# Start of article 1.

Such REXX you did not know earlier.

Over time, IBM mainframes are evolving, and getting closer to the “open” world. Today, even Linux docker under z/OS 2.4 doesn’t sound surprising. Functionality is constantly expanding and improving, but at this stage we cannot deny the mainframe legacy or forget the old-fashioned classical methods that we used in most of our professional projects. In addition, legacy issues force us to be cautious about upgrading our existing software when it becomes relevant. This is a kind of a junction. We can either rewrite our old-fashioned programs to adapt them to the new environment, or stay with the existing legacy and enrich it with new technological methods and paradigms.

Thus, the purpose of these articles is to share methods that can help us take the first steps to overcome this junction and provide classic mainframe applications with the functionality of modern APIs and data structures. This will allow us to make OLD and NEW applications coexist. Using these methods will minimize the changes needed in our legacy code, while providing many opportunities to enrich the legacy with modern APIs and data structures.

So, I am starting this manual for MF programmers with the confidence that the methods we will discuss here can be widely used in our software.

I will present some amazing techniques in REXX that we can use in our applications. Moreover, I will share a completed framework written in REXX to make writing in REXX convenient and the code as error-free as possible.

It’s assumed that the audience has basic knowledge of REXX , TSO/ISPF, 3270 and TCPIP . We will not discuss the basic points. You can find the appropriate knowledge in the z/OS manuals and many posts and classes.

Why REXX? Can it be a choice if you decide to write a serious application? A couple of years ago I would have laughed at this question. Over the years of programming in Assembler and JAVA I have always used REXX to seam between applications or to quickly prepare data without writing annoying programs in COBOL or JAVA. Only recently did I realize that this was the wrong approach to REXX because REXX deserves and can give a programmer much more.

Strange as it may seem, the use of "not strongly typed" languages such as javascript or Python in serious open source applications has influenced my choice recently when I had to start a new development. Of course, the closeness and ease of REXX to manipulating z/OS resources only strengthened my confidence in the right choice. Thus, the new approach to programming in REXX has led to a new framework.

The framework presented here, **SummeRx**, includes the following APIs:

* Injector
* Library controller
* File controller
* 3270 terminal controller
* TCPIP controller
* Trace controller
* Heap controller

In the code “controllers” are referred to as helpers or mini-framework managers.

At this point we’re almost ready to start. The only thing that remains to be discussed, which is very important, is called PIPELINE. The history of pipelines in mainframes began in the 90s in VM. It was developed by several enthusiasts and still works in z/VM CMS.

Over the years, IBM had adapted this tool for z/OS as well. But unfortunately, for some reason IBM decided to discontinue the PIPELINE project in z/OS less than a decade ago. The product is available over SMP/E and I found it in our ADCD z/OS 2.2 preinstalled. I also found the manuals in the IBM collections from the early 2000’s in .**boo** format and can share these books along with SummeRx code.

It is impossible to overestimate the value of z/OS pipelines. This is a product that was ahead of its time.

Unlike similar features of other languages such as lambda, a functional programming or monad, which appeared less than ten years ago, z/OS pipelines were developed almost thirty years ago when object-oriented languages were in their infancy. Even today, the paradigm of mainframe multi stream pipelines is not fully implemented in "open" languages, although it has existed in MF pipelines for quite a while.

Therefore, I will dedicate the next article to z/OS pipelines to get the idea of programming streams in REXX.

Originally, SummeRx was developed with pipelines. Later, in order to meet the users' preferences, I decided to make it work in two modes: with and without pipelines.

So, in anticipation of your question about the need for pipelines, I answer:

The more IBM clients use pipelines in mainframes, the more likely it is that IBM will continue to develop them.

## Starting to work with SummeRx

* Download the file SUMRX.V201.TRS from <https://github.com/geelapidus/SummeRx> to your mainframe in binary format to a files with: LRECL=1024, RECFM=FB, BLKSIZE=27648, SPACE=(CYL,5)
* Adapt the job in the file JCLSTART.txt from <https://github.com/geelapidus/SummeRx> to your site environment and submit it. The result will be 7 libraries as follows :

SUMRX.V201.ASSMBLD assembled programs’library

SUMRX.V201.CMDLIB SummeRx TSO launcher

SUMRX.V201.COPYLIB user applications’ parts

SUMRX.V201.FWLIB SummeRx framework main library

SUMRX.V201.JCLLIB JCLs which run programs from REXXLIB

SUMRX.V201.REXXLIB user applications’ samples and configs

SUMRX.V201.SAMPLIB this library members are used in apps from REXXLIB

* Move GEXEC member of CMDLIB to a library chained in SYSEXEC DD of your TSO procedure

That’s all.

Now follow the examples in my next articles to understand how to start working with SummeRx.

End of article 1.

# Start of article 2.

Today we will discuss IBM pipelines in z/OS.

*NOTE: All the programs in these blog* article*s have been developed in TSO/ISPF environment.*

Let’s look at the following REXX code snippet and explain how it works.

Example 1

stm.0 = 2

stm.1 = "first line"

stm.2 = "second line"

pipe " stem stm. " || ,

"| terminal **"**

The result of this will be 2 lines printed as follows:

**first line**

**second line**

as if we would have written the following code instead of “pipe” sentence:

**do i = 1 to stm.0**

**say stm.i**

**end**

Very simple! An incoming stream from stem “stm.” is displayed on the terminal.

Note, that we use a new term STREAM which is not used in the regular REXX without pipelines.

Please notice the following:

1. **Style**. I like to have pipelines as rectangles in my code. So, the concatenation operator at the right side provides the right frame boundary.
2. Each stage except the first in the pipeline should be prefixed by the stage separator, “|” in our case (default). In general, the stage separator is one of the pipeline parameters and may be chosen by you.

In that pipeline there are two stages: STEM and TERMINAL.

Each of them has either an input stream or an output stream or both.

STEM, when placed as the first stage of a pipeline has an output stream only.

TERMNAL has both input and output streams.

In general, the output stream of a stage is usually the input stream of next stage.

There are types of stages which can have more than one input or output streams. They are used for programming with multi stream pipelines.

Let’s take a look on an example of a multi stream pipeline:

In this example, the parameter **endchar** is set to “?” as the pipeline delimiter.

Example 2

**stm.0 = 4**

**stm.1 = "first line"**

**stm.2 = "second line"**

**stm.3 = "third line"**

**stm.4 = "fourth line"**

**pipe " (endchar ?) " ||,**

**" stem stm. " ||,**

**"| l:not locate w1 /th/ " ||,**

**"| f:faninany " ||,**

**"| terminal " ||,**

**"?" ||,**

**"l:" ||,**

**"| specs /TH/ 1 1-\* nw " ||,**

**"| f:**

This pipeline marks all input stream rows which contain “th” substring with TH prefix. Thus, the result will be as follows:

**first line**

**second line**

**TH third line**

**TH fourth line**

**How it works** ?

First of all I will explain the new syntax as shown below:

* **endchar ?** Character “?” is used as the delimiter in the multi stream pipeline.
* **l: ,f:**  Labels that are used to control a stream flow.
* **“l:not locate”** Breaks the input stream into two output streams; one with, and one without the searched substring.
  + Input stream #0: Output stream of stage “stem stm.”.
  + Output stream #0: All the rows of the input stream that do not contain the substring “th”.
  + Output stream #1: All the rows of the input stream that contain the substring “th”.
* **“f:faninany”** Combines two or more input streams, keeping the chronological order of their appearance in the input streams.
  + Input stream #0: The output stream #0 of “l: not locate”.
  + Input stream #1: The output stream of “specs /TH/”.
* **“specs /TH/ “** This stage prefixes each row from its input stream with the constant “TH”.

In the following example, we want the rows marked with “TH” to appear at the beginning of the output stream.

Example 3

**stm.0 = 4**

**stm.1 = "first line"**

**stm.2 = "second line"**

**stm.3 = "third line"**

**stm.4 = "fourth line"**

**pipe " (endchar ?) " ||,**

**" stem stm. " ||,**

**"| l:not locate w1 /th/ " ||,**

**"| elastic " ||,**

**"| f:fanin 1 0 " ||,**

**"| terminal " ||,**

**"?" ||,**

**"l:" ||,**

**"| specs /TH/ 1 1-\* nw " ||,**

**"| f: "**

What is the difference between this and the previous example ?

We used “fanin 1 0” instead of “faninany”. *Output stream #0 of the srage "fanin 1 0" is the concatenation of the whole input stream #1 followed by the input stream #0.*

It may happen that "l:not locate" is ready to put a line into its output stream #0 as "f:fanin 1 0" is still waiting for rows from its input stream #1 which is stage"f:". In this case the output stream #0 from "l:not locate" has to be cached, otherwise the pipeline will stall. The "elastic" stage serves as a feed cache for "f:fanin 1 0".

In most cases, a pipeline is a single task process. It tries to push a row from the pipeline beginning to its end. When it happens, everything works straightforward. A good example of this is a single stream pipeline or a simple multi stream pipeline that does not change the order of rows flowing through the pipeline. The examples 1 and 2 above have shown such pipelines.

However, in example 3, the flow has been delayed, so the order of rows passing through the pipeline has been changed. It may also happen that several rows of the input stream are combined or one row of output stream is divided. Programming of pipelines has several features and methods of blocking to avoid stopping of pipelines. All this turns programming REXX with pipelines to a joy, with improving the quality of coding.

The last example of this article demonstrates a FOREACH pipelines’ paradigm. The purpose of the example is to print the contents of all members of the source library.

Suppose, we need a pipeline to read a library directory and to prints all the members’ contents embraced by start and end markers as follows:

**MEMBER=<member name>**

**member line 1**

**member line 2**

**…**

**member last line**

**end-of-member**

The pipeline itself is:

Example 4

**rexxlib = "your.source.lib"**

**"pipe (endchar ? ) " ||,**

**"| listpds "rexxlib" " ||,**

**"| chop 8 " ||,**

**"| specs /callpipe (stagesep $) / 1 " ||, (1)**

**" /$ members "rexxlib"/ nw w1 nw " ||, (2)**

**" /$ chop 72/ nw " ||, (3)**

**" /$ drop last 1/ nw " ||, (4)**

**" /$ append literal end-of-member/ nw " ||, (5)**

**" /$ append literal/ nw " ||, (6)**

**" /$ literal MEMBER=/ nw w1 n " ||, (7)**

**" /$ \*.output:/ nw " ||, (8)**

**"| pipcmd " ||, (9)**

**"| terminal "**

Note that in the internal pipeline created by **callpipe**, the **stagesep** parameter is "$", and in the main pipeline it is "|" by default.

**How it works.**

It is based on the method of dynamic pipeline creation by CALLPIPE and its subsequent execution through PIPCMD stage.

1. **listpds** stage reads the library directory
2. **chop 8** extracts the member name in the first 8 bytes of the row. Output stream #0 of this stage will contain several rows, where each row is the name of the member.
3. **specs** creates an internal pipeline using **callpipe** command for each row of the input stream, i.e. for each library member.
4. Internal pipeline:
5. Sets a new stage separator
6. Reads the content of the member
7. Cuts out the first 72 bytes of each line
8. Removes the last line (“members” stage appends it by default to separate between members)
9. Adds an “end-of-member” separator
10. Adds a blank line
11. Prepends MEMBER=<member-name> to the content of the member
12. Binds the pipeline currently built to the input stream #0 of stage **pipcmd**
13. “pipcmd” executes the internal pipeline from its input stream #0 (9)
14. “terminal” displays the result of the internal pipeline execution

If you want to look deeper, replace “pipcmd” with “terminal“ in the pipeline and you will see a number of pipelines in the input stream #0 of stage TERMINAL. Each pipeline will be shown twice, as you now have two “terminal” stages.

If you want to save the result to a file, you can add a stage “>” or “>>” .

You can also place the result in the stem using the "stem" stage.

That is an introduction to z/OS pipelines. In reality, there are many variants of pipelines and I have a lot of examples to demonstrate them. In future I will prepare a class of z/OS pipelines for REXX.

**P.S.**

You can query the existence of pipelines in your zOS by issuing the following TSO/ISPF command:

**ddlist load pipe**

You can try to install pipelines from SMP/E , FMID HACH301 .

The BPWGDTEP module will be installed into SASxxx.SASFPLNK library (or another SASxxx library, depending on the product version) and the PIPE module will be an alias of BPWGDTEP.

You can find the manuals in in .BOO format in earlier z/OS Collections. Unfortunately, IBM has not convert them to PDF. So, use **IBM Softcopy reader** or old **IBM Library reader** to read the manuals

The master book is :

BatchPipes OS/390 V2R1 BatchPipeWorks Reference (SA22-7456-01) , collections SK3T-4273-01 and SK2T-6700

End of article 2.

# Start of article 3.

Today we will discuss stems. STEM is a compound symbol that allows a period. It turns a stem in a structure that can be used as array, map, hash or another data structure that can be invented by a REXX programmer. In the following articles I will present a HEAP built on STEM and give some examples of its use for code safety, pseudo object-oriented programming and other purposes.

Thus, it is very practical to use the stem if your data structure is more complex than trivial variables. Unfortunately, IBM restricts the use of the stem as an argument, so it cannot be available to the subroutine unless the subroutine uses the EXPOSE option of the internal PROCEDURE operator. INTERNAL means that the PROCEDURE operator using stem must be in the same source file where stem is used. There are several methods to solve the problem but none of them can be used due to lack of code safety, low performance or non-optimal use of resources. We will look at all the known methods of stem transfer between programs, and finally I will present my own method to your considerations.

Let’s start with a method that I like to use in small REXX programs:

Example 5

**stem.0 = 1**

**stem.1 = "aa"**

**i=5**

**call printst("stem")**

**say "program ended"**

**exit**

**printst:**

**st = arg(1)**

**interpret "say "st".0"**

**interpret "say "st".1"**

**say i**

**return**

In this example although **printst** is a subroutine it behaves like a label. It does not protect caller variables.

The result will be as follows:

**1**

**aa**

**5**

**program ended**

As it can be seen, the argument contains rather the name of the stem, than the stem data structure.

The stem variables are printed by index, but other variables of the caller, which have not been passed to the subroutine through the arguments are also accessible in the subroutine and this is very bad !!!

Thus, it cannot be used in a serious REXX project.

In the following example we will use PROCEDURE, which is the only change toward the previous example.

Example 6

**stem.0 = 1**

**stem.1 = "aa"**

**i=5**

**call printst("stem")**

**say "program ended"**

**exit**

**printst: procedure**

**st = arg(1)**

**interpret "say "st".0"**

**interpret "say "st".1"**

**say i**

**return**

Now, the result has been changed but it still is not one that expected, although the caller’s variables are protected:

**STEM.0**

**STEM.1**

**I**

**program ended**

Why are stems "hidden" for the procedure? That's a good question. I think that passing the stems as arguments to subroutines may probably disrupt the internal optimization of the REXX code.

Example 7

**stem.0 = 1**

**stem.1 = "aa"**

**i=5**

**call printst("stem")**

**say "prorgam ended"**

**exit**

**printst : procedure expose stem.**

**say stem.0**

**say stem.1**

**say i**

**return**

Now, by indicating PROCEDURE EXPOSE we “requested” to keep stem for PRINTST: PROCEDURE and the result will be :

**1**

**aa**

**I**

**program ended**

That is exactly what we need !!!

Unfortunately, this will not work if **printst** is an EXTERNAL routine, i.e. is stored in another member of the source code library. The PROCEDURE statement is not allowed in an external routine since there is an implicit PROCEDURE that hides all caller’s variables. The only method to pass parameters to an external routine is to use arguments, but as we know a stem is not passed as an argument !!!

According to the rules of stem usage, we need to create programs in **all-routines-in-one-file** way. The big problem with this approach is that we need to place our common routines and proprietary APIs into each program file. Thus, API subroutines will be repeated as many times as the number of source files (the main REXX programs). This approach makes projects less maintainable: a change in a common or API routine can cause a number of the same change being repeated in all the applications which call the routine. So,the question is how to achieve modularity in REXX applications that intend to use STEM ?

The idea is quite simple. Let’s consider the following piece of code :

Example 8

**/\* rexx \*/**

**a = 1**

**b = 2**

**say a**

**++inc m**

**say b**

**exit**

The result will be produced as follows :

**1**

**5 +++ ++inc m**

**Error running …, line 5: Bad arithmetic conversion**

We got the expected result. REXX interpreter advanced over the source code and executed it until it met a card with incorrect syntax “++INC M”. Indeed, **++INC** pseudocode is not known to REXX interpreter but is KNOWN to SummeRx framework.

Obviously, REXX is a single-pass compiler/interpreterter, so it is possible to use pseudo code if the program can replace it with correct REXX code at runtime. In the above example, I want the program to replace a card containing "++INC MEMBER" with the content of a MEMBER member before the execution pointer reaches that pseudo code. I call this an injection of code. This way, the program will work in two modes: **injection** mode and **execution** modes. All magical things will happen on the injection stage.The only thing left to decide is: who will be the INJECTOR ?

Example 9

|  |
| --- |
| **retcode = INJSTUB(arg(1))**  **if rexxname <> '?'**  **then**  **do /\* submitted by TSO \*/**  **ZISPFRC = retcode**  **"ISPEXEC VPUT (ZISPFRC)"**  **exit**  **end**  **else**  **do /\* submitted by injector \*/**  **if retcode <> 0**  **then**  **return retcode**  **end** |
| **INJSTUB : procedure**  **parse source . . rexxname . rexxlib .**  **rcode = PARM\_VALIDATION(arg(1))**  **if word(rcode,1) <> 0**  **then**  **return 8**  **else**  **do**  **msg\_level = word(rcode,2)**  **return INJREXX(rexxname rexxlib ,**  **msg\_level arg(1))**  **end** |
| **SUMRX\_MAIN : procedure expose msg\_level HEAP.**  **… application program code …**  **return retcode** |

|  |
| --- |
| The following code fragment makes the idea clearer : |

**How it works.**

INJSTUB is called at the very program beginning. It retrieves the program name, source library name , and the execution parameters(s). It then externally calls the injector **INJREXX** and passes all the parameters to it. The INJECTOR does magic (we will consider the INJECTOR functionality in the next article) and returns control to the point where the program was called at the beginning.

End of article 3.

# Start of article 4.

Let’s start with unveiling the secrets of the INJECTOR.

First, INJSTUB is called and then it in its turn calls INJREXX injector. The purpose of the INJECTOR is to assemble a module that will contain REXX code with all pseudo code resolved, i.e. replaced by the correct REXX code. The injector reads the main source program from the library, renames INJSTUB to INJSTUB\_NA to disable it in the assembled module, and resolves the ++INC dependencies by injecting the corresponding library members. Then, when assembling of the module has been completed, the INJECTOR issues a TSO EXEC command to execute it. The INJECTOR also inserts its own code into the assembled module. In this way, the SummeRx framework will function as a shell for a user application. ASSEMBLE here has the connotation of creation, but not of compilation into the machine code.

A screenshot of a cell phone

Description automatically generated

1. Running First Level application.

SummeRx application allows CALL another SummeRx application, which also has the SummeRx pseudo code for the injection, i.e. SummeRx frame allows the First Level application to call the Second Level application, having previously created it, as in Figure 2. It calls INJREXX, but now this call is internal because the INJREXX code has been earlier injected into the First Level application.

A screenshot of a cell phone

Description automatically generated

1. Call Second Level application.

Thus, we’ve achieved our aims:

We can keep our common routines in separate and use the INJECTOR loosely coupled mechanism at run time to assemble the execution file. We’ve solved the not trivial problem of accessing stems distributed over different members of source libraries and can keep a high maintainability of our applications.

NOTE : Since we are able to use such a useful INJECTOR feature as ++INC we can go ahead and present other INJECTOR features based on ++INC that are very useful when programming with SummeRx :

1. The ++INC dependency can be solved from either MAIN, COPY or SUMRX libraries. The MAIN library is the library from which an application runs. COPY library is a customer library that contains APIs which are dynamically injected into the application code. SUMRX library is a framework library, that contains special subprograms and APIs, which are necessary at runtime. MAIN has higher priority over COPY, and COPY, in its turn, has priority over SUMRX. This scheme allows the user to override likely named members. The names of COPY and SUMRX libraries are defined in the **base configuration element**, to which the ++INC directive refers from the main application program. For example, **++INC SETBASE,ENV**. The ENV suffix is used by SUMRX to recognize the name of the base configuration element. For using that in TSO background mode it is necessary to chain COPY and SUMRX into SYSEXEC DD. In TSO foreground mode you can use SummeRx launcher GEXEC (instead of standard EXEC), which obtains the names of COPY and SUMRX libraries from the base configuration element.
2. For the user convenience ++INC directive with parametric member name is supported. For example, ++INC &VAR. Variable VAR is resolved in the base configuration element :

**injstm.VAR = "MEMNAME"**. Thus, member MEMNAME will be included.

1. Tracing message level control. The trace level is entered into the message routine and you no longer need to “haul” it as an annoying parameter through the hierarchy of your routines.
2. You may use an extended REXX syntax in the source code if you update it with INJECTOR.

An example of that is **MSG(..)** command , which is replaced by injector with **RC=MSG(..)** to adapt the REXX syntax. SummeRx offers MSG() to save place in the source code line.

1. Also SummeRx has an opportunity to pass the application external parameters from SYSTGIN DD file (only in background TSO mode). Usually, these are environment parameters. SummeRx passes these parameters to all levels of injection nesting scheme. This can be used to move SummeRx application between environments without changing the basic configuration element, for example, from TEST environment to PROD environment.
2. In TSO foreground mode trace messages are displayed on the terminal. That’s illegal to do if your application is a SummeRx 3270 application. In this case INJECTOR allocates TRM3270 file and routes trace messages to this file. TRM3270 file is re-allocated in next run of SummeRx 3270 application.
3. The INJECTOR can operate in the same way in both the pipeline mode and the regular REXX mode, which is controlled by the PIPEMODE variable in program INJREXX. Value of 1 means pipeline mode while 0 means no pipeline mode.

In the next article we will consider an example program that uses the SummeRx framework. We will discuss the important techniques as follows:

1. Normalized stems
2. Tracing routine
3. Loosely coupling and common routines
4. File and Library frameworks
5. Tracing normalized stems
6. Saving assembled REXX code as a library member

End of article 4

# Start of article 5

Here we will discuss a simple REXX application working with SummeRx framework and introduce a tracing feature of SummeRx.

We will use an application example from the member BX11 of MAIN-library. The application prints contents of all library members.

You can run the application by TSO command:

**gexec '<your main lib> (BX11)' ‘summerx ALL’**

Since the application uses ISPF LM-commands (via LMHELPER) the ISPF environment is needed.

You can see that the application code is small, the rest of the code is the SummeRx pattern.

The actual code is as follows:

Example 10

**SUMRX\_MAIN : procedure expose appl\_parm msg\_level HEAP.**

**++INC SETBASE,ENV**

**lib = userid()".SUMRX.V201.CMDLIB" /\* choose name of your lib \*/**

**appl\_parm = word(arg(1),1)**

**MSG(".Application Started ,parm="appl\_parm)**

**/\* LIBMLIST will fill stem MEMSTAT with lib dir items \*/**

**rc = LIBMLIST(lib)**

**if rc <> 0**

**then**

**return 8**

**/\* list library members \*/**

**/\* call TRACE\_STEM "memstat" \*/**

**do i = 1 to memstat.0**

**mem = word(memstat.i,1)**

**if LIBMREAD(lib,mem) <> 0**

**then**

**return 8**

**if member.0 > 0**

**then**

**do**

**MSG("Member:"mem)**

**do j = 1 to member.0**

**MSG(substr(member.j,1,44))**

**end**

**MSG("End-of-member")**

**end**

**end**

**return 0**

**How it works**

Parameter value of “summerx ALL” is parsed by **application parameter** and **run trace level**. **ALL** means the highest trace level, which in this application is 10. The SummeRx framework will filter trace messages whose **trace level** is greater than **run trace level**.

You may use your logic of assigning current **run trace level** and when it’s not passed through the parameter you may set its default value. For example, when executing command **gexec '<your main lib> (BX11)' ‘summerx’** without passing message level through the parameter, **run trace level** will be set to the default value , which is 1 in this application.

SummeRx supports two types of **message trace level**.

The old type of **Message trace level** is a number of periods preceding the text in the message.

For example, **MSG("..End-of-member")** has **message trace level** of 2.

We will speak about **new type of message trace level** in the next articles.

Now, issue **gexec '<your main lib> (BX11)' ‘summerx 1’** from TSO and you will see fewer trace messages, basically almost all of the framework messages have been filtered out.

In addition, SummeRx offers the powerful tracing feature.

For example: **call TRACE\_STEM "memstat"** will trace stem **memstat.** To get to the framework trace a stem must have the stem size assigned to its elemen **.0** . SummeRx will process  **.1, .2** , etc. elements. This is a one-dimensional **normalized** stem.

The SummeRx tracer can also trace a two-dimensional normalized stem, as shown below:

**A.0 =2**

**A.1 = ”SINGLE”**

**A.2 = ”PAIR”**

**A.SINGLE.0 = 1**

**A.SINGLE.1 = ”MUG”**

**A.PAIR.0 = 2**

**A.PAIR.1 = ”CUP”**

**A.PAIR.2 = ”SAUCER”**

In general, the SummeRx supports more complex data structures and we will talk about this in the article that discusses SummeRx HEAP framework.

And the last thing to do is to add a stem to the TRCDATA member in order to introduce it to the SummeRx Tracer.

**NOTE**. To have SummeRx tracing stems the **trace message level** should be set to 5 or higher.

End of article 5

# Start of article 6

In this article we will speak about running SummeRx applications in Batch.

Generally, we need to use a JCL deck that includes TSO and ISPF environments, but here we will discuss the additional feature of SummeRx that works only in BATCH. This is an application trace segregation.

If you set the trace to a high level, you will obviously not like the mess of your application's messages and SummeRx messages mixed in one file. Thus, SummeRx splits tracing messages between 3 files as follows :

**SYSTSPRT** standard file for application messages

**SYSFWPRT** dynamically allocated file for the SummeRx framework messages

**SYSEXPRT**  dynamically allocated file for the SummeRx and application explaining messages. Usually, a user wants to get messages that less technical but have clear explanations in human language. We will talk later about the technique SummeRx uses to do this.

Since each message has a prefix with a timestamp, it is easy to find a case at runtime. It is enough to look at the messages of the framework around the time when the case appeared in the tracer log of the application.

Below is a sample of the JCL deck.

Example of SummeRx job deck

**// EXPORT SYMLIST=(COPYLIB,SUMRXLIB)**

**// SET VER=201**

**// SET MEM=HTEST06**

**// SET HLQ=TLVGL**

**// SET COPYLIB=&HLQ..SUMRX.V&VER..COPYLIB**

**// SET SUMRXLIB=&HLQ..SUMRX.V&VER..FWLIB**

**// SET PROFDSN=&SYSUID..S0W1.ISPF.ISPPROF**

**//\***

**//STEP1 EXEC PGM=IKJEFT01,DYNAMNBR=20,**

**// PARM='ISPSTART CMD(%&MEM H06 ALL)'**

**//SYSPROC DD DISP=SHR,DSN=ISP.SISPCLIB**

**//SYSEXEC DD DISP=SHR,DSN=&HLQ..SUMRX.V&VER..REXXLIB**

**// DD DISP=SHR,DSN=&HLQ..SUMRX.V&VER..COPYLIB**

**// DD DISP=SHR,DSN=&HLQ..SUMRX.V&VER..FWLIB**

**// DD DISP=SHR,DSN=ISP.SISPEXEC**

**//ISPMLIB DD DISP=SHR,DSN=ISP.SISPMENU**

**//ISPEXEC DD DISP=SHR,DSN=ISP.SISPEXEC**

**//ISPPLIB DD DISP=SHR,DSN=ISP.SISPPENU**

**//ISPSLIB DD DISP=SHR,DSN=ISP.SISPSLIB**

**// DD DISP=SHR,DSN=ISP.SISPSENU**

**//ISPTLIB DD DISP=SHR,DSN=ISP.SISPTENU**

**//ISPPROF DD DISP=SHR,DSN=&PROFDSN**

**//SYSTSPRT DD SYSOUT=\***

**//ISPLOG DD SYSOUT=\*,DCB=(LRECL=125,BLKSIZE=129,RECFM=VA)**

**//ISPCTL1 DD DISP=NEW,UNIT=SYSALLDA,SPACE=(CYL,(1,1)),**

**// DCB=(LRECL=80,BLKSIZE=800,RECFM=FB)**

**//ISPCTL2 DD DISP=NEW,UNIT=SYSALLDA,SPACE=(CYL,(1,1)),**

**// DCB=(LRECL=80,BLKSIZE=800,RECFM=FB)**

**//ISPLST1 DD DISP=NEW,UNIT=SYSALLDA,SPACE=(CYL,(1,1)),**

**// DCB=(LRECL=121,BLKSIZE=1210,RECFM=FBA)**

**//ISPLST2 DD DISP=NEW,UNIT=SYSALLDA,SPACE=(CYL,(1,1)),**

**// DCB=(LRECL=121,BLKSIZE=1210,RECFM=FBA)**

**//SYSTSIN DD DUMMY**

**//SYSTGIN DD \*,SYMBOLS=EXECSYS**

**COPYLIB='&COPYLIB'**

**SUMRXLIB='&SUMRXLIB'**

**/\***

**How it works**

When you move your application from TEST to PROD of from FORE to BACK ground you want to share **base configuration element** between the environments. You also want to keep **++INC <setbase>,ENV**  in the source code and not change any other member in your application libraries.

To achieve this you may use SYSTGIN file to override your environmental parameters in the base configuration element. The following parameters will be overridden on the JCL deck above:

**COPYLIB**

**SUMRXLIB**

Usually, you can select any parameter you want to override in any environment. You will only need to specify your logic in the base configuration element as in the following example:

**/\* COPYLIB \*/**

**if copylib <> "COPYLIB"**

**then**

**MSG(".Overridden COPYLIB :"copylib)**

**else**

**do**

**copylib = '<your\_test\_copylib>’'**

**MSG(".Default COPYLIB :"copylib)**

**end**

In the above code fragment, the COPYLIB variable is assigned a default value if this variable has not yet been initialized. SummeRx prepends the content of SYSTGIN file to the base configuration element. Thus, when moving to a new environment, the COPYLIB variable will be assigned a new environment value. In TSO foreground, the COPYLIB variable will be set to the default value because SYSTSGIN is not used for TSO foreground. The same technique can be used for any environment variable of the basic configuration element.

SummeRx inserts the contents of the SYSTSGIN file just before the **++INC <base>,ENV** line. I propose to put an override logic of the parameter at the very beginning of the base configuration element.

SummeRx will pass the overridden parameters to the second level application. Your responsibility is to avoid conflicts while a 2nd level application uses a different base configuration element with a different logic of values’ assigning to environment variables.

End of article 6

# Start of article 7

In this article we will discuss how to prepare an executable SummeRx application.

It is very convenient to work with SummeRx helpers, but you may have noticed that assembling an executable source code module from its parts may take a significant amount of time. And if you run your programs periodically, you would prefer to avoid the assembling process every time you start an application.

The injector can store the assembled program to a file instead of running it. $TSORDY member is an example of the job that assembles an application.

Note the differences between $TSORDY and $TSOHT1 JCL decks in the JCLLIB library.

We have already talked about SummeRx TSO JCL deck before.

$TSORDY uses the INJRDY control parameter which defines the name of the dataset in which the assembled source module will be stored. After its creation, you can run it as a usual REXX.

There are a number of things to consider:

1. The values injected into the code cannot be changed later by parameters. It is necessary to assemble the source module with the required input values. For example, during assembly the level of tracing messages is injected into the source code. There is no way to control this value in the assembled module through a parameter unless you decide to change it manually in the corresponding routine inside the assembled module.
2. If an application contains a number of programs that use the injector, you should assemble them all separately. But, unfortunately, this is not enough. Please pay attention on the INJ\_CALLEXEC injector routine and note that SOURCE\_LIB\_DYN is a variable which value is injected during assembly. This refers to the library where the 2nd level application should exist. In future version of SummeRx this problem will be fixed. In meantime, you can change it manually in the assembled code.
3. If an application uses calls to SummeRx 2nd level applications, you need to store your assembled modules as members of same library rather than in separate sequential files.
4. SummeRx supposes that the library to store the assembled modules already exists and is cataloged. This is a fixed block dataset with record length of 100. Why not 80? Because the injector can extend the source line, for example by replacing MSG(..) with RC=MSG(..)
5. Once you have assembled the source code and got all the members in the library, you can compile them to the machine code.

End of article 7

# Start of article 8

With SummeRx you can easily develop your 3270 applications in TSO/ISPF.

3270 framework is mainly based on the pipeline stage FULLSCREEN, so it is the only framework that uses pipelines , among others.

TRMHLPR is a helper that does a lot for you and turns the annoying programming work on the 3270 into a simple job. So, what is TRMHLPR functionality? And how do you use it to program a real application?

The TRMHLPR has the following very attractive features:

1. It keeps a current screen as a background and protects all unprotected fields of it.
2. It allows the programmer to develop applications with nested windows. It saves the buffers of previous windows to activate them during reverse navigation.
3. It allows the programmer to design simple windows and define repetitive lines and information areas in these windows.
4. It calculates the window size according to the actual number of lines on the physical screen and the volume of the data array to be displayed in the window.
5. It provides the vertical scrolling of the data inside the window.
6. It creates a stem of REXX variables whose values are filled in from the corresponding unprotected fields of the window. The programmer will never need to make the calculations of 3270 buffers’ offsets and replies. All variables and AIDs are available to him through REXX variables.
7. The 3270 window may be defined in an application program on the object manner. This approach makes development easier.
8. At the preparation stage SummeRx takes care of trace output and redirects trace messages to the dynamically allocated TSO file TRM3270. The application can work with the swapped TSO terminals. SummeRx will not clear/delete the TRM3270 file until the application on the swapped terminal is completed. This gives the programmer a complete trace of the application run. Only when starting a new 3270 application, the trace data of the previous run is deleted.

If you’re programming a SummeRx 3270 application the all you need is:

* In the INJSTUB section of the template, set the value of INJREXX 2nd parameter to ONLINE. This will let SummeRx know that this is a 3270 application .
* Textually design your windows as REXX stems
* Design PFs logics
* Design data arrays for the windows
* Design navigation between the windows
* Mark repetitive lines and information areas
* Mark unprotected fields and cursor position in textual windows and assign one-character names to the fields, so called short names. You can also specify a long name for each short name. Working with long names variables in REXX is much preferable.

Let’s consider an example BX10 that uses the TRMHLPR framework . This is quite self explanatory program, and you can understand a lot from this example. Please notice the following in BX10:

1. Routines OBTAIN\_WINDOW and OBTAIN\_DATA are called from the framework, so keep their signatures as in EXPOSE clauses
2. On the first screen in the textual field the value #C2A# means:

* In this field the cursor will be set
* The field length is 2 bytes
* The field short name is A
* The field long name is LIBRARY\_NUMBER
* You can also gain access to VARS. LIBRARY\_NUMBER to get the field value

1. ++R means a repetitive line
2. ++I means a line to display INFO or ERROR messages
3. Since the '#' character is used by default to mark a 3270 field position and boundary, the data provided by the OBTAIN\_DATA routine will be compiled by the framework to replace the '#' character with a period. Otherwise, creating a 3270 buffer from the textual window may cause an error.
4. It is assumed that the application will have 2 levels of windows and:

* You can assign actions to your PF keys. For example, AIDS.1.PF3 assigns actions to PF3 button in the window at level 1
* Note that you can restrict PFs/PAs or text command to use by routines’ names that start with ALLOW…
* The SCRSTAT.INFO\_MSG field will be displayed on 3270 screen as an informational or error message.
* The BX10 program code can only be changed in its subroutines’ blocks, not in the main one. Thus, SummeRx can guarantee the correct functionality of the program.

1. BX10 application will display the list of libraries which is produced by TSO LISTCAT command and filtered with your user ID as HLQ.

* You can change search criteria
* It’s assumed that all given files have the type of library
* Only the first 10 files are displayed
* If something goes wrong the application will not analyze it or send an error to the user
* The application is written to demonstrate the 3270 framework of SummeRx, not the most correct style of writing programs in REXX.
* The 2nd level screen is only displayed as a window but does nothing in point of view of the business logic

This framework can be improved by making the fields marked with different attributes characters. Extended attribute characters are not supported yet, so it can be nice if someone makes these screens rich colored under 3270 emulations.

To start BX10 example type from TSO COMMAND panel:

**gexec 'TLVGL.SUMRX.V201.REXXLIB(BX10)' 'ALL'**

The first application window will appear. You can navigate to the second window by selecting the line number with the desired library. PF3 will return you to the previous window or to ISPF. When finished, you can go to the SDSF panel and view the TRM3270 file of the current TSO session. This is a tracing file. Find the **VARS.** in this file. You will see the VARS with AID, the cursor position and a variable with a long name and its value, for example, **VARS.LIBRARY\_NUMBER:3**.

Try to run this application from another TSO session with different number of lines on the physical screen. As you can see, SummeRx adjusts the size of windows to the size of the screen.

End of article 8

# Start of article 9

In this article we will discuss writing a TCP/IP server with the SummeRx framework.

REXX is a single task process, but IBM offers the RXSOCKET API which provides a pseudo multitasking environment and thus makes REXX a suitable platform for server development. Of course, with some limitations.

RXSOCKET internally uses assembler code that controls a number of tasks, addressing multiple user sockets at the same time. But REXX itself remains a single task process that coordinates the data control with the assembler layer through strings. These strings contain the list of the work sockets. Each element in the list contains the event that occurred on the socket and the action required on it as follows :

* READ - the socket has data in the buffer, to set the REXX variable with data the action RECEIVE is required.
* Write - the client side is on RECEIVE, so SEND to the client is necessary.
* CONNECTION - loss of connection, for example

From a REXX programmer's point of view, writing a server becomes a simple thing. By watching the socket list, the programmer can determine which socket operation is requested at a given moment.

Thus, a server in REXX is an endless loop on the socket list. At each iteration of the loop, the decision of what to do is made based on the state of the socket in the socket list.

What is the drawback of such a server? Single task mode!!! The socket listener and all application sockets run within the same task and that degrades the performance. But if you want to work with request-response sessions with a small number of clients and short messages, RXSOCKET can be a good candidate.

The BX12 member of REXXLIB is an example of such a TCP/IP server. It uses a TCPHLPR helper which works in the RXSOCKET scheme. BX12 code has three custom subprograms: APPL\_RECEIVE, APPL\_SEND and APPL\_CLOSE. You can put your code there to parse the received data, collect the data to send and handle the connection closure. The server based on TCPHLPR runs in request-response mode with stateless sessions, allowing it to consume as few resources as possible.

To start a BX12 server, use member $TSOBX12 of REXXLIB. Note that the trace message level is set to ALL, so you will get many messages. BX12 uses port 12000. This value passed to the server through the parameter.

TCPCLNT member in REXXLIB library is an example of client that connects local server on port 12000. This can be a good test program to check the server.

It is possible to issue a CANCEL command from the z/OS console to stop the server. In future versions of SummeRx the server control through STOP and MODIFY commands will be added.

In addition, the BX12 server has a simple encoding recognition feature. It works well with text messages. When the server recognizes an incoming message in EBCDIC, it sends the message back to the client in EBCDIC as well. ASCII clients will receive server messages in ASCII encoding.

End of article 9

# Start of article 10

In this article I will present the SummeRx HEAP. Usually the heap makes sense when programming with object-oriented languages. But REXX is not one of them. So, what is the purpose of adding a heap to a straight forward language? What benefits will a programmer get from using it? Does REXX add new functionality? These and other questions arise in relation to the heap in REXX.

First of all, REXX remains the same as it was. I didn't add a new syntax, even at the pre-processing level, as it was with ++INC.

Secondly, the new HEAP function is an optional part of SummeRx.

But I recommend to use it for a number of reasons, which I try to discuss in my articles and considerations about the right programming style with the SummeRx framework.

1. Working with objects. When designing, it is necessary to formally describe the behaviour of program's elements and then implement it with REXX code. Let us consider the following design element:

DELETE MEMBER: remove the member from the library, but if it is the only member, then delete the library.

This is a good moment to start thinking about objects. Of course, you can just write this logic in REXX. But let's agree that if we can somehow describe such a behaviour as an object property, our REXX code will be more readable and reliable.

1. The program logic can be roughly divided into two parts: data access and data presentation. If the program prints out a report, it must first collect the data and then elaborate and present the data in the report tables. We often make a mess by mixing these two parts. OOP has an MVC (Model-View-Controller) architectural template, which is designed to avoid this clutter.

We can't use the MVC concept in REXX (we'll need a REXX studio and the corresponding project templates, which do not exist yet), but SummeRx offers a provider-consumer scheme that still has a number of advantages over the conventional approach as follows :

* SummeRx uses the so-called loose coupling of code parts, where any of the parts can be written and tested separately.
* These parts are injected into the code dynamically, so that both the provider and the consumer can be easily replaced.
* SummeRx provides an immediate level of data between provider and customer. This level is HEAP, which is used for data storage and access.

1. SummeRx HEAP offers several data primitives for both provider and consumer use. Their use can greatly simplify programming and help avoid many errors.
2. In REXX we usually "forget" about temporarily used variables. Thus, they remain allocated in memory as garbage. This can lead to serious memory leaks in a long running process. Using HEAP primitives will help to avoid such unpleasant situations as HEAP uses its internal garbage collector.
3. SummeRx provides a powerful tracing feature. With this you can trace both the variables in your program and the data structure that is stored in HEAP.
4. SummeRx offers SERVICEs. The service represents the data structure of HEAP, which can be run from a user program. The scheme of working with services is simple:

* The programmer defines a service for SummeRx and launches the main program (consumer).
* the consumer launches the service through the HEAP API.
* the service calls the provider by name (the name of the provider is part of the service definition).
* the provider receives the data and stores it in HEAP.
* the service converts the data stored in HEAP data into the required format, which is defined in the service.
* The consumer accesses the elaborated data in HEAP and continues to work with it
* The consumer finishes working with the current service and stops it.
* The garbage collector (GC) is called and the corresponding REXX variables are dropped.
* the user program continues to work and can start another service.

HEAP is a large REXX stem and has its own logical structure. The HEAP code can be found in the HEAPHLPR helper. It contains a number of data structures that are divided into 2 categories: system and user. The system data structures store the necessary information for HEAP itself and are mainly used by internal HEAP services such as tracer, garbage collector and serializer. User data structures can be accessed through a user application and can serve as an intermediary between provider and consumer. Below are the types of HEAP data structures :

* LIST is a bidirectional linked list containing data elements. It allows ADD, INSERT, FORE, BACK, SORT, DELETE and others.
* MAP is a <key,value> table. Can be used for quick search within HEAP by key.
* ELEMENT is a LIST or MAP data element. The element exists only as a list/map element, but not autonomously.
* VARIABLE is an elementary value field. Variable is an autonomous primitive. Variables are usually used to store global values.
* OBJECT - data structure similar to the interface in OOP. An object is a data structure different from LIST, MAP, ELEMENT or VARIABLE. It is good to manage z/OS resources (file, terminal, etc.). Unlike LIST, MAP, ELEMENT or VARIABLE which only define one type of pseudo-class, there can be several pseudo-classes with OBJECT type. The programmer decides on the required classes and their behaviours. Examples of classes may be LIBRARY, MEMBER, VOLUME, etc. Thus, the programmer can work with z/OS resources through his own CRUD (Create, Read, Update, Delete) subprograms.
* REFERENCE is direct addressing of all HEAP data structures. It is intended for quick search of a data structure and high performance of the Garbage Collector.
* SERVICE - used to define the data provider and data formats of the HEAP application. SERVICE is defined to HEAP as an internal #SERVICES list.
* Garbage Collector (GC). Frees up unused memory. Useful in long running REXX jobs.
* TRACER - traces selected data structures within HEAP or the entire HEAP.

**How it works.** The first time an application calls a HEAP service, the HEAP is self-initialized. It creates a couple of LISTs within itself to store all the necessary constants to use TRACER, GC and other internal HEAP services. After that the user program request continues.

Within each data structure, a unique 8-byte hash is assigned. The hash of the elements is randomly generated by HEAP. In addition to hash, there is a REFERENCE that points to the data structure. A REFERENCE contains more than 8 bytes. The user program is not aware of hashes, but can access the data structure by its type and name. For example, a user query of the LIST data structure type with the name WORK will be compiled by HEAP to the NAME\_LST\_WORK reference, by which HEAP will try to find the requested data structure. When the data structure is removed, its hash and reference are also removed from the REXX memory (dropped), and the hash area of HEAP is reduced.

A screenshot of a cell phone

Description automatically generated

*Figure 3. The main view of HEAP. HASH area.*

Figure 3 shows the hash area and the HEAP data area. In this example, there are only two data structures in HEAP. In general, the HEAP data area is limited only by the amount of memory available to your REXX program. Note, that data structures are pointed by both hash and reference from the HEAP hash area.

HEAP also supports no-named data structures. In this case an 8-byte hash is used as part of the data structure name.

In the next article we will consider HEAP, including the links between its elements and important internal procedures.

End of article 10

# Start of article 11

This article reveals the internal architecture of HEAP and can be useful for those programmers who decide to develop additional HEAP features. We will discuss important HEAP procedures and internal data structures here.

I will start with a HEAP layout that allows you to link an item, make a quick search by name and quickly remove it from HEAP when needed.

A screenshot of a cell phone

Description automatically generated

*Figure 4. Detailed HEAP structure.*

The HEAP anchor is a hash area which contains an array of hashes. A hash can be a hexadecimal constant of 8 bytes length that points ELEMENT, SPECIAL, MAP, OBJECT, VARIABLE data structurs (DS) or a REFERENCE (> 8 bytes) to any HEAP DS type except VARIABLE.

HEAPIX is a HEAP DS field that contains an index of the data structure hash in the hash area. This is a back pointer from DS to the HEAP hash.

A hash in the hash area points to a DS header that contains the hash value followed by the element type (LIST, MAP, ELEMENT or OBJECT name).

The LIST, MAP, ELEMENT and OBJECT types have the same header structure. They are equally coloured in the red transparent colour in Figure 4. MAPs, ELEMENT MAPs and OBJECT DS contain **normalized stem**  eachto store data passed from the application through STMXDATA. stem.

Historically, the OBJECT type was the first HEAP DS type. It was designed to implement CRUD API with HEAP. It was designed to manage z/OS resources using the HEAP API.

A DS reference is a unique value consisting of 3 parts ["NAME", item type, <object Name>] connected by the symbol "\_".

No-Name elements use their hashes as the object names.

**NOTE ! Stem STMXDATA is used in the HEAP API to exchange data between HEAP data structures and user applications.**

In the code below a no-named list will be created. The list will contain one no-named element. The element will contain an array (stem) with 2 items , GL01 & GL02 :

Example 11

**stmxdata.NAME=''**

**stmxdata.0 = 2**

**stmxdata.1 = "GL01"**

**stmxdata.2 = "GL02"**

**parse value CLSMNGR("create","list") with retcode retref .**

**if retcode <> 0**

**then**

**do**

**say "LIST creation failed with RC="retcode**

**end**

**else**

**do**

**say "LIST creating succeeded,reference="retref**

**end**

Note that the STMXDATA stem is a normalized stem. Using a non-normalized stem as an STMXDATA can lead to unpredictable results and program failures.

The data from the STMXDATA is copied by HEAP to the normalized stem within the new added element that is pointed by **HEAP .<hash>** . Hash and reference values are generated by HEAP for each new element and added to the hash area.

The completion code of this function API will be returned in **retcode** field.

The reference will be returned in **retref** field.

The example below will create a named list LIST01 that will contain a named element EL01. The element will contain an array (stem) with 2 items , GL01 & GL02 :

**stmxdata.NAME='EL01'**

**stmxdata.0 = 2**

**stmxdata.1 = "GL01"**

**stmxdata.2 = "GL02"**

**parse value CLSMNGR("create","list",”LIST01”) with retcode retref .**

The maximum length of name is 8 and names should not start with a '#' character. Only the names of HEAP internal items may star with ‘#’ character.

The "search by reference " is shown below on the fragment taken from the HTEST01 execution trace (member $TSOHT1 in JCLLIB). Note the following in the trace :

* HEAP.1 contains hash **LTVPJGMC** that points to #INTERNAL list.
* HEAP.2 contains the reference NAME\_LST\_#INTERNAL which points to #INTERNAL list.

Below is a trace fragment containing the important links around the #INTERNAL list.

20/10/19 13:25:43.591 HEAP.1 :LTVPJGMC

20/10/19 13:25:43.592 HEAP.LTVPJGMC\_<HEAD> :LTVPJGMC LIST

20/10/19 13:25:43.593 HEAP.LTVPJGMC\_NAME :NAME\_LST\_#INTERNAL

20/10/19 13:25:43.594 HEAP.LTVPJGMC\_<TYPE> :LIST

20/10/19 13:25:43.595 HEAP.LTVPJGMC\_HEAPIX :1

20/10/19 13:25:43.595 HEAP.LTVPJGMC\_SORTED :NO

20/10/19 13:25:43.596 HEAP.LTVPJGMC\_FIRST :QUOMDS6G ELEMENT

20/10/19 13:25:43.605 HEAP.LTVPJGMC\_LAST :WT7EV9OE ELEMENT

. . .

20/10/19 13:25:43.723 HEAP.WT7EV9OE.3 :<TYPE> <clslist

20/10/19 13:25:43.723 HEAP.WT7EV9OE.4 :HEAPIX HEAPIX

20/10/19 13:25:43.724 HEAP.WT7EV9OE.5 :SORTED SORTED

20/10/19 13:25:43.725 HEAP.WT7EV9OE.6 :FIRST FIRST

20/10/19 13:25:43.725 HEAP.WT7EV9OE.7 :LAST LAST

20/10/19 13:25:43.726 HEAP.2 :NAME\_LST\_#INTERNAL

20/10/19 13:25:43.727 HEAP.NAME\_LST\_#INTERNAL :LTVPJGMC 2

20/10/19 13:25:43.728 HEAP.3 :WT7EV9OE

20/10/19 13:25:43.729 HEAP.WT7EV9OE\_<HEAD> :WT7EV9OE ELEMENT

The following displays mutual data links from the above trace :

LTVPJGMC LIST - hash HEADER of list #INTERNAL

NAME\_LST\_#INTERNAL - RHASH (reference)

LTVPJGMC 2 - reference HEADER of list #INTERNAL

LTVPJGMC\_NAME - HEADER\_PATH to reference name of list #INTERNAL

Note NAME\_LST\_#INTERNAL points to the #INTERNAL list header "LTVPJGMC 2".

You can run HTEST01 (job $TSOHT1) to better understand the internal links in HEAP.

Note that trace fields with angle brackets are virtual and do not really exist in HEAP.

They are for TRACE only.

Below is a list of the most important procedures that manage HEAP :

GET\_ROUTINE\_NAME : procedure expose HEAP.

returns the routine name for requested action in class

FIND\_ACTION\_IN\_STEM : procedure expose HEAP.

returns index of element in stem or 0 if the action is not allowed

GC : procedure expose HEAP.

garbage collector

GC routines :

REMOVE\_HASH\_FROM\_HEAP\_INDEX : procedure expose HEAP.

call SHRINK\_HEAP\_INDEX index

drop removed hash by index

SHRINK\_HEAP\_INDEX : procedure expose HEAP.

move last HEAP index to the place of the removed index and shrink the HEAP index

HEAP optimization routines :

NOTE !!!

There are two HEAP internal objects : HEADER PATH and REFERENCE .

Assume hash is Q4Z0SAJK for object MEM001 of class MEMBER. (see OBJECT class explanation)

The qualified object name in HEAP will be set to NAME\_OBJ\_MEMBER\_MEM001 (so called reference) and it will be kept in HEAP.Q4Z0SAJK\_NAME (header path).

REFERENCE is a special hash which length > 8. REFERENCE is a unique value consisting of 3 parts ["NAME", item type, <object Name>] connected by the symbol "\_". The REFERENCE value is unique within an OBJECT class.

A short summary of HASH and REFERENCE :

* REFERENCE is generated on base of the class dependent prefix, class name and object name and stored in the HEAP index. It occupies more than 8 bytes.
* HEADER PATH keeps the object REFERENCE
* REFERENCE points to a pair <HEADER hash ,hash areaindex> where :
* HEADER hash points to the item HEADER
* Hash areaindex is the numeric index of this reference in the HEAP HASH AREA, a reference back index
* HEAP."hash\_"HEAPIX always contains the numeric index of the item in the HEAP HASH AREA, a hash back index
* HEAP.hash contains a pair of <hash classname>

The following is the HASH & REFERENCE internal HEAP API (subroutines’ signatures) :

GET\_HEAP\_INDEX\_BY\_HEADER : procedure expose HEAP.

returns HEAP index for a header pointed by HEAP.hash

GET\_HEAP\_INDEX\_FOR\_REFERENCE : procedure expose HEAP.

returns HEAP index for the reference. Used to GC the reference

GET\_HEAP\_HASH\_FOR\_REFERENCE : procedure expose HEAP.

returns HEAP hash for the reference. Used to quick search by reference

GET\_INSTANCE\_BY\_HEADER : procedure expose HEAP.

returns hash of an object by its header value

GET\_CLSNAME\_BY\_HEADER : procedure expose HEAP.

returns class name of an object by its header value

GET\_REFERENCE\_BY\_HEADER : procedure expose HEAP.

returns reference of an object by its header value

GET\_NAME\_BY\_HEADER : procedure expose HEAP.

returns NAME of an object by its header value

GET\_ITEM\_REFERENCE : procedure expose HEAP.

constructs the reference by the object name and class name

GET\_HASH\_BY\_REFERENCE : procedure expose HEAP.

returns the element hash by its reference

GET\_HEADER\_BY\_REFERENCE : procedure expose HEAP.

returns HEADER by REFERENCE

GETOBJ : procedure expose HEAP. STMXDATA.

returns class name and object name by the object HEAP hash

CHECK\_HASH : procedure expose HEAP.

checks existence of object in HEAP with a certain hash. YES = 1 , NO = 0

ADD\_HASH : procedure expose HEAP.

adds REFERENCE to HEAP

GET\_HASH : procedure expose HEAP.

generates a new unique hash

SET\_NEW\_HASH : procedure expose HEAP.

sets either type of hash or reference to HEAP with the back links.

Few words about data structures :

Each data structure has its corresponding set of procedures. For example :

* LIST\_CREATE, LIST\_ADD and other LIST\_<action> subroutines.
* MAP\_CREATE, MAP\_ADD and other procedures MAP\_<action>.

There is also an external HEAP API that provides the direct access to HEAP global subroutines (not through CLSMNGR). It can be used to query HEAP statistics, making HEAP serialization, etc. The names of the global subroutines end with \_EXT suffix. You can also use them if you want to have fewer HEAP tracing messages. For example:

GET\_HEAP\_STATISTICS\_EXT

HEAP\_SERIALIZE\_EXT

End of article 11

# Start of article 12

In this article we will discuss programming with HEAP .

The first example is BX13. It demonstrates how to work with CRUD using the OBJECT class.

Note the ++INC MBRHLPR, which provides a CRUD implementation for the library member. You only need to supplement the code in MBRHLPR by calling LMMNGR through the common LIBxxx API in CFHELPER. Use $TSOBX13 from JCLLIB to run BX13 example.

Next example is HTEST01 which runs $TSOHT1 from JCLLIB. This is a test that checks the functions of the LIST API - LIST: CREATE,DESTROY and ELEMENT:ADD,INSERT,EXCHANGE,DELETE. The test checks named and no-named lists. The LIST API also has a SORT feature which is not tested in HTEST01. The reason is that the SORT function cannot be tested offline like other API functions. The SORT function can only be run in a wrapper as part of a service. We will talk about this later.

The next example is HTEST02 which is executed with $TSOHT2 from JCLLIB. The purpose of this test is to check the getter and setter for the VARIABLE data structure. The source code is simple and clear enough.

And after all the main example is BX14 which is executed with $TSOBX14 in JCLLIB.

The user application launches the HEAP services. BX14 launches 3 jobs in the following order:

* <GETLIB,F> - a job to print all members of the library in direct order.
* <GETLIB,B> - print jobs for all library members in reverse order.
* <GETLIBS,F> - the task on the press of all members of library in the direct order from the sorted library directory.

Each of the above jobs performs HEAP service, which is defined in SRVCHLPR, a member of COPYLIB :

**usr\_services.0 = 2**

**/\* name provider\_mame class object sort \*/**

**usr\_services.1 = "GETLIB GETLIB\_PROVIDER LIST LISTLIB NO "**

**usr\_services.2 = "GETLIBS GETLIBS\_PROVIDER LIST LISTLIBS YES "**

There are two services here : GETLIB and GETLIBS. Both they define providers, list data names and sorting requests. Thus, for example, the service GETLIBS defines a provider GETLIBS\_PROVIDER, which will prepare a sorted LIST in HEAP with the name LISTLIBS. The column **class** in the service definition table may contain LIST only in SummeRx version 2.0. In future versions, another classes may be added. For example, MAP.

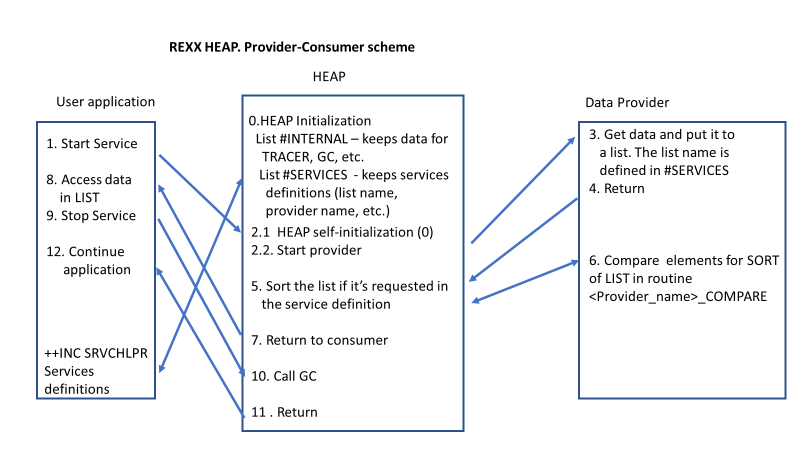
The code of both providers is included from the PRVHLPR member in COPYLIB. Note how providers access zOS libraries through the LIBxxx APIs, which in their turn call the LMMNGR APIs.

Also note the GETLIBS\_PROVIDER\_COMPARE routine, which sets the sort order according to the first card data of the member. (This only makes sense to demonstrate the flexibility of the service sorting function). The bubble sort is performed by HEAP and the trace messages are stored to SYSFWPRT file.

After all, when the data is stored in the HEAP list, the provider finishes and the consumer uses the prepared data.

The application goes through the LISTLIB list first forward and then backward and finally goes forward through the sorted LISTLIBS list. When the application stops the service, the GC removes the corresponding list from memory.

Below you can see the "provider-consumer" scheme:



I hope you mainly got the idea of using REXX with HEAP and can now continue to write your own applications using these features.

End of article 12

# Start of article 13

MAP and new message format

Let's discuss the tracing messages in the application. First, it is very important to have sufficient tracing data to show how the process works. Second, tracing messages should not spoil the output of your application with technical data. It is also a good idea to control the level of messages in your application by filtering unnecessary messages with high details. As you could see from the previous examples, these functions are implemented in SummeRx.

But with the existence of the HEAP, the following question comes up:

**Why not store text messages in HEAP?**

This is a good idea, especially for repetitive messages in your code. It also has other advantages. We will discuss them in this article.

You may notice that so far SummeRx uses the MSG functions in the following way :

MSG("...<message data>"). The number of periods preceding the message data corresponds to the message level and is used to filter the trace messages. This is the old SummeRx message format. It can also use the framework type as the first parameter to route SummeRx framework messages to a separate file. The following types of framework are available in SummeRx:

* H, heap
* T, TCP/IP
* D, 3270
* L, Library system
* F, File System
* C, Framework Common layer

For example, MSG("H","...<data messages>") indicates that the message originates from HEAP and SummeRx places the message in the SYSFWPRT file with the <H> prefix. Thus, only the application's messages are placed in SYSTSPRT. If we find an exception in the SYSTSPRT file, we can simply investigate the reason of the exception by looking at the messages around the timestamp in the SYSFWPRT file.

For some reasons, we need a new message format to show a brief report on running the application to the end user whose resolution to application problems is greater than that of the application developer. All he is interested in is whether there were problems and warnings during the application run. He would also like to receive messages in human language rather than technical "jargon".

SummeRx offers a solution for this with **explanatory messages**. An explanatory message is a message that is associated with an application message or with a framework message.

To provide such messages’ links SummeRx uses indexes of regular messages. Both normal and explanatory messages are elements of a certain stem. An explanatory message is linked to a regular message using the index, as in FRMARRAY members.

Consider a fragment of a FRMARRAY member:

**SMRX.62 = "4W $$P1 completed with RC=$$P2”**

**SMRX.79 = "062 LMHELPER:Request $$P1 commpleted with code $$P2”**

In the above code fragment , SMRX.79 is an explanatory message of SMRX.62. These messages are linked by index of 62.

**How is this used in code ?**

Find the following line in LMHELPER :

**call LMSG 62,arg(1),arg(2)**

During message processing, the content of SMRX.62 will be placed in the SYSFWPRT file and the content of SMRX.79 will be placed in the SYSEXPRT file. During the message processing, the parameter value will be changed. The values of arg(1) and arg(2) arguments will replace the $$P1 and $$P2 parameter fillers respectively.

**How this works.**

When loading new formatted messages into HEAP, SummeRx adds regular and explanatory messages to the list and binds them using the map.

There are 2 **<list,map>** pairs for applications and framework messages.

* <APPLLIST,APPLMAP> - used for application messages.
* <FWLIST, FWMAP> - used for framework messages.

SummeRx will automatically create a Message Store when it finds a new formatted message coming for the first time. HEAP will be implicitly created if it does not already exist. Take a look at the INJHLPR member to understand this in detail.

When preparing an application, you must set the name of the application message member to variable **injstm.APPLARR** in the base configuration element. See, for example, SETBASE member.

MSGARRAY is a COPYLIB member. Keep the signature of the MSGARRAY routine and SPIX as the stem name.

The regular message format is prefixed with 3 bytes :

Byte 1: number, message level.

Byte 2 : I/W/E - informational, warning or error message respectfully.

Byte 3 : action : empty, do nothing ; T - terminate the application abnormally ; A - put the framework message in the application file SYSTSPRT.

Using the termination 'T' option enriches the flexibility of the program, but 'breaks' the usual programming style. Take your own thoughts about whether you should use this option in your applications.

End of article 13

# Start of article 14

SummeRx offers HEAP marshalling across the application boundary. Let's assume that we have 2 applications working together at levels 1 and 2. The Application A starts on the level 1, builds HEAP and calls the Appllication B which works on the level 2. SummeRx can transfer HEAP between applications on a bilateral basis. This is done invisibly to user applications. A process is called serialization/deserialization and HEAP data is transmitted using a temporary sequential file. SummeRx calculates the file size required to contain HEAP data on its own, based on the HEAP statistics. The temporary file will contain REXX assign statements for the HEAP stem items. Thus, the deserialization process is simply to execute the statements of this file.

SummeRx supports 3 types of HEAP exchange between applications :

* **CPY** - A passes HEAP to B and remains unaware of updates made by application B.
* **SEP** - A and B use separate HEAPs. A does not pass HEAP to B. B builds the HEAP on its own.
* **SHR** - A and B logically share the same HEAP. A passes HEAP to B. B makes updates to HEAP and passes it back to A.

In HTEST06 you can see the following statement :

**retcode = INJ\_CALLEXEC(HEAP\_SHR, 'HTEST07' 'H071' msg\_level)**

HEAP\_SHR is a variable that can have one of the HEAP exchange types: CPY, SEP or SHR.

Work $TSOHT6 starts the program HTEST06, which calls the program HTEST07 four times:

1. Demonstrates work without HEAP
2. Demonstrates HEAP exchange in SHR mode. HTEST06 creates HEAP items and calls HTEST07 to remove these items from HEAP.
3. HTEST07 simulates the syntax error that is handled by SummeRx. The HEAP will be returned to the caller. We haven't discussed this SummeRx feature yet, but it is not difficult for understanding from the example.
4. HTEST07 produces a new format message with a termination option. SummeRx will terminate the program. The HEAP will return to the caller.

In addition, HTEST06 tests the situation when HEAP is created without a Message Store. It should not filter out old format messages.

You can test different share options in the HEAP\_SHR variable and learn how SummeRx exchanges HEAP with different share types.

End of article 14

End of article 14

# Start of article 15

In this article we will discuss another example of a TCPIP server that uses HEAP and then give a brief overview of the SummeRx framework.

The HTEST08 program is based on the BX12 example and can serve as a good server template.

At the beginning the program creates a HEAP MAP to keep information for all users successfully connected to the server.

When a user connects to the server, the program receives a request consisting of the user name, password and data. The usual USER\_VALIDATION procedure is designed to check users against ESM. You will need to fill in the routine code.

APPL\_RECEIVE routine generates a token for the verified user and adds it to the MAP.

APPL\_SEND routine looks for the token of the user in the MAP. If the token is found, it concludes that the user is authorized and sends him a greeting, otherwise the request is denied.

Run the $TSOHT8 job to start the server. Use TCPCLNT client for testing. You can run TCPCLNT a couple of times. Note that you can use HEAP for the server on manner of in-memory database. You can store detailed information about the user in HEAP. You can also send a token to the user so that he can submit a token for verification against the server next time, not a password. In other words, you can develop your own proprietary policy to identify the user of the request to allow or deny the server to perform the request. You can also use the z/OS AT-TLS function to protect your traffic. To stop the server, you must issue a CANCEL command. In future versions of SummeRx the server control through STOP and MODIFY commands will be added.

**Summary.**

The SummeRx framework can help to solve a couple of problems that a developer may encounter.

It offers the following features which can make writing code into REXX a real pleasure :

* Managing common routines separately from the main routine
* Dynamic Code Parts Injection
* REXX syntax extension
* Easy movement between TEST and PROD environments
* Simplicity of writing 3270 applications
* Templates for TCPIP server
* Powerful trace and messaging
* Runtime Error Handling
* Programming with HEAP primitives
* Sharing HEAP between applications

Can anyone now say that REXX is not a platform for writing serious applications?

Indeed, why not try to write a task scheduler or a long running task that can drastically reduce the time spent on night window processes?

Looking forward, I see SummeRx growing by improving existing controllers and developing new ones for DB2, VSAM, CONSOLE, Linux Containers and other APIs to the latest technologies of today.

End of article 15

# Start of article 16

Unix System Services, future development.

End of article 16

# Start of article 17

Rexx + java + Linux. Future development.

End of article 17

# Bibliography

BatchPipes OS/390 V2R1 BatchPipeWorks Reference (SA22-7456-01) SK3T-4273-01

All collections containing SK3T-4273-01

Collection SK2T-6700-21 , 2001