REXX/IUCV PACKAGE

REXXIUCV:
VM REXX PROGRAMMING SUPPORT FOR IUCV

SHARE 75 (NEW ORLEANS, LOUISIANA)
SESSION O739

August, 1990

Rainer F. Hauser

IBM Research Division
Zurich Research Laboratory
Säumerstrasse 4
CH - 8803 Rüschlikon
Switzerland

BITNET/EARN/VNET: RFH@ZURLVM1
INTERNET: rfh@ibm.com

Abstract

REXXIUCV allows the use of the VM/SP or VM/XA SP Inter-User Communications Vehicle (IUCV) with the Restructured Extended Executor (REXX) language under CMS. In the first part, communications in general and IUCV in particular are shortly introduced. The second part presents the design, implementation, and use of REXXIUCV. Finally, possible and real applications are discussed to demonstrate its use.

Introduction

REXXIUCV (or also called REXX/IUCV or RXIUCVFN) allows the use of IUCV (Inter-User Communications Vehicle) with the REXX language under CMS. Application Systems which involve multiple virtual machines can now be implemented entirely in REXX. The same holds for applications which access services provided by subsystems in other virtual machines or functions offered by CP via IUCV. All of the advantages of using REXX can be exploited for prototyping, realizing, and testing applications without the need of low-level interface programming. REXXIUCV supports multiple communication paths in parallel and offers flexible handling of events, including IUCV and timer interrupts.

Before we describe REXXIUCV in detail, we introduce shortly some basic communications concepts and terminology needed to understand IUCV as depicted in Figure 1.

```
+-------------------------------+  +-------------------------------+
|VM                             |  |VM                             |
|            |    A             |  |                               |
| +----------|----|-----------+ |  |                               |
| |REXXIUCV  +----+           | |  |                               |
| |                           | |  |                               |
| |          |    A           | |  |             |    A            |
+-+----------|----|-----------+-+  +-------------|----|------------+
+------------|----|------------------------------|----|------------+
|CP          V    |        --------------+       V    |            |
|                                     ||||                         |
|                          --------------+                          |
|                          +-------------                           |
|                          ||||||                                   |
|                          +-------------                           |
+------------------------------------------------------------------+
```

Figure 1. IUCV as a Communication Service

IUCV is a communication medium available to virtual machines to communicate among themselves and with certain CP facilities. The communication service offered by IUCV can be modeled as a set of message queues through which users can exchange data.

IUCV being a *connection-oriented* service requires that two virtual machines must first establish a connection (i.e. an IUCV path) between them, before they can actually communicate, and eventually will close the connection after termination of the data exchange. Accordingly, these three distinct phases are called connection establishment phase, data phase, and termination phase.

The IUCV service is available to its users through interaction across the service interface, i.e. across the boundary between CP and each virtual machine. This interaction, seen from a virtual machine, is described in “VM/SP: System Facilities for Programming” (SC24-5288) and “VM/XA SP: CP Programming Services” (SC23-0370) in terms of the *IUCV machine instruction*, the *IUCV interrupt*, and the many associated parameters with their formats. Seen as service primitives, these interactions include commands to initialize and terminate IUCV use (DCLBFR, RTRVBFR), to establish and close IUCV paths (CONNECT, ACCEPT, SEVER), and to to exchange message data (SEND, RECEIVE, REPLY).

In contrast to this assembler-level interface to IUCV, REXXIUCV offers the full IUCV service to REXX programs in terms of the REXX language. That is, IUCV is used through a single REXX function, where parameters and results are easy-to-parse data strings.

Description of REXXIUCV

REXXIUCV provides a REXX interface to IUCV or CMS IUCV in addition to the assembler interface available through VM/SP and VM/XA SP. When loaded, it offers a built-in function called *IUCV* in REXX which allows access to the complete IUCV facility.

RXIUCVFN Function Syntax

When the module RXIUCVFN is loaded, the built-in function *IUCV* can be issued through the REXX function call

$$var = \text{IUCV}(\textbf{function}, \text{arg}_1, \text{arg}_2, ..., \text{arg}_n)$$

in a REXX program, where **function** is a function keyword (service primitive) such as CONNECT, and the arg_i are function-dependent further arguments.

There are two functions manipulating the IUCV environment. The keyword INIT activates IUCV either with DCLBFR or through the the CMS IUCV interface depending on the other arguments. The keyword TERM deactivates IUCV after termination of all IUCV paths created using REXXIUCV. The functions CONNECT, ACCEPT, SEVER, QUIESCE, and RESUME manipulate IUCV paths. They correspond to the IUCV functions with the same name. Similarly, the functions SEND, PURGE, RECEIVE, REJECT, and REPLY handle IUCV messages. In addition, the two function keywords QUERY and WAIT allow determination of the status and waiting for events (interrupts), respectively.

Example

To demonstrate the use of REXXIUCV, we present two simple REXX programs. They are to be executed in two different virtual machines on the same VM system. One program (RCVSAMPL EXEC) — to be started first — reacts passively to connection requests from the other program (SNDSAMPL EXEC). In order to keep them as small as possible, both programs do not contain the code normally used for various checks and corresponding error messages. In other words, they are neither robust nor user-friendly.

After some initialization, the program RCVSAMPL will wait for IUCV interrupts and process them. When an IUCV path gets severed, it displays the number of messages received on this path. RCVSAMPL is given as follows:

```
/* Initialize REXX/IUCV                                                 */
address command 'CP SET TIMER REAL'                          /* Note  1 */
address command 'RXIUCVFN LOAD'                              /* Note  2 */
if rc¬=0 then exit rc
Temp = IUCV('INIT',2000,'Appl','EXAMPLE ')                   /* Note  2 */
if rc¬=0 then exit 100+rc

/* Process IUCV Interrupts                                              */
do forever                                                   /* Note  5 */
  Temp = IUCV('WAIT',900,'NOWAIT')                           /* Note  4 */
  QueryNextInterrupt = IUCV('QUERY','NEXT')                  /* Note  4 */
  parse var QueryNextInterrupt Pending Type PathId . TrgCls .
  select
    when Type=1 then do       /* Pending Connection          -  Note  7 */
      Temp = IUCV('ACCEPT',PathId,255,'No',left('OK',16))    /* Note  7 */
      Count.PathId = 0
    end
    when Type=3 then do       /* Path Severed                -  Note  9 */
      Temp = IUCV('SEVER',PathId,left('End',16))             /* Note  9 */
      say Count.PathId 'messages received on path' PathId
    end
    when Type=9 then do       /* Pending Message                        -  Note  8 */
      Message = IUCV('RECEIVE',PathId,TrgCls)                           /* Note  8 */
      if rc=0 then Count.PathId = Count.PathId + 1
    end
    otherwise nop
  end
end

/* Terminate REXX/IUCV                                                             */
Temp = IUCV('TERM')                                                     /* Note 10 */
address command 'NUCXDROP RXIUCVFN'                                     /* Note 10 */
```

After the necessary initialization, the program SNDSAMPL actively connects to the virtual machine running RCVSAMPL EXEC. When the IUCV path is successfully established, it starts sending messages. SNDSAMPL looks as follows:

```
/* Initialize REXX/IUCV                                                            */
address command 'CP SET TIMER REAL'                                     /* Note  1 */
address command 'RXIUCVFN LOAD'                                         /* Note  2 */
if rc¬=0 then exit rc
NumPath = IUCV('INIT',1,'Appl','Temp'||right(random(0,9999),4,'0')) /* Note  2 */
if rc¬=0 then exit 100+rc

/* Connect to RCVSAMPL EXEC                                                        */
parse arg VmId Count Message                                            /* Note  3 */
Temp = IUCV('CONNECT',VmId,255,'Noprio','EXAMPLE ')                     /* Note  6 */
if rc=0 then Error = 0
else Error = 1
parse var Temp . MsgLim .

/* Process IUCV Interrupts                                                         */
do while(¬Error)                                                        /* Note  5 */
  Temp = IUCV('WAIT',900,'NOWAIT')                                      /* Note  4 */
  Temp = IUCV('QUERY','NEXT')                                           /* Note  4 */
  parse var Temp Pending Type PathId Others
  select
    when Type=2 then do       /* Connection Complete                    -  Note  7 */
      Sent      = 0
      Received  = 0
      rc = SENDIT(PathId,Message)                                       /* Note  8 */
      if rc¬=0 then Error = 1
    end
    when Type=7 then do       /* Nonprio Msg Complete                   -  Note  8 */
      Received = Received + 1
      rc = SENDIT(PathId,Message)                                       /* Note  8 */
      if rc¬=0 then Error = 1
    end
    otherwise nop
  end
  if Sent=Received & Sent=Count then do
    Temp = IUCV('SEVER',PathId,left('End',16))                          /* Note  8 */
    leave                                                               /* Note  8 */
  end
end

/* Terminate REXX/IUCV                                                             */
Temp = IUCV('TERM')                                                     /* Note 10 */
address command 'NUCXDROP RXIUCVFN'                                     /* Note 10 */
exit Error
```

```
/* Send Procedure                                                - Note  8 */
SENDIT: procedure expose MsgLim Count Sent Received
  PathId  = arg(1)
  Message = arg(2)
  rc = 0
  do while(rc=0 & Sent-Received<MsgLim & Sent<Count)
    Temp = IUCV('SEND',PathId,Message,0,0,'No','No','No')          /* Note  8 */
    Sent = Sent + 1
  end
return rc
```

Notes:

1. On VM/SP systems, the virtual machine should set the timer to *real* for the WAIT function to work properly. Otherwise, no timer interrupts get generated, and the program may wait for-ever. (A more sophisticated program would set this parameter back to what it was before the program was called.)
2. In an initialization phase, REXXIUCV must be loaded before the IUCV function can be called, and the IUCV facility must be initialized before a path can be established. Since both programs initialize IUCV for CMS IUCV (argument 'Appl'), a name is needed. (One program uses the name 'EXAMPLE ', and the other builds a random name not to be known externally.)
3. The RCVSAMPL program is called without arguments. When calling the SNDSAMPL program, the userid (variable *VmId*) of the partner virtual machine, the number of messages to be sent (variable *Count*), and the message itself (variable *Message*) must be given. (Note that you have to enter *VmId* in uppercase.)
4. The function keywords WAIT and QUERY together provide the means to wait and react to IUCV events. When an IUCV interrupt occurs, an interrupt buffer (kept inside REXXIUCV) describes the event. This further information is accessible using the QUERY function.
5. The functions WAIT and QUERY are called in a loop until some exit conditions are met. (The program RCVSAMPL actually never ends.)
6. Before the SNDSAMPL program enters the loop, it requests a connection to the virtual machine running the RCVSAMPL program. (The *VmId* of this virtual machine was entered when calling SNDSAMPL.) It must specify the same name (here 'EXAMPLE ') as RCVSAMPL used when initializing.
7. The virtual machine running RCVSAMPL gets an IUCV interrupt indicating a pending connection and accepts it. Accepting the path results in a connection complete type IUCV interrupt in the virtual machine running SNDSAMPL. Now, the path is established.
8. In a subroutine, the SNDSAMPL program sends as many messages as allowed by IUCV (REXX variable *MsgLim*) or as requested by the user (REXX variable *Count*). RCVSAMPL receives these messages, and SNDSAMPL can send more messages. When as many messages as requested by the user have been sent and received, the SNDSAMPL program disconnects (severs) the path.
9. Also the RCVSAMPL program severs the path to complete the disconnect phase. (Note that the RCVSAMPL program can have established several paths to different virtual machines running SNDSAMPL at the same time.)
10. In a termination phase, IUCV and REXXIUCV are cleaned up and deactivated. (If the function TERM is not called explicitly, dropping REXXIUCV from the nucleus would call it implicitly.)

Design Principles

There are two fundamental differences between the /370 Assembler and REXX. First, the assembler language represents data as an address and a length of a piece of memory, but REXX uses variables and their values for this purpose. Second, events in an assembler program are handled by an asynchronous interrupt handler while REXX only knows synchronous function calls. These two differences and the following principles guided the design of REXXIUCV:

1. The status of REXXIUCV with all established paths, pending messages, and interrupt buffers should not be destroyed when the REXX program terminates without cleaning up properly. (For testing, it is very useful if one program can terminate and another can determine the status and continue where the former program stopped.)

2. REXXIUCV should provide access to the full IUCV facility, and not only to a limited subset. (The function not provided is usually the one you would need.)
3. There should be a one-to-one mapping between REXXIUCV function keywords and IUCV functions. (Combined higher order functions such as for a complete connection establishment phase can easily be written in REXX if needed.)
4. The information contained in the IUCV interrupt buffer describing an IUCV interrupt should be available to the REXX program, but in a representation appropriate for REXX parsing.
5. REXXIUCV should allow for high scheduling flexibility in processing pending events. (A REXX program may only process events from one path and leaving the interrupts from other paths unserviced.)
6. Because the original return code from IUCV bears important information, it should be available to the REXX program. (These return codes plus some REXXIUCV specific return codes are passed to the program in the REXX variable *RC*.)
7. Limits such as the maximum number of IUCV paths supported by REXXIUCV are necessary, but should be easy to change from one version to another.

Applications

Instead of describing what REXXIUCV could be used for, we describe where we actually used it in our own projects. We designed, built and used REXXIUCV to support the implementation of an OSI transport service for VM systems and workstations. The service allowed communication between virtual machines and LAN-attached workstations. The VM side of it was accessible from the user's virtual machine via IUCV.

Testing and Experimenting

Initially, we designed REXXIUCV as a testing tool for this IUCV path to the OSI transport service. The power of REXX allowed building any character string and sending it to the service under test. We generated systematically many protocol test cases in the client's virtual machine and observed the behavior of the service. This simple and thorough testing made the service very robust. (The performance of REXX and REXXIUCV allowed even stress testing.)

Later in the project, we started using an experimental IUCV line-driver of RSCS. Since we could not get any documentation for it except the running program, we started exploring its behavior experimentally. Besides the fact that we learned all essential detail about it, we also found several errors which we reported to the developers.

Rapid Prototyping

In a very natural way, we derived some complete client programs for the OSI transport service from the programs we used for testing. More and more phases of the protocol became completely tested, and the test programs worked through these phases the same way a client program would do.

We actually intended to replace the early REXX prototype of the client program by a compiled version in another programming language, but we never started this work since there was no need for it. The difference in speed between a compiled and an interpreted language did not cause any serious degradation of the service because the time-consuming part was in the server program which was already implemented in a compiled language, rather than in the REXX client part.

Final Application Programs

As one set of final applications, we used REXXIUCV to connect to the CP service '*MSG' in order to get access to RSCS messages, messages from batch machines or other similar service machines. Such programs can be written even by casual programmers without much difficulties since they are simple and small.

We also realized very large REXX programs using REXXIUCV. Connecting to the above mentioned IUCV line-driver in RSCS, we built a complete, full-function RSCS node which provided access to the RSCS network from LAN-attached workstations. Besides a few assembler routines to invoke certain CMS features, all software of this RSCS node was written in REXX. (This program was part of the setup to demonstrate an X.400 mail service at the TELECOM'87 exhibition in Geneva, Switzerland, October 20-27, 1987, in connection with the IBM X.400 PROFS Connection.)

When already REXX as an interpreted language allowed programmers development of large applications mainly or completely in REXX, the availability of the REXX compiler encouraged even more development of large applications directly in REXX. Needless to say, programs using REXXIUCV can be compiled the same way as programs without using it.

Conclusions

REXXIUCV is a system dependent REXX extension which runs on CMS under VM/SP or VM/XA SP and allows inter-VM communication in REXX. It provides access from a high-level language to the complete IUCV facility. It is an easy-to-use interface to IUCV and a flexible tool for IUCV programming.

REXXIUCV is available as a licensed program of IBM from the Europe/Middle East/Africa Group since March 1989 under the name “VM REXX Programming Support for IUCV” and with the program number 5785-LAT. As stated in the Availability Notice, GB11-8432-0: “General availability from the Asia/Pacific Group and the Americas Group program libraries is planned to be one month later.” In addition to the licensed program (including sample REXX programs), a Program Description and Operations Manual, SB11-8433, is provided as unlicensed documentation.