-block may serve as input for one or several action blocks down the chain. (...Or down the drain: There is a similarity between pipe-line programming and plumbing work). This makes RXS a complete language: The topology of the problem to be programmed can always be matched by a topology of action-blocks, thereby solving the problem.

A RXS program may consist of a sequence of action-blocks not contained in some master action-block. This is syntactically ok, and the action-blocks are executed in sequence, consuming their pipes and creating their output-pipes, and when done, the next action block down the chain or drain will be activated. Any text between the action blocks: that is, text not contained in action-blocks - will by copied unaltered to output.

0e. Binary-tree-searching in pipes is supported by RXS

A common way to use two input pipes to create one output pipe is to use binary search, making one of the input pipes accessible by key. This important feature for making complex topologies is supported by RXS. (Instructions `queuevar` and `getqueue`).

0f. RXS nodes has a number of pre-screening services when reading pipes

Pre-screening input using '`func`' is used when the program needs to 'see' input in some special way. Example: input must be sorted, or input must be transformed from one encoding to another. Or input is to be seen as XML, or to be seen as SQL. This is specified in RXS using a '`func`' modifying or interpreting input. So transformations of input using rules applied to each input-unit (unit.1) are programmed in the coding inside the `)action` block, while transformations which sees the input as a whole are not programmed, but specified as a parameter ('`func`') at the `)action`-level.

Example: `func='sorted'` will sort the units of input in ascending order. Sorting descending, and sorting on some restricted sort-field is also possible.

```
)action in='ourgroup.thisdata'
)&        func='sorted'
     /* some coding here */
)endaction
```

Example: `func='mqbrowse'` is assuming that `in` is naming an MQSeries-queue on the mainframe, and will undertake a non-destructive reading of messages in this MQ-queue.

Example: `func='sql'` will see input as some SQL, and will undertake the specified reading (or update or anything) on DB2 to create input to the action block. RXS has a tight connection to SQL, compared with other languages - accordingly it is relatively easy to mix SQL coding and RXS coding.

0g. Adressing other subsystem on the mainframe: The address concept

If *output* from an action-block in RXS is to be seen as anything other than plain textual output, the action-block may 'address' output to some subsystem on the mainframe, thus putting some special interpretation upon the output. Example: Specifying `address='unix'` will assume that output from this action-block is UNIX-commands, and accordingly try to execute them against the UNIX-system on the mainframe.

0h. Final disposal of output: the outfunc concept

If output from an action-block is to be seen as plain text (that is, if '*address'* is not used), then the standard addressing in RXS handles it: Default in RXS is handling of output by addressing to *'stdout*'. *'Stdout'* has some default features, which may be modified using 'outfunc'. Example: `outfunc='browse'` changes the normal behaviour of *'stdout'* from 'edit' to 'browse' on the created output.

If an environment addressed by RXS - for instance *unix* - produces sequential output (say error messages or confirmations) this is also handled by *stdout*, just like any other output from an action block. So: *stdout* is the final handler of all sequential output from RXS.

0i. Unattended (batch) execution of RXS. Macro execution of RXS

Normally RXS program are executed in foreground on the mainframe: You are in an edit session on the mainframe having created a RXS program. You write RXS in the command-line and presses enter. The RXS program starts executing, and the user interfaced is blocked. When execution is finished, the edit session switches to a view of the output from the RXS program. When pressing 'end' (F3 in ISPF) all output files are viewed in a sequence. In case of errors, instead the line in the program in error will be high-lighted with an error message and all changes and all output made op to the error point will be rolled back.

This normal picture may be altered:

- A RXS program may be catalogued in a file allocated to RXSLIB on the ISPF-session, thereby enabling RXS programs to be executed like REXX-programs and TSO commands.
- A RXS program may be executed unattended in background. That is: activated from JCL
- A RXS program may function as a 'macro' by using the current file in the edit session as a input source or a pipeline.

0j. Building user-dialogues in RXS

The pipeline concept of RXS includes the user (you) sitting in front of the 3270 screen on the workstation: The RXS program may set up a pipe requesting input from the user at any point in the program, by specifying `func='prompt'`. The user will then be prompted from a ISPF-panel which is generated by the RXS interpreter. Also a dialogue is generated: The user can go forward and backwards between these panels for correction of errors on previously displayed panels. So the coding in the RXS program just specifies some pipelines requesting user-input. The RXS interpreter generates panels and dialogues.

0k. Code generation using RXS

Code generation is a key feature of RXS, and code generation was the starting point and the key motivation for development of the RXS language. Code generation is nothing more than the ability to generate a text-file by combining the different chunks of text (chunks of 'coding') to be generated, conditionally governed by a set of rules; the chunks of text containing variables to be substituted. Code-generation is like any normal programming work: changing input to output, but the job is easier when using a language focused on straight specification of 'chunks of text' and having easy interfaces to all kind of input. As mentioned before: a language with a small footprint, making output clearly visible, is essential in code generation. Easy code generation is often essential: Being able to generate code can solve a lot of grave problems on a development project.

0l. Programming in RXS

The focus on pipelines in the RXS language encourages a programming style where logic is specified in pipelines more than in programmed logic. Example: you want to list all records in a mainframe file named `this.file` which contains any of the words 'sunshine' 'bicycle' or 'apple' as the first word of the record.

It can be programmed:

```
)action in='this.file'
     if word.1 = 'sunshine' | word.1 = 'bicycle' ,
               | word.1 = 'apple' then do
          unit.1
     end
)endaction
```

Or it can be programmed:

```
)action out='fine_words'
     'sunshine'
     'bicycle'
     'apple'
)endaction
)action in='this.file'
     if queuevar('fine_words', word.1) = 1 then do
        unit.1
     end
)endaction
```

The last version sees the problem as a match between two pipelines, which might be a clearer image of the problem. Thereby making the modification easier if the collection of 'fine words' is changed.

Another example: You want to concatenate some files in a new file called `salesall.data`, and therefore writes this program:

```
)action out='file_list'
     'sales.south'
     'sales.northwst'
     'sales.northeas'
     'sales.east'
     'sales.central'
)endaction
)action in='file_list'
     actual_file = unit.1
     )action in=actual_file
     )&       outfile='salesall'
          unit.1
     )endaction
)endaction
```

In RXS you might solve a problem by naming a list of the objects that the program is to act upon. While in a normal programming language like JAVA or COBOL you specify (and name)

a procedure able to do the action and then program a sequence of calls to this procedure, specifying the actual object to be acted upon in each of the calls.

RXS programs may use 'property-files' massively, like in the two examples above. Having the property-files inline in the coding paves for easy reading and changing of both the coding and the property-files.

RXS is the programming language a squirrel would like to use: When you stumble over some interesting information, you put it in a queue somewhere in your territory for later use, naming the queue to help finding the information later.

Being a scripting language, RXS is focused on quick and readable specification of a problem. RXS is not intended for programming the production system - but might be used for 'programming the programming' of parts of the production system, that is: code generation. Besides that, RXS is intended for the daily utilities needed to create and maintain and supplement the production system. RXS is more complicated than the original pipeline concept, but still with this simplistic idea as its basic, and RXS is radically simpler than production languages like JAVA or COBOL. This paves for an attitude of 'spend 5 minutes to program a utility' instead of manually solving 3 related problems separately. The threshold for when to program a utility is low when using RXS.

RXS is a mainframe language. This is true in a more subtle sense than just the choice of platform - RXS is a language for large projects with an internal handling of information in the development process. The idea of having a special language to assist the process of creating a system in another language - well, we are clearly in the realm of large systems.

1. Action blocks and dead text

A RXS program consists of one or more 'action blocks' containing coding in REXX syntax.

The action blocks are connected with each other and with the surrounding world through queues and files, thus implementing a 'pipes-and-nodes' pattern, the nodes being the action blocks containing the coding. Between the action block may reside 'dead' text: lines of text that are copied to final output file without any interpretation or altering. Using such 'dead' text is often essential in code generation: the job is to generate a complete program, but often parts of the program is not suited for code generation, but better written as is.

'The surrounding world' for RXS is all the common file formats and data store used on the mainframe: DB2, MQSeries, XML, COBOL-source, sequential and partitioned files, files on the mainframe UNIX-file system, and files on PC and LAN. Even the person sitting in front of the screen is - as seen from RXS - a source of data which could be tapped using a pipeline *(section 21*).

An *action block* is a part of the RXS program delimited by the two lines

```
)action
```

and

```
)endaction
```

In the wrapper for the coding, following the word `)action`, may be some assignments *('general orders'*), also in REXX syntax, describing the connection between the surrounding world and the coding: Which pipes leads to and from the coding, and how are these pipes consumed by the action block. Action block may be without general orders, in which case they are executed just once, consuming no external input. Otherwise an action block is triggered once per element (record or row or...) input to the action block.

Here is a trivial example: a RXS program consisting of 'dead' text only - no action blocks:

Example 1.1:

```
1.
What shall we do with the drunken sailor
What shall we do with the drunken sailor
What shall we do with the drunken sailor
Early in the morning

2.
Put him in the longboat till he's sober
Put him in the longboat till he's sober
Put him in the longboat till he's sober
Early in the morning
```

Notice that RXS coding (that is: what is to by consumed by the RXS interpeter) throughout this paper is in a coloured courier font. Dead text is coloured olive.

Writing the text in a dataset or member using ISPF edit, then writing `RXS` in the command field on the screen, and pressing *enter* will start the RXS interpreter, trying to see the text as a program. Because no part of the text is contained in an `)action )endaction` block, all lines will be copied unaltered to output.

Now, adding a couple of action blocks:

Example 1.2:

```
1.
)action
  do 3
    "What shall we do with the drunken sailor"
  end
)endaction
Early in the morning

2.
)action
  do 3
    "Put him in the longboat till he's sober"
  end
)endaction
Early in the morning
```

Output from example 1.2 is identical to output from example 1.1 The program is now a mixture of lines not contained in action blocks (dead text) (lines 1, 7, 8, 9 and 15), programmed logic (lines 3, 5, 11 and 13), and lines inside action blocks to be written to output (lines 4 and 12). Notice that no general orders are given in the)action lines: The above program is not to work *on* anything. Accordingly the action blocks are triggered only once.

How does the RXS interpreter see that a line inside an action block is to be written to output? RXS inherits and expands a core principle from the REXX language: Any line in the coding which is not executable is considered aimed at 'the addressed environment' and is sent to that environment. For RXS the 'adressed environment' is *output* (normally). A line in the coding contained inside quotes is not executable, and therefore is sent to output.

Accordingly, RXS is probably the only programming language not using som kind of 'write' command: Creating output is default.

Notice that execution of a RXS program is always top-down: Imagine some 'execution counter' sweeping down the program, line after line. When this execution counter hits a line of 'dead' text, the line is written to output. When the execution counter hits an action block, the whole block is read and then interpreted.

Besides the `)action )endaction` constructs, the following meta-structures exists in RXS:

`)text` `)endtext` (See *section 17*) - kind of action blocks containing only strings
`)imbed` (*section 31*) - indicating externally defined RXS coding
`)trigger` `)notrigger` (*section 12*) - sub structure inside action blocks: specifying reaction on empty input
`)interface` (*section 39*) - opening an interface into an internal queue in rxs
`)nop` (*section 03*) - doing nothing

2. Syntax inside action blocks

2a. Iterations

```
do i = 2 by 2 to 12 /* execute for i = 2, 4, 6, 8 ,10 ,12 */
     some-action
end

do forever /* execute until the command 'leave' is reached */
     if some-logical-evaluation-is-true then iterate /* start over */
     some-action
     if some-other-logical-evaluation-is-true then do
      leave /* leave this do loop now */
     end
end
```

2b. Evaluations

```
if some-logical-evaluation-is-true then do /* either or: */
     some-action
end
else do
     some-other-action
end

select /* evaluate or case structure: */
     when some-logical-evaluation-is-true then do
          some-action
     end
     when some-other-logical-evaluation-is-true then do
          some-other-action
     end
     otherwise do /* Note: 'otherwise' clause must always be stated */
          another-action
     end
end
```

RXS normally uses the syntax above: an if-statement is a separate line ending with '`then do`'. What is *to be done* is ended with an '`end`' in a separate line.

A short form may be used: `if some-logical-evalution-is-true then some-action`

2c. Operators

'some-logical-evaluation' means some expression containing an operator, that is '=', '<', '>' and the like. Example: `a < 14`. The operator <> (not equal) is valid in RXS.

Two or more *logical evaluations* may be combined into one using the logical operators '&' (and) '|' (or) and parenthesis. Example:

```
if (a > 14 & b >= 13) | c <> 12 then do
```

In Nordic countries, France, Germany, Austria and Italy, the 'or' operator is '!'. This oddity is enherited from REXX or more correctly, from EBCDIC.

This inconvience applies generally to RXS: Whenever the documentation specifies '|' then use '!' when in the Nordic countries, France, Germany, Austria or Italy.

2d. Variables

Names for *variables* in RXS must start with a letter; special characters are not allowed. Maximum length of a name is 250 characters. Underscore '_' is allowed, hyphen '-' is not allowed.

Names starting with the characters rx_ are reserved for internal use and cannot be used.

Variables in RXS are allocated automatically. Variables are *typeless*, the variable `a` may contain a number or an alphanumeric constant. But asking `if a > 14` will only evaluate to true or false if `a` is currently holding a numeric value. Otherwise execution is terminated in error.

Alphanumeric constants in RXS must be contained in quotes - single or double. Maximum length of a quoted string in an assignment is 256 characters. Internally in RXS, an alphanumeric string can hold up to 16 MB of data (see *section 2n* below).

Numeric constants in RXS hold up to 15 digits. Exponential notation is possible: `1.4E+02` is 1400.

Hexadecimal constants in RXS are in the form `"12AB"X`

A *'stem'* in RXS is an array. The form of a stem is *variabelname.number*. For example `mystem.14`, or `mystem.mynumbr`, provided that `mynumbr` is the name of a variable containing a number. (It is also accepted that `mynumbr` is not numeric).

The assignment `mystem. = ''` will initialize all possible values inside `mystem`.

2e. Active statements in RXS

'some-action' in the text above indicates one of the following active elements in RXS:

2f. Active statements: Assignments. Functions.

RXS uses assignments in the form: `a = 14` or `a = random()`. Here `random()` is an example of a function in RXS. RXS has a rich collection of functions, most of them dealing with strings - see *section 40*.

Assignments may contain arithmetic: multiply `a * b`, divide `a / b`, square `a ** 2`. And `a + b` and `a - b`. Parenthesis may alter the normal order of evaluation.

2g. Active statements: External calls

The call of an external function in RXS is in the form: `call 'mylib.mydsn(mypgm)'`

2h. Active statements: Instructions

An instruction in RXS is a command that alters the further proceedings of the program.

For example `iterate` (see *section 2a*) , `leave` (*section 2a*), `address` (see *section 4*), `exit` (see *section 2m* below). All instructions in RXS are listed in *section 39*.

2i. Active statement: Nested action blocks

An action block may be situated anywhere where an active statement is acceptable. This indicating that action blocks can be nested. Which doesn't make much sense for now: action blocks are just blocks of REXX coding. The purpose of this construct will be revealed in *section 18*.

2j. Active statements: Strings

A string may be situated any places where an active statement is acceptable. A string is a line surrounded by single or double quotes (plus some other situations which RXS classifies as strings: evaluations which transforms into strings after evaluation). In the case of a string, RXS will write the string to output. See *Section 4* for further discussion.

Example 2.1:

```
)action
   do i = 1 to 2
      i"."
      do 3
         if i = 1 then do
            "What shall we do with the drunken sailor"
         end
         else do
            "Put him in the longboat till he's sober"
         end
      end
      "Early in the morning"
      " "
   end
)endaction
```

Example 2.1 is a slightly more advanced programming of our current example: output will be as in example 1.1 Notice that lines in an action block may be intended - to increase readability.

The second line in the action block: `i"."` transforms into a string after evaluation. The evaluation results in a replacement of the variable `i` with its current value. The thereby created string will be written to output by RXS.

2k. Statements are separate lines: one statement per line

The above elements in RXS have to be written as separate lines in the action block. It is possible to continue a line: ending a line with ',' will concatenate the next line to this line. It is also possible to write several lines in one line, separating the parts by ';'.

Otherwise no constraints on layout exists: RXS statements in actions blocks can have any amount of leading 'white space', and can be written in any dataset.

2l. Comments

/* starts a comment, */ ends a comment.

2m. RXS compared to REXX

The syntax described above is equivalent to REXX syntax. It *is* REXX: RXS uses the interpreting engine supplied by REXX to interpret and execute coding inside action blocks. Accordingly even more advanced elements in the REXX language may be used in RXS - for instance REXX commands for handling input and output, and REXX commands for handling stacks and queues. But such advanced REXX elements are seldom used in RXS: the handling of text-blocks and other internal files inside the RXS program and the handling of files externally in the form of input and output is more easily handled using the basic machinery in RXS.

Two REXX constructs are not allowed in RXS:

- You may not call your own *internal* sub-routines ('call'). A RXS program is structured in action blocks, not in sub-routines.
- Goto constructs are not allowed (Go to is called *signal* in REXX, so therefore: signal is not allowed in RXS)

This implies that execution in a RXS program is always sequential: top down. An action block is executed when the execution hits the block; it cannot be 'called'.

Notice that call of *external* routines, for instance the calling of REXX or RXS programs, is supported by RXS (See *section 31*).

There are two differences between detailed functionality in RXS and REXX:

- `exit` and `return` has a strong side effect in RXS compared to REXX. Using them means that all output is rolled back (written lines are not written anyway) and any updates on DB2 and MQSeries from RXS are rolled back too. After that, the program stops. A RXS program is a *transaction*, and implements coherent commit / rollback of all resources of the program.
- In RXS it is strictly controlled that a variable is assigned a value before its content is used in a logical evaluation or in a string. You cannot ask `if a = 14 then...` if the variable `a` has not received a value at a prior point in the RXS program. If you ask anyway, the program will terminate in error, labelling the if-statement with a message: '*a has no value*'.

Finally: RXS differs from REXX by supplementing REXX with a (small) number of new functions and instructions. All relevant functions and instructions in RXS are listed in *Section 40*.

2n. "Program memory exhausted"

REXX variables have an implementation maximum: No single request for storage can exceed the fixed limit of 16 MB. This limit applies to the size of a variable plus any control information. It also applies to buffers obtained to hold numeric results.

The limit is often lower than 16 MB when running in *tso* (in foreground). It depends on the region size at logon. The maximum may by as low as 3 MB. The relation between region size and maximum length of a data structure in RXS is not obvious. If the limit is reached, execution halts with the error message "Program memory exhausted".

A file read or written from RXS cannot exceed about 1 GB. All external and internal files in RXS are kept in memory during execution of the program. Accordingly, RXS uses a lot of high memory during execution.

Running RXS in the background might be needed if the program is very data intensive. Here using REGION=0K in the JOB-card will maximise the amount of high memory. See *section 36*.

3.Write, execute and debug RXS programs

To execute a RXS program, write the program in any dataset on the mainframe using *ISPF edit*, write *rxs* in the command line in the edit screen (`==> rxs`) and press the *enter* key. No allocation of the dataset containing the program is necessary. It is not necessary to 'save' the program before execution: the RXS interpreter works as an ISPF-macro. The dataset may have COBOL numbers, these are ignored in the interpretation of the program, but output will be generated using COBOL numbers in this situation.

In case of errors, error messages from RXS will pop up over the line in error in the program.

ISPF command `==> hilite rexx` will *syntax colour* what is on the screen, and that can be very handy, especially because quotes are used quite intensive in RXS. This syntax colouring highlights unbalanced quotes.

RXS programs also may be executed as TSO commands. The prerequisite is that these RXS programs are members in a dataset allocated to the TSO session as file RXSLIB (*Section 35*).
RXS programs may also be executed from JCL (*Section 36*).
RXS programs must execute inside TSO + ISPF environment.

3a. Debugging

When debugging is needed, note the following:

- The RXS command `say xxx` is excellent for debugging. The command writes strings and variables on the screen during execution.
- Error messages from RXS always contains a header: "RXS error:", "REXX error:", "SQL error:" and so on. If some coding conflicts with REXX syntax rules, a REXX manual might be useful. Etc.
- `)interface in='q1'` will halt execution temporarily and will show actual content of internal queue 'q1' on the screen
- RXS encourages incremental programming: Write a couple of action blocks and execute. Build on when successful.
- Marking a block of lines with line commands 'cc' 'cc' will direct RXS to execute the marked block only

3b. To halt a RXS program

If execution of a RXS program has to be stopped, press the *'Esc'* button, which starts a termination dialogue. Now enter `hi` for 'halt immediate', or `he` for 'halt execution'. 'Halt execution' will kill any sub process from the RXS program as well - 'halt immediate' will not kill activities in SQL for instance, and not stopping such activities might block the RXS program from stopping.

3c. Upper case. Lower case.

RXS do not discriminate between upper and lower case.. `NUMB` and `numb` is the same variable name and `)action` and `)ACTION` works the same. Strings used in general orders don't discriminate either: `FUNC='SQL'` and `FUNC='sql'` will work identically. But for user variables case does matter: `a='B'` and `a='b'` are not identical assignments.

3d. COBOL line-numbers in RXS programs

RXS programs may be written in a dataset using COBOL line numbering (columns 1 thru 6 numeric). These line numbers are ignored (but influences the allocation of stdout - see *Section 32*)
Input datasets for RXS program having last qualifier of the name equal to 'COBOL', and having column 1 thru 6 numeric or blank, will be read ignoring column 1 thru 6.

3e. Line numbers in columns 73 thru 80

If columns 73 thru 80 are numeric, if column 72 is not entirely numeric, and if the record length of the file is 80, then columns 73 to 80 are ignored. This principle applies to both the RXS program itself and to input files used in the program.

3f. Continuation of lines

Any statement may be continued on the next line by ending the first line with a comma:

Example 33.1:

```
)action
   "Alas my love, you",
   "do me wrong"
```

```
   "to cast me off discour",
   ||"teously..."
)endaction
```

Output is:

```
Alas my love, you do me wrong
to cast me off discourteously...
```

If || (the concatenation operator) is written in front of the continued line, the two parts of the line are concatenated without an interleaving blank. Otherwise an interleaving blank is placed between the two parts of the line. In some european ebcdic (Nordic...) , use !! instead of ||.

3g. Indenting action blocks and text blocks

Blanks are allowed in front of `)action`, `)endaction` and the other RXS delimiters. Readability is augmented when intending is used for marking the logical nesting of blocks.

3h. Comments,)nop

Comments surrounded by `/*` and `*/` may be written anywhere inside action blocks - also in general orders.

Exception:

Any line starting with `)` is interpreted as a special RXS delimiter. Even if it is inside a comment block.

Outside action blocks (in text blocks and outside blocks) lines are not interpreted. Accordingly comments are not recognized as comments, but are written to output.

To place comments outside action blocks, use the special RXS marker `)nop`:

```
)nop "This is just a comment"
```

`)nop` lines are always ignored in the interpreting and do never qualify for output.

3i. Parameters for RXS programs

RXS programs may be executed using parameters. If in an IPSF edit session on a RXS program, you write `rxs what's up doc` in the command field on the edit screen, RXS will receive the string "what's up doc" in the RXS variable `rxsparm`.

The parameter is not allowed to start with characters | or `?`

4. Output from RXS

'Strings' in the RXS program - that is lines in the program not being commands, assignments, selections or iterations - are as a default handled over to an environment called *'stdout'*. It is possible to deviate from this default (*Section 7*).

The handling over of strings to *'stdout'* implies the strings are written to the current RXS dataset. Depending on the situation, this dataset is automatically created as RXS.DATA or RXS.COBOL. It is possible to deviate from this default (*Section 7*).

As default the RXS program terminates by bringing the current RXS dataset up in an edit session - providing that something is written in this dataset. It is possible to deviate from this default.

Example 4.1:

```
)action
   nbr = random() /* get a random number */
   'Square of' nbr 'is' nbr**2
)endaction
```

The third line in the program describes a string. Therefore this line will be written to the dataset <user>.rxs.data -where <user> is the actual TSO userid. The RXS program terminates by bringing this dataset up in ISPF edit.

Default environment for the receiving of strings (that is *stdout*) may be changed: By the general order `address...` strings may be sent to another environment or milieu. For instance `address'tso'` sends all following strings in the action block to TSO. Over here, the strings will be handled 'the tso way' that is: strings are seen as commands and are executed.

See *Section 32* for a more detailed discussion of output from RXS.

See *Section 33* for a more detailed discussion of the general order `address`.

5. General orders for action blocks

Putting the coding of RXS into action blocks offers the opportunity to specify what the coding is to work against: what is input, how to fundamentally interpret input, where to put output, and what to happen when the putting of output is finished. These kinds of specifications are called 'general orders'. *A 'general order' is an assignment placed in the line starting with the word* `)action`.

Some common general orders are:

`in`	what is input?
`func`	how to interpret input?
`outfile` or `out`	where to write output?
`outfunc`	how to treat output when terminating the RXS program.

General orders are in REXX syntax, that is assignments in the form `outfunc='edit'`. If the action block has more than one general order, the orders are written one per line, using the continuation marker for blocks which is `)&`. For instance:

```
)action out='mylib.mydsn(mymbr)'
)&      outfunc='browse'
```

Alternative: separate the general orders by one `';'` and write more than one general order in the same line. For instance:

```
)action out='mylib.mydsn(mymbr)';outfunc='browse'
```

If one general order is longer than what fits on the edit-screen, it may be split over more than one line, using the continuation marker `'||'` on the continuated line. For instance:

```
)action out='C:\ACME Cooporation\Management\Administration'
)& ||    '\Sales department\employees'
)& ||    '\empolyee of the month\october 2013.txt'
)&       outfunc='browse'
```

Lines continuated using the concatenation marker `'||'` on the continuated line will be concatenated without any interleaving blank.

General orders may be any executable statement in REXX syntax. Coding placed as general orders will execute when the action block 'starts up', that is, when the execution of the RXS program hits the top of the action block.

Again: imagine some 'execution counter' sweeping down the program, line after line. When this execution counter hits an action block, the general orders in the)action line is interpreted first and once, the rest of the action block is then read in and is interpreted repeatedly for each element in input for the action block. If the action block doesn't use input, the action block is interpreted once.

General orders are - when introduced first time - in the following sections written in typeset **`courier bold`**.

5a. General orders can be any coding

Anything may be programmed as general orders in the)action line. You may write

```
if a=4 then out='q1' else out='q2'
```

or

```
say 'ok - so far, no problems'
```

or whatever.

Exception: RXS block constructs cannot be used as general orders: you cannot inside a general order use `)action` and `)imbed` etc.

6. outfile='xx' Where to write output

`outfile`

General order `outfile` governs naming of the *stdout* dataset. As default, the middle qualifier in the name of this dataset is 'RXS'. Assigning a value to `outfile` as a general order will change this qualifier.

Example 6.1:

```
)action outfile='yrsa'
   nbr = random()
   'Square of' nbr 'is' nbr**2
)endaction
```

Writing will be done on the dataset '<user>.yrsa.data' where <user> is actual TSO user ident.

Default *stdout* dataset is automatically allocated by RXS. This applies as well if `outfile` has changed its name.

`outfile` is at maximum eight characters, and have to obey to IBM syntax for names, that is first character non numeric, and using no special characters.

7. out='xx' Where to write output

`out`

RXS may be conducted to write on any dataset.

As default - if `out` is not assigned a value - output is written to default *stdout* dataset according to section 5.

If general order `out` is assigned the name of an existing dataset, RXS will write to this dataset.

Dataset names must be fully qualified and stated in single or double quotes. (They have to be in quotes because RXS considers any non-quoted string as the name of a variable).

Example 7.1:

```
)action out='myuser.myout(hugo)'
   nbr = random()
   'Square of' nbr 'is' nbr**2
)endaction
```

Writing is done in member 'hugo' in dataset 'myuser.myout'

The dataset stated as `out` has to exist before the execution of the RXS program hits the action block. A member name stated as part of `out` may or may not exist. Writing will delete any prior content in the dataset or the member.

`out` may also denote an internal queue. This core feature of RXS is discussed in *section 11*.

The file concept in RXS is generalized. The table below lists the different file-types which RXS is able to write:

If `out` contains...	...and the file is...	then this happens...	example.... out contains:	section:
...at least one '.'	a mainframe file (FB, VB, U, spanned). (VSAM files are not supported in RXS)	the file is written	`'myuser.myoutput'`	
...at least one '.'	a MQSeries queue	the queue is written	`'our.reply.queue'`	27
...at least one '.' and one '(' and one ')'	a member in a partitioned mainframe file (FB, VB, U, Spanned)	the member is written	`'r2d2.c.text(yrsa)'`	
...at least one '/'	A UNIX (HFS) file on mainframe	the file is written	`'/home/r2d2/xx.txt'`	29
...at least one '\'	a PC file or local area network file	the file is written	`'c:\diverse\xx.txt'`	28
...none of these	an internal queue in RXS	the queue is written	`'very_bad_movies'`	11

7.1 Member statistics is always updated when writing a member

`zlcdate`	Current date. Formatted in the 'national' date format by ISPF
`zlmdate`	Current date in 'national' date format
`zlmtime`	Current time, hour and minute. tt:mm
`zlmsec`	Current time, seconds, ss
`zluser`	The constant 'RXS'
`zlcnorc`	The number of records written in the member. The maximum value is 65535, but actual number of written records have no limit.

8. outfunc='xx' What to do with output

`outfunc`

The *stdout* environment has a default way of handling the output dataset from RXS: The dataset will be shown in an edit screen - presuming that the RXS program has written something into the dataset. If the dataset is empty, nothing will happen.

If general order `outfunc` is assigned a value, this default for 'termination action' in the *stdout* environment is overruled. Following options exists:

- `outfunc='edit'` ISPF edit on output dataset (default)
- `outfunc='browse'` ISPF browse on output dataset
- `outfunc='view'` ISPF view on output dataset
- `outfunc='sub'` TSO submit of output dataset
- `outfunc='mqput'` Output is sent as MQSeries messages
- `outfunc='nop'` No action
- `outfunc=`*anything* Anything.

`Outfunc='sub'` prerequisites - for good results - that the RXS program has created JCL statements in the output dataset.

`Outfunc='mqput'` prerequsites that `out` is assigned the name af a MQSeries message queue residing on the queue manager which is defined as current for RXS. More information in *Section 27*.

`Outfunc='nop'` means that the RXS program just terminates after writing the dataset.

`Outfunc=`*anything*. Any TSO-command, CLIST / REXX may be used as `outfunc`. Such a command may use parameters. The command does not have to have any relationship to the output dataset. Example: `outfunc='t'` means that the actual time is written on the screen on the termination of the RXS program (`t` is a tso-command giving actual time). Observing the rules in *Section 27*, even a RXS program may be used as `outfunc`.

Example 8.1:

```
)action out='myuser.myout'
)&      outfunc='browse'
   nbr = random()
   'Square of' nbr 'is' nbr**2
)endaction
```

In this example both `out` and `outfunc` are general orders to the action block. Therefore they are wirinten on separate lines, using the contination marker for general oders: ')&' Alternatively bot order may be written in the same line separated by `;`. The execution of the RXS program results in the string 'Square of 25 is 625' (presuming the random number happens to be 25) being written to the dataset 'myuser.myout' and this dataset is finally presented on screen using ISPF browse.

9. in='xx' Input for action blocks

`in`

An action block may read an input dataset. General order `in` is to be assigned the name of such a dataset.

In very general, the action scheme in RXS is the following: Each record or unit (message, row, element...) in input will trigger the action block once. As default, input is seen as a collection of records. That is, if an input dataset contains four records, the action block will be executed four times.

As default, data from actual input record can be accessed inside the action block by the following variables:

`unit.1` The variable (which is a *stem* with one element) `unit.1` is assigned the value of the actual input record

`word.x` The 'words' of the actual input record is put into variables (*stem*) `word.1`, `word.2`, `word.3`,`...` depending on the number of words in the actual record. By 'word' is meant a string of non-blank characters. Say actual records contain three words, then `word.4`, `word.5` etc. will be assigned with strings of length zero.

(A *'stem'* in RXS is - as mentioned in *section 2* - an array or a one-dimensional table).

Example 9.1:

```
)action in='myuser.myinput'
)&      out='myuser.myout'
   'Square of' word.1 'is' word.1**2
)endaction
```

If the first 'word' of each record in the dataset 'myuser.myinput' is numeric, RXS creates a stream of linies in output clarifying how these input numerics are squared. If any record exists in input having a non numeric first 'word', RXS terminates with an error message and nothing is written.

Reading datasets is the default form of input to action blocks. In *section 19* is discussed how to access other kinds of input like DB2, MQSeries etc.
`in` may also denote an internal queue. This core feature of RXS is discussed in *section 11*.

`readlim`
If only a part of the file is to be read, then specify readlim

Example 9.2:

```
)action in='myuser.myinput'
)&      readlim=10
   'Square of' word.1 'is' word.1**2
)endaction
```

This works like example 8.1, but only the first 10 records of 'myuser.myinput' is read. Output is written to stdout (see section 4)

`readfrst`
To skip reading of the start of the, then specify readfrst.

Example 9.3:

```
)action in='myuser.myinput'
)&      readfrst=11
   'Square of' word.1 'is' word.1**2
)endaction
```

This works like example 8.1, but the first 10 records of 'myuser.myinput' is skipped. Reading starts on record number 11.

The file concept in RXS is generalized. The table below lists the different file-types which RXS is able to read:

If `in` contains...	...and the file is...	then this happens...	example... in contains:	section:
...at least one '.'	a mainframe file (FB, VB, U, spanned). (VSAM files are not supported in RXS)	the file is read	`'myuser.myinput'`	
anything compatible to MQ naming standards	a MQSeries queue	the queue is read	`'our.xx_queue_'`	27
...at least one '.'	a partitioned mainframe file (FB, VB, spanned).	a member list is created	`'myuser.cntl'`	9.1
...at least one '.' and one '(' and one ')'	a member in a partitioned mainframe file (FB, VB, U, Spanned)	the member is read	`'r2d2.c.text(yrsa)'`	
... one or several '*'	a search argument for creating a list of mainframe files	a list of files is created	`'r2d2.c*.*'`	9.2
...at least one '/'	A UNIX (HFS) file on mainframe	the file is read	`'/home/r2d2/xx.txt'`	29
...at least one '/'	A UNIX (HFS) directory on mainframe	a list of files is created	`'/home/r2d2/'`	29
...at least one '\'	a PC file or local area network file	the file is read	`'c:\diverse\xx.txt'`	28
...none of these	an internal queue in RXS	the queue is read	`'very_bad_movies'`	11

9.1 Reading a member

If `in` names a partitioned mainframe file and a member name - example in='r2d2.c.text(yrsa)' - the member is read, and variables zlcdate, zlmdate, zlmtime, zlmsec, zluser are given values according to the description in section 9.2:

9.2 Reading a member list

If `in` names a partitioned mainframe file, and `in` does not contain a member name, then unit.1 will form a list of the members in the partitioned file - one member name per triggering of the action block. In the triggering, the *ISPF member statistics* will be available in variables:

`zlcdate`	Creation date for the member. Formatted in the 'national' date format by ISPF
`zlmdate`	Modification date in 'national' date format
`zlmtime`	Modification time, hour and minute. tt:mm
`zlmsec`	Modifikation time, seconds, ss
`zluser`	User ident for the user who did the latest modification of the member

`unit.2` will contain 'MEM' as a hint that a member list is being communicated.

9.3 Reading a generic list of mainframe files

If `in` points to a generic name for a mainframe file list, then unit.1 will form a list of the files adhering to the generic name - one file name per triggering of the action block.

A *"generic name for at mainframe file list"* is a file name containing one or several placeholders: '*'. Each '*' means that characters in the file name up to the next '.' or up to the end of the name is ignored, that is: any value here will qualify for adding the file-name to the list.

`unit.2` will contain 'FIL' as a hint that a file list is being communicated.

9.4 Reading a UNIX directory

To read a UNIX file, you have to specify the path and the file name in one string.

Specifying only a path in `in` results in a reading of the directory, that is, a list of files and directories in the actual directory.

`unit.2` will contain 'FIL' of 'DIR' as a hint of whether the current `unit.1` is a file name or a directory name.

10. infile='xx' Input for action blocks

`infile`

General order `infile` may be used instead of general order `in`. The variable `infile` points to a file named <user>.xxx.data, where <user> is tso userident and xxx is the content of infile.

Example 8.2:

```
)action infile='myinput'
)&      out='myuser.myout'
   'Square of' word.1 'is' word.1**2
)endaction
```

The action block will read the dataset 'R2D2.MYINPUT.DATA' provided that tso userident is R2D2.

The variable `infile` is primarily used when executing RXS in the background via JCL. See *section 36*.

11. Internal queues as input and output

- If general order `in` or `out` is assigned not a dataset name but a name - that is: a string obeying the RXS rules for elementary names - it is a reference to an internal queue in RXS.

Example 11.1:

```
)action out='myqueue'
   9
   25
   121
   2
)endaction
)action in='myqueue'
  'Square of' word.1 'is' word.1**2
)endaction
```

Output will be the following lines in standard output: <user>.RXS.DATA:

```
Square of 9 is 81
Square of 25 is 625
Square of 121 is 14641
Square of 2 is 4
```

Notice:

- RXS programs may consist of several action blocks. These are executed in sequence, that is top-down.

  The first action block in example 9.1 writes to the queue 'myqueue'. The action-block uses no input, accordingly it is executed only once.

  The second action block uses the queue 'myqueue' as input, accordingly the block is executed once for each element in 'myqueue'. General order 'out' is not assigned any value in the second action block, accordingly this block writes to standard output: the dataset <user>.RXS.DATA.

- Queues are global: any succeeding action block in the RXS program may read a queue created in a preceding action block.
- A RXS program may read and write an unlimited number of datasets, members and queues. A RXS program may consist of an unlimited number of action blocks. Two or more action blocks may write to the same queue, dataset or member - the succeeding action block will in this case extend what the first action block has written. Two or more action blocks may read the same queue, dataset or member. Reading from an action block is always from beginning to end, regardless of any concurrent reading from other action blocks.
- Unlike records in MVS files, which contains up to a maximum of 32.760 byte of data, 'records' in queues may contain up to 16 MB of data.
- Assigning a value to general order `outfunc` has no meaning when the action block writes to a queue. The assignment will be ignored.

Example 11.2:

```
)action out='compressed_lyrics'
   "What shall we do with the drunken sailor"
   "Put him in the longboat till he's sober"
   "Pull out the plug and wet him all over"
   "Put him in the scuppers with a hose-pipe on him"
   "Heave him by the leg in a running bow-lin"
)endaction
)action in='compressed_lyrics'
)&      i=0
   i = i + 1
   i"."
```

```
   do 3
      unit.1
   end
   "Early in the morning"
   " "
)endaction
```

Line 9 in this example shows that the `)action` line itself may contain any RXS coding. This coding will be executed once as execution hits the action block. Therefore the `)action` line is well suited for initialization of counters. In this example the counter `i` is initialized.

Output will be like example 1.1, here adding three more verses to the lyrics.

12.)trigger and)notrigger; Reacting on empty input

Example 12.1:

```
)action out='myqueue'
   wcount = random(0,5)
   do wcount   /* loop is executed 0 to 5 times */
      random()
   end
)endaction
)action in='myqueue'
)trigger
   'Square of' word.1 'is' word.1**2
)notrigger
   'Sorry, this time no numbers were produced'
)endaction
```

In this example the first action block produces a number of elements in the queue 'myqueue'. The number is between zero and five. If by random the number is zero, the second action cannot be executed - normally resulting in an error message from RXS. "queue 'myqueue' does not exist". Using `)trigger` and `)notrigger` catches this situation:

If an action block reads from an empty (that is: non-existing) queue, RXS will terminate in error. But this exception may be catched: executing may continue in another branch of the program. This is accomplished by parting the interior of the action block in two parts via the headings `)trigger` and `)notrigger`. The `)notrigger` part *is only executed if input is empty*, while the `)trigger` part *is executed for each record in input*.

The condition 'is empty' is true if any of these conditions do describe input:

(a) input is a queue which does not exists
(b) input is a member (in a dataset) which does not exist
(c) input is a member (in a dataset) which is empty
(d) input is a dataset which does not exists
(e) input is a dataset which is empty

If the `)notrigger` part of the action block is not programmed, situation (*a*) and (*d*) results in termination in error, while situation (*b*), (*c*) and (*e*) results in nothing: no execution of the action block, and no termination in error.

RXS does not distinguish between an empty versus a non-existing queue.

13. cont: Flagging last record of input

If output from an action block is to be in the form of a list: some elements separated by comma, then the RXS variable `cont` may be used.

`cont` Contains the value `','` each time the action block is executed, except the last time, where `cont` is assigned the value `' '`.

Other programming tasks than the production of lists may also benefit from the flag `cont` signalling 'now last triggering' of the action block.

RXS contains no similar flag for first record. Needing to flag this, some `start_switch` may be initialized in the general orders. General orders are executed once, just before the first executing of the action block.

Example 13.1:

```
)action out='myqueue'
   9
   12
   121
   2
)endaction
)action in='myqueue'
)&      start=1
   if start = 1 then do
      "Here is the resulting list:"
      "("
      start = 0
   end
   word.1||cont
   if cont = "" then do
      ")"
      "That's all"
   end
)endaction
```

Output is:

```
Here is the resulting list:
(
9,
12,
121,
2
)
That's all
```

`||` is the concatenating operator in RXS, meaning consecutive writing of the strings before and after the operator.

Another way of concatenating strings in RXS is to write an empty string between the two strings. That is (referring to the example above):

```
word.1""cont
```

In the line above, `""` is the empty string.

This last solution may create a conflict with RXS' notation for hexadecimal strings: "12AB"x means the hexadecimal string 12AB.

14. queuevar: Joining data from two queues

The RXS function `queuevar('queue_name','queue_element')` returns `'1'` if `queue_element` does exist in the queue `queue_name`, otherwise `'0'` is returned.

If the queue `queue_name` does not exist, then `'0'` is returned.

Example 14.1:

```
)action out='q1'
```

```
   9
   121
   25
   2
)endaction
)action out='q2'
   7
   13
   25
   9
)endaction
)action in='q1'
   if queuevar('q2', word.1) = 1 then do
     word.1
   end
)endaction
```

Output is

```
9
25
```

That is: elements existing in both queues.

15. getqueue: Keyed data from queues

If you put some input record into a RXS queue holding a comma - outside quotes - the thereby established two parts of he record will be saved separately. In reading of the RXS queue, the first part is assigned to variable `unit.1` and the second part is assigned to variable `unit.2`. As mentioned before, `unit.1` is further split into variables `word.1`, `word.2`, `word.3` etc. `unit.2` may also be accessed by the command `getqueue`:

The RXS function `getqueue('queue_name', 'element_value')` makes a search through the queue `'queue_name'` to find a element where `unit.1` holds the value `element_value`. If the search is successful, the function returns the value of `unit.2`. Otherwise: if the element is not found, if the found element does not hold a value for unit.2 or if the queue `'queue_name'` is not found, getqueue returns an empty string (a string of length zero).

Example 15.1:

```
)action out='q1'
 'Yrsa Nielsen', 'bike mechanic'
 'Hugo Jensen', 'account manager'
 'Niels Olsen', 'brazier'
)endaction
)action in='q1' /*Example of accessing unit.1 & unit.2 in a queue: */
 unit.1 'is' unit.2
)endaction
)action  /* Example of accessing unit.2 in a queue via getqueue: */
 ' '
 'What position has Hugo Jensen: '
 getqueue('q1', 'Hugo Jensen')
)endaction
```

Ouput is:

```
Yrsa Nielsen is bike mechanic
Hugo Jensen is account manager
Niels Olsen is brazier

What position has Hugo Jensen:
account manager
```

The line `getqueue('q1', 'Hugo Jensen')` in the example, is a function call. The result of the function call is a string. And strings in RXS are written out to stdout.

16. dropqueue: Dropping a queue

The instruction `dropqueue` removes a queue:

```
dropqueue 'queue_name'
```

When this instruction is executed, the queue `queue_name` turns non-existing. If an action block tries to read the queue `'queue_name'` the RXS program will terminate in error - unless a `)notrigger` clause catches the exception (see *section 12*).

`dropqueue` is relevant when an action block wants to create a new content in a queue which is formerly created by an action block.

17.)text block

Lines in the program surrounded by lines `)text` *and* `)endtext` *constitutes a text block.*

Text blocks are not interpreted by RXS. The lines in the text block are written unaltered to *stdout*.

A text block may contain action blocks and these will be interpreted - see *Section 18.*

Text blocks may have general orders. A general order for at text block is an assignment in the line starting with `)text`.

If the general order `in` is stated for a text block, RXS is terminated in error: Text blocks cannot use input files or input queues.

In all other respect, general orders for text blocks are like general orders for action blocks.

See *section 18* for an example.

18. Mixing)text and)action blocks

Example 18.1:

```
)text out='q1'
   2
   4
   6
)endtext
)action in='q1'
   hideword = word.1
   if word.1 < 6 then do
      )action in='q1'
         hideword "multiply by" word.1 "is "word.1 * hideword
      )endaction
   end
)endaction
```

Output is:

```
2 multiply by 2 is 4
2 multiply by 4 is 8
2 multiply by 6 is 12
4 multiply by 2 is 8
4 multiply by 4 is 16
```

```
4 multiply by 6 is 24
```

In this example it is necessary to move `word.1` to the user defined variable `hideword`. This is done in the outer action block - to prevent overwriting of the variable when `word.1` is assigned values in the inner action block during the run of the RXS program.

Notice:

- An action block may contain another action block: An action block is syntactically like any other chunk of coding in the RXS program. That is, it may be conditioned by an `if`-statement etc.
- An action block may reside into an action block, which may reside inside an action block - etc.
- Action blocks may contain text blocks and text blocks may contain action blocks - in any combination.
- Accordingly `)trigger` and `)notrigger` parts of an action block may contain any hierarchy of action blocks and text blocks
- General orders for an action block are executed once when execution hits the block (In the example above, the general orders for the inner action block are executed twice because the flow of execution in the RXS program will hit this action block twice)

The principle of the 'execution counter' sweeping down the program, activating line after line, top to bottom, now becomes a bit complicated: When hitting an action block inside another action block, the outer action block is halted while the inner action block is interpreted consuming its own input. When finished, the outer action block turns active again.

If an action block tries to consume a RXS queue which is not yet produced, execution ends in error (more specific rules in *section 12*).

Assigning variables instead of values to general orders can be relevant:

Example 33.8:

```
)text out='q1'
   a1
   a2
   a3
)endtext
)action in='q1'
   )action out='myqualif.mydsn('word.1')'
      "What's up doc?"
   )endaction
)endaction
```

This RXS program will create (or replace) three members, A1, A2, and A3, in the dataset 'myqualif.mydsn'. All three members will contain a line with the text "What's up doc?"

19. func: Special interpretation of input

`func`

Not all data sources are simple collections of records and words. For instance, a DB2 table contains rows and named variables, and these are to be read as result sets, governed by some SQL. Combining this with RXS, it would probably be beneficial if the action block were triggered every time DB2 fetches a row. To accomplish this, state general order `func='sql'`, and follow the instructions in *Section 20* to write the SQL to govern the action.

Here is the list of values for `func`, constituting the list of data sources RXS can read:

- `sql` - *Section 20*

- `prompt` - *Section 21*
- `dcl` - *Section 23*
- `namespace` - *Section 24*
- `xml` - *Section 25*
- `sorted` and `sorted_desc` - *Section 26*
- `mqbrowse`, `mqdrain` and `mqdrainkey` - *Section 27*
- `binary` - *Section 28 and 29*
- `<utf8` - *Section 30*
- `>utf8` - *Section 30*
- `<ascii` - *Section 30*
- `>ascii` - *Section 30*

Default - when `func` is not given a value in the general orders for the action block - is to interpret input as described in *Section 9*.
Whatever value `func` is given, the result is the triggering of the action block by a number of instances of some set of data. The variable `cont` is implemented the same way in all cases: `cont` holds the value `','` in all triggering of the action block except the last one, where cont holds the value `' '`.

20. Func='sql' Accessing DB2

General order `func='sql'` means that input to the action block will be interpreted as SQL. That is, the lines of this input do not trigger the action block; instead the action block is triggered by the rows produced when executing the lines of the input as SQL against a DB2-system.

20a. General order insql

`insql`

Func=’sql’ is the most common interpretation in RXS, so a short form exists: Instead of writing

```
)action in='q1'
)&      func='sql'
```

you may write:

```
)action insql='q1'
```

this way indicating that the content of queue ‘q1’ is to be interpreted as SQL. This short form is used in the examples below.

Example 20.1:

```
)text out='q1'
   select account, name from myqualif.mytable
   where department = :w_department
)endtext
)action insql='q1'
)&      w_department='SALES'
   name" has account number "account
)endaction
```

Output will be something like:

```
Peter has account number 45476
Hugo has account number 32243
Yrsa has account number 11223
```

- Presuming that these three employees are running the SALES department according to the DB2 table *myqualif.mytable*.

20a. Host variables

'Host variables' might be used in the SQL clause, thereby using RXS variables as input to the SQL call. Host-variables are prefixed by `':'`. *Example 20.1* above uses the host variable `w_department`. The value 'SALES' is assigned to this RXS variable in the general orders for the action block. Assignment may also take place in an nesting action block:

Example 20.2:

```
)text out='q1'
   select account, name from myqualif.mytable
   where department = :w_department
)endtext
)action
   w_department = 1448
  )action insql='q1'
   name" has account number "account
  )endaction
)endaction
```

Assigning null values to host variables is done this way: If a RXS variable is assigned the value `'?'`, DB2 will read the value as null. This rule does not apply when calling stored procedure - see *Section 20e* below.

Example 20.3:

```
)text out='q1'
   update myqualif.mytable
   set account = :w_account
   where department = :w_department
)endtext
)action insql='q1'
)&      w_department='SALES'
)&      w_account='?'
   "Ok, number of rows affected is:" sqlerrd.3
)endaction
```

In this example 'account 'is updated to null for all rows having department = 'SALES'

Notice the variable sqlerrd.3 in DB2 which always contains the number of rows affected by an insert / update call.

20b. Output from a SQL select call

Output from the DB2 call is transported into RXS namespace using their DB2 names:

- `select count(*) from myqualif.mytable` will not work: `count` don't have a name and accordingly cannot be used by RXS.
- `select count(*) as w_count from myqualif.mytable` do work: the counter is transported out of DB2 and into RXS in the variable `w_count`.
- `select count(*) as "number of rows" from myqualif.mytable` will not work: "number of rows" is not a valid RXS variable name
- `select * from myqualif.mytable` do work: DB2 replaces the `*` by the names of all fields in the row, and the values are transported out of DB2 and into RXS using these names.

Therefore, use of SQL clause 'select into' is illegal. Mapping of SQL variables to RXS variables is handed automatically.

Data are presented to RXS in the same format as in SPUFI or in QMF:

- Decimal separator is '.'

- Numeric data are not prefixed by zeros
- Alphanumeric fields are post fixed by spaces up to their defined length in DB2.

Besides the fields read from DB2, a SELECT call assigns values to these 'extra' variables in RXS:

`sqlnames`	this variable holds all data names that are selected. The names are listed in the variable separated by one blank
`sqltypes`	this variable holds a 'type' for every data that are selected. The type is `'A'` for alphanumeric and `'N'` for numeric. The types are listed in the variable separated by one blank
`sqllengths`	this variable holds the length of every data that are selected. The lengths are listed in the variable separated by one blank (For BLOB/CLOB fields the length is set to 100000 - but the actual length of a BLOB/CLOB determines the length of the receiving RXS variable when a row is read - up to 16 MB)
`sqlvalues`	this variable holds all data read in the select. The data are listed in the variable separated by one blank. All data are of fixed length, according to `sqllengths`. Numeric data is prefixed by spaces, alphanumeric data are suffixed by spaces
`sqlnulls`	this variable holds a null indicator for all data read in the select. '–' indicates the data is read as null, '`0`' indicates the data is given a value. The indicators are listed in this variable separated by one blank

20c. Choosing DB2 system

`sql`

The DB2 system to be accessed is specified at the installation of RXS. The actual RXS program may deviate from this specification, and access another DB2 system. General order `sql` is used for this purpose. Use for instance `sql='ddb2'` if your installation holds a DB2 system called 'ddb2'. If you are to access more than one DB2 system during one execution of a RXS program, use qualification in your table names - you cannot use different values of `sql` to switch between different systems during one execution of a RXS program. Violating this rule, an error message will correct you.

20d. SQL update, delete, insert

See *Example 20.3* above.

If DB2 `update` or `delete` or `insert` is used in the RXS program, these changes are *committed* at termination of the RXS program.

If any part of the RXS program ends in error (error related to DB2 or any other error), if the user leaves the program reversing out through the first window of the dialogue (according to *Section 21a*), or if the RXS program terminates in a programmed `exit` or `return`, then all DB2 changes are rolled back.

If an action block is driven by a SQL-call which uses `set`, `update`, `delete` or `insert`, then the action block will by triggered once - provided that this SQL call results in SQLCODE = 0.

Variable `sqlerrd.3` contains the numbers of rows affected by an update or delete statement.

Any SQL call resulting in an SQLCODE not equal to zero and 100 will terminate the RXS program with an error message. Any SQL call giving SQLCODE = 100 (no data) will trigger a `)notrigger` part of the action block. If `)notrigger` is not coded, nothing happens: the action block is not triggered, and no error is raised.

20e. Calling a DB2 stored procedure

The handling of null values is somewhat different when calling a DB2 stored-procedure, as shown i example 20.4.

Example 20.4 demonstrates access of IMS data from RXS. CICS data may be accessed using similar principles.

Example 20.4:

```
)text out='sqa'
CALL SYSPROC.DSNAIMS (
   :IN_DSNAIMS_FUNCTION ,
   :IN_DSNAIMS_2PC      ,
   :IN_XCF_GROUP_NAME   ,
   :IN_XCF_IMS_NAME     ,
   :IN_RACF_USERID       :NULLP,
   :IN_RACF_GROUPID      :NULLP,
   :INOUT_IMS_LTERM      :NULLP,
   :INOUT_IMS_MODNAME    :NULLP,
   :IN_IMS_TRAN_NAME     :NULLP,
   :IN_IMS_DATA_IN      ,
   :OUT_IMS_DATA_OUT    ,
   :IN_OTMA_TPIPE_NAME   :NULLP,
   :IN_OTMA_DRU_NAME     :NULLP,
   :IN_OTMA_USER_DATA_IN :NULLP,
   :OUT_OTMA_USER_DATA_OUT,
   :OUT_STATUS_MESSAGE  ,
   :OUT_RETURN_CODE    )
)endtext
)action
   IN_DSNAIMS_FUNCTION    = 'SENDRECV'
   IN_DSNAIMS_2PC         = 'N'
   IN_XCF_GROUP_NAME      = LEFT('IMSOTMA',8)
   IN_XCF_IMS_NAME        = LEFT('IMS6',16)    /* IMS system */
   IN_RACF_USERID         = ''
   IN_RACF_GROUPID        = ''
   INOUT_IMS_LTERM        = ''
   INOUT_IMS_MODNAME      = ''
   IN_IMS_TRAN_NAME       = ''
   IN_IMS_DATA_IN         = LEFT('GETACC SALES',2000)
   OUT_IMS_DATA_OUT       = LEFT('',32000)
   IN_OTMA_TPIPE_NAME     = ''
   IN_OTMA_DRU_NAME       = ''
   IN_OTMA_USER_DATA_IN   = ''
   OUT_OTMA_USER_DATA_OUT = LEFT('',1022)
   OUT_STATUS_MESSAGE     = LEFT('',120)
   OUT_RETURN_CODE        = 0
   NULLP = -1
   )action insql='sqa'
      OUT_IMS_DATA_OUT
      OUT_OTMA_USER_DATA_OUT
      OUT_STATUS_MESSAGE
      OUT_RETURN_CODE
   )endaction
)endaction
```

Stored procedure SYSPROC.DSNAIMS is an IBM-provided procedure giving access to IMS from DB2: Any IMS transaction code or IMS system command may be issued in IN_IMS_DATA_IN, and if the stated function or command returns output, it will be presented in OUT_IMS_DATA_OUT.

SYSPROC.DSNACICS gives similar access to CICS.

20f. SQL limitations

A RXS program may contain up to 99 different SQL select call, and any number of other SQL calls. All calls to SQL can be active at the same moment - action blocks using SQL may be woven into each other in any pattern. (The precise rule is: action blocks are numbered from the top of the program. First 99 action blocks may use SQL select, the rest of the action blocks may not).

20g. SQL isolation level

Isolation level in SQL is 'Cursor stability' If isolation level is to be changed, execute the following SQL statement from inside RXS:

```
Set current packageset = 'DSNREXxx'
```

here `xx` is `RR` for 'Repeatable read', `RS` for 'Read stability', `CS` for 'Cursor stability' and `UR` for 'Uncommitted read'. Isolation level may be changed at any time in a RXS program, and the stated isolation level will govern all succeeding SQL call in the RXS program.

Example 20.8:

```
)text out='sqa'
        set current packageset = 'dsnrexur'
)endtext
)action insql='sqa'
)endaction
)text out='sqb'
   select name from myqualif.mytable
   where department = 'SALES'
)endtext
)action insql='sqb'
name
)endaction
```

This program will read all names from SALES department. Reading will be done under 'uncommitted read' that is without checking locks in DB2.

20h. Comments in SQL

RXS accepts the 'normal' way of adding comments to a SQL-call: Any text inside SQL prefixed by '--' will not be interpreted.

Example 20.9:

```
)action out='q1'
   "select account, name from myqualif.mytable"
   "where department = :w_department  -- killroy was here..."
   "-- and position = :w_position"
   "-- and salary > 13000  "
)endaction
)action
   w_department = "'1448'"
   )action insql='q1'
      name" has account number "account
   )endaction
)endaction
```

21. Func='prompt' Opening windows

Using `func='prompt'` in RXS opens for the programming of dialogs using windows. The programming is 'non procedural': you specify in the coding which input you want at which point

in your logic, and the RXS engine generates some windows and a dialogue to connect them. The user may iterate backwards among the windows when using the dialogue - this is not an issue for the programmer, but handled by RXS itself.

Example 21.1:

```
)action out='prm1'
   'member', 'enter member name'
)endaction
)action in='prm1'
)&       func='prompt'
)endaction
```

The first action block writes a record to the queue 'prm1'. Two parts of the record are written separately, separated by comma. The second action block reads the queue 'prm1' and uses the information to generate a window on the screen. The window will prompt the user of the dialogue to enter a value for the variable 'member'. The user may act as indicated, or may press F3 (end) to terminate the dialogue. RXS generates a guiding text in the window pointing out these two possible responses.

Example 21.2:

```
)action out='prm1'
   'member', 'enter member name'
)endaction
)action in='prm1'
)&       func='prompt'
   )action in='myqualif.mydsn('member')'
   )&       out='que1'
      unit.1
   )endaction
   )action  out='myqualif.mydsn('member')'
   )&       in='que1'
   )&       outfunc='submit'
   )&       start=1
      if start = 1 then do
         "//* This member is submitted" date()time() userid()
         start = 0
      end
      unit.1
   )endaction
)endaction
```

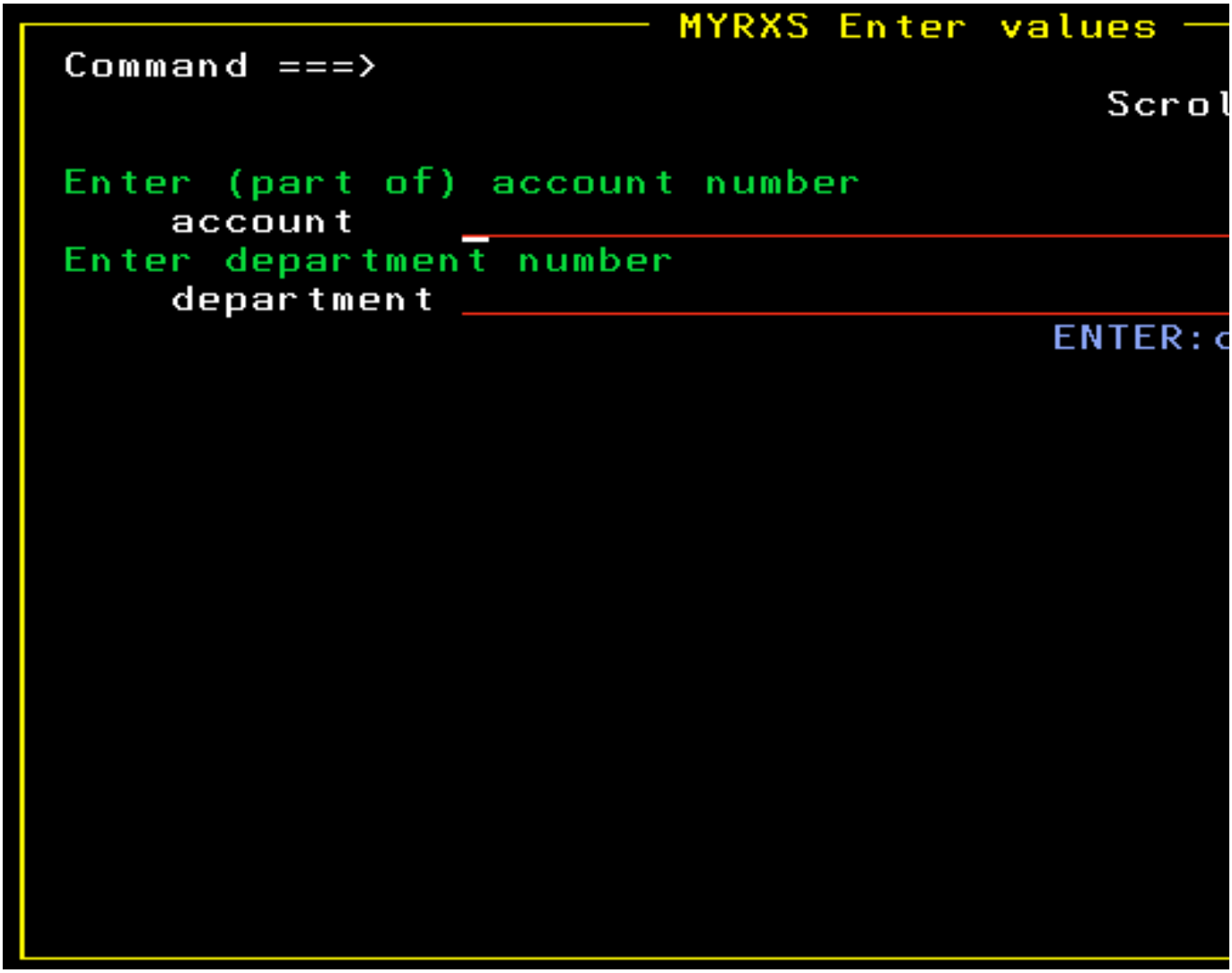
The example reads a member from a dataset, adds a new line at the top of the member, submits and saves the member. The user is prompted to enter the name of the member that is to be acted on.

Example 21.3:

```
)action out='prm1'
   'account', 'Enter (part of) account number'
   'department', 'Enter department number'
)endaction
)action in='prm1'
)&       func='prompt'
)&       out='sql1'
   "select account, name from myqualif.mytable    "
   "where account like '"account"%'               "
   "and department = '"department"'               "
)endaction
)action in='sql1'
)&       func='sql'
   "name: "left(name,15)" has account: "account" ,
   "(department "department")"
)endaction
```

Here we have two queues, `prm1` and `sql1`, and three action blocks.

The first action block writes two rows on a queue, `prm1`, each row containing two parts, separated by comma.
The other action block writes something (in SQL syntax) on a queue `sql1`.
Output from the first action block is input to the second, governed by general order `func='prompt'`
The window on the screen looks:



Repeating - with more precision:
`func='prompt'` implies that information in the input queue is used to format a window on the screen, prompting the user to enter the value for one or more variables. The format and lay-out of the window depends on the amount of data shown; when appropriate a full screen panel will be shown instead of a window.

- The first part of each element in the input queue names the variable which is to be assigned a value.
- The second part of each element in the input queue is a guiding text to be shown next to the name of the variable and the input field. This part of the element may be omitted. In such case, only the name of the variable and the input field is shown in the window.
- Strings in the elements of the queue are in quotes. Applies even to the name of the variable: it is not a variable; it is a string indicating the name of a variable.

- When the user has entered values for these variables (or for this variable) and presses *enter*, the action block having `func='prompt'` will be triggered once. Assigned values are at hand inside the action blocks by their names. If the user presses F3 (*end*) instead of *enter*, an optional `)notrigger` part of the action block is triggered. If no `)notrigger` is programmed, the RXS program returns to a former state according to the following rules:

21a. The dialogue generated by prompt

If a RXS program contains at least one action block which is using `func='prompt'`, the execution of the program - *as seen from the user* - will go like this:

If the user presses F3 (*end*) when output from RXS is shown on the screen, a re-display of the first window in the dialogue will take place

If the user presses F3 (*end*) when a RXS window is prompting for input, a re-display of the former window in the dialogue will take place. If the current window is the first window of the dialogue (or the sole window of the dialogue) then the RXS program terminates.

If the user presses *enter* in the same situation, the RXS program will prompt for input in the current window or display the next window, depending on whether the user has entered some input before pressing *enter*.

But do not put notice on these rules: As *seen from the programmer* of the RXS program, this is transparent. You indicate in the program which information you want at which point in the logic - the dialogue is not programmed (but optionally may be adjusted - see below).

Commit or rollback of changes made by the RXS program is affected by the way the user handles the dialogue. The commit point in RXS is - as mentioned - just prior to the display of output from RXS. If the program never reaches this point, a rollback is performed, and no update of DB2, MQSeries or sequential files is done. Therefore, if the user reverses out of the program by pressing F3 (*end*) in a re-show of each previous shown windows of the program, and continues to do this until the program terminates, then nothing is committed. If the user cycles several times through the logic, pressing F3 (*end*) after output is displayed, commit will take place each time the program reaches the point where output is to be displayed. Messages on screen will inform of commit and rollback.

If the user makes a reverse exit of the program, backing his way out through all windows including the first window, without having ever reached a display of output, RXS will set a return code +4 to ISPF.

21b. Tailoring the dialogue using programming

If the programmer uses `return` in the RXS program, this will force a jump to the first window of the dialogue - if the user has already seen this window. If not, the RXS program terminates.

If the programmer uses `exit` in the RXS program, this will unconditionally terminate the program.

Programming a `)notrigger` part of an action block which is using prompt will catch when the user presses F3 (*end*). For instance programming `exit` in the `)notrigger` block will cause the program to terminate if the user presses F3 (*end*) in this window.

Programming general order `outfunc='nop'` means that output is not displayed. Accordingly the user has no occasion to press F3 (end) in the termination situation. Therefore the program terminates unconditionally when the logic of the program is executed once. The same applies if the program does not produce any sequential output (any file). These are 'ok' situations, and any changes are committed.

To validate input from the user, use instruction `set_halt`:

Example 21.4:

```
)action out='prm1'
   'member', 'enter member name'
```

```
)endaction
)action in='prm1'
)&      func='prompt'
  if datatype(left(member,1)) = 'NUM' then do
    set_halt "Member-name must start with a chararacter"
  end
)endaction
```

If the first charater in the string entered on the screen is numeric, then the panel is re-displayed with the indicated message. If the error is corrected, the dialogue continues normally.

To put additional guidance on the input-screen, use instruction `set_message`:

Example 21.5:

```
)action out='prm1'
   'member', 'enter member name'
    set_message "Member-name starts with a chararacter"
)endaction
)action in='prm1'
)&      func='prompt'
)endaction
```

The message is displayed first time the RXS program reaches a panel-display. This could be the final display of output, but in this case it is the prompt.

21c. Tailoring the dialogue using general orders

prompt

Setting general order `prompt='yrsa'` will internally name the window: it now bears the name 'yrsa'. A named window will remember its content, entailing that a re-show of the window later in the same ISPF session will re-display its values. The re-display may even be initiated from another RXS program provided that the variable names and the guiding texts which are set up in the window by both programs are identical. And of course provided that the two programs name the window identical.

The variable `prompt` may hold a maximum of 6 characters (Window content is saved in ISPF-tables).

A window without a name will function the same way, and remember its content, provided that no other not-identical window without a name is used in the users ISPF-session. Again, windows are considered identical if they use identical variable names with identical guiding texts.

promptsource

Setting a value for general order `promptsource` governs the content of the window in the event of a re-display during the TSO-session.

There are three possible values:

- `'U'` (default): If the window is re-displayed, all values are re-displayed too - provided that the window is unique, according to the discussion of `prompt` above. If the window has not been displayed before, values assigned in the RXS program will initially be shown.
- `'P'` Values assigned in the RXS program will always be shown in a re-display of the window.
- `'I'` In a re-display of the window, all values are blank.

promptall

Setting a value for general order `promptall` governs whether the user is allowed to continue by pressing *enter* without entering a value in every field in the window.

There are two possible values:

- `'Y'` (Default) The user must enter a value in every field. Not obeying this, an error message will prompt the user to do so if he presses *enter*.
- `'N'` The user may do as he likes: enter a value in none, in some, or in all fields of the window when pressing *enter*.

promptlgth

Setting a value for general order `promptlgth` will determine the length of the input fields in the window. Default length is 42 characters; any other value between 0 and 130 may be entered in `promptlgth`.

Setting a length of zero makes it pointless to display the window. Accordingly it is not displayed. This may be used as a mechanism for transporting variables between RXS programs: making a prompt of a named window which is holding some values will make these values part of the RXS program. This is also true if the window is not actually displayed.

caps

Setting general order `caps='on'` will transform any input in the window to upper case. Setting `caps='off'` will not transform input. `caps='on'` is default.

zwinttl

This general order sets a windws-title: Assigning a value to variable `zwinttl` in the RXS program results in that value being written in the top part of the frame of the window - as a title.

21d. A more advanced example

In this example, window content - that is the names of variables and the guiding texts - reflects data originating from DB2, instead of being stated in the program:

Example 21.6

```
)text out='sql1'
  select distinct department from our.employee
)endtext
)text out='sql2'
  select name, salary from our.employee
  where department = :wdep
)endtext
)text out='sql3'
  update our.employee set salary = :wsalary
  where department = :wdep and name = :name
)endtext
)action insql='sql1'
)&      out='prm1'
  department
)endaction
)action in='prm1'
)&      func='prompt'
)&      prompt='prm1'
)&      promptall='n'
)&      zwinttl="Select department(s) entering 'X'"
)&      promptlgth=1
  )action in='prm1'
    if value(unit.1) = 'X' then do /*if selected on screen*/
      wdep = unit.1
      )action insql='sql2'
      )&      out='prm2'
        name , "(Salary: "salary")"
      )endaction
      )action in='prm2'
      )&      func='prompt'
      )&      prompt=left(wdep,6)
```

```
        )&       promptall='n'
        )&       promptlgth='1'
        )&      zwinttl=wdep": Grant a 10% salary rise entering 'X'"
          )action insql='sql2'
            if value(name) = 'X' then do /*if selected on screen*/
              wsalary = trunc(salary * 1.1)
              left(name "("wdep")",25) "is granted a 10% salary",
              "rise. New salary:" wsalary
              )action insql='sql3'
              )endaction
            end
          )endaction
        )endaction
      end
    )endaction
)endaction
```

It is assumed that DB2 table *our.employee* contains fields *Department*, *Name* and *Salary*. Unique key is Department + Name.

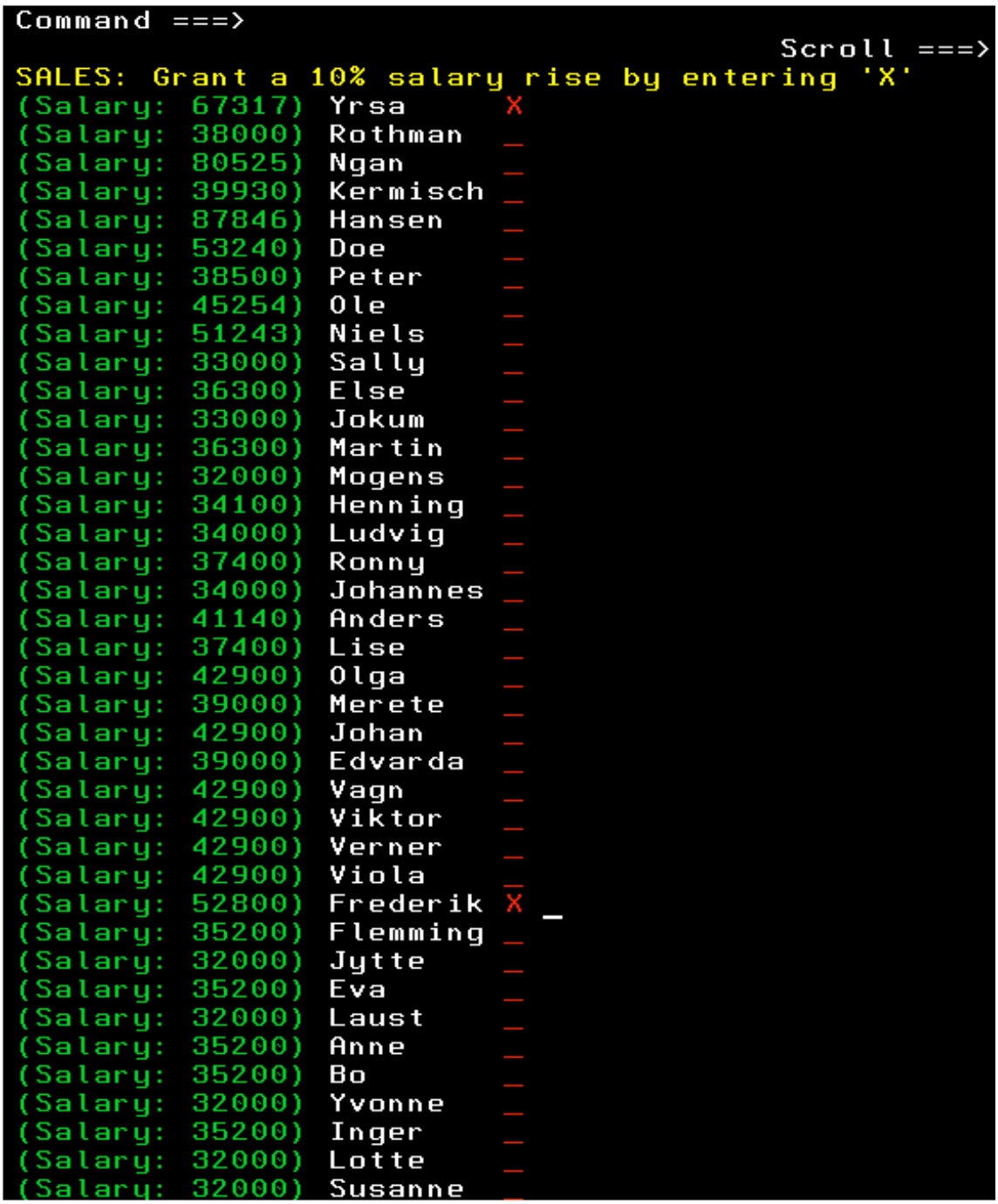
Running the example will at first show a window listing all departments:

If one or more departments are selected using an 'X', a window showing a list of all employees and their salary is shown for each selected department.

Any employee selected in this window using an 'X' will be given a 10% salary raise. The user (the manager...) may cycle between the windows using 'enter' or 'F3'.


MANAGEMENT: Grant a 10% salary ris
Command ===>
Sc
(Salary: 111465) Wrinkley X
(Salary: 96630) Hardy _
(Salary: 74048) Onslow _
(Salary: 41261) Johnson _
(Salary: 37000) Marenghi X
(Salary: 28000) Naughton _
(Salary: 45000) Abrahams _
(Salary: 47190) Quigley _
ENTE

SALES department actually contains 41 employees; therefore RXS switches to a full screen window with scrolling:



When the last one of the departments selected on the first screen is processed, a list of employees receiving a salary rise is shown (see below), and all updates are commited: the new salaries are written to DB2. If the user at any point before this final display leaves the program (pressing F3 repeatedly) all updates are rolled back, and no employee is receiving any salary rise. The user will in this situation be notified that all updates are rolled back.

If commit is reached, this is what is finally displayed: a message confirming 'commit' and a list of actions made:

```
EDIT       MYUSER.RXS.DATA                                    SQL commit
Command ===>                                                  Scroll ===> CSR
****** ***************************** Top of Data*********************************
000001 Wrinkley (MANAGEMENT)   is granted a 10% salary rise. New salary: 122611
```

```
000002 Marenghi (MANAGEMENT)   is granted a 10% salary rise. New salary: 40700
000003 Yrsa (SALES)            is granted a 10% salary rise. New salary: 74048
000004 Frederik (SALES)        is granted a 10% salary rise. New salary: 58080
****** *************************** Bottom of Data ****************************
```

If the program is executed several times, the first window (listing departments) will remember its selections, and make a redisplay of what was entered before. The other windows will not remember their selections because guiding texts have changed due to the already granted salary rises.

23. Func='dcl'. DB2 table information

A DCL library is a partitioned dataset containing DCL structures, created by the DB2 dclgen command. The DCL structure describes the fields of a DB2 table. It is intended for use in the access of DB2 from compiled languages like COBOL.

Using `func='dcl'` implies that input to the action block must be a member of a dataset containing DCL. The action block is triggered once for each field described in the DCL structure. Information about format and field name is then available inside the action block.

The same information may be read from the DB2 system catalogue by using `func='sql'`, but accessing the DCL area might require less coding in the RXS program.

The following variables are assigned values for use inside the action block:

`dataname`	DB2 field name for current field
`datatype`	Type of field. Contains `DATE TIMESTAMP CHAR DECIMAL VARCHAR, BLOB, CLOB` or `SMALLINT`
`length`	Number of bytes (Example: 26 for a timestamp) - or maximum number of digits including decimals - if the field type is numeric
`decimals`	Only if field type is numeric: Number of digits after the decimal point
`nulls`	A value of `'1'` if the field holds a null indicator. `'0'` if the field does not hold a null indicator

Example 23.1:

```
)action in='ourquali.dcl.cobol(ourtab)'
)&      func='dcl'
  if nulls = '1' then do
     "      IF OURTAB-"dataname"-I = -1 "
     if datatype = 'DECIMAL' ! datatype = 'SMALLINT' then do
        "         MOVE ZERO TO OURTAB-"dataname
     end
     else do
        "         MOVE SPACE TO OURTAB-"dataname
     end
     "      END-IF"
  end
)endaction
```

The resulting COBOL code will initialize all fields in a DB2 structure if they hold a null indicator and if the indicator actually indicates the field to be null. Such coding may come in handy after a successful DB2 select call in COBOL.

24. Func='namespace'. Using the internal RXS format: namespace

Example 24.1:

```
)text out='q1'
```

```
   name(Peter Jensen)
   age(32)
   position(bike mechanics)
   nationality(danish)
   ;
   name(Hugo Jensen)
   position(account manager) age(44)
   nationality(danish)
   ;
   name(Niels Olsen)
   age(19) position(CEO) nationality(greek)
)endtext
)action in='q1'
)&      func='namespace'
   if nationality = 'danish' then do
     left(navn,25)"  "left(position,20)" age:"age
   end
)endaction
```

Output is:

```
Peter Jensen              bike mechanics          age:32
Hugo Jensen               account manager         age:44
```

The core principle of the namespace file-format is assigning a value to a variable this way:

`variable(value)`

A namespace in RXS is a file containing groups of such assignments, the groups separated by '`;`'. Refer to the text block in top of *example 24.1* above.

Using `func='namespace'`, each group in the file triggers the action block. Example 24.1 contains three groups; accordingly the action block is triggered three times.

Notice:

- Writing more than one assignment per line is ok
- One assignment may span several lines. The assignment starts with `(` and ends with `)`. In between any number of lines may exist. In assigning the value to the variable, the lines are concatenated separated by one blank character. The interleaving blank can be avoided using the concatenation operator `||` at the end of the line.
- Assignments are not to be put in quotes. If assignments are quoted, the quotes will be part of the assigned value.
- Quotes, single and double, may be used anywhere inside assignments.
- Parenthesis may be used inside assignments, but they must be balanced: Same number of `)` and `(`.
- Variables assigned a value in a previous group of assignments, not given a value in the current group of assignments, are empty (that is: strings of length zero).

A pre screening of the namespace file may be conducted using the variable `spacerow`.

`spacerow`	Contains the current namespace group in namespace format

Example 24.2:
Using spacerow:

```
)text out='q1'
   name(Peter Jensen)
   age(32)
   position(bike mechanics)
   nationality(danish)
   ;
   name(Hugo Jensen)
   position(account manager) age(44)
```

```
   nationality(danish)
   ;
   name(Niels Olsen)
   age(19) position(CEO) nationality(greek)
)endtext
)action in='q1'
)&      func='namespace'
)&      out='qspace'
   if nationality = 'danish' then do
     spacerow
   end
)endaction
)action in='qspace'
)&      func='namespace'
   left(navn,25)"  "left(position,20)" age:"age
)endaction
```

At first, data is put into the queue `'q1'`, after which selected groups are copied to the queue `'qspace'` from which they are formatted to output in the last action block.

Output will be as in example 24.1

25. Func='xml' Accessing XML

Example 25.1

```
)text out='xx'
   <Order Salesrep="Yrsa" Date="2006-05-05">
     <CustomNumber>4711</CustomNumber>
     <CustomContact>
       Wilbur Jensen &amp; John Doe
     </CustomContact>
     <ShippingAddress A1="Solitudevej 14" A2="2840 Holte"/>
     <Detail>
       <Itemno>1864</Itemno>
       <Quantity>4</Quantity>
     </Detail>
     <Detail>
       <Itemno>1448</Itemno>
       <Quantity>2</Quantity>
     </Detail>
   </Order>
)endtext
)action in='xx'
)&      func='xml'
   do i = 1 to xml.0
     if i > xml_elem_unch then do
       intend = left('                ',i * 2)
       if i < xml.0 | xml = '' then do
         intend""xml.i
       end
       else do
         intend""xml.i "=" xml
       end
     end
   end
   do i = 1 to xml_attrib.0
     intend"      "xml_attrib.i"="value(xml_attrib.i)
   end
)endaction
```

Output is:

```
Order
     Salesrep=Yrsa
     Date=2006-05-05
   CustomNumber = 4711
   CustomContact = Wilbur Jensen & John Doe
   ShippingAddress
        A1=Solitudevej 14
        A2=2840 Holte
   Detail
        Itemno = 1864
        Quantity = 4
   Detail
        Itemno = 1448
        Quantity = 2
```

The RXS function `value` (third last line) performs a double de-reference: `xml_attrib.i` holds a name of a variable as a value, and the value of this variable is found.

Notice that XML is a 'generalized' language. Accordingly, the above RXS program (the second action block) will format (or de-format) any XML structure.

`func='xml'` presumes input is 'well formed' XML. Otherwise execution is terminated in error.

The input file is considered as being one single piece of XML, containing just one starting tag that is closed in the last record.

The input XML structure for an action block using func='XML' triggers the action block every time the XML structure assigns values for one or more variables. That is, every time an element is assigned a value, or some attributes at the same level are assigned values, or both.

'Escape sequences' in the XML are translated to their equivalent characters (notice the element in `CustomContact` in *Example 25.1* above).

'White space' around elements is ignored (again notice the element in `CustomContact` in *Example 25.1* above).

The following RXS variables are assigned values in the triggering:

`xml.`*i*	A *stem* containing the hierarchy of names behind the actual value. `xml.0` contains the number of elements in the stem, that is, the number of names in the actual hierarchy. 'XML namespace' is ignored: When namespace is used, only the part of the name behind ':' is recorded.
`xml_cnt`	The number of names in the actual hierarchy (equal to `xml.0`)
`xml_attrib.`*i*	A *stem* containing names of all attributes that are assigned a value at this level in the hierarchy. `xml_attrib.0` contains the number of elements in the stem, that is the number of attributes at this level in the hierarchy. 'XML namespace' is ignored: When namespace is used, only the part of the name behind ':' is recorded
`xml_attrib_cnt`	The number of attributes at this level in the hierarchy (equal to `xml_attrib.0`)
`xml`	The value of the element (might be content of a CDATA string in the XML)
`xml_elem_unch`	The number of names in the current hierarchy that belong to the same path compared to the previous triggering of the action block. That is: elements numbered from 1 up to `xml_elem_unch` are part of the same path as in the previous triggering.

Besides these variables, attributes are at hand inside the action block. If for instance the actual path in the XML contains the assignment `yrsa = "14"`, the variable `yrsa` will contain the value `14`.

Notice: RXS is not case sensitive; accordingly attributes `Yrsa` and `yrsa` are considered the same variable

If the actual path in the XML only assigns values to attributes, `xml` is empty (is a string of length zero).

If the actual path in the XML only assigns value to the element, then `xml_attrib_cnt` and `xml_attrib.0` is zero.

Attributes given value in a former triggering, are empty (string of length zero) in the actual triggering - unless the current path in XML assigns new values to these.

Referring to example 25.1: in the first triggering of the action block *xml.1* contains "Order", *xml.2* contains "Salesrep" and *xml* contains "Yrsa". *xml_elem_unch* is zero.

In the second triggering of the action block *xml.1* contails "Order", *xml.2* contains "Date" and *xml* contains "2005-05-05". *xml_elem_unch* contains 1 indicating that only first element in *xml.* is unchanged compared to the first triggering.

Constraints:

- XML elements may not contain unbalanced set of `'{'` and `'}'`.
- Maximum length of an XML structure is 16 MB.
- Maximum length of an element is 0,5 MB
- Maximum length of an attribute value is 1000 bytes
- Maximum length of a tag name or a attribute name is 1000 byte
- Maximum depth of hierarchy of tags behind an element is 100 tags
- Maximum number of attributes per tag is 100

In example 25.1 following values are assigned during the seven triggerings of the action block:

xml.1	xml.2	xml.3	xml.0	xml	xml_elem_unch
Order	-	-	1	-	0
Order	CustomNumber	-	2	4711	1
Order	ShippingAddress	-	2	-	1
Order	Detail	Itemno	3	1864	1
Order	Detail	Quantity	3	4	2
Order	Detail	Itemno	3	1448	1
Order	Detail	Quantity	3	2	2

Besides, attributes are given values.

Example 25.2

The input in this example is one file containing a sequence of separate xml-structures.

```
)text out='xx'
  <order no="1"><type>Pizza No 14</type><quant>3</quant></order>
  <order no="3"><type>Slush Ice</type><quant>2</quant></order>
)endtext
)action in='xx' /* separate input */
)&      i=0
  i = i + 1
  indv = unit.1
  )action out='q'i
    indv
  )endaction
)endaction
)action  /* read the separated input */
  do ii = 1 to i
  '-------------- XML-structure number ' ii '---------------'
  )action in='q'ii
  )&      func='xml'
    do i = 1 to xml.0
      if i > xml_elem_unch then do
        intend = left(' ',i * 2)
        if i < xml.0 | xml = '' then do
          intend""xml.i
        end
        else do
          intend""xml.i "=" xml
        end
      end
    end
    do i = 1 to xml_attrib.0
      intend" "xml_attrib.i"="value(xml_attrib.i)
    end
  )endaction
  end
)endaction
```

The first action block (marked with /* separate input */) puts every single input record into a separate queue, named 'q1', 'q2', 'q3' and so forth. The variable i contains the total number of queues.

The next top-level action block (marked with /* read the separated input */) reads these queues one after one, and analyzes them as XML.

Output is:

```
---------------- XML-structure number 1 ---------------
order
```

```
     no=1
  type=Pizza No 14
  quant=3
---------------- XML-structure number 2 ---------------
order
     no=3
  type=Slush Ice
  quant=2
```

26. Func='sorted' Func='sorted_desc' Sorting input

Example 26.1

```
)text out='myqueue'
  9
  25
  121
  2
)endtext
)action in='myqueue'
)&      func='sorted'
  'square of' word.1 'is' word.1**2
)endaction
```

Output is:

```
square of 2 is 4
square of 9 is 81
square of 25 is 625
square of 121 is 14641
```

`func='sorted'` performs a sorting of the input before it is presented to the action block. Input records are sorted ascending on the value of `unit.1`.

If all values of `unit.1` are numeric, a *numeric* sort is performed; otherwise an *alphanumeric* sort is performed:

Example 26.2

Alphanumeric sort:

```
)text out='myqueue'
  9
  25
  UPS
  121
  2
)endtext
)action in='myqueue'
)&      func='sorted'
  unit.1
)endaction
```

Output will be like this (Sorting is primarily on byte 1, that is 121 is ranked before 2 etc.):

```
UPS
121
2
25
9
```

If `unit.2` is present in input, it will also participate in the sort. In this case `unit.1` is the sorting key of the item and `unit.2` is the value of the item:

Example 26.3

```
)action out='myqueue'
  9, 'Road Runner'
  25, 'Winnie the Pooh'
  121, 'Cinderella'
```

```
  2, 'The Big Bad Wolf'
)endaction
)action in='myqueue'
)&      func='sorted'
  unit.2
)endaction
```

Output is:

```
The Big Bad Wolf
Road Runner
Winnie the Pooh
Cinderella
```

If length of any input (`unit.1`) record exceeds 256, then consider putting input in `unit.2` and create a sort criteria in `unit.1`.

If length of any input (`unit.1`) record exceeds 256 and is less than 4000, and if `unit.2` is not used, the sorting schema changes to a alphanumeric sort of `unit.1`.

If

- the length of any `unit.1` in input is larger than 4000
- the length of any `unit.1` is larger than 256 and `unit.2` is used

then the sort in RXS is terminated in error

26a. Sorted_desc

`func='sorted_desc'` works like `func='sorted'`, except that a descending sort on `unit.1` is performed.

27. Func='mqbrowse' and other access to MQSeries

RXS reads and writes queues defined in *IBM Webspere MQSeries*. Any changes on queues during the access are committed, provided the RXS program ends normally. If the program ends in error (error related to MQSeries or any other error), if the user leaves the program reversing out through the first window of the dialogue (according to *Section 21a*), or if the RXS program terminates in a programmed `exit` or `return`, then all MQSeries changes are rolled back.

`mq`

General order `mq` contains the name of the actual MQSeries system. `mq` may only be given one value in a RXS program: a RXS program cannot access several MQSeries queue managers. Default value for `mq` is stated at installation of RXS.

In RXS, a MQ message can hold up to 16 MB of data.

27a. Reading messages from MQSeries: Func='mqbrowse'

`in`

For `func='mqbrowse'` the name of the queue to read from is stated in general order `in`.

`readlim`

As general order for the action block may be stated `readlim`: the maximum number of messages to be read. Default for `readlim` is 3,000,000.

The reading assigns values to these variables:

`unit.1`	Contains the current message
`mq_backout`	*Backout-count*, that is the number of times this message previously

	has been read in vain, because of rollbacks in a reading application
`mq_messid`	*Message-ident*: a 24 character field containing the unique key assigned to the message by MQSeries
`mq_putdate`	Date at which the message was created on the queue
`mq_puttime`	Time at which the message was created on the queue (Greenwich mean time)
`mq_applname`	A name of the application which created the message on the queue
`mq_appltype`	The environment of the application that created the message on the queue (CICS, DOS, AIX, MVS, OS390, WINDOWS...)

Each message triggers the action block. In MQSeries terms, 'mqbrowse' is a MQGET with the browse-flag set.

27b. Destructive reading of MQSeries: Func='mqdrain'

`func='mqdrain'` functions like `func='mqbrowse'`, but reading is destructive, meaning that the accessed queue is empty after reading. In MQSeries terms, 'mqdrain' is a normal MQGET. A value in `readlim` will limit the destructive reading, for instance `readlim=30` will read and delete the 30 oldest messages on the queue.

27c. Destructive reading of one message: Func='mqdrainkey'

`mq_messid`

`func='mqdrainkey'` works like `func='mqdrain'`, but only one message is read. The message to be read is stated in general order `mq_messid` (24 character MQ-message-id).

Reading is destructive, meaning that the one message read is removed from the queue.

Example 27.1

```
)action in='ourqualf.inpque'
)&      func='mqbrowse'
  if substr(unit.1,4,10) = '2007-01-06' then do
    foundmess=mq_messid
    )action in='ourqualf.inpque'
    )&      func='mqdrainkey'
    )&      mq_messid=foundmess
       "Deleted: "left(unit.1,40)
    )endaction
  end
)endaction
```

The program deletes all messages having date '2007-01-06' from the stated queue. The date is located in position 4 in the messages and is 10 bytes long. A report is written on <user>.RXS.DATA containing the first 40 byte of every deleted message.

27d. Writing messages to MQSeries: Outfunc='mqput'

`out outfunc`

If an action block has `outfuc='mqput'` output will be written to the MQSeries queue named by general order `out`.

Example 27.2

```
)action out='ourqualf.thisque'
)&      outfunc='mqput'
)&      mq='mqdc'
  "What hath God wrought?"
  "One small step for man, a giant leap for mankind"
)endaction
```

This RXS program writes two messages on the MQSeries queue `'ourqualf.thisque'`. The queue belongs to MQSeries system `'mqdc'`. mq_putappltype is set to 'OS390' and mq_putapplname is set to the current userid.

28. Accessing files on PC and local area network

To access files on PC and on local area network (LAN): A prerequsite is that IBM ISPF program "Workstation Agent" is installed and is running on the PC which is to be used.

28.a Installation of "Workstation Agent"

The program "Workstation Agent" is part of ISPF. Check menu 3.7.1 for information regarding installation.

Alternative:

- Download mainframe file `'*.ispf.ispgui(ispguinx)'` as binary to `c:\ispfagnt\ispfinst.exe` (the exact name of the mainframe file can be found in menu 3.7.1 in ISPF)
- Download is done in menu '6' in ispf
- The name of the exe file must conform to Windows 95 standard, that is max 8 characters per name etc.
- Execute (double click) `c:\ispfagnt\ispfinst.exe`
- The installation asks about a 'base install directory'. Default is the actual directory. Keep it that way.
- The installer writes "Please add c:\ispfagnt\ispfinst to your path". This makes no sense.
- Now, in the library `c:\ispfagnt\` is the file `wsa.exe`
- Don't move this file, instead make a shortcut to it on the desktop.
- Double click on shortcurt to activate *wsa* ("workstation agent")

Activate *wsa* whenever the PC is started up if you want to access PC files via RXS.

28.b Accessing the PC from RXS

`pc`

General order `pc` must indicate the name of the actually used PC (The name of a PC is the name used for remote connection to the PC. Alternatively, the internet URL addressing the PC may be used).

If `in` or `out` holds a name containing one or several '\' it is considered the complete name (inclusive path) of a file on the indicated PC or on a local area network accessible by that PC.

Different action blocks in the RXS program may read or write different PC's.

As default characters are converted from ASCII to EBCDIC when reading from the PC - and the opposite when writing to the PC. 'Newline' characters ('0D'x + '0A'x) on the PC will break the mainframe file into records - and opposite around.

Constraint: Records having length zero are written to the PC as a line containing one blank.

28.c func='binary': Reading from the PC without character conversion

`func='binary'` implies that actual bits are transmitted 'as is' that is witout ASCII/EBCDIC conversion. 'Newline' characters ('0D'x + '0A'x) are also transmitted as it. Accordingly, input to RXS from the PC will consist of just one record (but notice that a record internally in RXS may consist of 16 MB of data).

28.d outfunc='binary': Writing to the PC without character conversion

`outfunc='binary'` implies that actual bits are transmitted 'as is' that is without ASCII/EBCDIC conversion

Example 28.1

```
)action out='c:\clutter\myfile.txt'
)&      pc='r2d2-2'
  "What's up doc?"
)endaction
```

The line: "What's up doc?", is written as the content of the indicated file on the indicated directory (C:) on the indicated PC. Character representation will be ASCII.

Adding:

```
)&      outfunc='binary'
```

as general order will transmit file as is, accordingly the PC will receive the file as EBCDIC.

Example 28.2

```
)action in='c:\clutter\myfile.txt'
)&      pc='r2d2-2'
  unit.1
)endaction
```

According to example 28.1 the string "What's up doc?" is written on mainframe standard output, <user>.rxs.data. Character representation will be EBCDIC.

Adding:

```
)&      func='binary'
```

as general order will transmit file as is, accordingly RXS on mainframe will receive the file as ASCII.

Example 28.3

```
)action in='o:\ourdept\ouroffice\ourfile.htm'
)&      pc='r2d2-2'
  myvar = unit.1
  )action out='c:\diverse\ourfile.htm'
  )&      pc='r2d2-2'
    myvar
  )endaction
  )action out='c:\diverse\ourfile.htm'
  )&      pc='yrsa-1'
    myvar
  )endaction
)endaction
```

The example copies one file from the local area network over to two different PC's. The users of the two PC's will be prompted for whether they will accept the connection.

Example 28.4

Reading a spreadsheet :

```
)action in='C:\diverse\this_sheet.txt'
)&      pc='R2D2-2'
)&      strt=1
)&      n.=''
)&      v.=''
  if strt = 1 then do
    stmt1 = "parse var unit.1 n.1"
    do ix = 2 to 255
      stmt1 = stmt1 "'05'x  n."ix
    end
```

```
      interpret stmt1
      stmt2 = "parse var unit.1 v.1"
      do ix = 2 to 255
        stmt2 = stmt2 "'05'x  v."ix
      end
      do ix = 1 to 255
        if n.ii = '' then leave
        change(' ', '_',n.ii)
      end
    end
    else do
      interpret stmt2
      do ii = 1 to 255
        if n.ii = '' then leave
        n.ii'('v.ii')'
      end
      ";"
    end
    strt = 0
)endaction
```

A spreadsheet is saved as a tabulator-separated .txt file on the PC. The RXS program transforms the spreadsheet to a mainframe file which is using the RXS namespace form. It is assumed that the columns of the spreadsheet are named in the first row. These names are used as names in the created namespace file. The namespace file is presented on screen at termination. To make sense, the example should use the namespace file to something - say putting the values into some DB2-table with fitting names.

If the spreadsheet contains

	A	B	C
1	Department	Name	Monthly Salary
2	SALES	Jens Olsen	30000
3	SALES	Peter Poulsen	42500
4	INFORMATION	Ole Jensen	45500

then the created file RXS.DATA will contain

```
Department(SALES) Name(Jens Olsen) Monthly_Salary(30000)
;
Department(SALES) Name(Peter Poulsen) Monthly_Salary(42500)
;
Department(INFORMATION) Name(Ole Jensen) Monthly_Salary(45500)
;
```

29. Accessing files on UNIX mainframe

If `in` or `out` points to a name containing one or several '/' it is considered the complete name (inclusive path) of a file on the UNIX HFS file system on the mainframe.

As default any 'newline characters' ('15'x) in the UNIX-file will break the file into records when the file triggers an action block in RXS - and the newline characters will be removed. And opposite: records are concatenated adding 15'x' between each record when writing from RXS to the UNIX file system. This entails that text files keep line breaks when transmitted between the two milieus.

The default behavior can be overruled:

`func='binary'` when reading from UNIX: indicates no handling of newline characters: all characters in the UNIX-file are transmitted. Accordingly the input data from UNIX into RXS

will consist of just one record. (But notice that a record internally in RXS may consist up to 16 MB of data).

`outfunc='binary'` when writing to UNIX: all records written are concatenated in the UNIX-file without any interleaving newline characters.

(Notice UNIX files are strings (or streams) of bits, because no notion of record exists in UNIX.)

If `out` points to a new (a non existent) file, the new file will get acces-control-bits set to 'write and execute' for the user, write and execute for the group (provided that a group has been assigned to the directory in which the file resides) and 'read' for all other users.

Example 29.1

```
)action in='/home/r2d2/example.txt'
   unit.1
)endaction
```

The example copies the indicated unix file to the mainframe file rxs.data. Being a text-file, each line triggers the action block separately, and the coding above results in a mainframe file where records maps the lines of the text file.

Example 29.2

```
)action in='/home/r2d2/example.bin'
)&      func='binary'
   do forever
      if length(unit.1) > 80 then do
         left(unit.1,80)
         unit.1 = substr(unit.1,81)
      end
      else do
         unit.1
      end
   end
)endaction
```

The example copies the indicated UNIX file to the mainframe file rxs.data. The action block is triggered just once, because func='binary' will treat the UNIX file as one single string of bytes. Writing this string on a mainframe file requires some programming to break the string into records, because a mainframe record only may contain 32.760 bytes. The programming in the action block solves this by creating 80 byte records when writing to the mainframe.

UNIX commands may be executed from inside RXS using the general order `address='unix'`. See *section 33*.

30. Character transformation between utf-8, ascii and ebcdic

Example 30.1

```
)action out='q1'
   "Alas my love"
   "You do me wrong"
   "To cast me off"
   "Discourteously"
)endaction
)action in='q1'
)&      func='>utf8'
)&      outfile='yrsa'
)&      outfunc='browse'
   unit.1
)endaction
```

The example transforms the text using character representation utf-8, writing the file to dataset <user>.yrsa.data

Notice ISPF browse primary command `==> display utf8`, which makes utf-8 characters readable.

Example 30.2

```
)action infile='yrsa'
)&      func='<utf8'
  unit.1
)endaction
```

The example assumes input being in character representation utf-8, converting it to ebcdic. The result is a re-creation of the original file from example 30.1 above.

The following transformations exists:

`func='>utf8'`	Transforms ebcdic characters to utf-8
`func='<utf8'`	Transforms utf-8 characters to ebcdic
`func='>ascii'`	Transforms ebcdic characters to ascii
`func='<ascii'`	Transforms ascii characters to ebcdic

Character transformation is mostly used when - but not limited to - communicating to and from UNIX and PC.

31. imbed='xx' and other ways of calling external

Programs external to RXS may be called the following way:

- Calling REXX and CLIST: `call 'progrnme' 'parm'` will call the REXX program `progrnme` using the parameter `parm`. Notice the use of quotes: RXS assumes every un-quoted name to be a variable. An alternative is to change address to *tso*, *ispexec* or *attach*, and then call the CLIST or REXX using the call format for these environments.
- Changing address: To use commands in *tso*, *ispexec*, *attach* and any other environment: change address to the environment, and send one or more strings to output, the string(s) containing the commands. See *Section 33h*.
- Calling RXS programs: `call 'rxs' 'progrnme' 'parm'` will call the RXS program `progrnme` using the parameter `parm`. Notice: this same way, a REXX program may call a RXS program. Notice the use of quotes: RXS assumes every un-quoted name to be a variable. The prerequisite for calling is that a RXSLIB library is allocated - see *'Installation of RXS'*. If both calling and called RXS program uses SQL, it is better to use imbed:
- *Imbed* is an alternative way of calling (or imbedding) a RXS program from RXS. The rest of this section will describe imbed:

`)imbed` in RXS works like *copy / include* in other languages. If execution of a RXS program reaches this line

```
)imbed imbed='mymbr'
```

execution continues in the member `'mymbr'`. This member must by found in a dataset allocated to file RXSLIB in the TSO-session. As execution reaches the end of `mymbr`, then the line after the imbed-point in 'main' is executed next. All variables and queues in RXS before the

imbed-point is also known inside `mymbr`. The connection also goes opposite: New values assigned to these values inside `mymbr` are at hand after return to 'main' RXS. The same applies to queues created inside `mymbr`.

imbed

Imbed uses one general order: `imbed` must be assigned the name of the member containing the RXS code to be imbedded.

Imbedded RXS coding must by syntactically correct considered on its own. For example, you cannot write an action block starting with `)action` in main RXS and terminated with `)endaction` in imbedded RXS.

Imbed has two special features:

- `imbed=xvar` do work: If a variable (here `xvar`) is assigned the name of a member, the variable can be used to govern imbed. That is: imbed is solved during execution, not in some pre-processing phase.
- The coding inside the imbed member may itself contain `)imbed`. This chain of imbed may be of any depth.

Executing external code using *imbed* means that the imbedded coding will participate in the same *unit-of-work* as main RXS. Any error or a programmed `exit` will result in a *rollback* of any changes in files, MQSeries queues or DB2 tables.

If a RXS program uses SQL, and the program transfers control to another RXS program, the other program also using SQL, the transfer ought to be done by `)imbed`. Imbed supports any pattern of concurrent SQL cursors in the two RXS programs, and both programs will run in the same unit-of-work.

Using other call interfaces than imbed, the RXS program and the external coding will work in separate *units-of-work*. Meaning that the called coding may *commit* some change, while main RXS makes a *rollback* - or the opposite.

Example 31.1

```
)text out='degree'
  set current degree = 'ANY'
)endtext
)text out='isola'
  set current packageset = 'DSNREXUR'
)endtext
)action in='isola'
)&      func='sql'
  say "Using isolation level UR, that is: no DB2 locks checked"
)endaction
)action in='degree'
)&      func='sql'
  say "Using current degree='any', that is: maximum parallelism "
)endaction
```

The above coding could be written in a member called 'SQLTURBO' in a dataset allocated to file RXSLIB in the ISPF-session, opening up for this coding in another RXS program:

```
)imbed imbed='sqlturbo'
)text out='sqlcoding'
  select name, salary from our.employee
)endtext
)action in='sqlcoding'
)&      func='sql'
  left(name,20) salary
)endaction
```

Executing this RXS program will execute the specified select-call against DB2. No locks will be checked, and - if possible - the DB2 system will split the query in parallel executing parts and thereby speed up execution. These two options are not programmed, but imbedded.

Variables to govern the imbedded coding may be stated anywhere in the RXS coding in the main program above the imbedding. To emphasize that some variables are used inside imbedded coding, you may also use the construct:

Example 31.2

```
)imbed imbed='createit'
)&     object='yrsa'
)&     input='r2d2.c.txt(hugo)'
```

That is, using the notion for stating of general orders to state orders for an imbed.

32. Output: Specific rules

32a. The stdout dataset

If general order `out` or `outfile` is not given a value, strings in the RXS program will be written to *stdout*. This dataset has the following characteristics:

- If the RXS program is written in a dataset using COBOL line numbers (that is columns 1 thru 6 numeric and column 7 not numeric) then *stdout* is created on dataset `<user>.RXS.COBOL`, where `<user>` is actual TSO userident. Created lines in *stdout* will have COBOL line numbers
- Exception: When executing RXS as macro (*Section 37*) output is always created on dataset `<user>.RXS.DATA` and without COBOL line numbers
- If the RXS program is written in a dataset without COBOL line numbers, *stdout* is created on dataset `<user>.RXS.DATA`
- LRECL for *stdout* dataset will be 256 bytes - if all records in output are smaller than 256 bytes. Otherwise LRECL will be 32756 bytes. RECFM will be VB. Exception: RXS.COBOL will have LRECL 80 byte, and RECFM will be FB. Exception two: If `outfunc='sub'` LRECL is 80 byte and RECFM=FB.
- Which *storage group* or *unit* is used in the allocation of *stdout*, is determined by parameters stated at the installation of RXS.
- The dataset used in *stdout* is automatically allocated, using the above rules. If a dataset with the right name and the right DCB parameters exists, it is reused. If not, a new dataset is allocated. Any old dataset with the same name but wrong DCB parameters will be deleted.

32b. General order outfile: changing the name of the stdout dataset

If `outfile` is assigned a value as general order to an action block or text block, this value will be used as the middle qualifier of the name of the *stdout* datset. Example: if `outfile='yrsa'` output will be written to the dataset `<user>.yrsa.data` or `<user>.yrsa.cobol` according to the rules in *Section 32a*.

`outfile` will be inherited to any action or text block contained in the block on which it is stated. Otherwise `outfile` is local, and accordingly a RXS program may use different values for `outfile` in different blocks.

32c. General order out: state the output dataset or output queue

If `out` is assigned a value as general order to an action block or text block, and if the value contains at least one period, the value is presumed to be the name of an existing dataset. If the presumption turns out to be wrong, execution of RXS is terminated in error - unless this exception

is catched by a `)notrigger` clause. If the presumption is right, output is written to this dataset. Record format for the dataset may be FB or VB, and LRECL (logical record length) can have any value.

If the value of `out` does not contain a period, the value is interpreted as the name of an internal queue. Writing on queues is immediate, that is, another action block in the same RXS program may read the records that are written. Several action blocks in a RXS program may write to the same queue, this will not overwrite previously written records.

The value for `out` will be inherited to any action or text block contained in the block on which it is stated. Otherwise `out` is local, and accordingly a RXS program may use different values for `out` in different blocks. Therefore a RXS program may write any number of datasets and members.

To replace an inherited value for `out` by *stdout*, assign `out = "-"` in the action block.

If both `out` and `outfile` are assigned values for an action or text block, `out` will work, `outfile` will be ignored.

If `out` contains a file name having last qualifier equal to COBOL then output will be generated with COBOL numbering, unless the written lines from the RXS program all are numeric in columns 1 thru 6, or all are equal to spaces in columns 1 thru 6.

32d. Writing members

`out` may point to a member of a partitioned dataset. Example `out='myqualif.mydsn(mymbr)'`.

ISPF-statistics for the member will be updated. As *userid* in ISPF statistics is assigned 'RXS'. If the member does not exist, it is created.

32e. Outfunc in a situation with several action blocks or text blocks

`outfunc` indicates a terminating action when output is created (*Section 8*).

If `outfunc` is 'edit', 'browse' or 'view' it will be inherited to any action or text block contained in the block on which it is stated. Otherwise `outfunc` is local, and accordingly a RXS program may use different values for `outfunc` in different blocks. But notice: two action blocks writing to the same dataset cannot use two different `outfunc`. Last stated `outfunc` will be activated. This is related to the fact that you cannot access the same dataset in , say, *edit* and *browse* at the same time.

32f. Setting global values for stdout

Any RXS program may be put inside a text block by writing `)text` above the program and `)endtext` below. This does not change anything. But by stating general orders to such a text block, default values for `out`, `outfunc` and `outfile` may be changed for the whole RXS program.

32g. Commit, rollback: when is writing done?

The physical writing of any output dataset from RXS is postponed to the termination of the RXS program. Any content in the dataset prior to the execution of the RXS program is overwritten. More than one action block in a RXS program may write to the same dataset - this will not overwrite records previously written from other action blocks in RXS. Reading an `out` dataset from RXS will read what was initially on the dataset - records just written from RXS cannot be read until the RXS program has terminated. If the RXS program does not reach *commit*, initial content of the dataset is preserved, and nothing is written. The following situations means that a commit is not reached:

- if the program ends in error

- if the RXS program uses dialogues (using `func='prompt'`) and the user leaves the program reversing backwards through the first window of the dialogue (according to *seciton21a*)
- if the RXS program terminates in a programmed `exit` or `return`.

33. Address: Special interpretation of output

33a. Changing address

The RXS general order `address='xxx'` will change the basic behaviour of RXS: strings are no longer sent to *stdout*, but are sent to the environment xxx stated by `address`.

`address` is local for an action block.

Default addressing in RXS is therefore `address='stdout'` meaning that strings are sent to the environment *stdout*.

The environment *stdout* is the normal handler of (sequential) output from RXS. *Stdout* normally writes strings created by RXS to the dataset RXS.DATA, but the behaviour of *stdout* can be modified - see *section 4* and *section 7*.

Any environment that can be addressed in REXX can be addressed in RXS. Below are some often used environments for RXS:

33b. Addressing ISPEXEC

`address='ispexec'` indicates that all strings from this action block are handled over to *ispexec* to be interpreted as orders for ISPF.

If an addressed command gives a return code, RC > 11, then RXS is terminated in error. If the addressing in this situation is ispexec or isredit then the error message in the addressed system will be displayed.

Example 33.4:

```
)action address='ispexec'
   "display panel(mypanel) cursor(myfld)"
)endaction
```

The example uses ISPF for displaying a window. In case of errors in the ISPF display, an ISPF error message will be shown.

33c. Addressing UNIX

`address='unix'` will direct strings in the action block to UNIX for execution.

Output from such commands is written to stdout for the action block.

Errors from such commands is written on the screen. Errors will halt the RXS program with RC = 20

Example 33.5:

```
)action address='unix'
   "cd /home/r2d2/mess"
   "cksum myfile.rxs>cksumfile.txt"
)endaction
```

The example changes the actual directory in UNIX, and executes the UNIX command cksum against a file on the actual directory.

Example 33.6:

```
)action address='unix'
)&       out='q1'
   "ls /home/r2d2/"
)endaction
)action in='q1'
  word.1
)endaction
```

The example list all files in the directory /home/r2d2/. The listing of the directory will be written on mvs dataset RXS.DATA

33d. Addressing Java via UNIX

Example 33.7:

Zipping a unix file om manframe:

```
)action address='unix'
  "cd /main_dir/our_dir/      "
  "jar cfv hovsa.zip r2d2.txt "
)endaction
```

The example zips the file /main_dir/our_dir/r2d2.txt to the zip-archive hovsa.zip which is created on the same directory. More than one file may be zipped:

```
  "jar cfv hovsa.zip yrsa.txt r2d2.txt "
```

Example 33.8:

UNZIP a zip.archive to a unix file om manframe:

```
)action address='unix'
  "cd /main_dir/our_dir/      "
  "jar xfv hovsa.zip "
)endaction
```

The original file(s) are extracted from the zip-archive hovsa.zip on the directory /main_dir/our_dir. The file(s) are placed on the same directory

33e. Addressing TSO

`address='tso'` will direct strings in the action block over to tso for execution.
Output from such commands is is written to stdout for the action block.
If a tso command sets a return code 8 or more, the RXS program is halted.

33f. Communicating to a remote system by FTP

Getting an unix-file from remote:

Example 33.9:

```
)action address='tso'
   queue "R2D2"
   queue "is_secrt"
   queue "binary"
   queue "lcd /home/R2D2"
   queue "cd /home/Stranger"
   queue "get lyrics.txt (replace "
   queue "quit"
   "FTP EXMACHINE.REMOTE.COM"
)endaction
```

The example performs an FTP transport of unit-file lyrics.txt from unix-directory home/stranger on EXMACHINE.REMOTE.COM over to /home/R2D2 on the local mainframe. User R2D2 with password is_secrt is authenticating the transport.

Note the use of instruction 'queue' to set op a list of answers to the questions we know that the FTP tso command is going to ask.

Putting a member from a MVS partitioned dataset or library to remote:

Example 33.10:

```
)action
  queue "R2D2"
  queue "is_secrt"
  queue "lcd 'ourqual.ourlib.cntl'
  queue "cd 'remqual.theirlib.cntl'"
  queue "put killroy "
  queue "quit"
  )action address='tso'
  )&      out='ftp_mess'
    "FTP EXMACHINE.REMOTE.COM"
  )endaction
)endaction
)action in='ftp_mess'
)&   errorc=1
  select
    when word.1 = 'EZA2644I' then do
      errorc = 2
      say substr(unit.1,9)
    end
    when word.1 = 'EZA2836I' then do
      say substr(unit.1,9)
      errorc = 2
    end
    when word.1 = 'EZA1684W' then do
      say substr(unit.1,9)
      errorc = 2
    end
    when word.1 = 'EZA1617I' then do
      say substr(unit.1,9)
      errorc = ''
    end
    otherwise nop
  end
)endaction
)action
  select
    when errorc = 2 then do
      say 'FTP fails'
      exit 16
    end
    when errorc = 1 then do
      say 'FTP fails. Probably wrong password on extern, or',
      'wrong filename on extern'
      exit 16
    end
    otherwise say 'FTP was a success'
  end
)endaction
```

The example performs an FTP transport of member 'killroy' from dataset 'remqual.theirlib.cntl' on external mainframe EXMCHINE.REMOTE.COM . The data will be recived on 'ourqual.ourlib.cntl(killroy)'. User R2D2 with password is_secrt is authenticating the transport.

Note the use of instruction 'queue' to set op a list of answers to the questions we know that the FTP tso command is going to ask.

The RXS program performs an analysis on the output from FTP to verify whether the transfer succeeded.

34. Scope of variables

The only pre-processing phase in the execution of a RXS program is a scan through the program to find all user defined variables:

All user-defined variables are made global, that is, all action blocks shares a common definition of the data. A variable is user defined if the variable exists in the RXS coding and if the variable is not a general order or output from a general order.

Output variables `word.x` and `unit.x` and variables carrying output from `'sql'`, `'namespace'` and `'prompt'` input, plus variables created in imbedded coding, are visible 'downwards' in the RXS program: The variables are visible in the action block where they are created plus all action blocks contained in - or imbedded in - this action block. But do notice that `word.x` and `unit.x` are given new content whenever a new action block using default *func* is entered during execution of the RXS program.

Variables created dynamically using the REXX `interpret` command are local. Using the RXS instruction `make_global 'varname'` to make such a variable visible in contained or imbedded action blocks.

RXS queues are always global.

General orders `out`, `outfunc`, `outfile` are local for the action block or text block at which they are stated, and are visible in blocks contained in - or imbedded in - this action block.

Remaining general orders `in`, `func`, `prompt`, `imbed`, `caps` etc., are local for the action block.

Example 34.1

If a general order is to receive an assignment prior to the execution of the action block, this can be accomplished using a user defined variable. All user defined variables are global.

This example uses variable w_outfunc to transport an assignment into the inner action block:

```
)action
   w_outfunc = word('browse view', random(1, 2))
   )action outfunc=w_outfunc
      "What's up doc?"
   )endaction
)endaction
```

About half the times this program is executed, the user will end in *browse* on output, about half the time in *view*.

The reason for these rather uneven principles is the possible use of RXS for code generation. A COBOL program using RXS code generation normally consists of separate islands of RXS code, separated by sequences of normal COBOL code ('dead code' as seen from the RXS program). This is because you do not generate the whole COBOL coding; but only the parts of the program that is to reflect some specification file. These separate islands of RXS code must be able to communicate, therefore the use of global variables. A schema of clean inheritance of variables will not do.

34a. 'Signal on novalue'

A variable in RXS is not to be referenced before it is assigned a value. Violating this rule will cause the program to end in error.

This strict rule helps finding typing errors in the program. It also helps finding errors caused by referencing the variable in an action block outside the variable's scope.

Example 33.3:

When writing:

```
)action
   do 5
```

```
      if x = 'x' then x = 0
      x = x + 1
      say x
   end
)endaction
```

the RXS program will end in error, giving an error message in line 3: `"x has no value"`.

To see if a variable has a value or not, use the following logic:

```
)action
   if symbol('x') = 'LIT' then do
      say 'x does not contain a value'
   end
)endaction
```

Notice: the variable name `x` used in `symbol` is quoted - elsewhere we would find ourselves back in the tarpit again, with the program making an immediate end saying `"x has no value"`.

35. Execution RXS as TSO commands and from REXX

a) Handling RXS programs located in allocated libraries:

RXS programs can be be executed as TSO commands if they reside in an allocated library: The TSO session must allocate the file `RXSLIB`, pointing to one or more partitioned datasets containing the RXS programs. Such a dataset may have any attributes, but RECFM=VB and LRECL=255 is recommended.

- A RXS program may be executed from any command line in ISPF by writing:
  `tso rxs myrxs`
  provided that the RXS program `myrxs` resides in the RXSLIB library.
- The RXS program may use parameters:
  `tso rxs myrxs what's up doc?`
  (*Section 03)*
- If the program terminates in error, the error message plus line number for the error will be written on screen. TSO will receive return code RC=8.
- Any other return code may be given in a programmed exit. If you program:
  `exit 20`
  in the RXS program, the program will terminate in error, and TSO will receive return code RC=20.

A RXS program can be executed from a REXX program:

Example 35.1

```
/* REXX */
XVAR = 'Killroy was here'
ADDRESS TSO
"RXS YRSA "XVAR
IF RC > 5 THEN DO
  SAY "'YRSA' set a return code" RC
END
```

The example calls an RXS named YRSA. YRSA will receive the string 'Killroy was here' in its variabel RXSPARM. The REXX program will receive a RC = nn if the RXS programs issues an EXIT nn.

b) Handling RXS programs located anywhere:

A RXS program may be executed from any commando line in ISPF by writing:

```
tso rxs 'ourgroup.ourlib.type(myrxs)'
```

That is, naming file and member name in normal ISPF syntax.

Executing a RXS program this way changes the behaviour of)IMBED: the search for imbed'ed RXS program is done solely in the indicated dataset

Naming both dataset and member is relevant when executiong a RXS program form a REXX program:

Example 35.2

```
/* REXX */
XVAR = 'Killroy was here'
ADDRESS TSO
"RXS 'OURGROUP.OURLIB.RXS(YRSA)' "XVAR
IF RC > 5 THEN DO
  SAY "'YRSA' set a return code" RC
END
```

Calling RXS this way, the call will function regardless of the allocations for the ISPF session.

c) Handling RXS programs when shown in an extended member-list:

Writing RXS in the command field of a member list will execute the indicated member as a RXS program.

36. Execution in background (JCL)

To execute a RXS program in the background, make the following allocations in JCL:

- DD name `//SYSPROC` must point to a library containing the REXX program RXS
- DD name `//ISPLLIB` must point to a library containing the load module RXSDO (If *IBM language environment* is not a default allocation on your installation, the library `*.adcycle.le370.sceerun` must be allocated too)
- Remaining DD names for the execution of ISPF in the background must be found in the JCL (`//ISPPROF`, `//ISPPLIB`, `//ISPSLIB`, `//ISPMLIB` and `//ISPTLIB`). Allocate `//ISPPROF` as in the example below, and use the allocations normally used at your installation for the other DD names.
- DD name `//SYSTSIN` must contain the string:
  `ISPSTART CMD(RXS)`
- DD-name `//RXSPGM` must point to a dataset or member containg the program to be executed. Alternatively the program is stated inline as in the example below.
- If general order `out` is given a value in the RXS program, this will function as usual: if the dataset exists, RXS will write to it. No JCL allocation of the dataset is needed.
- If the RXS program writes to standard output, stdout, writing will take place in DD name `//RXS`. If no such DD name exists in the JCL, RXS will allocate a dataset according to its normal rules (*section* 32a).
- If general order `outfile` is used in the RXS program, writing will take place in a file having this same name in the JCL. Example: If an action block writes to `outfile='yrsa'`, then DD name `//YRSA` will be used in the writing. If no such DD name exists in JCL, RXS will allocate a dataset according to its normal rules (*section* 32a).
- If general order `infile` is used in the RXS program, reading will take place from a file having this same name in the JCL
- If general order `func='prompt'` is used in the RXS program, DD name `//PROMPT` must be found in the JCL. Assignments for the variables in the prompt must be stated under this DD name - one assignment per line. The syntax `variablename(assignment)` must be used (See *Example 36.2*)

- If general order `prompt='xyz'` is used in the RXS program, the DD name `//XYZ` must be found in the JCL. Assignments for the variables in the prompt must be stated under this DD name - one assignment per line. The syntax `variablename(assignment)` must be used
- If `)imbed` is used, DD name `//RXSLIB` must point to a library (partitioned dataset) containing the RXS chunks to be imbedded
- Error messages from RXS are written to DD name `//SYSTSPRT`. The RXS step in error is terminated with return code RC=16. A message on screen notify the user that //SYSTSPRT should be checked
- A programmed exit is propagated up to the background job: When programming, say, `exit 12` in RXS, the job will terminate with return code RC=12.

Example 36.1:

```
//PROFALL EXEC PGM=IEFBR14
//PROFDSN DD DSN=&&PROFIL,DISP=(NEW,PASS),UNIT=VIO,
//         DCB=(BLKSIZE=6080,LRECL=80,RECFM=FB,DSORG=PO),
//         SPACE=(TRK,(1,1,1))
//*
//RXSSTEP EXEC PGM=IKJEFT1B,DYNAMNBR=30
//SYSPROC DD DSN=MYQUALIF.MYPROC,DISP=SHR
//ISPPROF DD DSN=&&PROFIL,DISP=OLD,UNIT=SYSDA
//ISPLLIB DD DSN=??.ADCYCLE.LE370.SCEERUN,DISP=SHR
//         DD DSN=MYQUALIF.ISPLLIB,DISP=SHR
//ISPTLIB DD DSN=??
//ISPMLIB DD DSN=??
//ISPSLIB DD DSN=??
//ISPPLIB DD DSN=??
//ISPLOG  DD DUMMY
//SYSTSPRT DD SYSOUT=T
//SYSTSIN  DD  *
 ISPSTART CMD(RXS)
//RXSPGM DD *
)action
   nbr = random()
   'Square of 'nbr' is 'nbr**2
)endaction
//RXSLIB DD DSN=MYQUALIF.RXSLIB,DISP=SHR
//RXS     DD DSN=MYQUALIF.MY.OUTPUT,DISP=(NEW,...
```

Submitting this JCL will create the dataset MYQUALIF.MY.OUTPUT consisting one line:
`Square of 117 is 13689`
or whatever the random number is.

`//ISPMLIB` and `//ISPSLIB` is not used by RXS, but they are needed to start ISPF in the background

In RXS everything will function normally, except:

- `outfunc = 'edit'` or `'browse'` or `'view'` is ignored
- The command `say` writes to `//SYSTSPRT` - not to the screen.

Notice: Even reading from and writing to a PC dataset works in the background: A background job on the mainframe may read or write to any PC, if permission is granted by the 'IBM workstation agent' program on the PC. According to *section 28*.

A debugging tip: Marking the RXS program in the JCL above using line commands `cc` `cc` and entering `==> rxs` in the command prompt, will execute the RXS program directly.

Warning:

'/*' in column 1 in JCL unfortunalely marks the end of a SYSIN-dataset. Therefore comments in the RXS program having /* in column 1 will terminate the reading of the program.

Example 36.2:
Execution of the RXS-program 'myrxs' from JCL. The program resides as a member on the dataset MYQUALIF.RXSLIB. The program 'myrxs' uses an action block using `func='prompt'` to get values for *account* and *department*. ('myrxs' could be the RXS program in example 21.3).

```
//PROFALL EXEC PGM=IEFBR14
//PROFDSN DD DSN=&&PROFIL,DISP=(NEW,PASS),UNIT=VIO,
//         DCB=(BLKSIZE=6080,LRECL=80,RECFM=FB,DSORG=PO),
//         SPACE=(TRK,(1,1,1))
//*
//RXSSTEP EXEC PGM=IKJEFT1B,DYNAMNBR=30
//SYSPROC DD DSN=MYQUALIF.MYPROC,DISP=SHR
//ISPPROF DD DSN=&&PROFIL,DISP=OLD,UNIT=SYSDA
//ISPLLIB DD DSN=MYQUALIF.ISPLLIB,DISP=SHR
//ISPTLIB DD DSN=??
//ISPMLIB DD DSN=??
//ISPSLIB DD DSN=??
//ISPPLIB DD DSN=??
//ISPLOG  DD DUMMY
//SYSTSPRT DD SYSOUT=T
//SYSTSIN  DD  *
 ISPSTART CMD(RXS MYRXS)
//RXSLIB DD DSN=MYQUALIF.RXSLIB,DISP=SHR
//RXS     DD DSN=MYQUALIF.MY.OUTPUT,DISP=(NEW,...
//PROMPT DD *
ACCOUNT(1448)
DEPARTMENT(SALES)
```

This last example is omitting `//RXSPGM` and pointing `//SYSTSIN` to
`ISPSTART CMD(RXS MYRXS)`
This way RXS programs using parm-strings may be executed in background:
`ISPSTART CMD(RXS OURRXS what's up doc?)`
will execute the command `OURRXS` using parameter `what's up doc?`

Example 36.3
Building a JCL-procedure:
Due to an error in ISPF, running multiple instances of RXS in background will collide when using //ISPTLIB. The solution is this modificated JCL:This is the way to build a JCL procedure for background execution of RXS:

```
//COPYTLIB EXEC PGM=IEBCOPY
//SYSUT1 DD DSN=SYS2.DCISPF.ISPTLIB,DISP=SHR
//SYSUT2 DD DSN=&&ISPTLIB,DISP=(NEW,PASS),UNIT=VIO,
//            DCB=(BLKSIZE=6080,LRECL=80,RECFM=FB,DSORG=PO),
//            SPACE=(TRK,(3,3,3))
//SYSPRINT DD SYSOUT=T
//SYSIN DD *
COPY OUTDD=SYSUT2,INDD=SYSUT1
SELECT MEMBER=(ISPCMDS,ISPFCMDS,ISPKEYS,ISPPROF,ISPSPROF)
//*
//RXSBAGAL EXEC PGM=IEFBR14
//DSN1     DD DSN=&&PROFIL,DISP=(NEW,PASS),UNIT=VIO,
//            DCB=(BLKSIZE=6080,LRECL=80,RECFM=FB,DSORG=PO),
//            SPACE=(TRK,(1,1,1))
//*
//RXSDO    EXEC PGM=IKJEFT1B,DYNAMNBR=30
//SYSPROC  DD DSN=SDBNYSL.PROJ.CLIST,DISP=SHR
```

```
//ISPPROF  DD DSN=*.RXSBAGAL.DSN1,DISP=OLD
//ISPMLIB  DD DSN=SYS2.DCISPF.ISPMLIB,DISP=SHR
//ISPSLIB  DD DSN=SYS2.DCISPF.ISPSLIB,DISP=SHR
//ISPPLIB  DD DSN=SYS2.DCISPF.ISPPLIB,DISP=SHR
//ISPLLIB  DD DSN=SYS2.ADCYCLE.LE370.SCEERUN,DISP=SHR
//ISPLLIB  DD DSN=SDBISPF.BSPF.LOAD,DISP=SHR
//ISPTLIB  DD DSN=&&ISPTLIB,DISP=SHR
//ISPLOG   DD DUMMY
//RXSLIB   DD DSN=SDBNYSL.PROJ.RXSLIB,DISP=SHR
//SYSTSIN  DD *
ISPSTART CMD(RXS)
//SYSTSPRT DD SYSOUT=T
//SYSUDUMP DD SYSOUT=T
```

//RXSPGM is to be added when calling the JCL procedure.

Notice: handling large data structures in RXS consumes large amounts of memory. Using REGION=0K in the JOB-card is recommended, as it will maximise the possible amount of high memory.

37. Writing ISPF edit macros

Consider the following situation: When editing a dataset in ISPF-edit, you want to execute the RXS program *myrxs*. *Myrxs* exists on the ISPRLIB library in the TSO session.

Writing in the command line:

```
==> tso rxs myrxs
```

and pressing enter, the program *myrxs* is executed

Writing

```
==> rxs |myrxs
```

and pressing enter, the program *myrxs* is executed too. Execution this way opens the following possibilities in myrxs:

In the program you may read the queue `edit_screen`. This queue contains all lines in the edit dataset on the screen - in the state as currently seen on the screen. COBOL line numbers are ignored; linie-numbers in col 73-80 are ignored if record-length is 80.

The following special situations exists:

- If a block of lines on the edit screen has been marked using line commands 'cc' marking start and end of the block, `edit_screen` will only contain the lines in the marked block.
- If a line command `a` or `b` is stated somewhere on the edit screen, it will have these effects:
  - the queue `edit_screen` will by empty
  - *stdout* will write on the screen after or before the stated position, unless `out` or `outfile` is specified
- If the edit screen is empty, it will have these effects:
  - the queue `edit_screen` will by empty
  - *stdout* will write on the screen, unless `out` or `outfile` is specified
- Otherwise, *stdout* will function as normal

Format and blocksize for the *stdout* dataset will be copied from the dataset underlying the edit session

Due to an oddity in REXX (that is in fact, an oddity in EBCDIC) the | above is to be replaced by an ! when using a keyboard from a nordic country, France, Germany, Austria and Italy

Example 37.1: myrxs contains:

```
)action in='edit_screen'
)&      start=1
  if start = 1 then do
    xx = change('yrsa', 'hugo', unit.1, 'first')
    xx
    if xx <> unit.1 then start = 0 /* if a change has occured */
  end
  else do
    unit.1
  end
)endaction
```

If command ==> rxs |myrxs is fired, the first occurence of the string 'yrsa' is changed to 'hugo'. The changed dataset is written to <userid>.rxs.data

Example 37.2: myrxs contains:

```
)action in='edit_screen'
)&      func='sql'
  sqlvalues
)endaction
```

If the edit screen contains some SQL, writing ==> `rxs |myrxs` will execute this SQL. If the execution creates output, the rows will be shown in a new edit screen.

Example 37.3

```
)action in='ourqualif.ourdsn' /* a partitioned dataset */
)&      out='q1'
   mbr = unit.1
  "edit dataset('ourqualif.ourdsn("mbr")') macro(mymacro)"
  )action in='q1'
  )&      address='ispexec'
    unit.1
  )endaction
  dropqueue('q1')
)endaction
```

ISPF macro 'mymacro' is executed on all members in dataset 'ourqualif.ourdsn'.

'Mymacro' could be:

```
/* REXX */
address isredit
"macro"
"change 'yrsa' 'hugo' first"
"end"
```

Resulting in a change of 'yrsa' to 'hugo' in first occurence in each member.'Mymacro' must be a member on a dataset allocated to SYSPROC in the ISPF session.

Example 37.4

```
)action in='edit_screen'
)&      cnt=0
  if pos(' IF ', unit.1) > 0 then do
    cnt = cnt + 1
    cnt unit.1
  end
  if pos('END-IF', unit.1) > 0 then do
    cnt = cnt - 1
    cnt unit.1
  end
  if pos('SECTION', unit.1) > 0 then do
    unit.1
  end
)endaction
```

This coding will trace any un-balance in IF / END-IF in a COBOL program.

38. Reserved names

All variable names starting with the three characters `rx_` are reserved for internal use in RXS. Failure is raised if such a name is used in a RXS program.

If RXS uses DB2, the IBM DB2 REXX interface is active. This interface reserves the following names for internal use: All names starting with `SQL`, `RDI`, `DSN`, `RXSQL` and `QRW`, plus names `C1` to `C100`. The reservation applies only to the action block which uses `func='sql'`. No problem using these names is ever reported, and accordingly RXS raises no failure if they are used.

The variable names mentioned in this paper as being general orders, and mentioned for being variables for communicating the result of general orders, of course are reserved: they have a special meaning in the context in which they are used. Otherwise they are not reserved. For instance, if an action block does not access MQSeries, the variable named `mq` is not reserved. The only variable in RXS that have its special meaning in every context is `cont`. Accordingly `cont` cannot by used for any other purpose.

Syntactically reserved words like `if`, `else`, `return` may - but should not - be used as variable names. The same applies to all function and instruction names.

The index of 'RXS documentation (the PDF-document) contains references to all variables with special meaning in RXS.

39.)interface

`)interface` halts the RXS program and starts an edit-session (or browse session) on the indicated internal RXS queue. When the user presses F3 (end) in the edit-session, the RXS program is resumed.

`in`

General order `in` names the internal RXS queue to be presented in the ISPF-editor in the user interface. The queue may or may not exist prior to beeing presented in `)interface`. If the queue does not exist, the user will be shown an empty edit screen.

`)interface` is a way of getting complex input from the user. This might for instance be used in a RXS programming of an editor for some special resources: a MQ-queue, a SQL-table etc.

`)interface` may also be used for debugging by showing the content of internal queues during the execution of the RXS program.

If a queue contains both `unit.1` and `unit.2`, then `)interface` will in browse, not edit, show both elements of the queue, separated by ##.

`interface(q_name)`

The function `interface(q_name)` using a queue-name as its argument, returns '1' if the indicated queue `q_name` has been changed by the user in an `)interface` session in the current RXS program.

Example 39.1

```
)action in='ourgrp.thisque'
)&      func='mqdrain'
)&      out='q1'
  unit.1
)endaction
)interface in='q1'
)action in='q1'
```

```
)&       outfunc='mqput'
)&       out='ourgrp.thisqueue'
  unit.1
)endaction
```

This is a very primitive editor for a MQSeries queue. 'ourgrp.thisque' is a MQSeries queue. The messages of the queue is presented in an edit-session, one line per message, and the user may alter, delete or add messages. When pressing F3, these messages are written back to the MQSeries queue.

40. Functions and instructions in RXS

The definition of the two terms in the heading just above is:

- A *function* replaces itself with the value it creates.
- An *instruction* is an executable line in RXS.

The notation in this appendix is:

Syntax is described using typeset `courier`. Required elements are **`bold`**, optional are `not bold`. If an optional element is omitted, the comma in front of it is to be omitted too.

Elements written using `UPPER CASE` must be written exactly as stated here (you may use lower case), elements written using `lower case` must be replaced by a string or a variable containing a value.

Do notice: non-numeric strings in RXS must always be written in quotes.

If nothing is noted below, the concept is a function, and its heritage is REXX. Which means that further information can be found in a REXX manual.

Functions in RXS can be nested - for instance `left(date(),2)` will create a two digit string containing the current day in the month.

The list below is not exhaustive - any REXX functions and instructions may be used in RXS, except those mentioned in *Section 2*.

`ABS(number)`
ABS returns the absolute value of a number (stripping of the sign, returning a positive number or zero)

`BITAND(string1,string2,pad)`
BITAND returns a string containing the two input strings ANDed together bit for bit.

`BITOR(string1,string2,pad)`
BITOR returns a string containing the two input strings logically ORed together bit for bit.

`BITXOR (string1,string2,pad)`
BITXOR returns a string containing the two input strings logically exclusive ORed together bit for bit.

`B2X(binary_string)`
Binary to hexadecimal - returns a string in character format, representing *binary_string* converted to hexadecimal.

`CALL extprocedure` `parameter`
The external REXX or CLIST program *extprocedure* is executed, optionally using the parameter *parameter*. (REXX instruction)

`CENTER(string,length,pad)`
CENTER returns a string of length *length* containing *string* centered inside it. *pad* characters may be added to reach the length.

`CHANGE(oldval,newval,string,``option`**`)`**
CHANGE returns a *string* in which first, last or all occurences of another string, *oldval* is changed to the string *newval*. Option is F(irst) L(ast) or A(ll). A is default. (RXS function)

`COMPARE(string1,string2,``pad`**`)`**

COMPARE compares *string1* and *string2*. COMPARE returns 0 if the strings are identical. If the strings differ, the position of the first character not in match is returned.

COPIES(string,n)

COPIES returns *n* concatenated copies of *string*.

C2D(string,n)

Character to decimal - returns the decimal value of the binary representation of *string*. This function converts an IBM 'binary' field to a REXX numeric - for example, if *string* contains '05A8'x, the REXX numeric '1448' is created.

C2X(string)

Character to hexadecimal - converts a string to its hexadecimal representation. This function conterts an IBM 'packed decimal' to a REXX numeric - for example, if string contains '01448D'x, the REXX string '01448D' is created. If the last character is 'D' then multiply by -1. Remove last character.

DATATYPE(string,type)

DATATYPE returns - when only *string* is specified - NUM if *string* is a valid REXX number. Otherwise CHAR is returned. If *type* is specified, 1 is returned if *string* matches *type*, otherwise 0 is returned.

DATE(option)

DATE returns the actual date in the format *dd mmm yyyy* (if *option* is omitted) or in the format according to *option*. *Option* can be Base, Century, Days (number of day inside year), European (the format 13/03/92) , Julian, Month, Normal (the format 13 Mar 1992), Ordered, Standard (the format 19920313), Usa, Weekday (the day of the week in letters). Only the first letter of *option* has to be written.

DELSTR(string,n,length)

DELSTR deletes the substring of *string* starting at the *n*'th character and being *length* long.

DELWORD(string,n,length)

DELWORD deletes the substring of *string* starting in the *n*'th word, being *length* blank-delimited words long.

DO

Lines in RXS contained inside lines DO and END are considered a block of lines (*section 2a*) (REXX instruction)

DROPQUEUE queue

DROPQUEUE removes the queue *queue*. (See *Section 16*). (RXS instruction).

DROP name

DROP restores variables to their original uninitialized state. If *name* is not enclosed in parentheses, it identifies a variable to drop. If a single name is enclosed in parentheses, then the value of name denotes a subsidiary list of variables to drop. (REXX instruction)

D2C(wholenumber,n)

Decimal to character - returns a character-string being the binary representation of the decimal number *wholenumber*.

D2X(wholenumber,n)

Decimal to hexadecimal - returns a character-string being the hexadecimal representation of the decimal number *wholenumber*.

END

Lines in RXS contained inside lines DO and END are considered a block of lines (*section 2a*) (REXX instruction)

EXIT number

The RXS program is halted immediate. If *number* is written the environment of the RXS program will receive number as return-code (normally in the variable RC). Any updates from the RXS program (DB2, MQ, and writing of files) will be rolled back. If the RXS program is exe-

cuted directly from an edit-screen or as a command, an informative message is written. (RXS instruction)

FIND(string,phrase)

FIND searches *string* to find first appearance of *phrase* (where phrase is a string of blank-delimited words), returning the number of the word in *string* where the appearance starts. If *phrase* is not found, or *phrase* is empty, 0 is returned.

FORMAT(number,`before,after,expp,expt`**)**

FORMAT rounds and format *number* according to the stated: *Before* states the number of digits before the decimal separator, *after* states the number of digits after the decimal separator.

FROMISPF(dsname)

Converting a dsname from ISPF naming standard to RXS naming standard: If *dsname* begins with a quote, qutoes before and after is removed. If no quote is found, dsname is prefixed `USERID()"."`. (RXS function).

GETQUEUE(queue_name, element_value)

GETQUEUE returns the value of unit.2 in the queue *queue_name* for the entry having unit.1 = *element_value* (See *Section 15*). (RXS function)

INDEX(haystack,needle,`start`**)**

INDEX returns the position of a string, *needle*, in another string *haystack*, starting the examination at *start*. If the string *needle* is not found, 0 is returned.

INSERT(new,target,`n,length,pad`**)**

INSERT inserts the string *new*, padded up to length *length*, into the string *target* starting at character *n*.

INTERPRET expression

INTERPRET executes *expression*: a string or a variable containing a valid statement in REXX syntax, or several valid statements, separated by ";". Example: `interpret "if w = 14 then do;w = w - 1;end"` Notice that RXS constructs is not allowed inside *expression*. (REXX instruction)

INTERFACE(in)

The function INTERFACE() using a queue-name *in* as argument, returns '1' if the indicated queue *in* has been changed by the user during an edit session using an `)interface` in the current RXS program (RXS function)

ITERATE

Jump to the beginning of the current block of coding (DO END block) and execute from here. (REXX instruction)

JUSTIFY(string,length,`pad`**)**

JUSTIFY formats blank-delimited words in *string* by adding *pad* characters between the words so that the words fill out *length*.

LASTPOS(needle,haystack,`start`**)**

LASTPOS returns the last position of a string, *needle*, in another string *haystack*, starting the examination at *start*. If *needle* is empty or if *needle* is not found inside *haystack*, 0 is returned.

LEAVE

Jump past the end of the current block of coding (DO END block) and execute from here (REXX instruction).

LEFT(string,length,`pad`**)**

LEFT returns a string containing characters from the left of *string* up to length *length*.

LENGTH(string)

LENGTH returns the length of *string*.

MAKE_GLOBAL varname

The variable with the name *varname* is made globally accessible. See *Section 33*. (RXS instruction)

MAX(number1,number2,...)

MAX returns the largest number in the list

MIN(number1,number2,...)

MIN returns the smallest number in the list.

NOP

Dummy instruction with no effect. (REXX instruction)

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

OVERLAY overlays *tartget* - starting at the *n*'th character - with the string *new*, padded and truncated to length *length*.

PARSE VAR string varname1 "," varname2 "," varname3

PARSE comes in a lot of flavors. The above form splits a *string* into three strings. The split occurs when the character ',' is found in *string*. For other uses of PARSE, see a REXX manual

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

POS returns the first position of a string, *needle*, in another string *haystack*. The examination starts at position *start*.

QUEUE string

Concatenate *string* at buttom of the current queue. The current queue is normally used to hold sub-commands for tso commands using several sub-commands, like FTP (REXX instruction) (*section 33j)*

QUEUEVAR(queue_name, queue_element)

Queuevar returns 1 if *queue_element* is found in the queue *queue_name*, otherwise 0 is returned (See *Section 14*). (RXS function)

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

RANDOM returns a pseudo random non-negative number in the sequence from *min* to *max* inclusive. A specific *seed* for the generation may be given.

REVERSE(string)

REVERSE returns the bytes of *string* in reverse order.

RETURN

If the RXS program contains a series of prompts (*Section* 21) the first prompt will re-appear. Otherwise: The RXS program is halted immediate. Any updates from the RXS program (DB2, MQ, and writing of files) will be rolled back. (RXS instruction)

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

RIGHT returns a string of length *length*, including the rightmost character of *string*

SAY expression

expression is written on screen (REXX instruction)

SET_HALT string

string is presented as a message on the current screen presented to the user, and the execution is temporarely halted - the current screen in a prompt is re-displayed. (RXS instruction).

SET_MESSAGE string

string is presented as a message on the next screen presented to the user. (RXS instruction).

SIGN(number)

If number is negative, -1 is returned, if number is zero, 0 is returned, otherwise 1 is returned.

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

SPACE formats blank-delimited words in *string* using *n* pad characters between each word. If *n* is zero, all blanks are removed.

STRIP(string,option,char**)**

STRIP removes Leading, Trailing or Both *char* from *string*, according to *option* being L(eading), T(railing) or B(oth). Default is B. Default for *char* is space.

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

SUBSTR returns the substring af *string* starting at the *n*'th character. If *length* is omitted, the rest of the string is returned.

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

SUBWORD returns the substring of *string* starting at the *n*'th word, and being *length* long.

SYMBOL(name)

If *name* is not a valid REXX symbol, BAD is returned. If *name* is the name of a variable, VAR is returned. Otherwise LIT is returned.

TIME(option**)**

TIME returns local time in the format '14:19:03' if *option* is not stated. *Option* is: Civil '2:19pm', Elapsed '0.000028' (number of seconds after *reset*), Hours '14' (number of hours since midnight), Long'14:09:03.050683', Minutes '859' (minutes since midnight), Normal '14:09:03', Reset (that is: resetting *elapsed*) and Seconds '51535' (Seconds since midnight). As *option* you may state the first letter, for instance 'c' for 'civil'.

TIMESTAMP()

TIMESTAMP returns the current timestamp in DB2-format:

yyyy-mm-dd-hh.mm.ss.mmmmmm

(RXS function)

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

TRANSLATE 'translates' the characters of *string* to other characters. If *tableo* and *tablei* is not stated, the string is translated to upper case.

TRUNC(number,n**)**

TRUNC returns the integer part of *number* and - if stated - *n* decimal places.

USERID()

Returns tso-userident

VALUE(name,newvalue**)**

VALUE returns the value that has been assigned to a variable *name*, and optionally assigns a new value *newvalue* to the variable *name*.

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

VERIFY verifies that *string* only contains characters from *reference*. If true, 0 is returned. If not true, the position of first character in *string* that is not in reference is returned. If *option* is 'nomatch', the function is reversed: it returns the position of the first character in *string* that is in *reference*. If *start* is stated, the examination starts at *start*.

WORD(string,n)

WORD returns the *n*'th blank-delimited word in *string*.

WORDINDEX(string,n)

WORDINDEX returns the position of the first character in the *n*'th blank-delimited word in *string*.

WORDLENGTH(string,n)

WORDLENGTH returns the length of the *n*'th blank delimited word in *string*.

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

WORDPOS searches string for the first occurrence of the sequence of blank-delimited words, *phrase* in *string*. The words in both strings may be separated by any number of blanks. If *phrase* is not found in *string*, 0 is returned.

WORDS(string)

WORDS returns the number of blank-delimited words in *string*.

X2B(hexstring)

Hexadecimal to binary - returns a string in character format representing *hexstring* binary.

X2C(hexstring)

Hexadecimal to character - converts a string of hexadecimal characters to character format.

X2D(hexstring,n**)**

Hexadecimal to decimal - converts a string of hexadeical characters to decimal

Index