Introduction to TSO and REXX APIs

Introduction to TSO and REXX APIs - Course Objectives

On successful completion of this class, the student, with the aid of the appropriate reference materials, should be able to:

1. Code application programs in Assembler, COBOL, PL/I, or C that run under TSO (including batch TSO) and that can:

   Accept and use parm data
   Set normal and abnormal return / completion codes
   Issue TSO commands (using IKJEFTSR)
   Access, modify, create, and delete REXX variables (using IKJCT441 and IRXEXCOM)
   Work with data in DB2 data bases (using DSN)

2. Use TSO and REXX EXECs to run programs in the Foreground or the Background (batch)

3. Run EXEC's in the batch, in TSO/E-integrated address spaces (IKJEFT01) or non-TSO/E-integrated address spaces (IRXJCL)

4. Code programs in Assembler, COBOL, PL/I, or C that can be invoked using the TSO REXX facilities for LINK and ATTACH (address link, address attach, address linkmvs, address attchmvs, address linkpgm, address attchpgm), accepting parm data in the multiple various formats provided by these alternatives.

Introduction to TSO and REXX APIs - Topical Outline

Day One

Introduction to TSO and REXX APIs - Topical Outline, 2

Day Two

Course Overview

❒ **This course is for programmers who need to design, code, debug, or maintain application programs that run under:**

Native TSO - TSO READY prompt

REXX execs invoked from Native TSO

REXX execs invoked under ISPF

❒ **In addition, we explore how to run TSO commands and REXX execs from compiled programs running in batch or from TSO READY or under ISPF**

- ♦ **And how to access, change, create, and delete REXX variables from a compiled program invoked from an exec running in batch or TSO**

❒ **Our focus is the programming interfaces to TSO and REXX from Assembler, COBOL, PL/I, and C**

TSO-Based Applications

❒ **You may code parts of TSO-based applications in REXX:**

- ♦ **To gain access to TSO commands and services including**
  - ✗ File create, repro, print, rename, delete
  - ✗ Provide an environment for programs that access DB2 data
- ♦ **To capture sets of commands in procedures and use symbolic substitution capabilities**
- ♦ **To use the parsing, compound symbol, EXECIO, LINK, and ATTACH facilities of REXX**

❒ **You may code parts of such applications in compiled (or Assembled) programs:**

- ♦ **For performance reasons**
- ♦ **To use non-sequential I/O, including access to DB2 data**
- ♦ **To use system services such as Data In Virtual, WAIT / POST, ENQ / DEQ, and so on**
- ♦ **To work with records in VSAM files**
- ♦ **To use Language Environment (LE) services**

TSO, REXX, and Program Interfaces

❒ **In this course we shall concentrate on the skills necessary to implement the following techniques:**

- ♦ **Running programs (written in Assembler, COBOL, PL/I, or C) using TSO CALL**

  - ✗ Including passing parameters, handling termination codes, and working with files (allocation and disposition)

- ♦ **Issuing TSO commands from programs running in a TSO environment (either from TSO READY or TSO running in batch)**

- ♦ **Intercepting abend codes and reason codes produced by a program running from an exec under TSO**

- ♦ **Creating, referencing, altering, or deleting REXX variables from a program invoked from an exec**

  - ✗ Including running under TSO or in a non-TSO address space

- ♦ **Running programs using TSO DSN**

  - ✗ Including accessing DB2 databases and communicating values to or from an exec

- ♦ **Coding programs to be invoked by REXX ADDRESS LINK or ADDRESS ATTACH**

- ♦ **Coding programs to be invoked by any of ADDRESS: LINKMVS, LINKPGM, ATTCHMVS, ATTCHPGM**

Section Preview

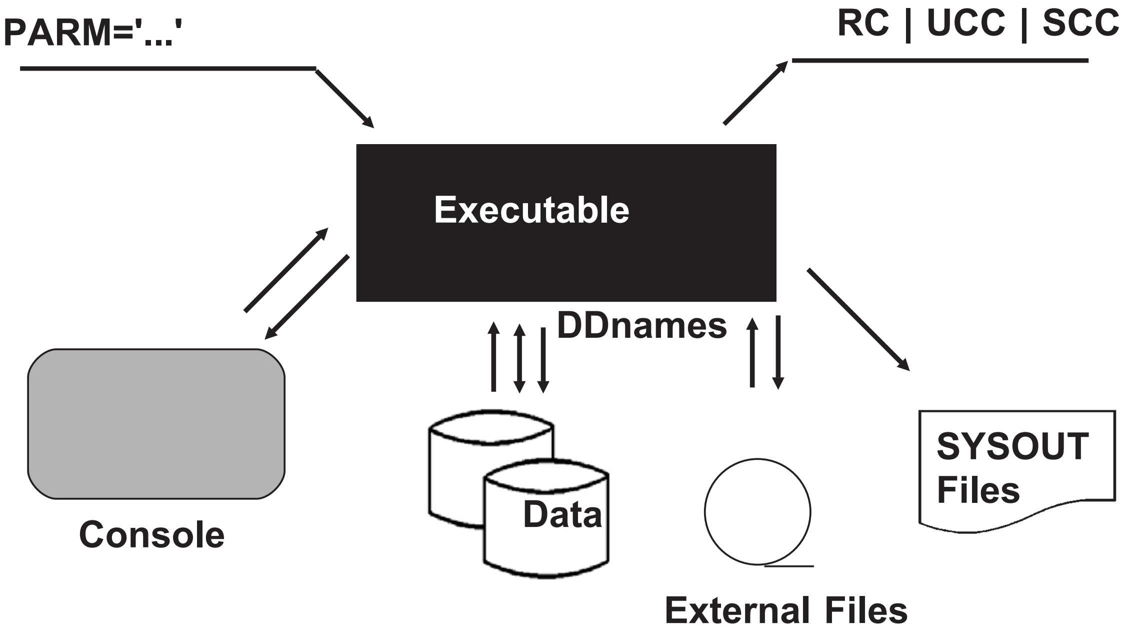
❒ **Basic Program Interfaces - Batch**

- ♦ **Accessing the data in the PARM field on the JCL EXEC statement**
- ♦ **Gaining access to external files**
- ♦ **Operator Console I/O**
- ♦ **Setting normal termination value**
- ♦ **Setting abnormal termination value**
- ♦ **Running jobs in batch**
- ♦ **Setting up files and running batch programs (Machine Exercise)**

Basic Program Interfaces

❒ **An assembled or compiled and bound program (load module or program object ["executable" for short]) accepts / produces the following inputs and outputs**

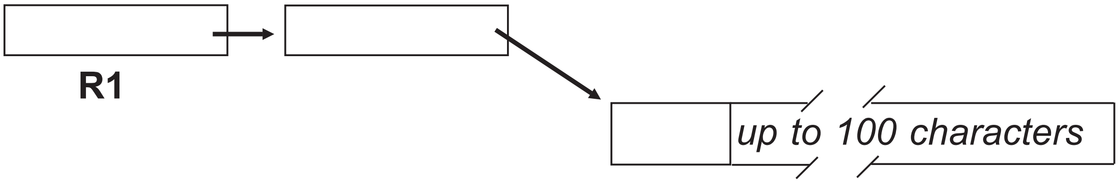
- ♦ **Parm field from the EXEC statement**
- ♦ **Records to / from files**
- ♦ **Lines to / from console**
- ♦ **Termination code**
    - ✗ Normal termination: Return Code (also Condition Code)
    - ✗ Abnormal termination: Completion Code (System or User)



Gaining Access To The PARM Field - Assembler

❒ On entry to your program, Register 1 points to a fullword in memory that points to a halfword length field followed by the PARM data

- ♦ The length field contains the length of the data only



❒ After doing basic save area chaining, if R1 has not been modified:

```
L     1,0(1)     PICK UP ADDRESS OF LENGTH FIELD
LH    2,0(1)     PICK UP LENGTH OF PARM DATA
LA    3,2(1)     POINT TO PARM DATA
```

❒ To move this variable length field to a location, say PARM_IN, use the EX instruction, something like:

```
       BCTR  2,0         DECREMENT LENGTH
       EX    2,MOVEIT    EXECUTE REMOTE INSTRUCTION
       ...
MOVEIT MVC  PARM_IN(0),0(3)    MOVE PARM DATA
```

❒ At execution time, the parm data is passed using this syntax:

```
//SNAME EXEC PGM=pgm,PARM='up to 100 characters'
```

Gaining Access To The PARM Field - COBOL

```
Id Division.
Program-id. MORTGAGE.
. . .
Data Division.
. . .
Linkage Section.
01  In-Parms.
    02  Length-of-parms Pic S9999 Binary.
    02  Data-in-parms.
        03  Date-choice Pic  X.
        03  Parm-Month  Pic 99.
        03  Parm-Day    Pic 99.
        03  Parm-Year   Pic 99.
        03  Parm-Rate   Pic V99999.
Procedure Division using In-Parms.
. . .
    If Length-of-parms = 0
    Then perform default-settings
    Else perform set-from-parm.
```

❒ **At execution time, code:**

```
//SNAME  EXEC  PGM=MORTGAGE,PARM='E011520yy825/'
```

- ♦ **and the data from the PARM field on the EXEC statement is passed into your program's "In-Parms" field automatically**

Gaining Access To The PARM Field - PL/I

❐ In mainline program, code something like this:

```
INTPMTS: PROC (IN_PARMS) OPTIONS (MAIN);
...
DCL IN_PARMS CHAR(100) VARYING;
...
IF LENGTH(IN_PARMS) = 0
THEN CALL SET_DEFAULT_SETTINGS;
ELSE  IF LENGTH(IN_PARMS) > 50
      THEN CALL BAD_PARMS;
      ELSE CALL SET_FROM_PARMS;
...
```

❐ At execution time, code something like:

```
//SNAME  EXEC  PGM=INTPMTS,PARM='/E011520yy825'
```

- ♦ and the data from the PARM field on the EXEC statement is passed into your program's “IN_PARMS” field automatically

Gaining Access To The PARM Field - C

- **In C, the situation is a little more complex**

- **C first parses any parm data, creating an array of words**
  - **A word is defined to be blank-delimited; extra leading and trailing blanks are deleted**

- **The C main function sees two arguments, one is the count of the number of words, the second is an array of character strings, one word per element, so your argument definition looks like this:**

```
void main(int argc, char *argv[]);
```

  - **We can only approximate the same output as in the other examples, something like this:**

```
char * parm_ptr;
char   char_work [100];
short  i;
void main(int argc, char *argv[]);
{
  if (argc >1)
      {
        strcpy(char_work, "Parm= ");
        for (i=1;i<argc;i++)
         {
         parm_ptr = strcat(char_work, argv[i]);
         parm_ptr = strcat(char_work, " ");
         }
        printf("%s",parm_ptr);
      }
```

  - **Also, argv[0] will contain the program name**

Gaining Access To The PARM Field - C, continued

- **Note that** printf **output goes to a DD statement named SYSPRINT if one is available; otherwise it uses a DD name of SYS0000*n*, which will be dynamically allocated at run-time**

- **If a C program is compiled with the NOARGPARSE option, then you will get the string as it is entered from: 1) the code on the previous page; 2)** scanf **of the parm (argv[1]); 3) the LE CEE3PRM service (not discussed in this course); and 4) this code:**

```
 _VSTRING ParmMsg;
. . .
  strcpy(ParmMsg.string), "Parm = ");
  parm_ptr = strcat(ParmMsg.string, argv[1]);
  ParmMsg.length = strlen(ParmMsg.string);
  CEEMOUT(&ParmMsg, &dest, &fc);
```

- **If you include a #pragma runopts(noexecops) in your program, at run time, the LE run-time options are not passed to LE but the whole parm string is passed to your program…**

```
#pragma runopts(noexecops)
#include <stdio.h>
#include <string.h>
char * parm_ptr;
.
.
.
```

Notes On PARM Data

- Non-LE-conforming Assembler programs take their PARM data “straight”:

```
          PARM='user-data'
```

- COBOL Programs expect any user PARM data followed, optionally, by a slash and any LE run-time parms:

```
          PARM='user-parms/LE-parms'
```

- PL/I, C, and LE-conforming Assembler programs expect any LE run-time parms followed, optionally, by a slash and any user PARM data:

```
          PARM='LE-parms/user-parms'
```

- Generally, accessing the PARM from your program using the above techniques returns the user-parms portion

Gaining Access To External Files

❒ **Basic process the same in all languages**

- ♦ **Define / declare files, specifying a DDname**
  - ✗ Assembler: DCB or ACB macro, DDNAME= parameter
  - ✗ COBOL: SELECT filename ASSIGN TO ddname
  - ✗ PL/I: DCL filename FILE RECORD …
    - ➢ filename is DDname unless OPEN with TITLE option
  - ✗ C: declare a variable as type file pointer: FILE **file-handle*, then ddname or filename in fopen() for that file

- ♦ **Define / declare data areas, end-of-file switches, etc.**
  - ✗ Assembler: DS and DC statements
  - ✗ COBOL: Data Division definitions
  - ✗ PL/I: DECLARE statements
  - ✗ C: item and structure declares

- ♦ **Issue I/O verbs, for example (not exhaustive):**
  - ✗ Assembler: OPEN, CLOSE, GET, PUT, PUTX
  - ✗ COBOL: OPEN, CLOSE, READ, WRITE, REWRITE, DELETE
  - ✗ PL/I: OPEN, CLOSE, READ, WRITE, REWRITE, DELETE
  - ✗ C: fopen(), fread(), fwrite(), fclose()

Operator Console I/O

❒ **This is discouraged, but is occasionally useful**

- ♦ **Assembler: WTO and WTOR macros**
- ♦ **COBOL: DISPLAY UPON CONSOLE and ACCEPT FROM CONSOLE statements**
- ♦ **PL/I: DISPLAY and DISPLAY … REPLY statements**
- ♦ **C: __console() function**

❒ **Also, can send some output to job listing instead of console**

- ♦ **Assembler: WTO with ROUTCDE=11**
  - ✗ goes to system message dataset for job: the JCL listing
- ♦ **COBOL: DISPLAY UPON SYSOUT**
  - ✗ goes to SYSOUT DDname
- ♦ **PL/I: PUT {DATA|LIST|EDIT}**
  - ✗ goes to SYSPRINT DDname
- ♦ **C: printf() goes to SYSPRINT, SYSTERM, or SYSERR if any of these are allocated; otherwise the runtime dynamically allocates SYS000*n* and uses that**
- ♦ **All languages: the LE CEEMOUT and CEEMSG routines**

Setting Normal Termination Value

❒ **To set the Return Code, or Condition Code, for testing in JCL:**

- ♦ **Assembler: Value in R15, then RETURN (14,12),,RC=(15)**

- ♦ **COBOL: Value in RETURN-CODE special register**

- ♦ **PL/I: CALL PLIRETC (*value*);**

  - ✗ Remember to declare PLIRETC as BUILTIN

- ♦ **C: Specify the value in a 'return' statement: return(*value*);**

  - ✗ Note that for this to work you must specify the prototype for the main function to return an integer instead of 'void':

```
int  main (int argc, char *argv[]);
```

- ♦ **In all languages you can just call the LE CEE3SRC service**

Setting Abnormal Termination Value

❐ **To set an Abnormal Termination value (System Completion Code or User Completion Code)**

- ♦ **z/OS assigns System Completion Code (*e.g.*: S0C7)**

- ♦ **Assembler User Completion Code**

  - ✗ ABEND nnnn

- ♦ **COBOL User Completion Code**

  - ✗ CALL 'ILBOABN0' USING identifier

- ♦ **PL/I User Completion Code**

  - ✗ Requires installation modification of IBM-supplied module IBMBEER

- ♦ **C User Completion Code: issue a return() from a signal catcher**

- ♦ **All languages: call the LE services CEE3ABD or CEE3AB2**

Accessing Termination Codes

❒ In batch, the Return Code value may be tested on subsequent steps by the COND parameter on the EXEC statement

- ♦ The values are also displayed on the JCL listing

❒ Completion Codes are not testable, although they are displayed on the JCL listing

❒ IF in JCL can test condition codes and completion codes

Some Examples

```
//STEPPER  IF  STEP5.RC > 8 THEN
...
//  ENDIF


//STAMPER  IF  ABEND  THEN
...
//  ENDIF


//STOMPER  IF  ABENDCC=S013  THEN
...
//  ENDIF
```

Coding JCL To Run Jobs In The Batch

- To prepare for the various interfaces for a program to be run in batch, JCL might look like this:

```
 //jobname  JOB  --job statement parameters
[//JOBLIB    DD   DSN=libraryname,DISP=SHR]
 //stepname EXEC  PGM=pgmname,PARM='parm data'
[//STEPLIB   DD   DSN=libraryname,DISP=SHR]
 //ddname    DD   DSN=dsname,--dd statement parms
 //ddname    DD   DSN=...
 //ddname    DD   SYSOUT=.
 //stepname EXEC  PGM=pgmname,PARM='...'[,COND=]
[//STEPLIB   DD   DSN=libraryname,DISP=SHR]
 //ddname    DD   DSN=dsname,--dd statement parms
 //ddname    DD   DSN=...
 //ddname    DD   SYSOUT=.
 . . .
```

- Plus any other special DD statements:
  - //SYSUDUMP for dumps in the event of ABEND or CEEDUMP for an LE dump
  - //SYSOUT for COBOL messages and LE messages
  - //SYSPRINT for PL/I messages
  - //PLIDUMP for PL/I debugging information
  - //SYSTERM or //SYSPRINT or //SYSERR for C printf() output

Running Jobs In Batch

❒ **JCL set up to run a job is placed into the batch job queues by way of the SUBMIT command**

♦ **In ISPF/PDF edit or view of the JCL, on the command line type**

```
SUBMIT
```

♦ **In ISPF/PDF, outside of edit and option 6, on the command line type**

```
TSO SUBMIT 'libraryname(membername)'
```

or just

```
TSO SUBMIT name(membername)
```

if library name is of the form: <userid>.name.CNTL

♦ **Outside of ISPF (from TSO 'READY') or at ISPF/PDF option 6, or from a CLIST or REXX exec:**

```
SUBMIT 'libraryname(membername)'
```

or

```
SUBMIT name(membername)
```

❒ **SUBMIT may be abbreviated SUB in all cases**

Computer Exercise: Running Batch Programs

This machine exercise is designed to provide setup for all the remaining class exercises.

First, you need to run A780STRT, a supplied REXX exec that will prompt you for the high level qualifier (HLQ) you want to use for your data set names; the exec uses a default of your TSO id, and that is usually fine. Then the exec creates data sets and copies members you will need.

From ISPF option 6, on the command line enter:

```
   ===> ex '__________.train.library(a780strt)' exec
```

A panel displays for you to specify the HLQ for your data sets, with your TSO id already filled in. Press <Enter> and you get a panel telling you setup has been successful. Press <Enter> again and you are back to the ISPF command panel.

The allocated data sets:

<hlq>.TR.EXEC	for REXX EXECs
<hlq>.TR.CNTL	for all your JCL
<hlq>.TR.COBOL	for all COBOL source code
<hlq>.TR.SOURCE	for all other source code
<hlq>.TR.LOAD	for load modules
<hlq>.TR.DBRMLIB	if you might be running the DB2 labs

Computer Exercise: Running Batch Programs, p. 2.

A number of programs and EXECs have been provided by the setup, copied into your various libraries. For this first lab, you will work with a program in the language of your choice; choose one of:

ALCFTF	- for Assembler programmers
COBFTF	- if you are working in COBOL
PLIFTF	- for PL/I programmers
CFTF	- for C developers

Note that COBFTF is found in your TR.COBOL library, while the other programs are found in your TR.SOURCE library.

Now, Assemble or compile and bind your source program into your LOAD library. In all cases, the program logic is:

> If the length of any parm data passed is between 1 and 25, a message is issued containing an image of the parm data; other wise issue a default error message
>
> An input file is read and copied (input file uses a ddname of INDD and output file uses a ddname of OUTDD)
>
> On completion, the length of the parm field is used as the Return Code value.

In the TR.CNTL library are members to Assemble or compile, bind, and run the various programs, as follows:

A780A1	for ALCFTF
A780COB1	for COBFTF
A780P1	for PLIFTF
A780C1	for CFTF

Run your program(s) several times with various parms, including no PARM, a parm value larger than 25 characters, and a parm value between 1 and 25 characters, to test the logic works as expected.

Computer Exercise: Running Batch Programs, p. 3.

The expected outputs are:

- Return code = length of parm
- Message displaying the parm value or an error message
- A listing of the input records, something like this:

```
On the ramparts flaming stood Aragon
The mighty warrior of wide renown
Burning with anxious charcoal eyes
Drooping down from purple skies
Arms extended, holding arms:
Broken lances, stolen glances,
Country dances, quick he prances away.

Never far from victory, but never final
Reciting poetry always banal
With wretched scansion, hackneyed rhymes
Stolen from poems written in ¾ time,
By better poets with truer souls and
Fairer hearts; his little band
Of literary agents seeking ten percent
Or more of earnings from the meager rent
He makes through the slaughter of several
Modern languages.

Still onward presses Aragon, to new
Lands and battles, with ever few
Fair memories to save for sweet future
Times of rest and respite from butcher
-ing of honest words and feelings
Known by nobler souls and underlings
But not by him, for he feels not.

Never dreaming of defeat in battle
Or in dance contest or in spelling bee
He staggers to the future, from past
He can't remember and will not last.
Not noticing the lines that do not scan
Neither those that do not rhyme
Nor those that make no sense, content
In blissful ignorance his time is spent
Avoiding life and objectivity
In preference to his own reality
Which is real for him alone, but
For him that is enough.

                               -Anon.
```