# TSO Tutorial

**TSO** is an acronym for **T**ime **S**haring **O**ption, which is an accurate summation of the primary service that the TSO feature provides. Time sharing allows the resources of the computer – CPU, memory, and datasets – to be shared by all users by dividing up computer time into segments or *slices*. The program or command being executed by each TSO user is allocated a single slice of time and then that user's program is swapped out onto paged storage and the next user is given their slice of computer resources. These time slices are extremely small, but the cycle of swapping occurs so quickly relative to human time perception that it appears to each user as though they are the only user of the computer's resources.

TSO is a basic product component of MVS (and also MVT). The more recent versions of MVS (OS/390 and zOS, as well as the "for charge" versions of MVS) also provide an extension product, TSO/E (TSO/Extensions), however the base component TSO is still present in these systems. TSO/E adds additional facilities that are required for modern applications/utilities, such as ISPF and REXX, but much of the functionality is provided by basic TSO. Therefore we can make use of many of the same functions on our freely available Operating Systems (MVT 21.8f and MVS 3.8j) running under Hercules as are available on zOS. A variety of tasks may be accomplished using basic TSO features:

- create or modify data contained in datasets,
- develop, test, and execute programs,
- manage datasets and catalogs,
- monitor and manage batch job facilities,
- communicate with other TSO uses and system console operators,
- use help facilities to learn about TSO commands,
- create, edit, and execute TSO Command Lists (CLISTs).

My goal in writing this tutorial is to provide at least the minimum instruction required to begin using TSO to perform the tasks shown above. As I wrote this tutorial I drew upon my own work experience using TSO for a number of years in development environments. The basic outline for what I decided to include was constrained by an an OS/VS2 TSO Command Language Reference Summary (IBM publication number GX28-0647-3) from my library. All material presented in this tutorial has been verified under MVS 3.8j running under Hercules (version 3.01) installed on Windows 2000. The 3270 terminal emulator I use is Tom Brennan's Vista (version 1.24) and illustrations of command input and output are trimmed screen captures from the terminal emulator window. The case of commands and operands typed during a TSO session is generally irrelevant; however in this document I have capitalized commands and operands, or portions thereof, to connote the required (or minimal) syntax required for recognition by TSO. The order in which I have chosen to present TSO commands in this tutorial is that which seems most logical to me and is what I would deem useful if I were using TSO for the first time. There is an Index at the bottom of this page which contains links by which a number of topics may be rapidly accessed.

Although the TSO monitor program may be run as a batch process, it is designed to function as an interface between the user, through a video display terminal, and the operating system. The actual management of the display system input/output is handled for TSO by either VTAM (under MVS) or TCAM (under MVT). A rudimentary security system is implemented by TSO, although in a production environment a more robust security application would probably be active (RACF, ACF, or similar third party product). When a user logs on to TSO through a terminal, an address space is created in which the user may execute programs, some basic datasets are allocated for the user, and a VTAM session is initiated between the address space and the user's terminal.

## TSO Security and Access Control

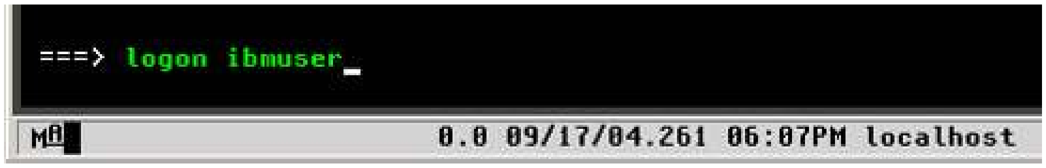
TSO provides a rudimentary security and access control implemented through the use of the TSO User Attribute dataset (SYS1.UADS), which is created during System Generation. TSO User IDs are administered (added, changed, and deleted) by using the Account TSO command. Access to the TSO system, and the functions the user is allowed to complete once access is granted are controlled by settings in the profile of the ID record in this dataset. Before any TSO commands may be entered, the LOGON command must be used with a valid TSO User ID.

## Logging On - LOGON Command

In order to use TSO, a valid TSO User ID and password must be submitted to the TSO monitor via the Logon TSO command. (The requirement of a password may be optional, depending upon settings specified when the User ID was created.) The syntax of this command is:

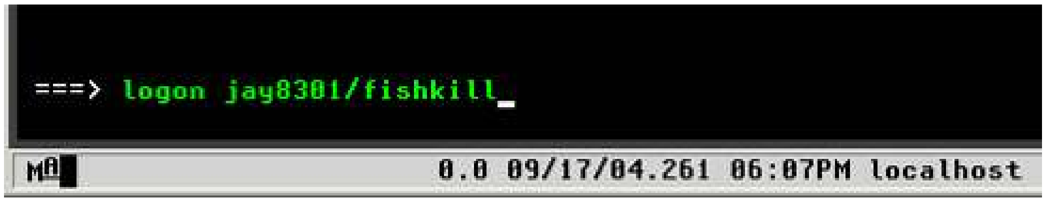
```
LOGON <id>[/<password>] [other optional operands]
```

A standard User ID - IBMUSER - is always installed in the system during System Generation. To log onto TSO using this User ID, you would type:



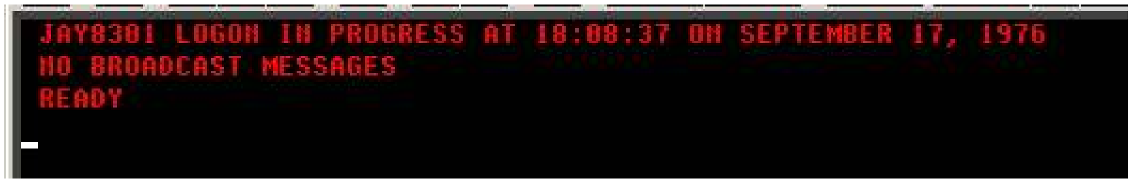
```
===> logon ibmuser_
MA                    0.0 09/17/04.261 06:07PM localhost
```

and press the ENTER key. Note: All TSO commands are entered into a terminal emulator client connected to an address managed by either TCAM or VTAM (depending upon which is installed in your environment). If you are logging on with an ID which requires a password, a slash (/) is entered immediately following the ID and the password is typed immediately following the slash:



```
===> logon jay8301/fishkill_
MA                    0.0 09/17/04.261 06:07PM localhost
```

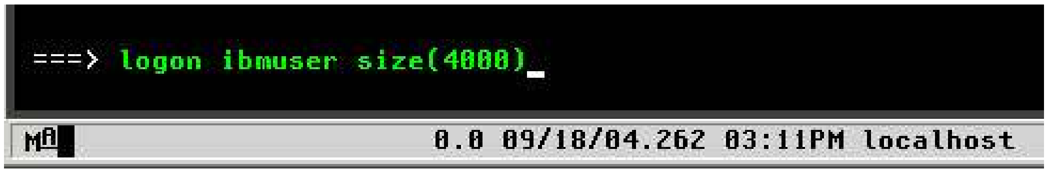
After successfully logging onto TSO, some system messages (which can vary from logon to logon) will be displayed on your terminal followed by the TSO **READY** prompt:



```
JAY8301 LOGON IN PROGRESS AT 18:08:37 ON SEPTEMBER 17, 1976
NO BROADCAST MESSAGES
READY
```

There are several optional operands that may be entered with the LOGON command. The ones which may be of occasional use under Hercules/MVS 3.8j are SIZE, PROC, and ACCT.

The SIZE operand allows you to specify the amount of virtual memory to allocate to your TSO address space. The integer value you provide with the SIZE operand indicates the amount of memory allocated to the virtual region in which your TSO session will execute. The amount specified, less some overhead required for control blocks used to manage your TSO session, becomes the maximum memory available for use by programs and buffers during your session. To logon with the IBMUSER ID and allocate 4,000k region:



```
===> logon ibmuser size(4000)

MA                          0.0 09/18/04.262 03:11PM localhost
```

If you do not specify the SIZE operand, the default value used is the value stored in the User Attribute Dataset for the User ID. Note: In the case of IBMUSER, the default (44k) is too small to provide the necessary buffer space to access VSAM objects (which precludes the use of IDCAMs commands) - the reason for this is that the IBMUSER ID is intended for emergency use only; not for productive work.

The PROC operand allows you to specify the name of a catalogued procedure to be used to allocate system datasets for your TSO session. If you do not specify this operand, the default procedure used is the one that is stored in the User Attribute Dataset for the User ID. If you have multiple logon procedures catalogued, you may use the PROC operand to select the one to be used for a particular session. The default procedure catalogued during System Generation is IKJACCNT.

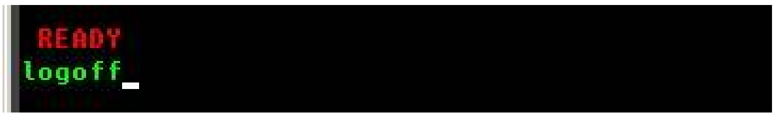
The ACCT operand allows you to specify the accounting information that is written to SMF records created during your TSO session. If used, up to 40 characters may be specified. The default string used is the one stored in the User Attribute Dataset for the User ID.

## Logging Off - LOGOFF Command

When you have logged onto TSO, the terminal remains connected to your TSO session until you terminate the session with the LOGOFF command. The syntax of this command is:

```
LOGOFF
```

Although there are two optional operands, they are not relevant in the Hercules/MVS 3.8j environment. To terminate your TSO session, simply type the command and press ENTER:



```
 READY
logoff_
```

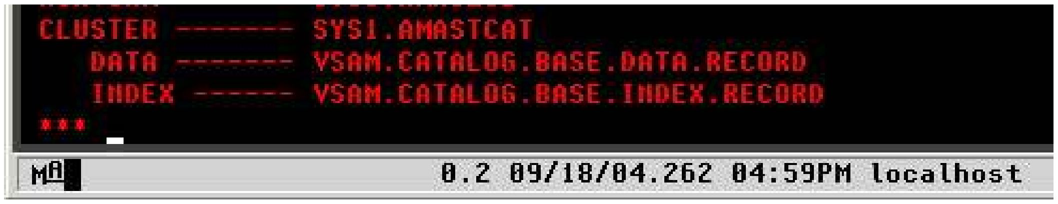
The LOGOFF command releases system resources allocated to your address space, frees virtual memory and terminates the address space. In order to issue further TSO commands, it is necessary to issue the LOGON command again.

Note: If your terminal is allowed to remain idle for a period of time, MVS will automatically log your session off.

## The 3270 Display System

A typical 3270 display screen consists of 24 rows of 80 characters. The screen is divided into fields which have specific characteristics, the most important of these determine whether the field is used for output -- characters written onto the screen by the active program -- or input -- characters typed by the terminal operator (user). In the screen images captured for use in this tutorial, characters written to the screen by the active program appear in red and characters typed by the terminal operator appear in green.

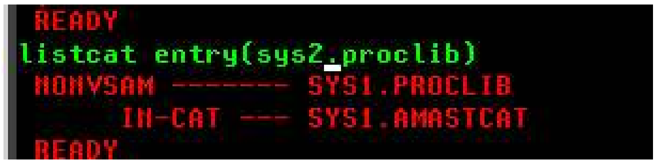
During a TSO session, information appears on the display as a conversation. Characters are typed by the user and sent to the computer when an attention key is pressed. The computer responds to the input by writing characters on the display screen; at the conclusion of the output the cursor remains positioned under the last output and the keyboard is unlocked to await the next input from the operator. The process repeats with the exchange progressing from the top of the screen until the bottom is reached. When the bottom of the screen is reached, the TSO monitor writes three asterisks on the bottom line of the display screen:



```
 CLUSTER ------- SYS1.AMASTCAT
    DATA ------- VSAM.CATALOG.BASE.DATA.RECORD
    INDEX ------ VSAM.CATALOG.BASE.INDEX.RECORD
 ***
 MA              0.2 09/18/04.262 04:59PM localhost
```

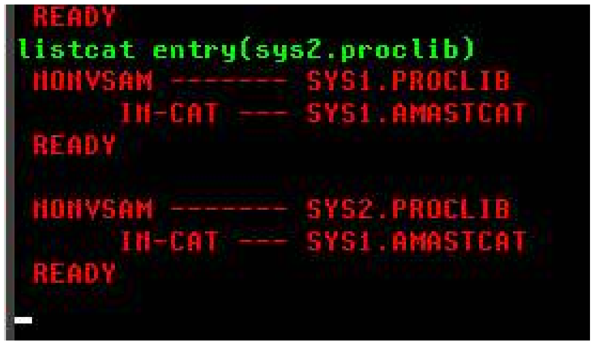
This is a signal to let you know that more information is available in the current output from the computer than will fit on the screen. To retrieve the additional output the ENTER key should be pressed; the screen will be cleared and the output will resume at the top of the screen. If you are not interested in viewing additional output from whatever command/program is executing, the output may be interrupted (see PA1 key description in the next section).

If an input field is still visible on the display screen that closely matches information that is needed again, the cursor position keys may be used to move the cursor from the current line back up to the prior input field. By typing any character into the field area, the prior input field is tagged for transmission to the system the next time an attention key is pressed. Any required changes to the input characters may be made before the attention key is pressed:



```
 READY
listcat entry(sys2.proclib)
 NONVSAM ------- SYS1.PROCLIB
      IN-CAT --- SYS1.AMASTCAT
 READY
```

In this example, the prior input command -- **listcat entry(sys1.proclib)** -- is being changed by altering the ***sys1*** high level index to become ***sys2*** (the cursor remains under the period following ***sys2*** after the change). When the ENTER key is pressed, the entire input field -- **listcat entry(sys2.proclib)** -- will be transmitted as though it had been typed in its entirety following the prior output:



```
 READY
listcat entry(sys2.proclib)
 NONVSAM ------- SYS1.PROCLIB
      IN-CAT --- SYS1.AMASTCAT
 READY

 NONVSAM ------- SYS2.PROCLIB
      IN-CAT --- SYS1.AMASTCAT
 READY

```

## Attention Keys

The 3270 display terminal's interaction with TSO functions quite differently from what many people have come to expect from their experience using a personal computer. When characters are typed on the screen by the user, they are not read and processed by the computer as they are typed. Instead, they are held in a buffer in the display terminal (or in the 3270 client, since it is unlikely that many in the Hercules' community are actually utilizing a physical 3270 terminal). The accumulated characters are sent to the computer for processing only when an attention key is pressed. The most frequently used attention keys are ENTER, CLEAR, and PA1.

The ENTER key, which may be the Carriage Return key (usually regarded as the ENTER key in the personal computer world), provides the same function in TSO as in the personal computer world. On some 3270 clients, the ENTER key is mapped to an alternate key on the keyboard. In the Vista emulator, it is by default mapped to the right CTRL key. When the ENTER key is pressed, all input fields tagged for transmission to the computer for processing are sent as a single stream.

The CLEAR key performs exactly the function it describes -- the terminal screen and character buffer are cleared. The CLEAR key is an attention key because input/output occurs to inform TSO that you have cleared the screen and the output position maintained by the TSO monitor should be reset to the top of the display screen. Since there is no CLEAR key on most personal computer keyboards, the function of the CLEAR key is mapped to another key (or key combination). In the Vista emulator, it is by default mapped to the PAUSE key.

The PA1 key (Program Access 1) interrupts the current command or program. When you press this key, the current command or process is interrupted. If you are executing a subcommand, you will receive a prompt for the parent command. If you are executing a command or program, you will receive a READY prompt. If you want to resume the interrupted command (or subcommand), you may press the ENTER key. Since there is no PA1 key on most personal computer keyboards, the function of the PA1 key is mapped to another key (or key combination). In the Vista emulator, it is by default mapped to CTRL-INSERT key combination.

Other attention keys that are less frequently used are PA2 (redisplay last screen sent from the computer) and the Program Function Keys (1 through 24).

No data is transmitted to the computer from input fields, even if it is tagged, when the CLEAR, PA1, PA2, or PA3 keys are pressed.

Although it is not an attention key, another key which is frequently useful is the RESET key. There are circumstances in which your terminal keyboard becomes *locked*. For example, when you attempt to type characters at a position where input is not permitted. You can use the RESET key to unlock your terminal keyboard without generating an attention interrupt. Since there is no RESET key on most personal keyboards, the function of the RESET key is mapped to another key (or key combination). In the Vista emulator, it is by default mapped to the left CTRL key.

## Command Syntax

TSO Commands are free form and consist of a command name which may be followed by one or more parameters (which I have tried to consistently refer to hereafter as operands in this document). Examples of this can be seen in the description of the LOGON and LOGOFF commands described above. Operands may be either positional or keyword. In the LOGON command, the User ID is a positional operand. The operand is recognized as the User ID by the fact that it is entered immediately following the LOGON command. The SIZE operand is a keyword operand -- it is recognized by the inclusion of the keyword SIZE and the value given for the operand is typed enclosed in parenthesis following the keyword.

TSO employs a liberal abbreviation policy for its commands and keyword operands. Any command (or keyword operand) may be abbreviated to the shortest form that eliminates ambiguity. As long as TSO can differentiate the command (or operand ) you are typing from similar commands (or operands in the same context), it will accept the command you intend. Some examples:

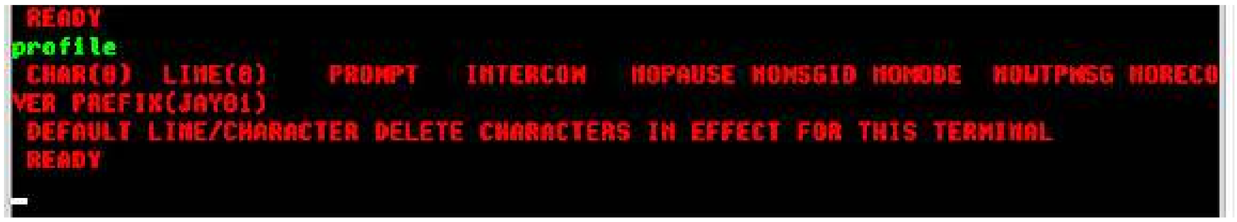
| Command | Acceptable Abbreviations |
|---|---|
| DELETE | DEL |
| LISTCAT | LISTC |
| EDIT | E |
| SUBMIT | SUB |

## User Profile - PROFILE Command

Some behavior of a user's TSO session is determined by settings that are maintained in the user profile. These settings persist from logon to logon. They are initially set to defaults when the User ID is created. If they are subsequently changed by use of the PROFILE command, the new settings specified will remain in effect until they are changed again. The settings are displayed and changed with the PROFILE command (which may be abbreviated PROF). The syntax of this command is:

PROFile [zero or more optional operands]

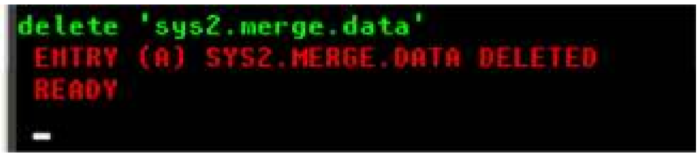
To display the current settings, type the command with no operands and press ENTER:



```
 READY
profile
 CHAR(0)  LINE(0)    PROMPT   INTERCOM   NOPAUSE NOMSGID NOMODE  NOWTPMSG NORECO
VER PREFIX(JAY01)
 DEFAULT LINE/CHARACTER DELETE CHARACTERS IN EFFECT FOR THIS TERMINAL
 READY
```

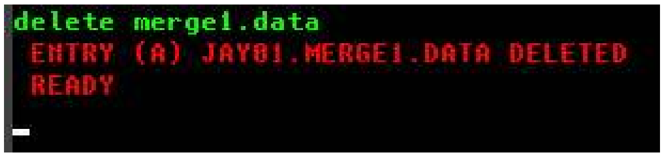
Some of the settings maintained in the profile originate from when line mode (non-display) terminals were in wide use and will not be utilized in a 3270 environment. Probably the most significant setting in the user profile is PREFIX.

When specifying datasets in a TSO session, a dataset name may be supplied as fully qualified or not. If you supply a fully qualified dataset name, you supply the entire name and enclose it in single apostrophes:



```
delete 'sys2.merge.data'
 ENTRY (A) SYS2.MERGE.DATA DELETED
 READY
```

If you omit the apostrophes, TSO will supply additional qualifiers for the dataset name. The most significant of these is the high level qualifier (located at the left most position of the name), which will be taken from the prefix setting of your user profile:



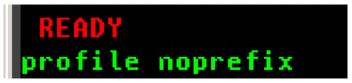
```
delete merge1.data
 ENTRY (A) JAY01.MERGE1.DATA DELETED
 READY
```

As you see in the example above, the user ID - JAY01 - has been prepended to the portion of the dataset name entered by the user - **merge1.data** - to form the complete dataset name used by the command - **jay01.merge1.data**.

The prefix is usually set to your User ID. Typically your User ID is set up as an alias to a User Catalog in order to exert some control over where datasets are cataloged (and thus prevent extraneous user datasets from being cataloged in the Master Catalog). With the prefix set to your User ID, it ensures that datasets you create during your TSO session will be cataloged in the appropriate User Catalog. There are two circumstances in which you might want to temporarily change the prefix --

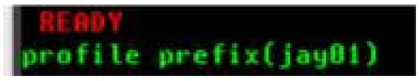
- when you intend to specify a number of system datasets and don't wish to remember to enclose them in apostrophes, and
- when you intend to specify a number of datasets that consistently use an identical high level qualifier (other than your TSO User ID).

In order to remove the prefix, you use the NOPREFIX operand of the PROFILE command:



```
 READY
profile noprefix
```

After setting the profile to ***noprefix***, subsequent dataset names not enclosed in apostrophes will have no high level qualifier prepended to them. To again set the prefix (either to your User ID or some other relevant value), use the PREFIX operand of the PROFILE command:



```
 READY
profile prefix(jay01)
```

Other settings in the user profile that might be useful to a typical user include PAUSE, MSGID, and WTPMSG, but I will not be discussing them here. To obtain more information use the HELP PROFILE command.

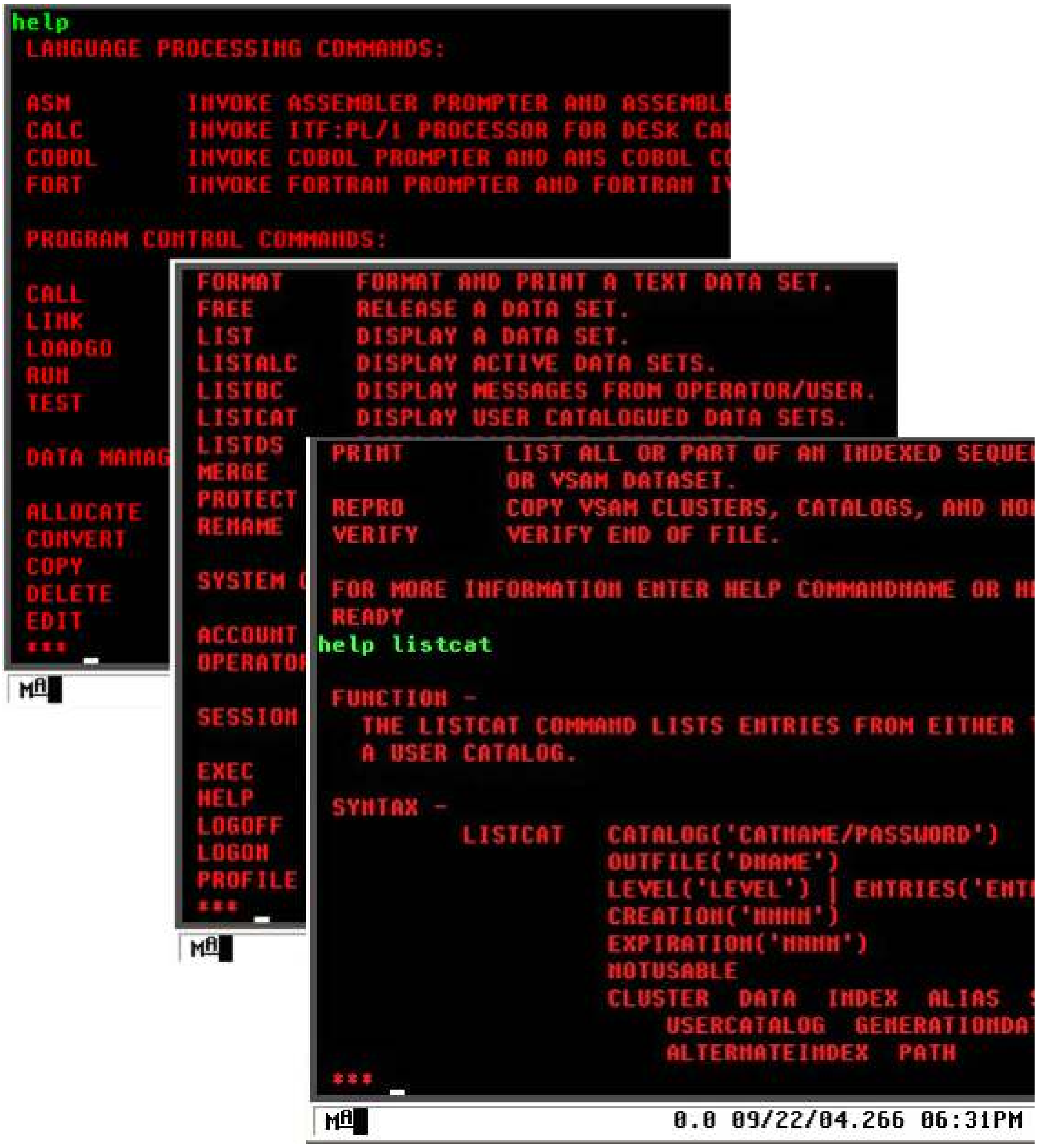
## TSO Help System

TSO conveniently provides a HELP command that may be used to obtain information about the function, syntax, and operands of commands and subcommands. The source of this information is contained in the members of the partitioned dataset SYS1.HELP, which is created during System Generation. (It is a convention followed by many Systems Programmers to place help text for user written commands in a secondary dataset -- SYS2.HELP -- which is also accessed by the HELP command.)

The syntax of this command is:

```
HELP [zero or more optional operands]
```

The HELP command with no operands displays a list of valid commands (note that this list is produced solely from SYS1.HELP so it will not include user commands for which help text resides in SYS2.HELP unless it has been specifically updated to do so). To obtain additional information about a particular command, enter the HELP command and specify the name of the command for which you want additional information displayed as an operand. The following overlapped screen captures shows the first two screens of output produced by the HELP command with no operands and the first portion of the output of the HELP command with LISTCAT specified as an operand:

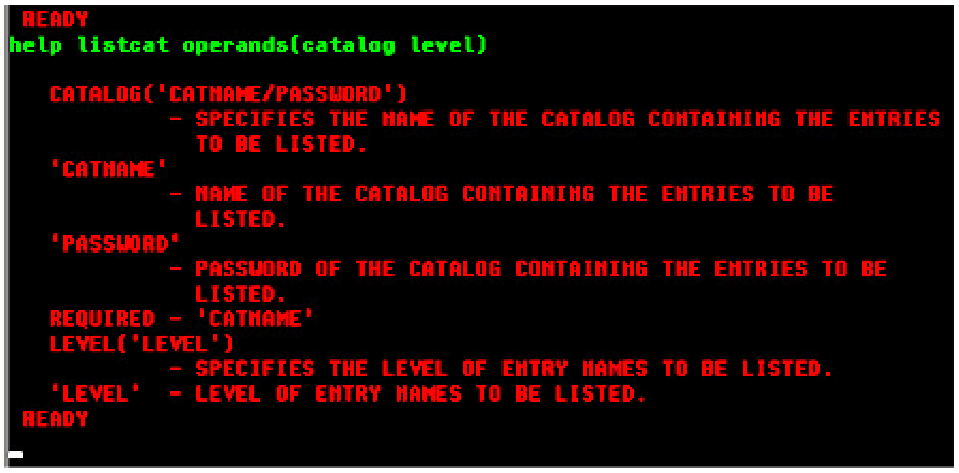


```
help
 LANGUAGE PROCESSING COMMANDS:

 ASM        INVOKE ASSEMBLER PROMPTER AND ASSEMBLE
 CALC       INVOKE ITF:PL/1 PROCESSOR FOR DESK CAL
 COBOL      INVOKE COBOL PROMPTER AND ANS COBOL CO
 FORT       INVOKE FORTRAN PROMPTER AND FORTRAN I

 PROGRAM CONTROL COMMANDS:

 CALL
 LINK
 LOADGO
 RUN
 TEST

 DATA MANAG

 ALLOCATE
 CONVERT
 COPY
 DELETE
 EDIT
 ***
MA
```

```
FORMAT     FORMAT AND PRINT A TEXT DATA SET.
FREE       RELEASE A DATA SET.
LIST       DISPLAY A DATA SET.
LISTALC    DISPLAY ACTIVE DATA SETS.
LISTBC     DISPLAY MESSAGES FROM OPERATOR/USER.
LISTCAT    DISPLAY USER CATALOGUED DATA SETS.
LISTDS
MERGE
PROTECT
RENAME

SYSTEM C

ACCOUNT
OPERATOR

SESSION

EXEC
HELP
LOGOFF
LOGON
PROFILE
***
MA
```

```
PRINT        LIST ALL OR PART OF AN INDEXED SEQUE
             OR VSAM DATASET.
REPRO        COPY VSAM CLUSTERS, CATALOGS, AND NON
VERIFY       VERIFY END OF FILE.

FOR MORE INFORMATION ENTER HELP COMMANDNAME OR HE
READY
help listcat

FUNCTION -
  THE LISTCAT COMMAND LISTS ENTRIES FROM EITHER T
  A USER CATALOG.

SYNTAX -
          LISTCAT   CATALOG('CATNAME/PASSWORD')
                    OUTFILE('DNAME')
                    LEVEL('LEVEL') | ENTRIES('ENT
                    CREATION('NNNN')
                    EXPIRATION('NNNN')
                    NOTUSABLE
                    CLUSTER  DATA  INDEX  ALIAS  S
                        USERCATALOG  GENERATIONDA
                        ALTERNATEINDEX  PATH

***
MA                          0.0 09/22/04.266 06:31PM
```

For the specified command, you will be shown a brief description of the function(s) of the command, the syntax of the command, and a description of each operand. Keyword operands for the HELP command -- FUNCTION, SYNTAX, OPERANDS -- may be specified to limit the output displayed by the HELP command. FUNCTION limits the output displayed to the description of the function; SYNTAX limits the output displayed to the syntax required for the command, and OPERANDS limits the output displayed to the operands available for the command. OPERANDS will also accept one or more keywords specified in parentheses to request information on specific operands:



```
 READY
help listcat operands(catalog level)

   CATALOG('CATNAME/PASSWORD')
           - SPECIFIES THE NAME OF THE CATALOG CONTAINING THE ENTRIES
             TO BE LISTED.
   'CATNAME'
           - NAME OF THE CATALOG CONTAINING THE ENTRIES TO BE
             LISTED.
   'PASSWORD'
           - PASSWORD OF THE CATALOG CONTAINING THE ENTRIES TO BE
             LISTED.
   REQUIRED - 'CATNAME'
   LEVEL('LEVEL')
           - SPECIFIES THE LEVEL OF ENTRY NAMES TO BE LISTED.
   'LEVEL'  - LEVEL OF ENTRY NAMES TO BE LISTED.
 READY
```

## Managing Datasets and Catalogs

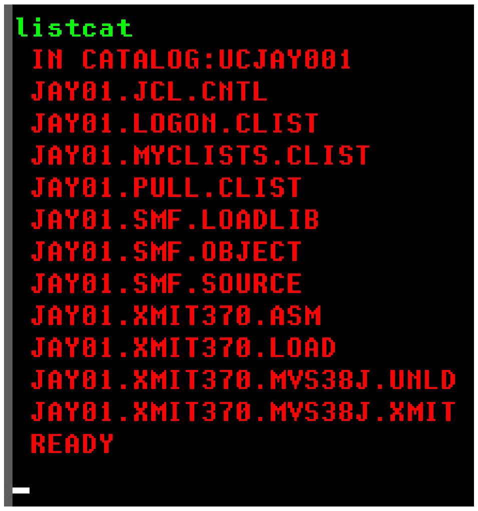
TSO most closely resembles the Command Line environment of Linux or Windows, in that it provides a rich set of commands that may be used to investigate and modify the environment accessible by the user from their TSO session. In fact, almost the entire set of IDCAMS commands (the VSAM Access Methods Services utility) may be executed directly from the TSO session prompt to manipulate Catalogs, Datasets, Aliases, and other VSAM objects. The following section describes some of the basic data management tasks that may be accomplished from the TSO session.

### Displaying Catalog Information - LISTCAT Command

The LISTCAT command (which may be abbreviated LISTC) is one of the most frequently used TSO commands. It is used to list information contained in the Master and User Catalogs of the system for Non-VSAM datasets and VSAM objects. The syntax of this command is:

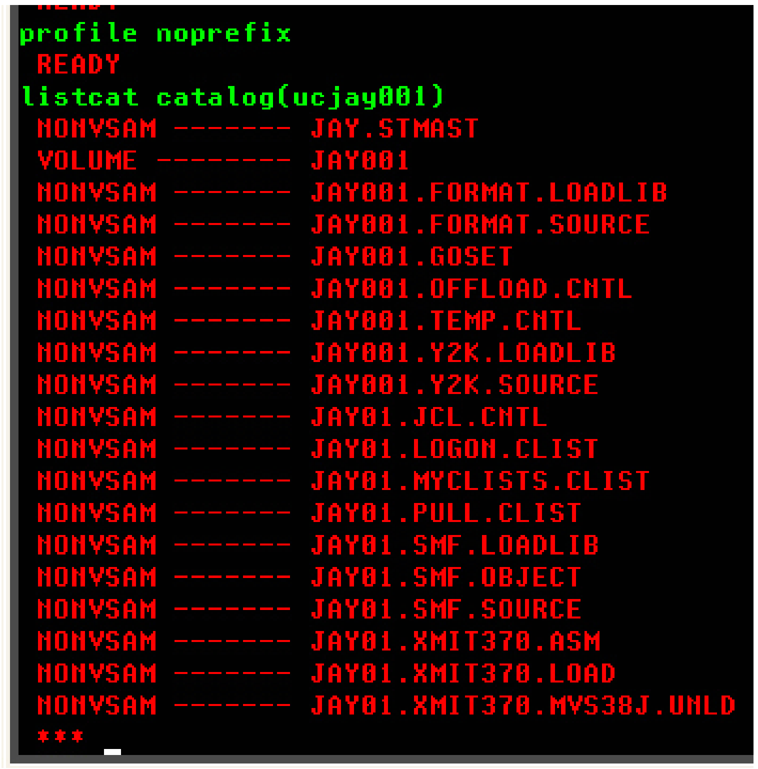
```
LISTCAT [zero or more optional operands]
```

Issued with no operands, the command will use the current value of the *prefix* from the User's profile to list catalogued datasets for that user:



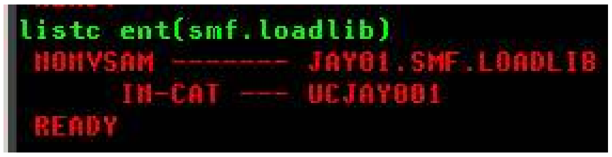
```
listcat
 IN CATALOG:UCJAY001
 JAY01.JCL.CNTL
 JAY01.LOGON.CLIST
 JAY01.MYCLISTS.CLIST
 JAY01.PULL.CLIST
 JAY01.SMF.LOADLIB
 JAY01.SMF.OBJECT
 JAY01.SMF.SOURCE
 JAY01.XMIT370.ASM
 JAY01.XMIT370.LOAD
 JAY01.XMIT370.MVS38J.UNLD
 JAY01.XMIT370.MVS38J.XMIT
 READY
```

The first line of the output shown in the session above indicates that the User ID prefix has been used to select the User Catalog - UCJAY001 - based upon the prefix value - JAY01 (this association is specific to this particular User ID on my system and will differ on other systems). The second and following lines are the datasets from this catalog in which the high level qualifier match the prefix. There may be additional datasets (as well as other VSAM objects) catalogued in the User Catalog that are excluded from this list, which is illustrated by setting the prefix to null and listing the contents of the User Catalog by specifically selecting the catalog to be listed:



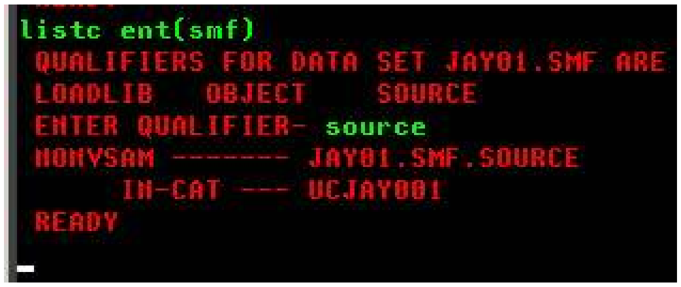
```
profile noprefix
 READY
listcat catalog(ucjay001)
 NONVSAM ------- JAY.STMAST
 VOLUME -------- JAY001
 NONVSAM ------- JAY001.FORMAT.LOADLIB
 NONVSAM ------- JAY001.FORMAT.SOURCE
 NONVSAM ------- JAY001.GOSET
 NONVSAM ------- JAY001.OFFLOAD.CNTL
 NONVSAM ------- JAY001.TEMP.CNTL
 NONVSAM ------- JAY001.Y2K.LOADLIB
 NONVSAM ------- JAY001.Y2K.SOURCE
 NONVSAM ------- JAY01.JCL.CNTL
 NONVSAM ------- JAY01.LOGON.CLIST
 NONVSAM ------- JAY01.MYCLISTS.CLIST
 NONVSAM ------- JAY01.PULL.CLIST
 NONVSAM ------- JAY01.SMF.LOADLIB
 NONVSAM ------- JAY01.SMF.OBJECT
 NONVSAM ------- JAY01.SMF.SOURCE
 NONVSAM ------- JAY01.XMIT370.ASM
 NONVSAM ------- JAY01.XMIT370.LOAD
 NONVSAM ------- JAY01.XMIT370.MVS38J.UNLD
 ***
```

Frequently the goal will be to have a smaller, more relevant, list returned by the LISTCAT command instead of having all objects included from the catalog that match the User ID prefix. Two operands allow you to limit the information selected - ENTRIES and LEVEL (which may be abbreviated to ENT and LVL, respectively). ENTRIES is used to limit the information returned to that for specific objects named in parentheses following the operand. Usually this operand is used to return information for a single object:



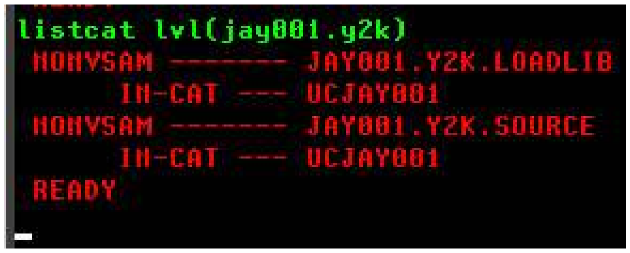
```
listc ent(smf.loadlib)
 NONVSAM ------- JAY01.SMF.LOADLIB
     IN-CAT --- UCJAY001
 READY
```

If the value you specify for the ENTRIES operand is ambiguous (more than one object in the catalog matches the value you enter, but with different lower level qualifiers), you will be prompted with a list of those lower level qualifiers so that you may specify which qualifier is to be used to complete the execution of the command:



```
listc ent(smf)
 QUALIFIERS FOR DATA SET JAY01.SMF ARE
 LOADLIB   OBJECT    SOURCE
 ENTER QUALIFIER- source
 NONVSAM ------- JAY01.SMF.SOURCE
     IN-CAT --- UCJAY001
 READY
```

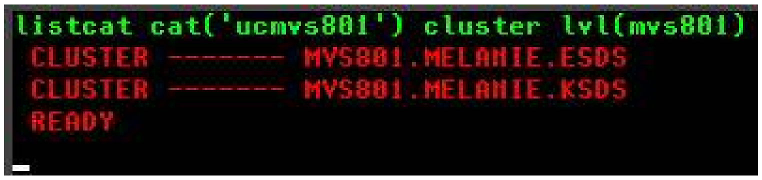
In order to list a group of objects from the catalog matching a specified qualifier level, you use the LEVEL operand:



```
listcat lvl(jay001.y2k)
 NONVSAM ------- JAY001.Y2K.LOADLIB
     IN-CAT --- UCJAY001
 NONVSAM ------- JAY001.Y2K.SOURCE
     IN-CAT --- UCJAY001
 READY
```

Because you are specifying the high level qualifiers to be used to filter the output returned, the prefix value is not relevant when the LEVEL operand is used. **However, the prefix value is still used to select the User Catalog from which the entries are listed.**

Other operands may be specified to designate the exact user catalog from which listings are made and the type of objects to be listed:



```
listcat cat('ucmvs801') cluster lvl(mvs801)
 CLUSTER ------- MVS801.MELANIE.ESDS
 CLUSTER ------- MVS801.MELANIE.KSDS
 READY
```

To learn more about the LISTCAT command, enter HELP LISTCAT at the TSO READY prompt.

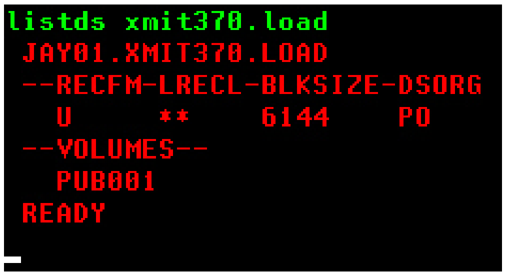
## Displaying Dataset Attributes - LISTDS Command

The LISTDS command (which may be abbreviated LISTD) is used to list attributes of Non-VSAM datasets. The syntax of this command is:

LISTDS [zero or more optional operands]

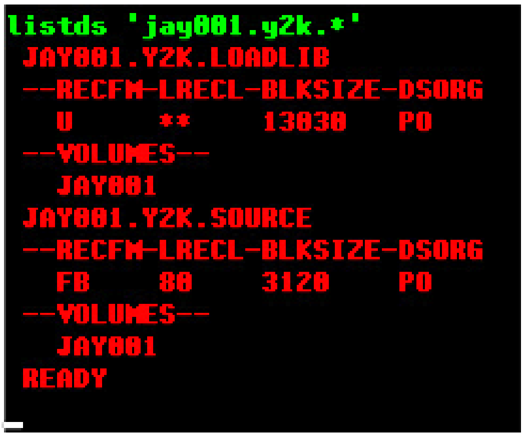
LISTDS [zero or more optional operands]

If the command is entered with no operands, you will be prompted for a dataset name; however, it is typical practice to enter the name of the dataset for which attributes are to be listed as an operand of the command:



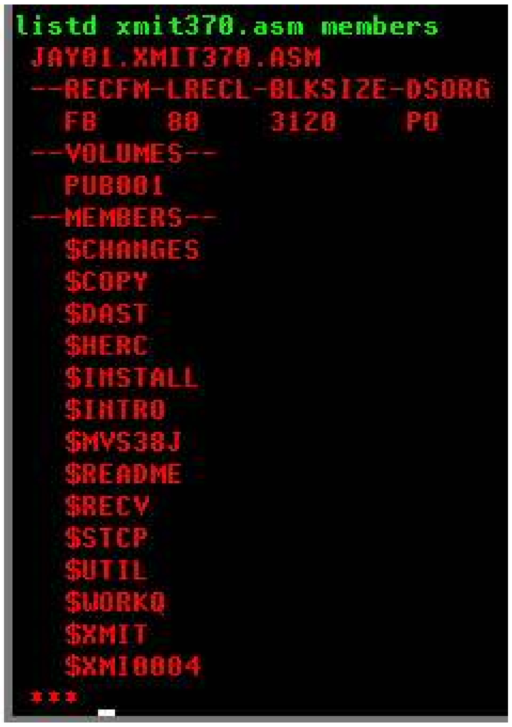
```
listds xmit370.load
 JAY01.XMIT370.LOAD
 --RECFM-LRECL-BLKSIZE-DSORG
   U     **     6144     PO
 --VOLUMES--
   PUB001
 READY
```

You may list more than one dataset by replacing one or more levels of qualification in the dataset name with an asterisk (*):



```
listds 'jay001.y2k.*'
 JAY001.Y2K.LOADLIB
 --RECFM-LRECL-BLKSIZE-DSORG
   U     **     13030    PO
 --VOLUMES--
   JAY001
 JAY001.Y2K.SOURCE
 --RECFM-LRECL-BLKSIZE-DSORG
   FB    80     3120     PO
 --VOLUMES--
   JAY001
 READY
```

The attributes listed by default for the specified dataset include Record Format (RECFM), Logical Record Length (LRECL), Block Size (BLKSIZE), Dataset Organization (DSORG), and the Volume Serial identifier(s) on which the dataset resides. Additional attributes may be displayed by including additional operands; a frequently used operand - MEMBERS - will return a list of the members for partitioned datasets:



```
listd xmit370.asm members
 JAY01.XMIT370.ASM
 --RECFM-LRECL-BLKSIZE-DSORG
   FB    80     3120     PO
 --VOLUMES--
   PUB001
 --MEMBERS--
   $CHANGES
   $COPY
   $DAST
   $HERC
   $INSTALL
   $INTRO
   $MVS38J
   $README
   $RECV
   $STCP
   $UTIL
   $WORKQ
   $XMIT
   $XMI0004
 ***
```

To learn more about the LISTDS command, enter HELP LISTDS at the TSO READY prompt. You might also want to investigate the DD command from the CBT Tape.

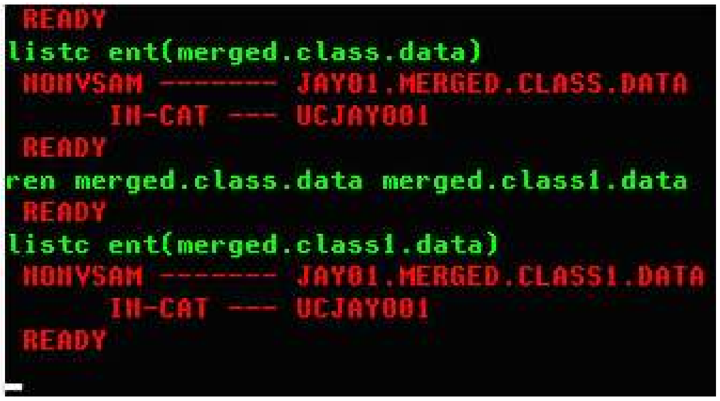
## Renaming a Dataset - RENAME Command

The RENAME command (which may be abbreviated REN) is used to rename either a physical sequential dataset or a member of a partitioned dataset. The RENAME command may not be used to rename a VSAM object (you must use the IDCAMs ALTER command to accomplish that). The syntax of the RENAME command is:

RENAME [zero or more optional operands]

The name of the dataset is changed in both the VTOC and the catalog in which the dataset is cataloged.

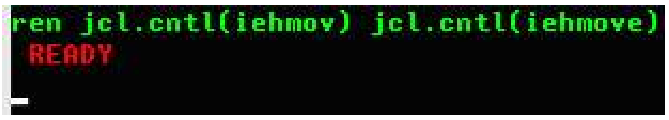
If the command is entered with no operands, you will be prompted first for the name of an existing dataset which is to be given a new name, followed by the new name the dataset is to be given. However, the typical practice is to enter both the old and new names for the dataset as operands of the command:



```
 READY
listc ent(merged.class.data)
 NONVSAM -------- JAY01.MERGED.CLASS.DATA
      IN-CAT --- UCJAY001
 READY
ren merged.class.data merged.class1.data
 READY
listc ent(merged.class1.data)
 NONVSAM -------- JAY01.MERGED.CLASS1.DATA
      IN-CAT --- UCJAY001
 READY
```

In the session above the RENAME command is preceded and followed by a LISTCAT command to show the catalog entries for the dataset before the RENAME command is entered and after. Note that for a successful RENAME operation, there is no output produced for the command.

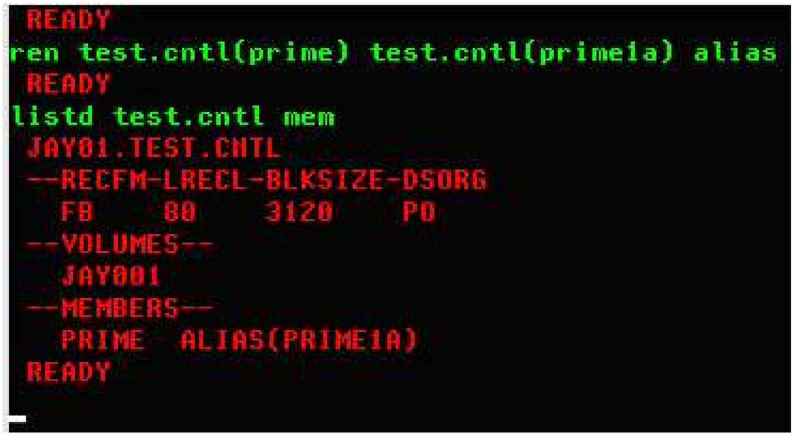
The RENAME command may also be used to rename members of a partitioned dataset. In that context, the operands given for both the old and new dataset name must include the name of a single member of a partitioned dataset:



```
ren jcl.cntl(iehmov) jcl.cntl(iehmove)
 READY
```

Instead of modifying the name of the dataset in the VTOC and catalog, the name of the member is changed in the directory of the partitioned dataset.

Another optional operand - ALIAS - may be specified for partitioned datasets and is used to create a new directory entry for an existing member of the dataset while leaving the original directory entry intact. This new entry, or alias, may then be used to access the contents of the member interchangeably with the original member name:



```
 READY
ren test.cntl(prime) test.cntl(prime1a) alias
 READY
listd test.cntl mem
 JAY01.TEST.CNTL
 --RECFM-LRECL-BLKSIZE-DSORG
   FB    80    3120    PO
 --VOLUMES--
   JAY001
 --MEMBERS--
   PRIME  ALIAS(PRIME1A)
 READY
```

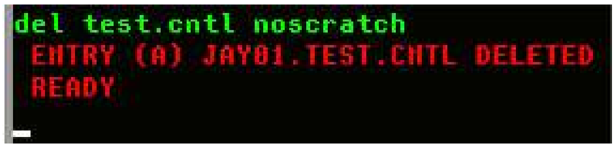
In the session above the LISTDS command is used following the RENAME command to show that an alias has been created - prime1a - for the existing member prime. Either member name, prime or prime1a, may now be used to access the contents of the member.

### Deleting a Dataset - DELETE Command

The DELETE command (which may be abbreviated DEL) is used to delete either VSAM objects or non-VSAM datasets from the and VTOC and free the space occupied by their contents for reuse. The syntax of this command is:

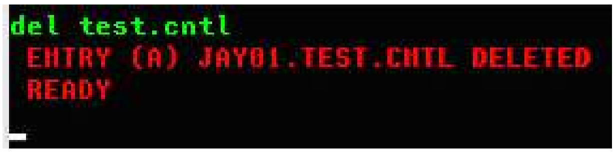
DELETE <object or dataset name> [optional operands]

The most common use of the DELETE command is to remove a non-VSAM dataset that is no longer required. By default, the command removes both the catalog entry for the dataset and the VTOC entry, freeing the space that was used by the dataset for reuse. An optional operand - NOSCRATCH - may be specified to remove the catalog entry while retaining the dataset contents and its entry in the VTOC.



```
del test.cntl noscratch
 ENTRY (A) JAY01.TEST.CNTL DELETED
 READY
```

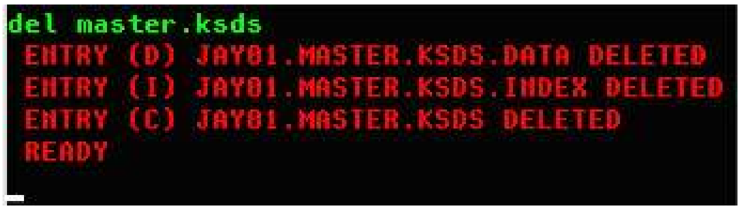
The outcome of the DELETE command in the session above is the removal of the catalog entry for the specified dataset. The entry for the dataset remains in the VTOC and the contents of the dataset remain intact on the DASD volume. If the **noscratch** operand is omitted, the default action, **scratch**, is in effect:



```
del test.cntl
 ENTRY (A) JAY01.TEST.CNTL DELETED
 READY
```

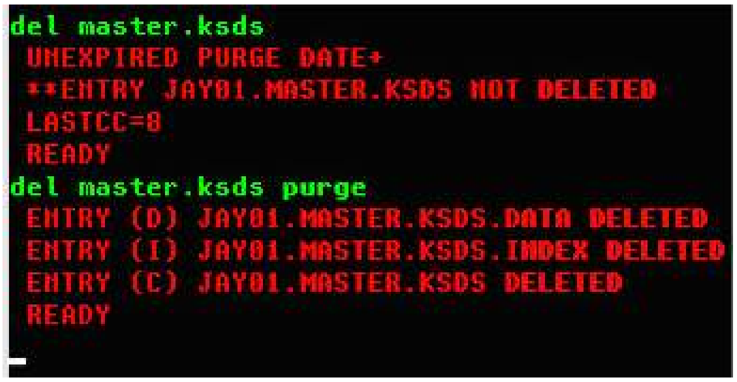
The entry for the dataset is removed from the VTOC and the space formerly occupied by the contents of the dataset are available for reuse. Note that the output from the command is identical regardless of the scratch/noscratch specification.

The DELETE command may also be used to remove VSAM objects:


```
del master.ksds
 ENTRY (D) JAY01.MASTER.KSDS.DATA DELETED
 ENTRY (I) JAY01.MASTER.KSDS.INDEX DELETED
 ENTRY (C) JAY01.MASTER.KSDS DELETED
 READY
```

In the session shown above, the VSAM object specified for deletion was a VSAM cluster. The subcomponents of the cluster, the data and index objects, were deleted before the cluster object.

Another operand that may be used frequently is PURGE. The opposite of this operand, which is the default, is NOPURGE and specifies that the object will not be deleted if the expiration date specified when the object was created has not yet been reached. Specifying the PURGE operand overrides the expiration date, allowing the object to be deleted anyway.


```
del master.ksds
 UNEXPIRED PURGE DATE+
 **ENTRY JAY01.MASTER.KSDS NOT DELETED
 LASTCC=8
 READY
del master.ksds purge
 ENTRY (D) JAY01.MASTER.KSDS.DATA DELETED
 ENTRY (I) JAY01.MASTER.KSDS.INDEX DELETED
 ENTRY (C) JAY01.MASTER.KSDS DELETED
 READY
```

As shown in the session above, if DELETE is specified for an object which has an expiration date that has not yet passed, an error message is issued and the object is not deleted. Adding the PURGE operand to the delete command causes the deletion to proceed regardless of the expiration date.

## Creating and Modifying Data - EDIT Command

The EDIT command (which may be abbreviated E) is used to create and modify data contained in user datasets. It has very basic capabilities and in the real world has mostly been replaced by ISPF. In the Hercules/MVS3.8j community, a much more robust editor exists in RPF, which provides an editor that is much like ISPF. However, there are times when EDIT is all that is available and can save the day. I am only going to cover the very basics of EDIT as it is most likely you will be using RPF if you regularly create or edit data under TSO.

The most basic syntax of the EDIT command is:

```
Edit <dataset>(<member>) [<type>] [OLD | NEW]
```

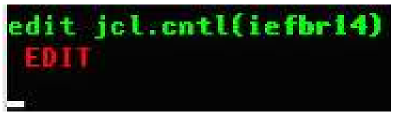
Dataset and member specify the partitioned dataset and member that is to be edited. EDIT can modify physical sequential datasets, but the majority of its use is in maintaining source code or JCL libraries (another name used frequently to refer to partitioned datasets).

Type specifies the type of records contained in the dataset and is used by the EDIT command to interpret where sequence numbers can be expected. Some valid types are:

- CNTL - used for Job Control Language
- COBOL
- PLI
- DATA
- TEXT
- ASM
- CLIST

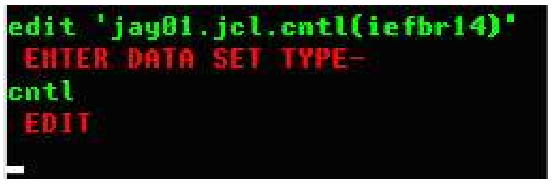
Old or New designate whether the dataset (or member if editing a partitioned dataset) already exists or is being created. Usually this operand can be omitted as EDIT will make the correct assumption based upon the existence of the dataset (or member) when EDIT is invoked.

If the dataset name is specified without enclosing quotes (ie. is not qualified), EDIT will attempt to use the last qualifier of the dataset name to determine the type of records contained in the dataset:



```
edit jcl.cntl(iefbr14)
 EDIT
```

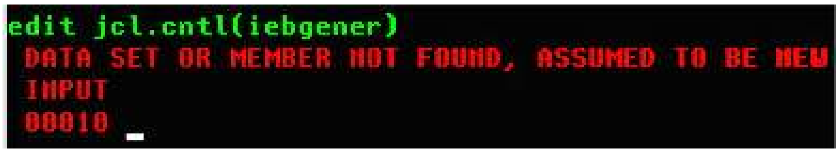
If the dataset name is enclosed in quotes and the dataset type is omitted (or EDIT is unable to determine a valid type from the dataset name), EDIT will prompt for the type:



```
edit 'jay01.jcl.cntl(iefbr14)'
 ENTER DATA SET TYPE-
cntl
 EDIT
```

**Two Modes of Operation - Input or Edit**

EDIT operates in two basic modes - INPUT or EDIT. The initial mode is determined by whether the data being edited is NEW or OLD. As shown above, since the member - iefbr14 - already exists, the initial mode is EDIT, which is indicated by the prompt. If the data being edited is NEW, the initial mode is INPUT:



```
edit jcl.cntl(iebgener)
 DATA SET OR MEMBER NOT FOUND, ASSUMED TO BE NEW
 INPUT
 00010
```

In INPUT mode, the sequence number for the next available data record is displayed and the cursor is positioned at the beginning of the record awaiting input of the data for the next record to be typed.

The mode is switched between EDIT and INPUT by pressing ENTER at the prompt without typing any other characters.

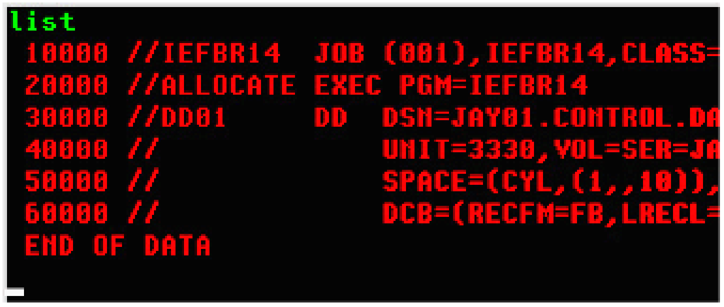
EDIT is line mode oriented, since it dates from the time when line mode terminals were the majority of devices in use. The commands you enter will identify one or more lines to which they will be applied. The current line in the dataset you are editing is referenced by the current line pointer, which is specified with an asterisk (*) when it is referenced in EDIT subcommands.

**Displaying Current Dataset Contents - LIST Subcommand**

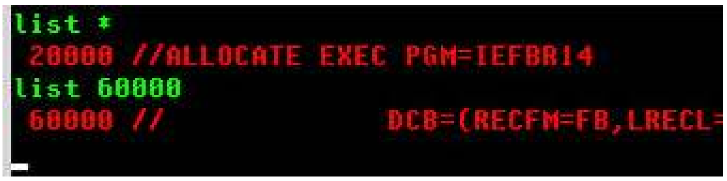
To display the data currently existing in a dataset, use the LIST subcommand:

```
LIST [ {<starting line sequence number> [<ending line sequence number>] |
       * [number of lines] }
```

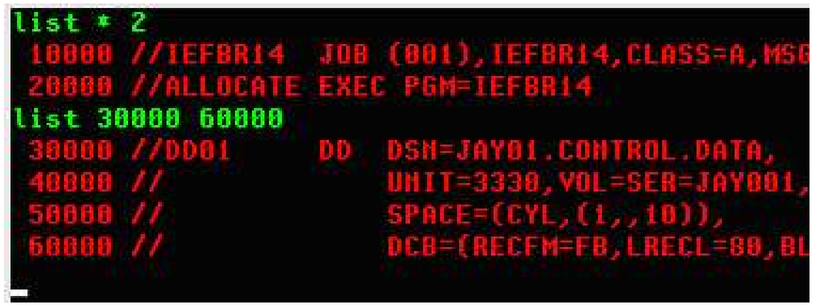
If no operands are included, the entire dataset is listed. If the dataset is large, this can result in numerous screens of output. A single line can be displayed by specifying either the line sequence number or the current line pointer (*). A range of lines may be displayed by specifying either a starting and ending sequence number pair or the current line pointer and a number designating the number of lines to display. Examples of each of these options are shown below:



```
list
 10000 //IEFBR14  JOB (001),IEFBR14,CLASS=
 20000 //ALLOCATE EXEC PGM=IEFBR14
 30000 //DD01     DD  DSN=JAY01.CONTROL.DA
 40000 //             UNIT=3330,VOL=SER=JA
 50000 //             SPACE=(CYL,(1,,10)),
 60000 //             DCB=(RECFM=FB,LRECL=
 END OF DATA
```



```
list *
 20000 //ALLOCATE EXEC PGM=IEFBR14
list 60000
 60000 //             DCB=(RECFM=FB,LRECL=
```



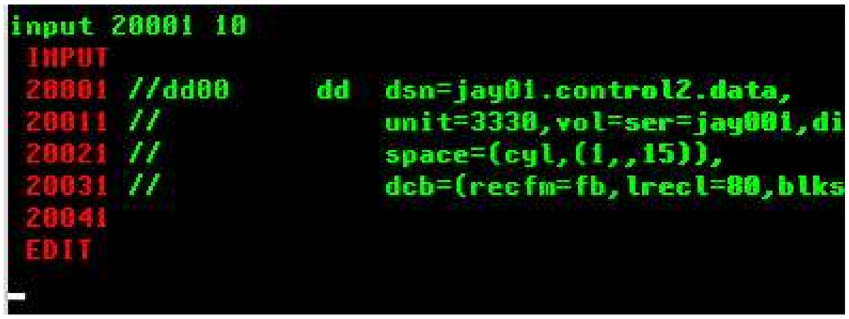
```
list * 2
 10000 //IEFBR14  JOB (001),IEFBR14,CLASS=A,MSG
 20000 //ALLOCATE EXEC PGM=IEFBR14
list 30000 60000
 30000 //DD01     DD  DSN=JAY01.CONTROL.DATA,
 40000 //             UNIT=3330,VOL=SER=JAY001,
 50000 //             SPACE=(CYL,(1,,10)),
 60000 //             DCB=(RECFM=FB,LRECL=80,BL
```

**Inserting New Lines - INPUT Subcommand and Input Mode**

To insert lines, the mode must be switched to INPUT. If you press ENTER without typing anything when you are in EDIT mode, you will be placed in INPUT mode and the lines typed will be added at the next available line number (usually the end of the dataset). If you use the INPUT subcommand and specify a line number, the lines are added at the position indicated by the line number you have specified. The syntax of the INPUT subcommand is:

INPUT <line sequence number> [<increment>]

The line sequence number operand specifies the line number at which the first new line is to be inserted. The optional increment operand specifies the value to be used to increment the line sequence number for subsequent lines.



```
input 20001 10
 INPUT
 20001 //dd00      dd  dsn=jay01.control2.data,
 20011 //              unit=3330,vol=ser=jay001,di
 20021 //              space=(cyl,(1,,15)),
 20031 //              dcb=(recfm=fb,lrecl=80,blks
 20041
 EDIT
```

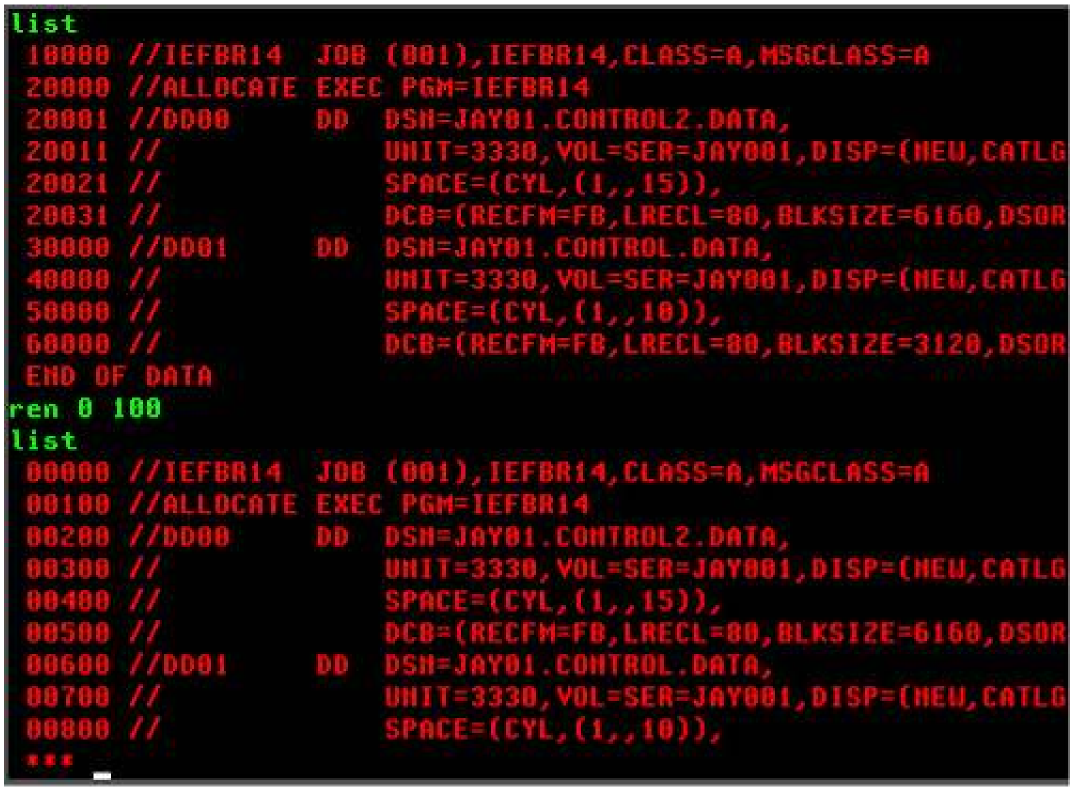
After all new lines have been entered, the mode is switched back to EDIT by pressing ENTER without typing any additional characters at the input prompt. When inserting new lines between existing lines of a dataset, you must have enough unused line numbers to accommodate the new lines you are adding. When all available sequence numbers have been added, an error message will be displayed and the mode will switch back to EDIT. In order to insert additional lines when you have used all available sequence numbers, you must RENUMber the dataset.

## Renumber Sequence Numbers in the Dataset - RENUM Subcommand

The RENUM subcommand (which may be abbreviated REN) is used to renumber the lines of a dataset in order to make room for additional new records. The syntax of RENUM is:

RENum [beginning sequence number] [increment]

The beginning sequence number operand specifies sequence number to be assigned to the first line of the dataset. If omitted it defaults to the value 10. The increment operand specifies the value to be added to each previous sequence number to derive the sequence number of each successive line. If omitted, the increment value will default to the value 10.



```
list
 10000 //IEFBR14   JOB (001),IEFBR14,CLASS=A,MSGCLASS=A
 20000 //ALLOCATE EXEC PGM=IEFBR14
 20001 //DD00      DD   DSN=JAY01.CONTROL2.DATA,
 20011 //               UNIT=3330,VOL=SER=JAY001,DISP=(NEW,CATLG
 20021 //               SPACE=(CYL,(1,,15)),
 20031 //               DCB=(RECFM=FB,LRECL=80,BLKSIZE=6160,DSOR
 30000 //DD01      DD   DSN=JAY01.CONTROL.DATA,
 40000 //               UNIT=3330,VOL=SER=JAY001,DISP=(NEW,CATLG
 50000 //               SPACE=(CYL,(1,,10)),
 60000 //               DCB=(RECFM=FB,LRECL=80,BLKSIZE=3120,DSOR
 END OF DATA
ren 0 100
list
 00000 //IEFBR14   JOB (001),IEFBR14,CLASS=A,MSGCLASS=A
 00100 //ALLOCATE EXEC PGM=IEFBR14
 00200 //DD00      DD   DSN=JAY01.CONTROL2.DATA,
 00300 //               UNIT=3330,VOL=SER=JAY001,DISP=(NEW,CATLG
 00400 //               SPACE=(CYL,(1,,15)),
 00500 //               DCB=(RECFM=FB,LRECL=80,BLKSIZE=6160,DSOR
 00600 //DD01      DD   DSN=JAY01.CONTROL.DATA,
 00700 //               UNIT=3330,VOL=SER=JAY001,DISP=(NEW,CATLG
 00800 //               SPACE=(CYL,(1,,10)),
 ***
```

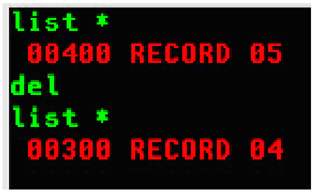
The session above includes a LIST subcommand to show the dataset before executing the RENUM subcommand, the RENUM subcommand specifying a starting sequence number of 0 and an increment value of 100, and another LIST subcommand showing the dataset with the new sequence numbers.

## Deleting Records from the Dataset - DELETE Subcommand

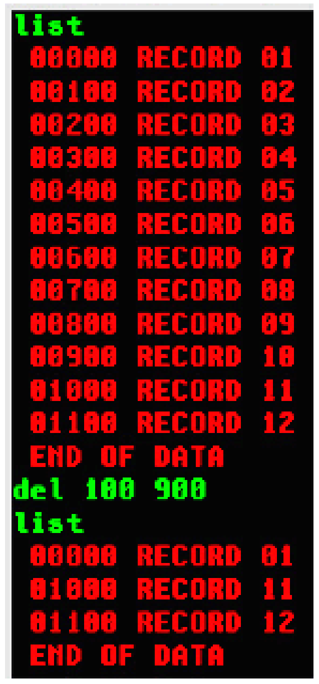
The DELETE subcommand (which may be abbreviated DEL) is used to remove records from the dataset. The syntax of the subcommand is:

```
DELete { [starting sequence number] [ending sequence number] |
         [*] [count of records to delete] }
```

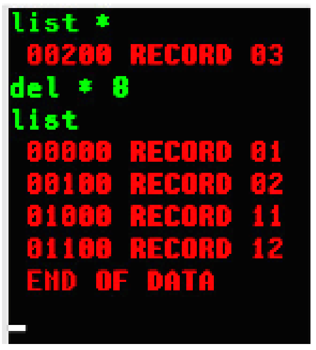
Like the LIST subcommand, the optional operands may be specified using sequence numbers of records in the dataset or by reference to the current line pointer. If no operands are included, the record referenced by the current line pointer is deleted. If a single sequence number is included, the record containing that sequence number is deleted. If a beginning and ending sequence number are included, the records containing those sequence number **and all records located between them** are deleted. More than a single record may also be deleted using the current line pointer by including the current line pointer indicator, an asterisk (*), followed by an integer indicating the total number of records to delete.



```
list *
 00400 RECORD 05
del
list *
 00300 RECORD 04
```



```
list
 00000 RECORD 01
 00100 RECORD 02
 00200 RECORD 03
 00300 RECORD 04
 00400 RECORD 05
 00500 RECORD 06
 00600 RECORD 07
 00700 RECORD 08
 00800 RECORD 09
 00900 RECORD 10
 01000 RECORD 11
 01100 RECORD 12
 END OF DATA
del 100 900
list
 00000 RECORD 01
 01000 RECORD 11
 01100 RECORD 12
 END OF DATA
```



```
list *
 00200 RECORD 03
del * 8
list
 00000 RECORD 01
 00100 RECORD 02
 01000 RECORD 11
 01100 RECORD 12
 END OF DATA
```

The three sessions above illustrate, from left to right, deleting a single record using the current line pointer (after the deletion the current line pointer moves to the preceding line), deleting records with sequence numbers 100 through 900, and deleting eight records beginning with the current line pointer referencing record number three.

### Repositioning the Current Line Pointer

Four subcommands are provided to allow you to reposition the current line pointer:

- TOP - moves the current line pointer to the first record in dataset
- BOTTOM - moves the current line pointer to the last record in dataset
- UP [count] - moves the current line pointer the number of records specified towards the beginning of the file (if count is not specified, the default is 1)
- DOWN [count] - moves the current line pointer the number of records specified towards the end of the file (if count is not specified, the default is 1)

The subcommands used to reset the location of the current line pointer are most often used in conjunction with the FIND and CHANGE subcommands.

### Verifying Current Line Pointer and Changes

A subcommand that is very useful when doing FIND and CHANGE subcommands is VERIFY (which may be abbreviated V). The syntax of the subcommand is:

    Verify [ON | OFF]

The subcommand changes the verify setting for the current edit session. When Verify is on, each time the current line pointer is changed or changes are made to a dataset record, the record is displayed. With Verify set to off, the records are not displayed.

## Finding Records Using Content - FIND Subcommand

The FIND subcommand (which may be abbreviated F) is used to search for a specified character string in the records of the dataset. The search begins with the current line. If the string is found the current line pointer is left positioned at the first record that contained the string being searched for. If the string is not found, the current line pointer is left positioned at the last record in the dataset. The syntax of the subcommand is:

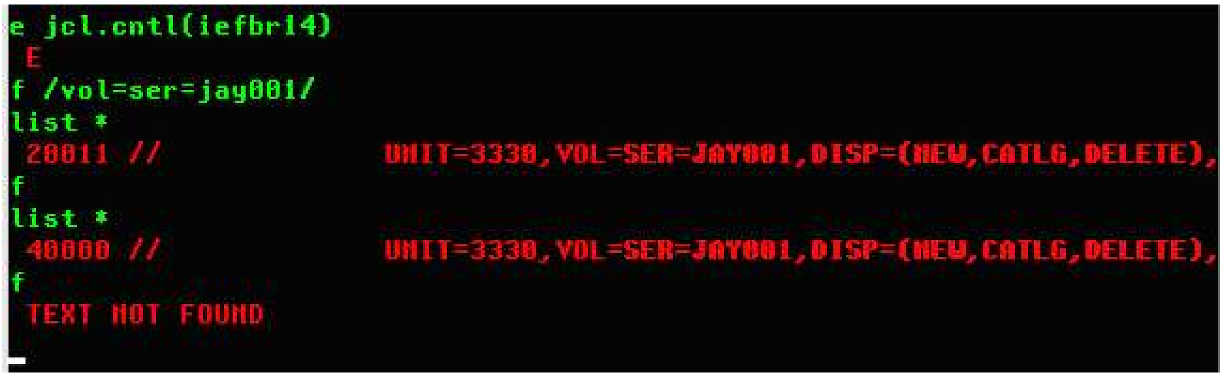
Find <character string> [<position>]

The character string operand may be delimited by any non-numeric character which does not appear within the specified character string, except blank, comma, tab, parenthesis, asterisk, apostrophe, or semicolon.

If the optional position operand is specified, each record is tested for the specified character string **only** beginning at the specified character position. If omitted, each character position of each record is tested for the presence of the character string.

After executing a FIND subcommand, the same search may be repeated without respecifying the character string by entering the subcommand with no arguments.

If the FIND subcommand does not find the specified character string, the message TEXT NOT FOUND will be displayed.



```
e jcl.cntl(iefbr14)
 E
f /vol=ser=jay001/
list *
 28011 //              UNIT=3330,VOL=SER=JAY001,DISP=(NEW,CATLG,DELETE),
f
list *
 40000 //              UNIT=3330,VOL=SER=JAY001,DISP=(NEW,CATLG,DELETE),
f
 TEXT NOT FOUND
```

The session above illustrates a FIND subcommand using the forward slash (/) as a delimited of the character string to be searched for. Following the first execution, LIST is used to display the record which satisfied the search. A second FIND command with no operands continues the search using the same string; the second search begins with the record following the one which satisfied the previous search. Again LIST displays the record where the search was satisfied. A third execution of FIND results in the string not being found.

## Changing Contents of Records - CHANGE Subcommand

The CHANGE subcommand (which may be abbreviated C) is used to modify a character string in one record, or a range of records, in the dataset. The syntax of the subcommand is:

Change <line specification> <change specification> [ALL]

The <line specification> may be one of the following:

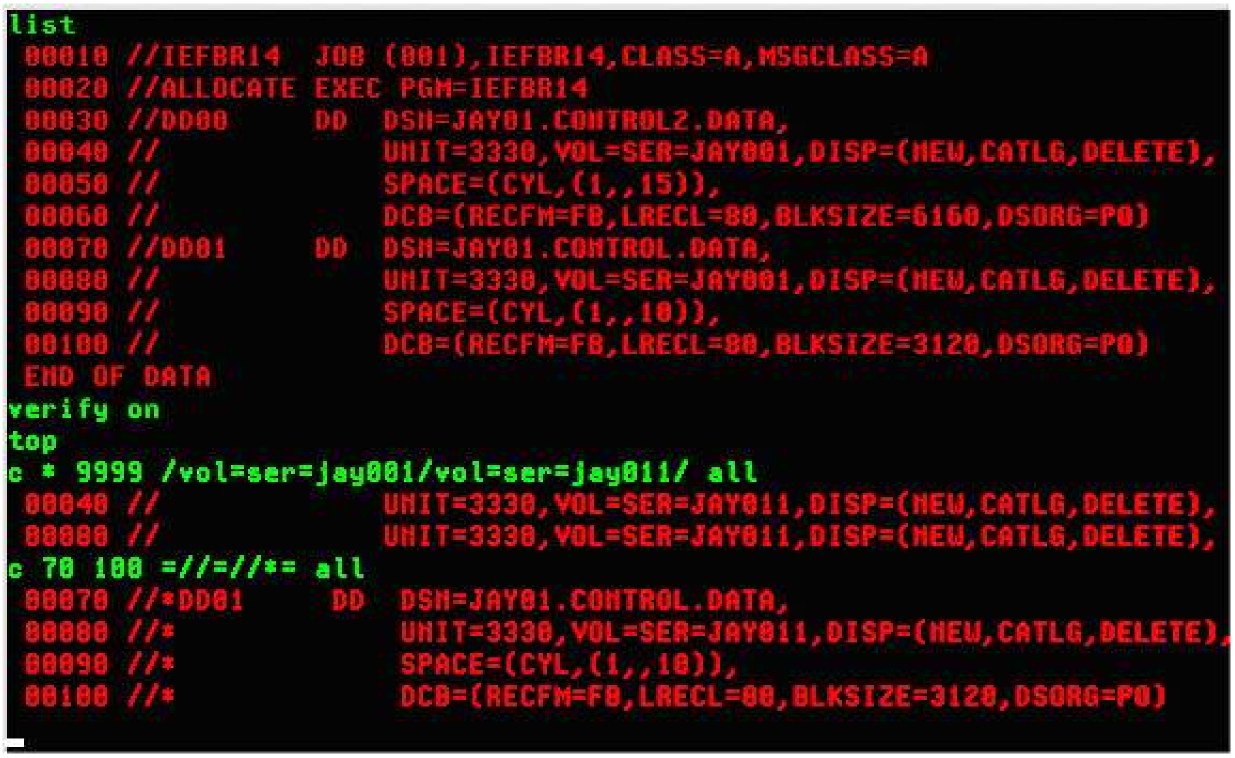
- current line pointer (*) to limit the search to the current record
- current line pointer and <count> to limit the search to the range of lines consisting of the current record plus the following number of records indicated by the specified <count>
- <single sequence number> to limit the search to a specified record
- <beginning sequence number> and <ending sequence number> to limit the search to the range of lines consisting of the records containing the specified sequence numbers and the records between them.

The <change specification> is a pair of character strings, delimited and separated by a unique delimiter character. The delimiters may be any non-numeric character which does not appear within the specified character strings, except blank, comma, tab, parenthesis, asterisk, apostrophe, or semicolon. Some examples:

| | |
|---|---|
| /VOL=SER=MVSRES/VOL=SER=SMP001/ | search for VOL=SER=MVSRES; if found change to VOL=SER=SMP001 |
| %DSORG=PS%DSORG=PO% | search for DSORG=PS; if found change to DSORG=PO |
| =//SYSGEN06=//SYSGEN6A= | search for //SYSGEN06; if found change to //SYSGEN6A |
| /RETAIN// | search for RETAIN; if found remove it |

Note in the last example: if the second string is omitted, the first string, if found, is removed from the record.

If the third operand, ALL, is specified, all occurrences of the search string found are replaced. If ALL is omitted only the first occurrence will be replaced.



```
list
 00010 //IEFBR14  JOB (001),IEFBR14,CLASS=A,MSGCLASS=A
 00020 //ALLOCATE EXEC PGM=IEFBR14
 00030 //DD00     DD  DSN=JAY01.CONTROL2.DATA,
 00040 //             UNIT=3330,VOL=SER=JAY001,DISP=(NEW,CATLG,DELETE),
 00050 //             SPACE=(CYL,(1,,15)),
 00060 //             DCB=(RECFM=FB,LRECL=80,BLKSIZE=6160,DSORG=PO)
 00070 //DD01     DD  DSN=JAY01.CONTROL.DATA,
 00080 //             UNIT=3330,VOL=SER=JAY001,DISP=(NEW,CATLG,DELETE),
 00090 //             SPACE=(CYL,(1,,10)),
 00100 //             DCB=(RECFM=FB,LRECL=80,BLKSIZE=3120,DSORG=PO)
 END OF DATA
verify on
top
c * 9999 /vol=ser=jay001/vol=ser=jay011/ all
 00040 //             UNIT=3330,VOL=SER=JAY011,DISP=(NEW,CATLG,DELETE),
 00080 //             UNIT=3330,VOL=SER=JAY011,DISP=(NEW,CATLG,DELETE),
c 70 100 =//=//*= all
 00070 //*DD01     DD  DSN=JAY01.CONTROL.DATA,
 00080 //*             UNIT=3330,VOL=SER=JAY011,DISP=(NEW,CATLG,DELETE),
 00090 //*             SPACE=(CYL,(1,,10)),
 00100 //*             DCB=(RECFM=FB,LRECL=80,BLKSIZE=3120,DSORG=PO)
```

## Saving Changes

All changes made to the dataset during the Edit session are made in a work space. In order to write the changes back to the dataset, you must specify the SAVE subcommand (which may be abbreviated S). The syntax of the subcommand is:

Save [<dataset name>]

If the optional <dataset name> operand is included, the changes are written to this dataset rather than to the dataset specified at the beginning of the edit session.

The Save subcommand may be issued any number of times during the edit session to provide a *snapshot* of changes up to that point in the session.

## Terminating a Edit Session

To terminate the edit session, the END subcommand is used. If there have been changes made to the work space without issuing a SAVE subcommand, you will be prompted to SAVE the changes or lose them. You can use one of the optional operands for the END subcommand to explicitly SAVE or abandon the changes you have made:

END SAVE - saves changes made to the dataset and ends the session
END NOSAVE - abandons changes made to the dataset and ends the session

## Develop, Test, and Execute Programs

In my professional experience, other than maintenance of the MVS Operating System, the main use of TSO has been the development and testing of user programs for eventual use in a production environment. Since just about everything that happens during a TSO session involves the execution of a program, it goes without saying that TSO is designed for executing programs in the time shared environment. So in this section I will discuss some of these aspects of using TSO.

In a typical *real world* environment, every TSO user has allocated for their personal use a set of partitioned datasets, usually referred to as libraries. These libraries contain source code for programs in a variety of languages, job control language for running batch jobs, collections of data for testing programs, etc. The structure of dataset names in MVS isn't just a random arrangement; by design the naming structure fits and supports the TSO user's environment. An MVS dataset name is composed of one or more qualifiers separated by periods. Each qualifier consists of one to eight characters (A through Z, 0 through 9, or national characters - #, $, @), and the first character of each qualifier may not be numeric. The length of the entire dataset name, composed of the individual qualifiers separated by periods, must not exceed 44 characters.

By convention, the first qualifier of a TSO User's dataset consists of the TSO User ID. The second qualifier is chosen to designate a project or function as a means of collecting related groups of data together. The third qualifier designates the type of data contained in the dataset. Of course, in a real world environment where ISPF is being used, there could easily be more than three qualifiers, but the convention of first and last indicating the TSO User ID and the last the data type would probably still apply. So examples of the types of datasets (libraries) a typical TSO User might have are:

<userid>.PROJECT1.COBOL - a library of source programs in COBOL
<userid>.PROJECT2.RPG - a library of source programs in RPG
<userid>.PROJECT3.PLI - a library of source programs in PL/I
<userid>.JCL.CNTL - a library Job Control Statements for batch jobs
<userid>.TEST.DATA - a library of test data
<userid>.CLIST - a library of the user's Command Lists
<userid>.PROJECT1.OBJLIB - a library of compiled programs
<userid>.LOADLIB - a library of executable programs

When you are creating datasets in your own Hercules/MVS3.8j environment, if you maintain an organizational structure similar to this you will find that at times the final qualifier of the dataset will be used by TSO to correctly determine the type of data contained in the dataset. Also, you should find this to be a very intuitive means of organizing your datasets.

### Editing Program Source Code and Job Control Language

Although I have described the basics of the EDIT command in the previous section, I would strongly recommend using RPF to edit datasets. EDIT was designed in the days of line mode terminals and is just too awkward to be very productive.

### Foreground Compilation

Although the HELP system lists the foreground compiler commands - ASM, COBOL, FORT - these commands are not included in the basic operating system. You can write your own Command Lists that will implement the equivalent functionality and I believe some members of the Hercules' community have made some effort in that area. There are also some of these programming language "command prompters" available on the CBT Tape, but the versions there may require modifications to work with MVS 3.8j and the early versions of the compilers we have available for use from MVT.

But, since these commands are not available in the basic system, I will not be discussing them and recommend you use batch jobs to assemble, compile, and link your programs. The main focus of this section is allocating the required datasets for testing and executing programs which are already compiled and link-edited. (For a discussion of batch procedures available to assemble, compile, and link-edit programs, see Assembling, Compiling, Link-Editing ...)

### Foreground Execution of Programs

Almost any program can be executed in foreground, even if it was written to be executed in a batch jobstream. The prerequisite to loading the program and beginning execution is that the datasets that are expected by the program, which would normally be set up by the batch scheduler using parameters provided by Job Control Language statements, are your responsibility. Most TSO Commands are in actuality nothing more than specialized programs written in assembler. These programs, simply by virtue of the fact that they are designed and written to be run in the TSO environment, allocate the datasets they require dynamically before the program begins interacting with the user session. They also do the required cleanup before terminating, which includes de-allocating datasets that were dynamically allocated by their code initially. The commands you need to be familiar with that provides these functions are ALLOCATE, ATTRIB, and FREE.

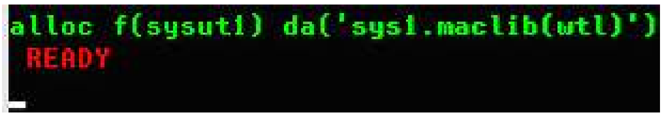
### Allocating Datasets - ALLOCATE Command

The ALLOCATE command (which may be abbreviated ALLOC) is used to perform two separate functions, although there are times when both are performed simultaneously. Most frequently, the ALLOCate command is used to perform the functions of the DD statement in Job Control Language. That is to provide the parameters necessary for a program to open a dataset using the DD Name that was assigned in the Dataset Control Block when the program was compiled/assembled. The ALLOCate command may also be used to create a new, empty dataset, even though the user doesn't intend to execute a program immediately to utilize it.

Many years ago I taught classes to accountants and clerical type employees at my company who needed to use TSO to execute programs through remote terminals that were sometimes located remotely from the data center and the programming staff. I found they easily grasped the concept of the ALLOCate command if I had them visualize their terminal screen surrounded by *hooks* and then had them visualize that the ALLOCate command was used to hang *tags* onto the hooks prior to executing their programs. The first, and most important, *tag* was the DD Name, in order to allow the program to locate a particular file by name. The second most important tag was the dataset name, which allowed the operating system to associated the physical dataset with the file name. Of course, any other *tags* that were required could also be placed on the *hook* for a particular DD Name - dataset organization, disposition, volume serial number, etc - in order to provide the necessary attributes for the dataset. The syntax of the command is:

```
ALLOCate DSName(<dataset name to allocate>)
                [{ DDName(<dd name>) | File(<dd name>) } ]
                [ { NEW | OLD | SHR | MOD } ]
                [ { KEEP | DELETE | CATALOG | UNCATALOG } ]
                [ UNIT(<unit name>) ]
                [ VOLume(<volume serial number>) ]
                [ SPACE(<primary quntity> [<secondary quantity>] ]
                [ TRACKS | CYLINDERS ]
                [ DIR(<directory blocks>) ]
                [ USING(<attribute set name>) ]
```

There are additional operands for the command, but these are the most frequently used for datasets other than print/punch datasets that are to be handled by JES2. Not all operands are used every time, in fact some are mutually exclusive, and in many situations only a very few operands are necessary to provide the required information to execute a program. Remember, for an existing dataset, many of the Dataset Control Block fields are populated from the dataset label, not from the JCL statement or ALLOCate command. So, to allocate a catalogued dataset for input, it can be as simple as:



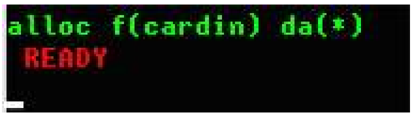
```
alloc f(sysut1) da('sys1.maclib(wtl)')
 READY
```

The complexity of the operands required for the ALLOCate command vary from the simple:

- catalogued datasets used only as input
- card image input read from the TSO User's terminal
- print/punch output directed to the TSO User's terminal
- print/punch output directed to JES2 for processing
- dd names which are to be assigned NULL files (ie. DD DUMMY)

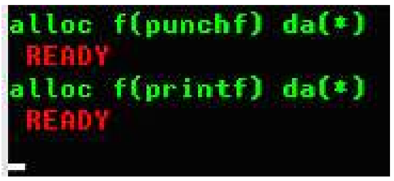
to the most complex, which would be creating a member in a new partitioned dataset that is to be allocated when it is opened and catalogued when it is closed, with DCB attributes provided by the ALLOCate statement rather than relying upon the program code to provide them.

Files which a program expect to read card image data provided by JES2 may easily be ALLOCated to receive input from the TSO User's session:



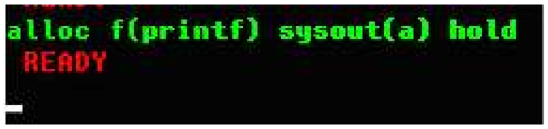
```
alloc f(cardin) da(*)
 READY
```

Likewise, files which are designed to produce card image or line printer output, utilize the same ALLOCate syntax to direct the output of the program to the TSO User's screen:



```
alloc f(punchf) da(*)
 READY
alloc f(printf) da(*)
 READY
```

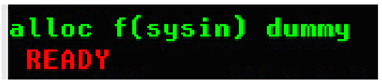
A special operand is provided to ALLOCate a JES2 dataset to a DD Name - SYSOUT. When a JES2 dataset is allocated, records produced by the program are written to the JES2 spool, just as they would be if the program was executed in batch. When the dataset is closed and the allocation is released, the data becomes a print (or punch) job on the spool and its disposition is handled according to the JES2 parameters supplied with the ALLOCate command and the Job Class as defined in JES2.



```
alloc f(printf) sysout(a) hold
 READY
```

Instead of the usual operands required for a dataset, such as dataset name (DSName), SYSOUT(<class>) is used to ALLOCate JES2 to the DD Name with the output spooled to the class specified. The optional operands HOLD or NOHOLD may be included to modify the disposition of the print/punch output. The default is NOHOLD. You may also include the optional operand DEST(<station id>) to specify a remote site to which the output is to be routed.

A special operand - DUMMY - is provided to assign NULL files to DD Names:



```
alloc f(sysin) dummy
 READY
```

When new datasets are being created more information is required, which requires more operands to be specified. Some information must be supplied by a user attribute list, which is provided by the ATTRIB command.

## Providing Dataset Attribute Lists - ATTRIB Command

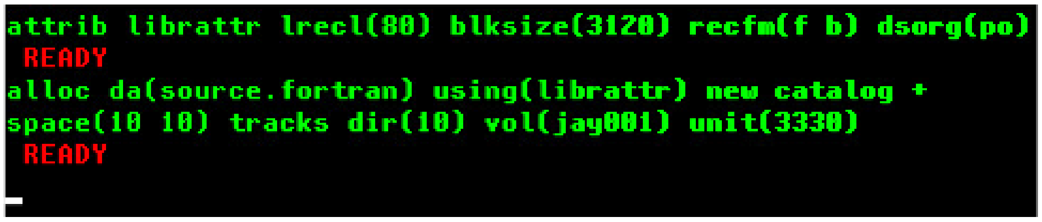
The ATTRIB command specifies a list of dataset attributes. The list is given a name by which it is subsequently referred to in an ALLOCate command. The syntax of the command is:

```
ATTRIB <attribute list name>
        [ LRECL(<logical record length>) ]
        [ BLKSIZE(<logical block size>) ]
```

[ RECFM(<record format>) ]

[ DSORG(<dataset organization>) ]
[ INPUT | OUTPUT ]
[ EXPDT(<expiration date>) | RETPD(<retention period>) ]

Like the ALLOCate command, not all operands will be used for all datasets. The attributes you will most frequently use are LRECL, BLKSIZE, RECFM, and DSORG. Once specified with an ATTRIB command, an attribute list remains defined for the duration of the TSO session or until explicitly released with the FREE command. Here is an example of an ATTRIB and ALLOCate command pair creating a new library:



```
attrib librattr lrecl(80) blksize(3120) recfm(f b) dsorg(po)
 READY
alloc da(source.fortran) using(librattr) new catalog +
space(10 10) tracks dir(10) vol(jay001) unit(3330)
 READY
```

The characteristics of the attribute list

- logical record size of 80
- block size of 3,120
- record format fixed, blocked
- dataset organization partitioned

are stored in the list name libattr. These are the commonly used attributes for source code libraries, so this attribute list might be used to create, with the ALLOCate command, several library datasets without having to reenter the attribute characteristics.

The ALLOCate command references the attribute list with the second operand - using(librattr). Since no DDName parameter is specified for the ALLOCate command, the purpose of this ALLOCate command may be assumed to be simply the creation of the new dataset. In fact, a DDName is automatically generated by TSO for the ALLOCate command, and you can learn what the name is using the LISTALC command described below.

Also note that in the ALLOCate command, the line continuation character (+) is used at the end of the first line to indicate that the command continues on the next line following. This allows for the orderly entry of long commands to make the more readable.

After you become familiar with the functions of the ALLOCate and ATTRIB commands, you might want to investigate an alternative from the CBT tape: DDN.

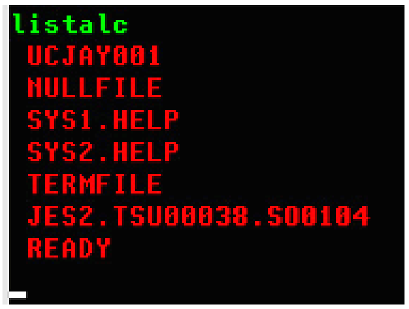
### Listing Allocated Datasets - LISTALC Command

The LISTALC command is used to list the datasets allocated to your TSO session. The syntax of the command is:

LISTALC [STATUS] [HISTORY] [MEMBERS] [SYSNAMES]

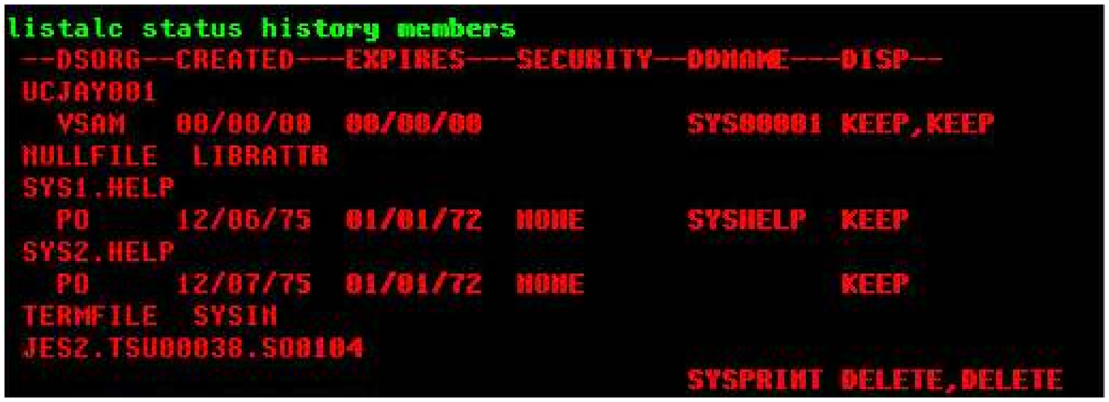
LISTALC [STATUS] [HISTORY] [MEMBERS] [SYSNAMES]

If no operands are included, only the dataset names are listed for the datasets currently allocated:



```
listalc
 UCJAY001
 NULLFILE
 SYS1.HELP
 SYS2.HELP
 TERMFILE
 JES2.TSU00038.S00104
 READY
```

The default information provides very few clues about the datasets, so adding STATUS (provides DD Name and disposition), HISTORY (provides creation date, expiration date, and owner ID), and MEMBERS (provides a list of member names for partitioned datasets) will provide much more useful information:



```
listalc status history members
 --DSORG--CREATED---EXPIRES---SECURITY--DDNAME---DISP--
 UCJAY001
   VSAM   00/00/00  00/00/00            SYS00001 KEEP,KEEP
 NULLFILE  LIBRATTR
 SYS1.HELP
   PO     12/06/75  01/01/72  NONE      SYSHELP  KEEP
 SYS2.HELP
   PO     12/07/75  01/01/72  NONE               KEEP
 TERMFILE  SYSIN
 JES2.TSU00038.S00104
                                        SYSPRINT DELETE,DELETE
```

If you are planning on using TSO frequently, you might want to investigate the ALIST TSO command from the CBT tape.

## Releasing Allocated Datasets - FREE Command

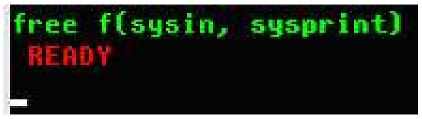
The FREE command is used to dynamically release datasets allocated during the TSO session. All allocated datasets are automatically released when a TSO session is terminated (via LOGOFF command or session time out). However, there are several reasons you might want to release datasets before the end of the TSO session:

- a dataset allocated to the terminal session is required for exclusive access by a batch job and the batch job will not proceed until the dataset is released
- the disposition specified for a dataset allocated to a TSO session will not be processed until the dataset is released - SYSOUT datasets will not be available for printing and CATALOG, UNCATALOG, or DELETE disposition won't occur
- you have used up all the control blocks available for allocation of datasets and need to free some up to continue working

The syntax of the command is:

FREE { File(<list of DD Names to be released>) |
DAtaset(<list of dataset names to be released>) }

The datasets listed in the FREE command are dynamically released and the disposition specified when they were allocated is processed.



```
free f(sysin, sysprint)
 READY
```

There are times when you might want to free all allocated datasets. This may be accomplished with the FREEALL TSO command from the CBT Tape.

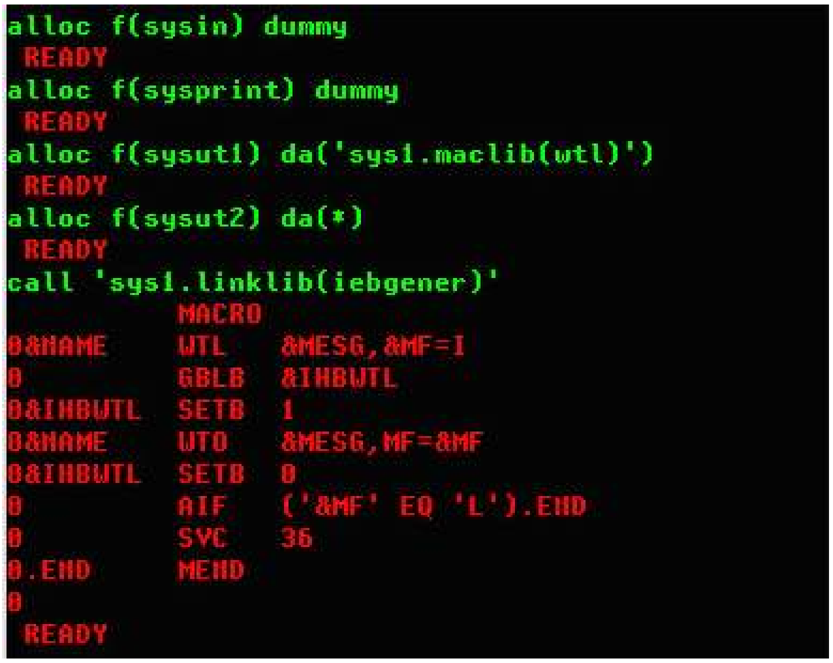
## Load and Execute Program - CALL Command

The CALL command is used to retrieve a load module (which has been processed by the Linkage Editor) from a partitioned Dataset and transfers control the the first executable instruction in the load module. The load module executes in the TSO session's address space. The syntax of the command is:

CALL <dsname>(<member>) [ '<parameter values>' ]

The <dsname> operand specifies a partitioned dataset that contains executable load modules as members. The <member> operand names the specific load module to be loaded/executed. If the TSO User ID has created a library with the name <userid>.LOAD, the dataset name may be omitted and only the (<member>) operand is required. If <member> is omitted, CALL attempts to load a member named **TEMPNAME** from the library. The optional <parameter values> may be used to specify up to 100 characters of information that is passed to the program as a parameter value.

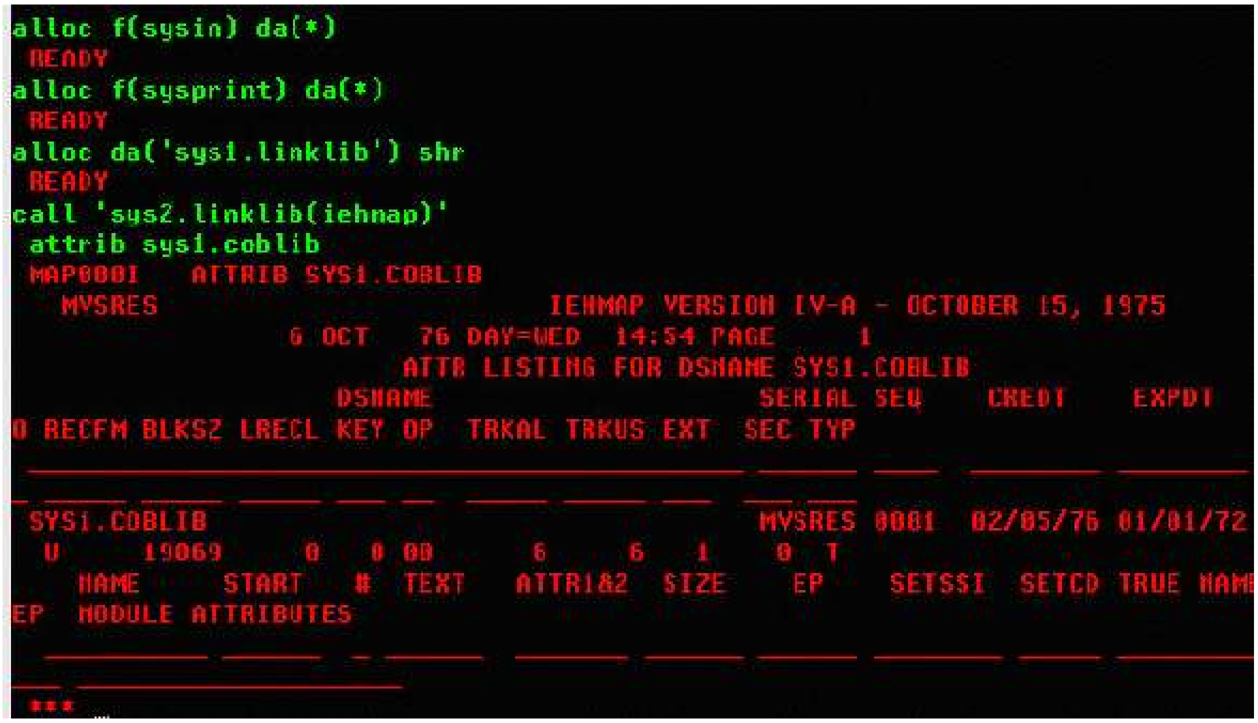
An example executing a utility program in foreground:



```
alloc f(sysin) dummy
 READY
alloc f(sysprint) dummy
 READY
alloc f(sysut1) da('sys1.maclib(wtl)')
 READY
alloc f(sysut2) da(*)
 READY
call 'sys1.linklib(iebgener)'
          MACRO
0&NAME    WTL   &MESG,&MF=I
0         GBLB  &IHBWTL
0&IHBWTL  SETB  1
0&NAME    WTO   &MESG,MF=&MF
0&IHBWTL  SETB  0
0         AIF   ('&MF' EQ 'L').END
0         SVC   36
0.END     MEND
0
 READY
```

IEBGENER is typically executed with SYSIN as a NULL file. Since there is little useful information produced by IEBGENER in the SYSPRINT dataset, in this example both SYSIN and SYSPRINT are ALLOCated to NULL files. SYSUT1 is read as input by IEBGENER and is ALLOCated to a short member of SYS1.MACLIB. SYSUT2, to which IEBGENER copies the contents of the dataset ALLOCated to SYSUT1, is ALLOCated to the TSO terminal. When IEBGENER is loaded from SYS1.LINKLIB and executed, the records read from the input DD Name - SYSUT1 - are written to the terminal screen.

An example of a more complex utility, showing SYSIN input read from the TSO terminal:



```
alloc f(sysin) da(*)
 READY
alloc f(sysprint) da(*)
 READY
alloc da('sys1.linklib') shr
 READY
call 'sys2.linklib(iehmap)'
 attrib sys1.coblib
 MAP0001    ATTRIB SYS1.COBLIB
   MVSRES                        IEHMAP VERSION IV-A - OCTOBER 15, 1975
              6 OCT   76 DAY=WED  14:54 PAGE     1
                     ATTR LISTING FOR DSNAME SYS1.COBLIB
                 DSNAME                    SERIAL SEQ    CREDT    EXPDT
0 RECFM BLKSZ LRECL KEY OP  TRKAL TRKUS EXT  SEC TYP
 _____________________________________________ ______ _____  _______ _______
_ _____ _____ _____ ___ ___  _____ _____ ____  ___ ___
 SYS1.COBLIB                                MVSRES 0001  02/05/76 01/01/72
  U     19069     0    0 00     6     6   1    0  T
    NAME      START    #  TEXT    ATTR1&2  SIZE    EP    SETSSI  SETCD TRUE NAME
EP  MODULE ATTRIBUTES
   __________ ______  _ _______  ______ ______ ______ ________ ______ ______
___ _____________________
 ***
```

Obviously this program is designed to produce 132 column output, so the lines *wrap* when displayed at the terminal, but executing such a program interactively can provide the information you need quickly and without submitting the job to batch and, either examining the output on the spool (with the OUTPUT command, QUEUE, or RPF) or printing the output to a Hercules' attached file and examining it there.

## Debugging Programs - TEST Command

The TEST command (which may be abbreviated T) allows programs to be executed under precise control of the user for the purpose of debugging. Under TEST:

- breakpoints may be set at any instruction location; when a breakpoint is reached, execution is halted and control is given to the terminal operator
- initial values may be supplied to data areas in program storage before execution commences
- register and data storage values may be displayed and altered
- the program status word may be displayed
- contents of control blocks (DEB, DCB, TCB, etc.) may be displayed and altered
- execution of program instructions may proceed one instruction at a time or through program sections

- execution of program instructions may proceed one instruction at a time or through program sections under control of breakpoints

Obviously, the most successful use of the TEST command may be made if the program being executed was written in assembler **and** the if user is familiar with object code debugging, MVS addressing, MVS control blocks, and the structure of the program being executed. The TEST command allows more versatile addressing of the program's structure if the program being executed under TEST has been assembled and link-edited with the TEST option, which inserts specific information into the load module that is accessible to the TEST command. The syntax of the command is:

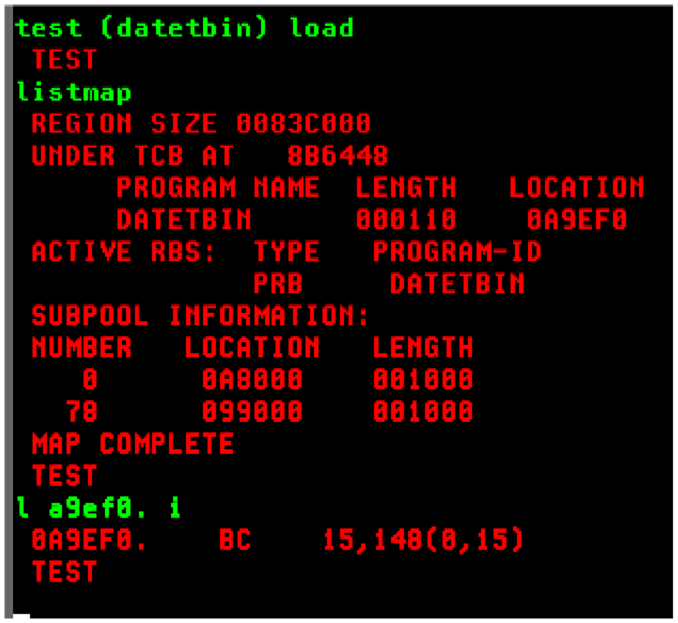
Test <program name> [ 'parameters' ] { LOAD | OBJECT }

<program name> specifies the dataset from where the program to be tested is to be loaded. The optional 'parameters' may specify a character string of up to 100 characters to be passed to the program. The LOAD/OBJECT operand specifies whether the named program is either a load module (has been processed by the linkage editor) or an object module (produced by the assembler or a compiler, but not link-edited).

After the program is loaded into the TEST environment, there are several subcommands that are available to control the execution of the program and examine and/or change values in storage or control blocks. A few of the more frequently used subcommands are:

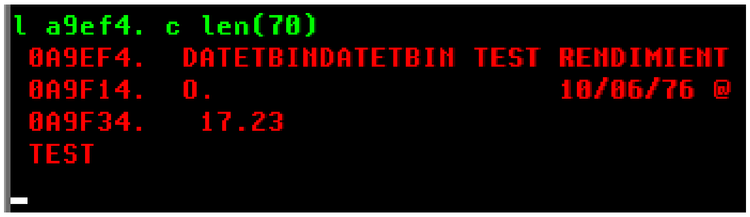
| | |
|---|---|
| ASSIGN <address> = <type> <value> | place data values into storage locations |
| AT { <address> \| (<address list>) } | set breakpoint at address (or addresses) |
| GO [ <address> ] | execute program (optionally from address) |
| LIST { <address> \| (<address list>) } <type> [ LENGTH(<length>) ] | display contents of storage |
| OFF { <address> \| (<address list>) } | remove breakpoints at address (or addresses) |

There are more subcommands that a regular user of TEST might need to become familiar with, but I limit my discussion here to the most basic functions.


```
test (datetbin) load
 TEST
listmap
 REGION SIZE 0083C000
 UNDER TCB AT   8B6448
      PROGRAM NAME  LENGTH   LOCATION
      DATETBIN      000110    0A9EF0
 ACTIVE RBS:  TYPE   PROGRAM-ID
              PRB     DATETBIN
 SUBPOOL INFORMATION:
 NUMBER   LOCATION   LENGTH
    0       0A8000    001000
   78       099000    001000
 MAP COMPLETE
 TEST
l a9ef0. i
 0A9EF0.    BC    15,148(0,15)
 TEST
```

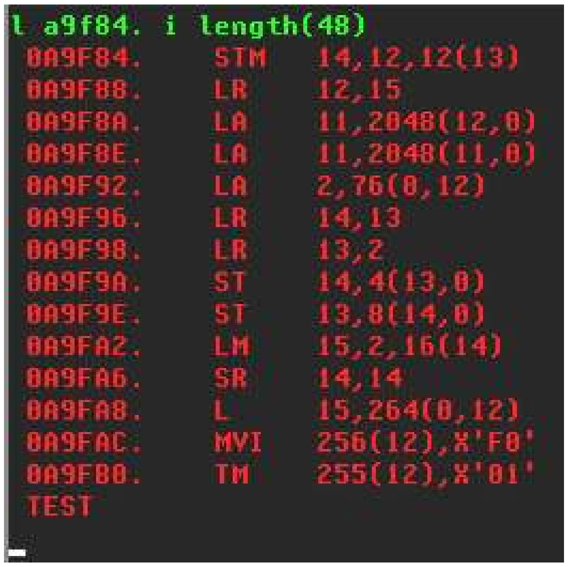
In the session above, a program is loaded into TEST. The program is a short assembler load module that is used to benchmark a simple routine. It is loaded from the <userid>.LOAD dataset; the member name is DATETBIN; the member has been processed by the linkage editor - LOAD operand.

The LISTMAP subcommand displays key areas of the program's storage. The first location of the storage area is 0A9EF0 - a LIST subcommand of that address with a TYPE designation I (instruction) shows that the first instruction is an unconditional branch relative to Register 15. This matches what is expected from the listing produced when the program was assembled and is a branch around identifying constants at the beginning of the program.



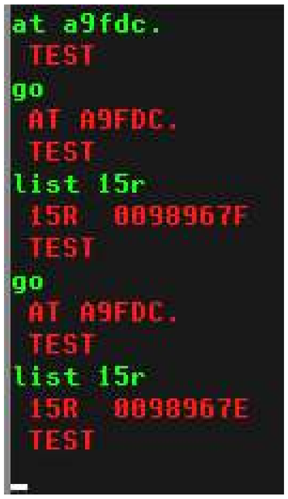
```
l a9ef4. c len(70)
 0A9EF4.  DATETBINDATETBIN TEST RENDIMIENT
 0A9F14.  0.                     10/06/76 @
 0A9F34.   17.23
 TEST
```

In the session above, the address immediately following the relative branch instruction (0A9EF0 + 4 = 0A9EF4) is displayed with a TYPE designation C (character) for a length of 70 characters. This reveals that a portion of the identifying constants are the CSECT name, a comment, and the date/time of assembly.



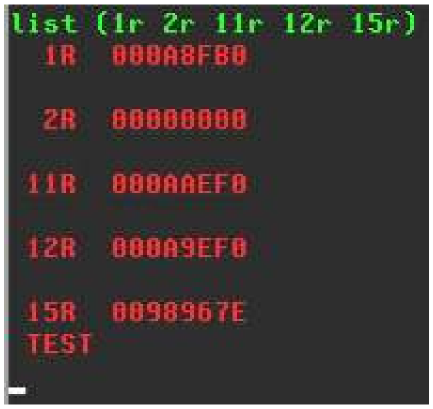
```
l a9f84. i length(48)
 0A9F84.     STM    14,12,12(13)
 0A9F88.     LR     12,15
 0A9F8A.     LA     11,2048(12,0)
 0A9F8E.     LA     11,2048(11,0)
 0A9F92.     LA     2,76(0,12)
 0A9F96.     LR     14,13
 0A9F98.     LR     13,2
 0A9F9A.     ST     14,4(13,0)
 0A9F9E.     ST     13,8(14,0)
 0A9FA2.     LM     15,2,16(14)
 0A9FA6.     SR     14,14
 0A9FA8.     L      15,264(0,12)
 0A9FAC.     MVI    256(12),X'F0'
 0A9FB0.     TM     255(12),X'01'
 TEST
```

Continuing to examine the storage for the load module, using the first instruction address (which is the address contained in Register 15 when the module is loaded by TEST) and the length from the relative branch instruction at that address, the next instruction that will execute is located at address 0A9F84 (0A9EF0 + 94 [the 148 offset in the displayed instruction is shown in decimal] = 0A9F84). A LIST subcommand for that address with a TYPE designation I (instructions) and a length of 48 shows the expected instructions establishing standard linkage (the instructions at addresses 0A8F84 through 0A9FA6).



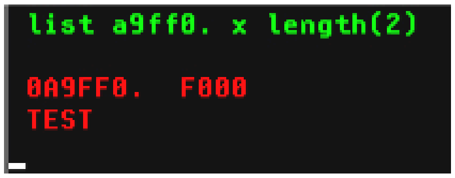
```
at a9fdc.
 TEST
go
 AT A9FDC.
 TEST
list 15r
 15R  0098967F
 TEST
go
 AT A9FDC.
 TEST
list 15r
 15R  0098967E
 TEST
```

The routine being benchmarked by the program is executed a number of times in a loop controlled by a counter maintained by Register 15. The last instruction in the loop is a BCT instruction at address 0A9FDC. In the session above, a breakpoint is established at that address. When the GO subcommand is given, the program runs until the breakpoint is reached, at which time the TEST prompt is displayed. A LIST subcommand is used to display the contents of Register 15 - **list 15r** - and the value displayed is 0098967F, or 9,999,999 in decimal, which is the initial value of the counter established by the loop. A second pair of GO and LIST subcommands show that after one iteration of the loop code, the value of Register 15 has been decremented by 1.



```
list (1r 2r 11r 12r 15r)
  1R  000A8FB0

  2R  00000000

 11R  000AAEF0

 12R  000A9EF0

 15R  0098967E
 TEST
```

Multiple addresses may be displayed with a single LIST subcommand, as shown above, by including a list of the addresses to be displayed in parentheses.



```
 list a9ff0. x length(2)

 0A9FF0.  F000
 TEST
```

The LIST subcommand above shows a display of the target of the MVI instruction located at address 0A9FAC.

## View and Manage Batch Job Facilities

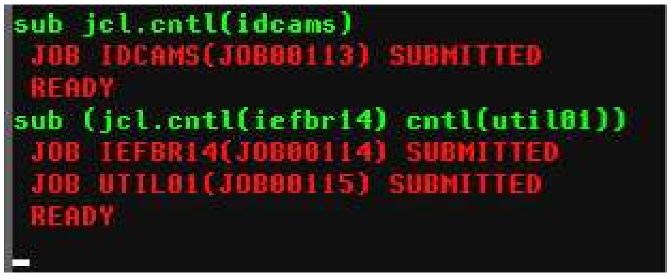
Although TSO provides a rich environment in which many tasks may be accomplished, there is still a necessity to utilize batch jobstreams for some tasks. And TSO provides a set of commands specifically to enable the TSO user to submit jobstreams to the batch environment, manage their progress, and retrieve and direct the disposition of the output those jobstreams create.

### Submitting Batch Jobs - SUBMIT Command

The SUBMIT command (which may be abbreviated SUB) is used to pass the contents of one or more datasets, or members of partitioned datasets, to JES2 for background processing. The syntax of the command is:

```
SUBmit { <dataset> | ( <dataset list> ) }
```

If a single dataset, or member, is to be submitted, the operand of the SUBmit command names the dataset. Multiple datasets may be submitted with a single invocation of the command by providing a list of datasets, enclosed by parentheses, as the operand. Note: Any dataset submitted may contain multiple batch jobs, which are defined by the presence of a JOB control statement.



```
sub jcl.cntl(idcams)
 JOB IDCAMS(JOB00113) SUBMITTED
 READY
sub (jcl.cntl(iefbr14) cntl(util01))
 JOB IEFBR14(JOB00114) SUBMITTED
 JOB UTIL01(JOB00115) SUBMITTED
 READY
```

The life cycle of a batch job submitted from TSO is:

- dataset containing Job Control Language statements is read by TSO and passed to JES2 by way of an internal card reader
- the card images are scanned and for each JOB card encountered, a JES2 job identifier is assigned to the job; a message is written to the TSO user showing the job identifier assigned by JES2
- the card images following the JOB card, which may be additional JCL statements and instream card image data, are written to the JES2 queue as belonging to the job identified by the assigned job identifer
- when an initiator becomes available, the JOB is selected for execution and processed by MVS
- output from the job is placed on the JES2 output queue

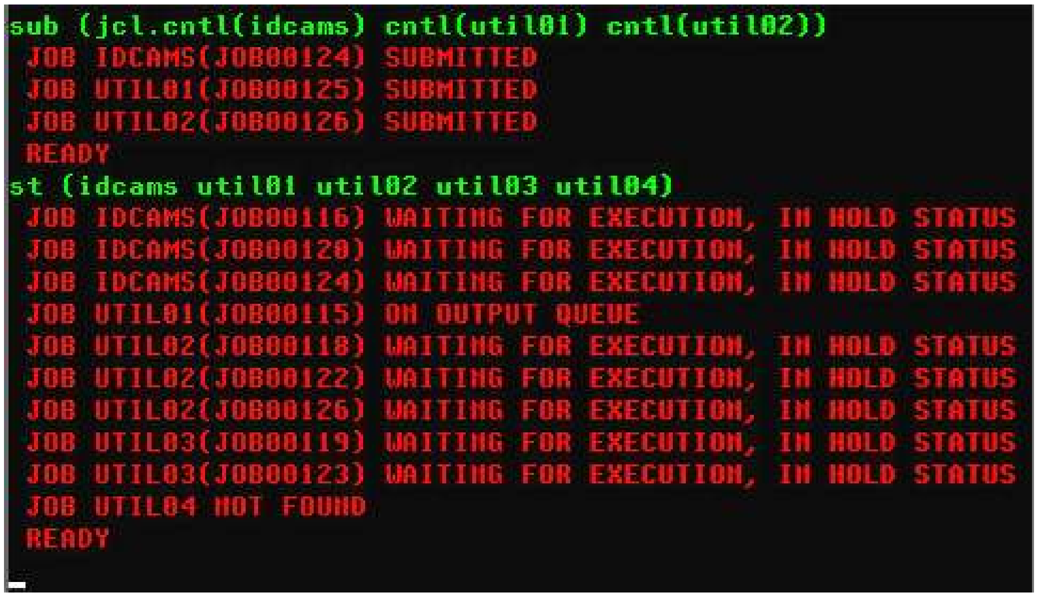
Datasets containing Job Control Language statements may also be submitted from within other command environments, for example: EDIT, RPF, and REVIEW.

### Requesting Batch Job(s) Status - STATUS Command

The STATUS command (which may be abbreviated ST) may be used to request information about batch jobstreams submitted by the TSO user. The format of the command is:

STatus { <jobname> | ( <joblist> ) }

The operand for the STATUS command should be either a single jobname or a list of jobnames separated by spaces and enclosed in parentheses. If the STatus command is issued with no operands, TSO attempts to find jobs whose name consists of the TSO User ID plus any one additional character (ie: <userid>A, <userid>B, <userid>C, ... <userid>9)



```
sub (jcl.cntl(idcams) cntl(util01) cntl(util02))
 JOB IDCAMS(JOB00124) SUBMITTED
 JOB UTIL01(JOB00125) SUBMITTED
 JOB UTIL02(JOB00126) SUBMITTED
 READY
st (idcams util01 util02 util03 util04)
 JOB IDCAMS(JOB00116) WAITING FOR EXECUTION, IN HOLD STATUS
 JOB IDCAMS(JOB00120) WAITING FOR EXECUTION, IN HOLD STATUS
 JOB IDCAMS(JOB00124) WAITING FOR EXECUTION, IN HOLD STATUS
 JOB UTIL01(JOB00115) ON OUTPUT QUEUE
 JOB UTIL02(JOB00118) WAITING FOR EXECUTION, IN HOLD STATUS
 JOB UTIL02(JOB00122) WAITING FOR EXECUTION, IN HOLD STATUS
 JOB UTIL02(JOB00126) WAITING FOR EXECUTION, IN HOLD STATUS
 JOB UTIL03(JOB00119) WAITING FOR EXECUTION, IN HOLD STATUS
 JOB UTIL03(JOB00123) WAITING FOR EXECUTION, IN HOLD STATUS
 JOB UTIL04 NOT FOUND
 READY
```

If there are multiple jobs in the JES2 queues with names identical to a jobname included on the STATUS command, the status display will contain all matching jobs. Notice in the session above there are duplicate jobs with the names IDCAMS, UTIL02, and UTIL03. But all jobs have a unique job identifier assigned by JES2 (JOBnnnnn).

In the session above, the display shows that some of the matching jobs are awaiting execution (because they are HELD), one job is on the output queue (has been processed by MVS and has output available for viewing), and one jobname specified was not found in the system.

## Viewing/Routing Output from Batch Jobs - OUTPUT Command

The OUTPUT command (which may be abbreviated OUT) may be used to:

- retrieve output produced by a batch job and display it at the terminal
- retrieve output produced by a batch job and place it in a dataset
- change the class of output produced by a batch job to allow printing/punching by JES2

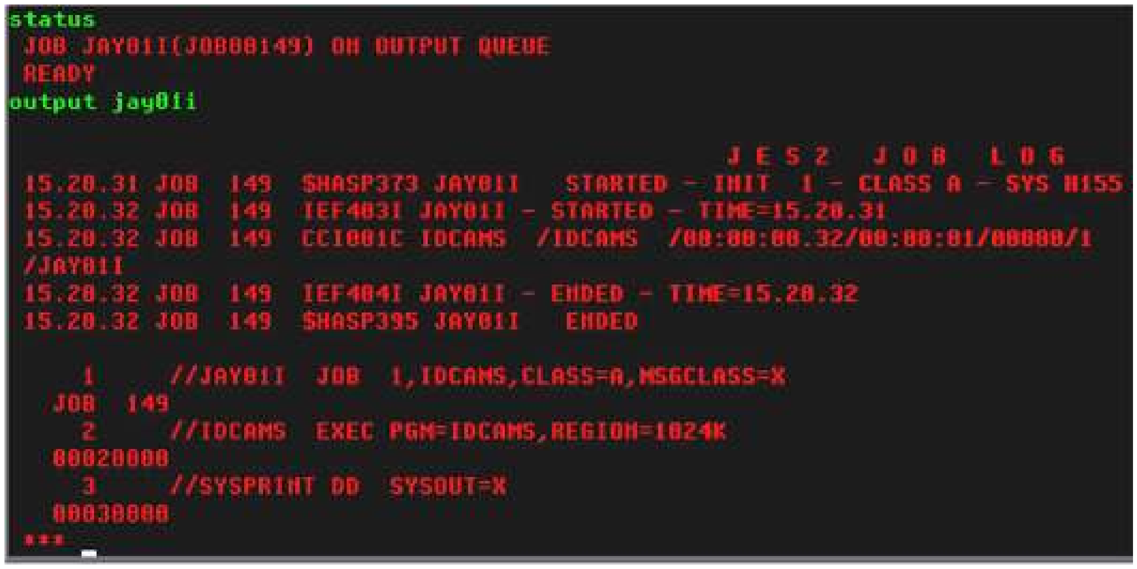
The format of the command is:

OUTput <jobname>[<job id>] [ PRINT( * | <dataset> ) | NEWCLASS(<class>)]

Some restrictions apply to which jobs are eligible to be processed by the OUTPUT command:

1. the jobname must consist of your TSO User ID, optionally followed by additional characters
2. the output must have been written to JES2 output class X

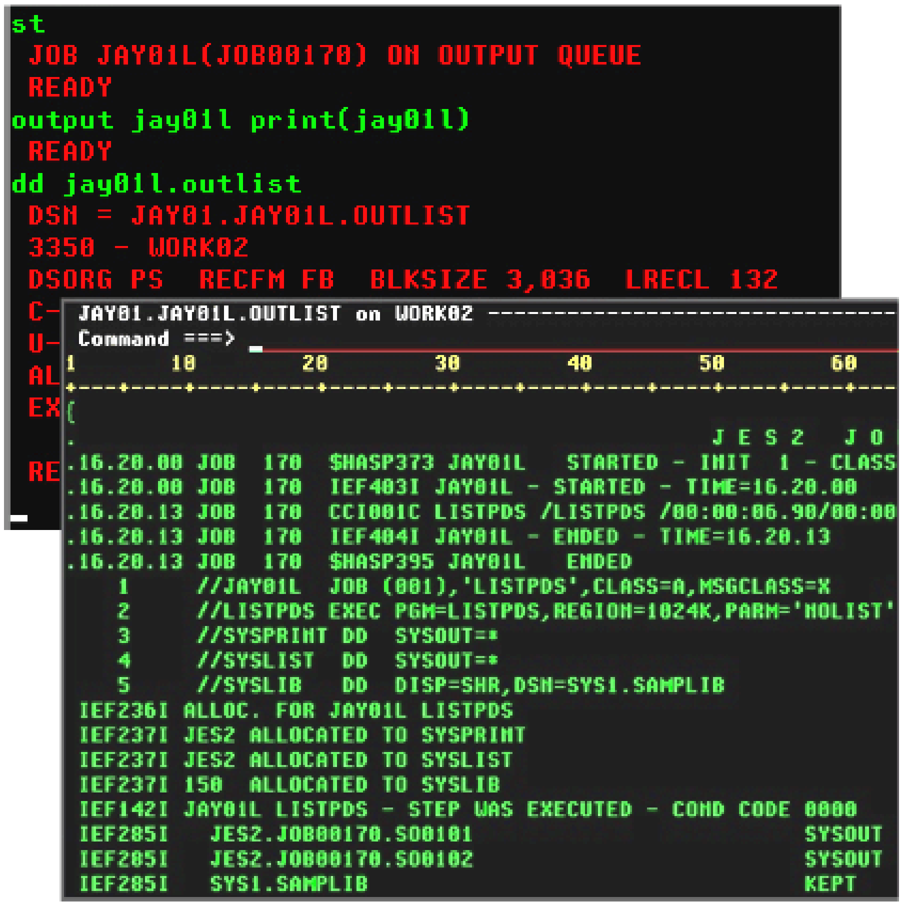
The only required operand for the OUTPUT command is the jobname (and optional JES2 job id). If only the jobname is provided as an operand to the OUTPUT command, the operand **PRINT(*)** is assumed and the output produced by the background job is retrieved from the JES2 queue and displayed at the TSO User's terminal:



```
status
 JOB JAY01I(JOB00149) ON OUTPUT QUEUE
 READY
output jay01i

                                                  J E S 2   J O B   L O G
 15.20.31 JOB  149  $HASP373 JAY01I   STARTED - INIT  1 - CLASS A - SYS H155
 15.20.32 JOB  149  IEF403I JAY01I - STARTED - TIME=15.20.31
 15.20.32 JOB  149  CCI001C IDCAMS  /IDCAMS  /00:00:00.32/00:00:01/00000/1
 /JAY01I
 15.20.32 JOB  149  IEF404I JAY01I - ENDED - TIME=15.20.32
 15.20.32 JOB  149  $HASP395 JAY01I   ENDED

     1      //JAY01I  JOB  1,IDCAMS,CLASS=A,MSGCLASS=X
   JOB  149
     2      //IDCAMS  EXEC PGM=IDCAMS,REGION=1024K
   00020000
     3      //SYSPRINT DD  SYSOUT=X
   00030000
 ***
```

The drawback of using the OUTPUT command in this way is that once viewed at the terminal, the output is no longer available on the JES2 queue to be viewed again or released for actual printing.

By specifying the PRINT operand with a dataset name as an argument, the background output is retrieved from the JES2 queue and written to the indicated dataset where it may be viewed repeatedly and, using some additional utility or TSO command, eventually be copied back to the JES2 queue for eventual disposition:

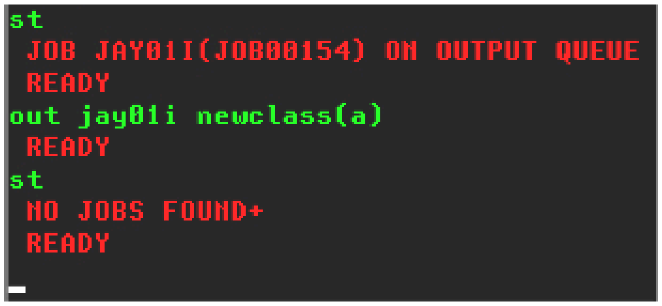


```
st
 JOB JAY01L(JOB00170) ON OUTPUT QUEUE
 READY
output jay01l print(jay01l)
 READY
dd jay01l.outlist
 DSN = JAY01.JAY01L.OUTLIST
 3350 - WORK02
 DSORG PS  RECFM FB  BLKSIZE 3,036  LRECL 132
 C-
 U-
 AL
 EX

 RE
```

```
 JAY01.JAY01L.OUTLIST on WORK02 --------------------------------
 Command ===> _
1       10        20        30        40        50        60
+----+----+----+----+----+----+----+----+----+----+----+----+---
(
.                                                J E S 2   J O
.16.20.00 JOB  170  $HASP373 JAY01L   STARTED - INIT  1 - CLASS
.16.20.00 JOB  170  IEF403I JAY01L - STARTED - TIME=16.20.00
.16.20.13 JOB  170  CCI001C LISTPDS /LISTPDS /00:00:06.90/00:00
.16.20.13 JOB  170  IEF404I JAY01L - ENDED - TIME=16.20.13
.16.20.13 JOB  170  $HASP395 JAY01L   ENDED
    1     //JAY01L   JOB (001),'LISTPDS',CLASS=A,MSGCLASS=X
    2     //LISTPDS EXEC PGM=LISTPDS,REGION=1024K,PARM='NOLIST'
    3     //SYSPRINT DD  SYSOUT=*
    4     //SYSLIST  DD  SYSOUT=*
    5     //SYSLIB   DD  DISP=SHR,DSN=SYS1.SAMPLIB
 IEF236I ALLOC. FOR JAY01L LISTPDS
 IEF237I JES2 ALLOCATED TO SYSPRINT
 IEF237I JES2 ALLOCATED TO SYSLIST
 IEF237I 150  ALLOCATED TO SYSLIB
 IEF142I JAY01L LISTPDS - STEP WAS EXECUTED - COND CODE 0000
 IEF285I   JES2.JOB00170.S00101                         SYSOUT
 IEF285I   JES2.JOB00170.S00102                         SYSOUT
 IEF285I   SYS1.SAMPLIB                                 KEPT
```

The two overlayed sessions above show the output for a batch job - JAY01L(JOB00170) - retrieved by the OUTPUT command to a sequential dataset - JAY01.JAY01L.OUTLIST - and the contents of the dataset being browsed using the REVIEW TSO Command. Note that unless the specified dataset is fully qualified, the OUTPUT command will prepend the TSO User's ID as the first qualifier and append 'OUTLIST' as the last qualifier to the value supplied in the PRINT operand in order to generate the full dataset name used to store the retrieved job output.

OUTPUT may also be used to change the class of the printed output so that it can be selected by a JES2 writer task for printing to a device attached to Hercules.



```
st
 JOB JAY01I(JOB00154) ON OUTPUT QUEUE
 READY
out jay01i newclass(a)
 READY
st
 NO JOBS FOUND+
 READY
```

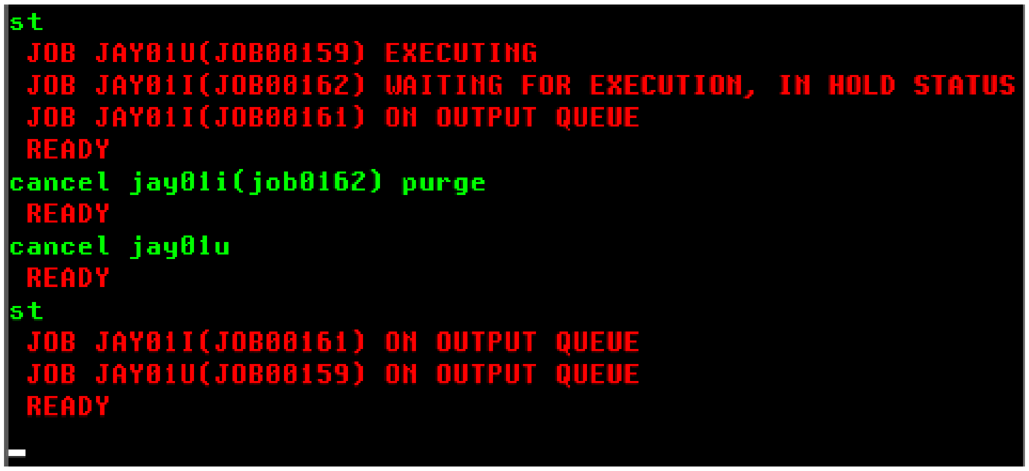
In my personal experience, there has almost always been some third party software available everywhere I have worked that was easier to use and provided more functionality than the OUTPUT command. If you are planning on viewing JES2 spooled output at your TSO session, I highly recommend that you investigate installing the QUEUE command or RPF. The specific purpose of the QUEUE command is viewing data contained in JES2 queues and RPF has a utility function that allows background job output to be viewed, deleted, and requeued for printing by JES2.

### Cancelling Background Jobs - CANCEL Command

The CANCEL command may be used to cancel background jobs, either while they are executing or in the JES2 input queue. The format of the command is:

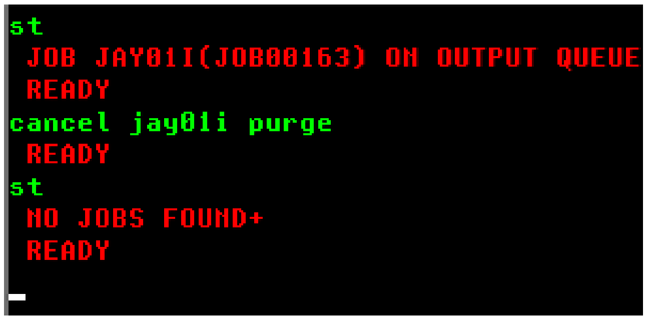
CANCEL <jobname>[<job id>] [ PURGE ]

The optional PURGE operand specifies that output produced by the job is to be deleted from the JES2 output queue; the default is to place any output generated prior to the execution of the CANCEL command on the JES2 output queue for processing as indicated by the JES2 parameters specified when the job was submitted for execution.



```
st
 JOB JAY01U(JOB00159) EXECUTING
 JOB JAY01I(JOB00162) WAITING FOR EXECUTION, IN HOLD STATUS
 JOB JAY01I(JOB00161) ON OUTPUT QUEUE
 READY
cancel jay01i(job0162) purge
 READY
cancel jay01u
 READY
st
 JOB JAY01I(JOB00161) ON OUTPUT QUEUE
 JOB JAY01U(JOB00159) ON OUTPUT QUEUE
 READY
```

If there are duplicate jobs with the same job name, the JES2 job ID may be used to qualify the target of the CANCEL command, as shown in the session above. The cancel command with the PURGE option may also be used to remove output from the JES2 queue:



```
st
 JOB JAY01I(JOB00163) ON OUTPUT QUEUE
 READY
cancel jay01i purge
 READY
st
 NO JOBS FOUND+
 READY
```

## Communication With Other Users and Operators

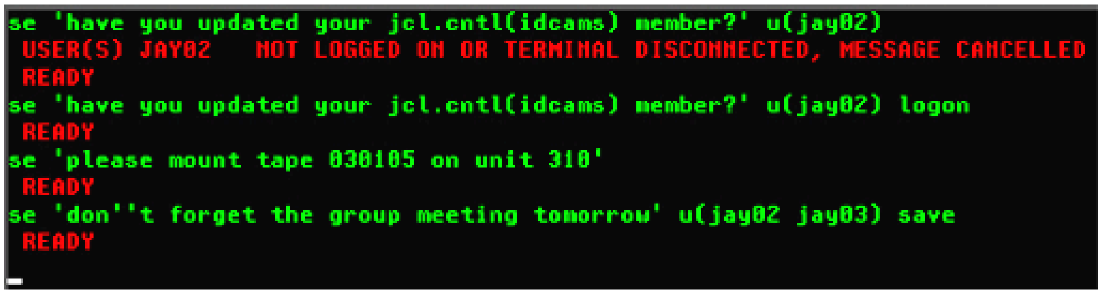
The SEND command (which may be abbreviated SE) is used to send text messages from a TSO User to one or more other TSO Users or the MVS operator's console. The syntax of the command is:

```
SEnd '<text of message>'
        { USER(<userid list>) | CN(<console id>) }
        { NOW | LOGON | SAVE }
```

The <text of message> is any string of characters to be sent as a message to the designated recipient. The text must be enclosed in apostrophes; if the message text contains an apostrophe, it must be entered as two consecutive apostrophes.

The USER or CN operands designate the recipient of the message. USER (which may be abbreviated U) may specify one or more TSO User IDs to which the message text is to be sent. Alternately, CN may be used to specify the operator console ID to which the message text is to be sent. If neither USER or CN is specified with the SEND command, the text will be displayed on the main operator's console - CN(00).

By default, the message is displayed at the recipients TSO session (or operator's console) as soon as ENTER is pressed (the NOW operand, which is the default if none is specified). The LOGON operand may be used to designate that the message is to be held for a TSO User that is not currently logged on and the message will be displayed the next time they log onto TSO. If the LOGON operand is specified and the target User ID is currently logged on, the message is delivered immediately, the same effect as if NOW had been specified. SAVE may be used to designate that the message is not to be displayed at the recipient's terminal, even though they are currently logged on, but saved until their next log on, at which time it will be displayed.



```
se 'have you updated your jcl.cntl(idcams) member?' u(jay02)
 USER(S) JAY02   NOT LOGGED ON OR TERMINAL DISCONNECTED, MESSAGE CANCELLED
 READY
se 'have you updated your jcl.cntl(idcams) member?' u(jay02) logon
 READY
se 'please mount tape 030105 on unit 310'
 READY
se 'don''t forget the group meeting tomorrow' u(jay02 jay03) save
 READY
```

## Command Lists

TSO provides the capability for executing sequences of stored basic commands. This eliminates the redundancy of repeatedly entering sequences of commands that are frequently executed. Using this facility, very sophisticated stored procedures may be created, used, and shared with others. A sequence of stored TSO commands is referred to as a Command List (or CLIST). In addition to the usual TSO commands that may be entered at the TSO READY prompt, a number of specialized commands that are available for use only from within a CLIST may be utilized to modify the operation of these stored procedures. Variables may be created and used to control the sequence and actions of the procedure. Specialized system variables and functions to manipulate the contents of user variables are also available from within a CLIST.

CLIST statements assign values, set controls, select options, and control the conditions under which CLISTs execute. Statements operate in both the command and subcommand environment, either by the EXEC command or by the EXEC subcommand of EDIT. In general, statements fall into the categories of control, assignment, conditional, and file access:

Control

Control

- ATTN
- CONTROL
- DATA - ENDDATA
- ERROR
- EXIT
- GLOBAL
- GOTO
- PROC
- RETURN
- TERMIN
- WRITE / WRITENR

Assignment

- READ
- READDVAL
- SET

Conditional

- DO - WHILE - END
- IF - THEN - ELSE

File Access

- CLOSFILE
- GETFILE
- OPENFILE
- PUTFILE

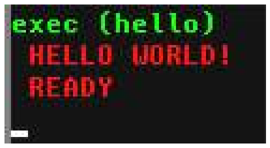
A Command List may be stored in either a sequential dataset or a member of a partitioned dataset. The contents of a CLIST dataset are created/modified with any text editor and are not processed by a compilation process; ie the lines of a CLIST are ***interpreted*** each time they are executed. Although the records in a CLIST dataset may be variable length, most of the editors we are using in the Hercules' community are restricted in some way when editing variable length datasets. Also, the dataset allocated during System Generation to contain system CLISTs - SYS1.CMDPROC - is allocated to contain fixed length, 80 byte records. Therefore, it would be my recommendation to allocate a partitioned dataset to contain the CLISTs that you write using the characteristics: DSORG=PO, RECFM=FB, LRECL=80, BLKSIZE=3120, and DSN=<your TSO ID>.CLIST. The following commands may be used to accomplish that task:

```
ATTRIB CATTR DSORG(PO) RECFM(F B) LRECL(80) BLKSIZE(3120)
ALLOC F(CLIST) USING(CATTR) +
      DA('<your TSO user id>.CLIST') +
      UNIT(SYSDA) VOL(<volume serial>) +
      SPACE(15 15) TRACKS DIR(20)
FREE F(CLIST)
```

substituting your TSO User ID and a valid DASD Volume Serial number in the appropriate places. Collecting all of your CLISTs in a single partitioned dataset will keep them organized and will allow you to execute them easily using only the member name.

It will probably be helpful if you think of Command Lists as a simple programming language. Before launching into details of Command List structure and syntax of the additional commands available for use in CLISTs, here is an example CLIST that implements the usual "Hello World" logic:

```
PROC 0                  /* NO PARAMETERS REQUIRED */
WRITE HELLO WORLD!      /* DISPLAY CONSTANT AT TERMINAL */
EXIT                    /* RETURN TO TSO READY PROMPT */
```



```
exec (hello)
 HELLO WORLD!
 READY
```

### Executing Command Lists - EXEC Command

The TSO command used to invoke a Command List is EXEC (which may be abbreviated EX). The syntax of the command is:

EXec <dataset name> [ <parameters> ] [ LIST | NOLIST ] [ PROMPT | NOPROMPT ]

The <dataset name> specifies the dataset or member that contains the stored Command List. If the CLIST is a member of a partitioned dataset which is named '<your TSO User ID>'.CLIST, you can execute the contents of the member by specifying the member name enclosed in parentheses (as shown in the session above).

Two types of parameters may be supplied to the CLIST. Positional parameters, if used, must follow the <dataset name> operand. Keyword parameters, if used, may be specified following the last positional parameter. The use of the two types of parameters is explained in more detail below.

If the LIST operand is specified, TSO commands contained in the CLIST are displayed on the terminal before they are executed.

If the PROMPT operand is specified, the operator is prompted to supply missing information for TSO commands contained in the CLIST.

#### Implicit Execution of Command Lists

Because the name of a Command List is identical in format to the name of standard TSO commands, TSO will first search its own command libraries for a command matching the name specified on the EXEC statement. Since you know when you are executing a Command List, you can eliminate the time to perform this search by using the implicit form of Command List execution:

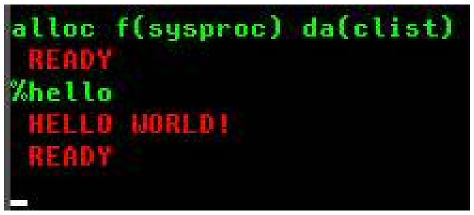
%<Command List name> [ <parameters> ] [ LIST | NOLIST ] [ PROMPT | NOPROMPT ]

%<Command List name> [ <parameters> ] [ LIST | NOLIST ] [ PROMPT | NOPROMPT ]

Prior to using the implicit form, you must allocate the reserved DD Name ***SYSPROC*** to point to the partitioned dataset containing your Command Lists:

```
ALLOC F(SYSPROC) DA(<your TSO User id>.CLIST)
```

As long as the allocation for SYSPROC is not freed, when the implicit form of EXEC is used, TSO will search only the library allocated to the SYSPROC dataset for the specified member name:



```
alloc f(sysproc) da(clist)
 READY
%hello
 HELLO WORLD!
 READY
```

## Comments in Command Lists

For simple CLISTs, there may be no need to include any comments to document what the commands on a particular line, or group of lines, is intended to accomplish. But for more complex CLISTs, you should consider using comments as documentation for you, or someone else, if they later need to make modifications to the CLIST or just to help understand how to use it.

Comments begin with a forward slash followed immediately by an asterisk (/*) and are terminated with an asterisk followed immediately by a forward slash (*/). Comments may be continued on multiple lines by following the same rules for continuation of regular TSO commands - terminate the line to be continued with either a minus (-) or plus (+) sign and continue the comment on the next line of the CLIST.

Some examples of comments:

```
/* THIS ENTIRE LINE IS A COMMENT - THERE IS NO EXECUTABLE COMMAND */

CLS  /* LINES WITH EXECUTABLE COMMANDS MAY INCLUDE SHORT COMMENTS */

/* VERY LONG COMMENTS MAY BE CONTINUED ACROSS MULTIPLE LINES BY     +
   UTILIZING THE SAME CONTINUATION RULES ALLOWED FOR ANY OTHER      +
   TSO COMMAND                                                     */

/* PROBABLY A BETTER WAY TO INCLUDE COMMENTS THAT SPAN MULTIPLE    */
/* LINES WOULD BE TO MAKE EACH LINE A WHOLLY CONTAINED COMMENT     */
```

## Defining Variables with the Procedure Statement

A procedure statement defines parameters and relates them to symbolic variables used in the Command List statements where the values passed through the parameters are to be substituted. The function of the keyword operands on the procedure statement in a CLIST is identical to the dummy variables contained on the PROC statement in a procedure for Job Control Language. The syntax of the procedure statement is:

PROC <count> [ <positional parameters> ] [ <keyword parameters>(<value>) ]

<count> is the number of positional parameters that are coded following the <count> value. If there are no positional parameters, <count> must be zero.

<positional parameters> are the names of variables that will receive values typed when the CLIST is executed, without the ampersand (&).

<keyword parameters> are the names of variables that will be specified by key words, without the ampersand (&). For each <keyword parameter> specified, <value> provides the default value to be given to the parameter if none is specified when the CLIST is executed. Null values are coded as empty parentheses.

Some examples:

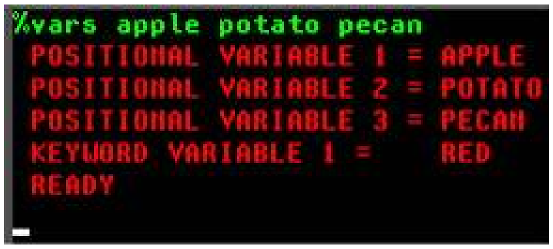
```
PROC 0        /* NO PARAMETERS REQUIRED OR EXPECTED */

PROC 1 CVALUE /* ONE VALUE MUST BE SUPPLIED AT EXEC */

PROC 1 MEMBER LIBRARY(SYS1.PROCLIB) /* MEMBER MUST BE SPECIFIED */
                                    /* LIBRARY MAY BE SPECIFIED, BUT */
                                    /* WILL BE GIVEN THE DEFAULT     */
                                    /* VALUE 'SYS1.PROCLIB' IF NOT   */

PROC 0 FROMLIB() TOLIB() /* NO POSITIONAL PARAMETERS, BUT TWO */
                         /* OPTIONAL KEYWORD PARAMETERS WITH  */
                         /* NO DEFAULT VALUES SUPPLIED        */
```

**Variables in Command Lists**

The real power of Command Lists is the ability to use symbolic variables when writing the operands for TSO commands. When the CLIST is executed, the values contained in the variables are substituted for the symbolic variable name wherever that name is coded in the statements coded in the CLIST.

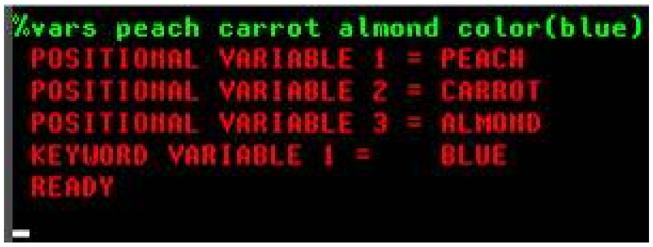
The name for a variable used in a CLIST may consist of up to 31 characters, alphabetic or numbers, beginning with a letter. When used in a command, the name of the variable is preceded by an ampersand (&), which indicates to the interpreter that the value contained in the variable whose name follows the ampersand is to be substituted for the variable name before the interpretation of the line continues. Here is a simple illustration of variable substitution:

```
PROC 3 FRUIT VEGETABLE NUT COLOR(RED)
WRITE POSITIONAL VARIABLE 1 = &FRUIT
WRITE POSITIONAL VARIABLE 2 = &VEGETABLE
WRITE POSITIONAL VARIABLE 3 = &NUT
WRITE KEYWORD VARIABLE 1 =    &COLOR
EXIT
```



```
%vars apple potato pecan
 POSITIONAL VARIABLE 1 = APPLE
 POSITIONAL VARIABLE 2 = POTATO
 POSITIONAL VARIABLE 3 = PECAN
 KEYWORD VARIABLE 1 =    RED
 READY
```

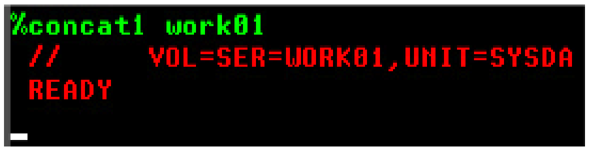
At execution time, the values specified for the three positional variables are assigned to the variable names included on the PROC statement as positional parameters - FRUIT, VEGETABLE, NUT. The value of the keyword variable is assigned from the default value coded on the PROC statement. When the WRITE statements are interpreted prior to execution, the values of each variable is substituted for the symbolic form of the variable name coded in the CLIST line. Each time a CLIST is executed, different values may be specified for each variable, including those designated as ***keyword*** variables:



```
%vars peach carrot almond color(blue)
 POSITIONAL VARIABLE 1 = PEACH
 POSITIONAL VARIABLE 2 = CARROT
 POSITIONAL VARIABLE 3 = ALMOND
 KEYWORD VARIABLE 1 =    BLUE
 READY
```

Two special circumstances arise during variable substitution. The first involves the rules the interpreter follows for concatenating the value of a variable with other characters, or another variable, in the CLIST line. As long as the variable name is followed by a blank or special character, simply writing the variable name adjacent to the following text is acceptable:

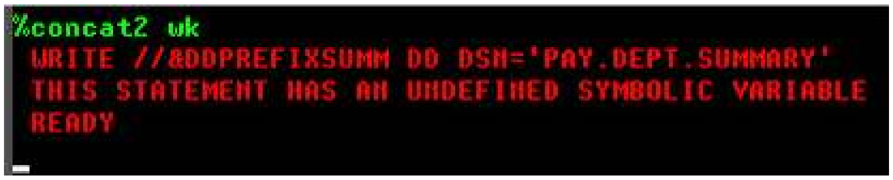
```
PROC 1 SERIAL UNIT(SYSDA)
WRITE //   VOL=SER=&SERIAL,UNIT=&UNIT
EXIT
```



```
%concat1 work01
 //     VOL=SER=WORK01,UNIT=SYSDA
 READY
```

However, if the character following the variable name is also alphabetic or numeric, the interpreter will fail to recognize the variable name correctly:

```
PROC 1 DDPREFIX
WRITE //&DDPREFIXSUMM DD DSN='PAY.DEPT.SUMMARY'
EXIT
```



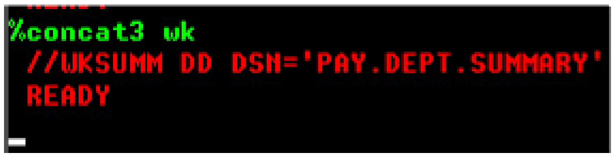
```
%concat2 wk
 WRITE //&DDPREFIXSUMM DD DSN='PAY.DEPT.SUMMARY'
 THIS STATEMENT HAS AN UNDEFINED SYMBOLIC VARIABLE
 READY
```

An error message is issued because the interpreter doesn't recognize that what is intended is that the value

An error message is issued because the interpreter doesn't recognize that what is intended is that the value contained in the symbolic variable **&DDPREFIX** is to be concatenated with the letters **SUMM** that are

following the variable name. The interpreter incorrectly tries to locate a variable named **&DDPREFIXSUMM** that doesn't exist. This situation is resolved by using a period (.) as a delimiter between symbolic variable names and the following text which is to be concatenated:

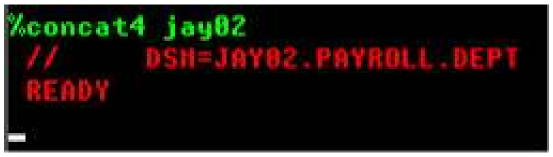
```
PROC 1 DDPREFIX
WRITE //&DDPREFIX.SUMM DD DSN='PAY.DEPT.SUMMARY'
EXIT
```



```
%concat3 wk
 //WKSUMM DD DSN='PAY.DEPT.SUMMARY'
 READY
```

However, allowing the period for a delimiter causes an exception to the rule of only requiring the period as a delimiter when the following character is alphabetic or numeric. Because the delimiting period is removed by the interpreter, if the character following the delimiter is a period, in order to retain a period in the output after interpretation, you must include a second period:

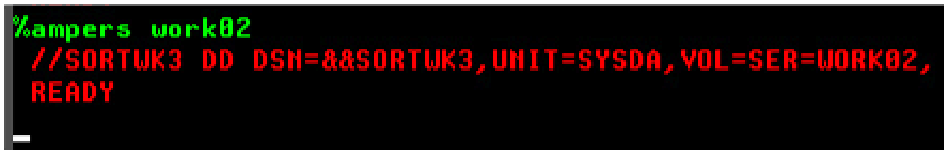
```
PROC 1 USER PROJECT(PAYROLL)
WRITE //     DSN=&USER..&PROJECT..DEPT'
EXIT
```



```
%concat4 jay02
 //     DSN=JAY02.PAYROLL.DEPT
 READY
```

The second circumstance that arises as a result of the rules governing symbolic variable interpretation is that any place in a CLIST that you need to code an ampersand (&), such as the specification of a temporary dataset name, you must code two consecutive ampersands:

```
PROC 1 SERIAL UNIT(SYSDA)
WRITE '//SORTWK3 DD DSN=&&&&SORTWK3,UNIT=&UNIT,VOL=SER=&SERIAL,'
EXIT
```



```
%ampers work02
 //SORTWK3 DD DSN=&&SORTWK3,UNIT=SYSDA,VOL=SER=WORK02,
 READY
```

## Controlling the Command List Environment - CONTROL Statement

There are a number of settings/options that control how a Command List execute. These may be modified by using the CONTROL statement. The syntax of the statement is:

CONTROL [ FLUSH | NOFLUSH ]

```
[ PROMPT | NOPROMPT ]
[ LIST | NOLIST ]
[ CONLIST | NOCONLIST ]
[ SYMLIST | NOSYMLIST ]
[ MSG | NOMSG ]
[ MAIN ]
[ END(<string>) ]
```

The settings specified by a CONTROL statement remain in effect for the duration of the CLIST, or until another CONTROL statement is executed. You might have a control statement at the beginning of your CLIST to provide the settings you wish to have in effect for the majority of your CLIST, but have additional CONTROL statements before and after a particular segment of your CLIST to provide an alternate option to be in effect for just that segment.

The FLUSH setting causes immediate termination of the CLIST if an error occurs. If NOFLUSH is set, execution will continue after an error message is issued. This can allow some amount of post error processing under the control of your CLIST.

The PROMPT setting causes the operator to be prompted to supply any required operands for TSO commands if the operands are missing at the time of the command's interpretation.

The LIST setting causes TSO commands to be displayed before execution but after the substitution of symbolic variables.

The CONLIST setting causes command procedure statements to be displayed before execution but after the substitution of symbolic variables.

The SYMLIST setting causes TSO and command procedure statements to be displayed before any symbolic substitution takes place.

The MSG setting causes messages generated by TSO command execution to be displayed.

The MAIN setting prevents the CLIST from being terminated in the case of an error or an attention interruption. If MAIN is specified, NOFLUSH is also assumed.

The END operand allows an alternate character string to be specified as the terminator for a DO group.

## Displaying Messages - WRITE Statement

You have seen the WRITE statement used in my examples above. The WRITE statement is used to display output as the Command List is executed. The syntax of the statement is:

```
[<label>:] WRITE[NR] <text>
```

If present, the <label> allows the statement to be the target of a branch from another part of the CLIST. Any <text> following the command is displayed. The text displayed can be used for messages, information, prompting, debugging, or whatever the writer of the command procedure desires.

If **WRITENR** is specified, as opposed to **WRITE**, the cursor position for the next output will immediately follow the last character of text displayed. When **WRITE** is specified, a carriage return/line feed are automatically issued following the text displayed as a result of the WRITE statement.

### Assigning Values to Variables - SET Statement

The SET statement is used to assign a value to a variable. The syntax of the statement is:

[<label>:] SET <variable name> { EQ | = } <expression>

If present, the <label> allows the statement to be the target of a branch from another part of the CLIST. The variable name need not have been defined already. The variable to be set cannot be a built-in function. It is not necessary for the <variable name> to include an ampersand (&); the interpreter will recognize it as a variable by its placement on the left side of the equal sign. The <expression> may consist of:

- a null value: SET MYVAR =
- a simple numeric value: SET COUNT = 1
- a non-numeric value: SET DSN = MY.TEST.DATASET
- the contents of another variable: SET OLDUNIT = &UNIT
- a concatenated value: SET CVAR = &DSN1..&DSN2..DATA
- a built-in function: SET DSNLEN = &LENGTH(&DSN)
- an arithmetic result: SET RESULT = &COUNT + 1

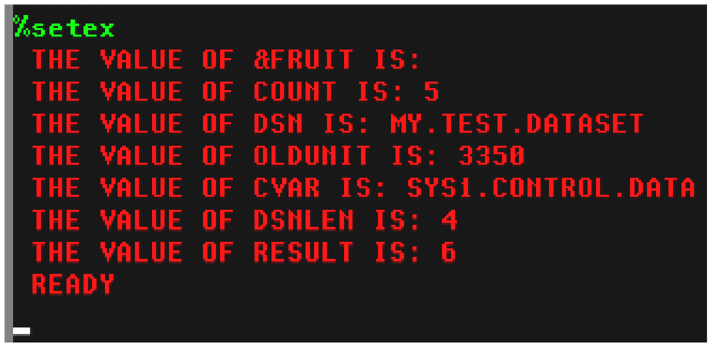
The arithmetic operators that may be used in the SET statement are:

| **Operator** | **Function** |
|---|---|
| + | Addition |
| - | Subtraction |
| * | Multiplication |
| / | Division |
| ** | Exponentiation |
| // | Remainder |

A CLIST illustrating these SET statements:

```
PROC 0 FRUIT(APPLE) UNIT(3350) DSN1(SYS1) DSN2(CONTROL)
SET FRUIT =                    /* NULL VALUE */
SET COUNT = 5                  /* NUMERIC VALUE */
SET DSN = MY.TEST.DATASET      /* NON-NUMERIC VALUE */
SET OLDUNIT = &UNIT            /* VARIABLE VALUE */
SET CVAR = &DSN1..&DSN2..DATA  /* CONCATENATED VALUE */
SET DSNLEN = &LENGTH(CVAR)     /* BUILT-IN FUNCTION */
SET RESULT = &COUNT + 1        /* ARITHMETIC RESULT */
```

```
WRITE THE VALUE OF &&FRUIT IS: &FRUIT
WRITE THE VALUE OF COUNT IS: &COUNT

WRITE THE VALUE OF DSN IS: &DSN
WRITE THE VALUE OF OLDUNIT IS: &OLDUNIT
WRITE THE VALUE OF CVAR IS: &CVAR
WRITE THE VALUE OF DSNLEN IS: &DSNLEN
WRITE THE VALUE OF RESULT IS: &RESULT
EXIT
```



```
%setex
 THE VALUE OF &FRUIT IS:
 THE VALUE OF COUNT IS: 5
 THE VALUE OF DSN IS: MY.TEST.DATASET
 THE VALUE OF OLDUNIT IS: 3350
 THE VALUE OF CVAR IS: SYS1.CONTROL.DATA
 THE VALUE OF DSNLEN IS: 4
 THE VALUE OF RESULT IS: 6
 READY
```

## Built- In Functions

A set of built-in functions is provided to perform evaluations of data expressions, character strings, and substrings. To use a built-in function, include the appropriate symbolic variable on a CLIST statement. The interpreter will substitute data for the symbolic variable at execution time.

&DATATYPE(<expression>)

The &DATATYPE function is used to determine whether an expression is entirely numeric. The interpreter will return the string **NUM** if the <expression> is all numeric or **CHAR** for anything else.

&EVAL(<expression>)

The &EVAL function is used to find the result of an arithmetic expression. The interpreter will return the value calculated as the numeric result of <expression>.

&LENGTH(<expression>)

The &LENGTH function is used to find the number of characters that <expression> occupy.

&STR(<string>)

The &STR function is used to suppress the evaluation of the <string> specified, however symbolic substitution does take place.

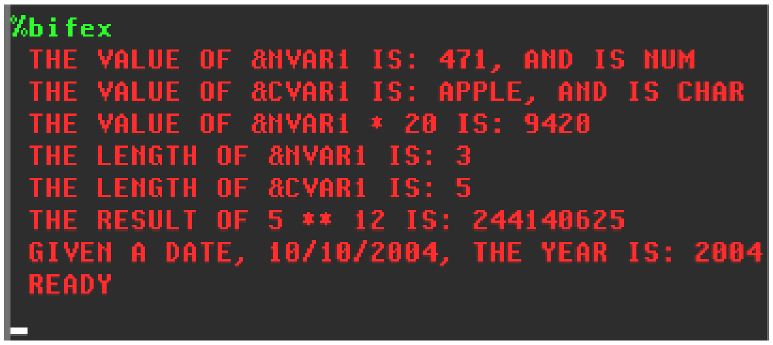
&SUBSTR(<starting position>[:<ending position>],<string>)

The &SUBSTR function is used to select a segment from a character string. The <starting

position> specifies the first character to be selected from <string>. The <ending position> specifies the last character to be selected. If omitted, <ending position> will default to the same value as <starting position>, resulting in a single character selected from <string>.

A CLIST illustrating the Built-In functions:

```
PROC 0
SET NVAR1 = 471
SET CVAR1 = APPLE
SET DVAR1 = &STR(10/10/2004)
WRITE THE VALUE OF &&NVAR1 IS: &NVAR1, AND IS &DATATYPE(&NVAR1)
WRITE THE VALUE OF &&CVAR1 IS: &CVAR1, AND IS &DATATYPE(&CVAR1)
WRITE THE VALUE OF &&NVAR1 * 20 IS: &EVAL(&NVAR1 * 20)
WRITE THE LENGTH OF &&NVAR1 IS: &LENGTH(&NVAR1)
WRITE THE LENGTH OF &&CVAR1 IS: &LENGTH(&CVAR1)
WRITE THE RESULT OF &STR(5 ** 12) IS: &EVAL(5 ** 12)
WRITE GIVEN A DATE, &DVAR1, THE YEAR IS: &SUBSTR(7:10,&DVAR1)
EXIT
```



```
%bifex
 THE VALUE OF &NVAR1 IS: 471, AND IS NUM
 THE VALUE OF &CVAR1 IS: APPLE, AND IS CHAR
 THE VALUE OF &NVAR1 * 20 IS: 9420
 THE LENGTH OF &NVAR1 IS: 3
 THE LENGTH OF &CVAR1 IS: 5
 THE RESULT OF 5 ** 12 IS: 244140625
 GIVEN A DATE, 10/10/2004, THE YEAR IS: 2004
 READY
```

## Control Variables

TSO maintains a set of variables whose values contain information about the current command procedure environment and the TSO user. The values of four of the variables may be set by CLIST statements - &LASTCC, &MAXCC, &SYSDVAL, and &SYSSCAN. Attempts to modify the other control variables will produce an error.

&LASTCC

Use &LASTCC to examine the return code from the last operation, whether TSO command/subcommand or statement. Values that may be returned by CLIST statements are:

| | |
|---|---|
| 16 Not enough virtual storage | 836 (reserved) |
| 300 User tried to update an unauthorized variable | 840 Not enough operands |
| 304 Invalid keyword on EXIT statement | 844 No valid operators |
| 308 Code specified, but no code given on EXIT statement | 848 Attempt to load character from numeric value |

| | | | |
|---|---|---|---|
| 312 | Internal GLOBAL processing error | 852 | Addition error - character data |
| 316 | TERMIN delimiter greater than 256 characters | 856 | Subtraction error - character data |
| 324 | GETLINE error | 860 | Multiplication error - character data |
| 328 | More than 640 delimiters on TERMIN | 864 | Divide error - character data |
| 332 | Invalid file name syntax | 868 | Prefix found on character data |
| 336 | File already open | 872 | Numeric value too large |
| 340 | Invalid OPEN type syntax | 900 | Single ampersand found |
| 344 | Undefined OPEN type | 904 | Symbolic variable not found |
| 348 | File specified did not open (for example, the filename was not allocated) | 908 | Error occurred in an error action range that received control because of another error |
| 352 | GETFILE - filename not currently open | 912 | Substring range invalid |
| 356 | GETFILE - the file has been closed by the system (for example, file opened under EDIT and EDIT has ended) | 916 | Non-numeric value in substring range |
| 360 | PUTFILE - file name not currently open | 920 | Substring range value too small (zero or negative) |
| 364 | PUTFILE - file closed by system (see code 356) | 924 | Invalid substring syntax |
| 368 | PUTFILE - CLOSFILE - file not opened by OPENFILE | 932 | Substring outside of the range of the string (for example, 1:3,AB - AB is only two characters) |
| 372 | PTFILE - issued before GETFILE on a file opened for update | 936 | A built-in function that requires a value was entered without a value |
| 400 | GETFILE end of file (treated as an error, which can be handled by ERROR action) | 940 | Invalid symbolic variable |
| 8xx | Evaluation routine error codes | 944 | A label was used as a symbolic variable |
| 800 | Data found where operator was expected | 948 | Invalid label syntax on a GOTO statement |
| 804 | Operator found where data was expected | 952 | GOTO label was not defined |
| 808 | A comparison operator was used in a SET statement | 956 | GOTO statement has no label |
| 812 | (reserved) | 960 | &SYSSCAN was set to an invalid value |
| 816 | Operator found at the end of | 964 | &LASTCC was set to an invalid |

| | | | |
|---|---|---|---|
| 816 | Operator found at the end of a statement | 964 | &LASTCC was set to an invalid value and EXIT tried to use it as a default value |
| 820 | Operators out of order | 999 | Internal command procedure error |
| 824 | More than one exclusive operator found | Sxxx | A system ABEND code |
| 828 | More than one exclusive comparison operator | Uxxx | A user ABEND code |

&MAXCC

Use &MAXCC to examine the highest return code issued up to now in this CLIST, or passed back from any nested procedures executed.

&SYSDATE

Use &SYSDATE to get the present date in the format mm/dd/yy.

&SYSDLM

Use &SYSDLM to identify which delimiter string of those specified by the TERMIN statement the user entered to give up control. This allows a user to select options that a CLIST may provide.

&SYSDVAL

Use &SYSDVAL for one of three purposes:

- When TERMIN passes control to the terminal user, get the value of any parameters the user enters besides the delimiter string entered to pass control back to the CLIST.
- When READ requests specific terminal input, get the value of the user response line.
- Parse records read from datasets into fields

&SYSICMD

Use &SYSICMD to get the name by which the user implicitly invoked the currently executing CLIST. The name value will be null if the user invoked this CLIST explicitly.

&SYSNEST

User &SYSNEST to get character strings YES or NO, signifying whether the currently executing CLIST is nested (invoked from another CLIST instead of from the terminal directly).

&SYSPCMD

Use &SYSPCMD to get the name of the most recently executed TSO command in this CLIST. The initial value is "EXEC" in the command environment and "EDIT" in the subcommand environment.

&SYSPREF

Use &SYSPREF to get the dataset name prefix specified in the user profile table of the CLIST user.

&SYSPROC

Use &SYSPROC to get the logon procedure name for the current CLIST user.

&SYSSCAN

Use &SYSSCAN to get the maximum value for the number of times that symbolic substitution is allowed to rescan a line to resolve symbolic variables.  The default value is 16; minimum value is 0; maximum value is the largest fixed-point number.

&SYSSCMD

Use &SYSSCMD to get the name of the subcommand currently executing.  The iniital value is null if EXEC is issued in the command environment and "EXEC" if EXEC is issued in the subcommand environment (under EDIT).  The value is null whenever the procedure is in the command environment.

&SYSTIME

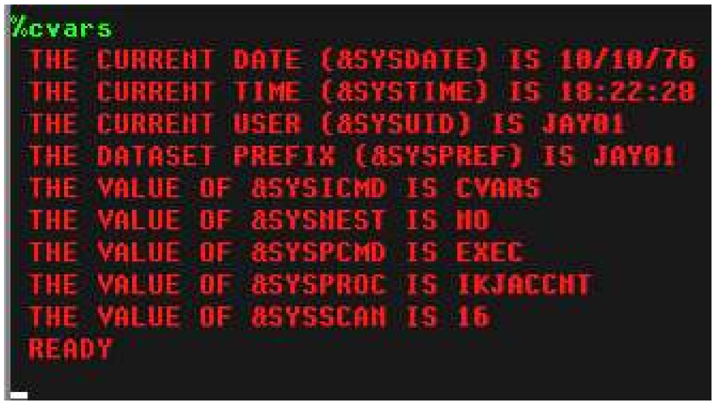
Use &SYSTIME to get the present time in the format hh:mm:ss.

&SYSUID

Use &SYSUID to get the TSO User ID of the individual currently executing the CLIST.

A CLIST illustrating some of the Control variables:

```
PROC 0
WRITE THE CURRENT DATE (&&SYSDATE) IS &SYSDATE
WRITE THE CURRENT TIME (&&SYSTIME) IS &SYSTIME
WRITE THE CURRENT USER (&&SYSUID) IS &SYSUID
WRITE THE DATASET PREFIX (&&SYSPREF) IS &SYSPREF
WRITE THE VALUE OF &&SYSICMD IS &SYSICMD
WRITE THE VALUE OF &&SYSNEST IS &SYSNEST
WRITE THE VALUE OF &&SYSPCMD IS &SYSPCMD
WRITE THE VALUE OF &&SYSPROC IS &SYSPROC
WRITE THE VALUE OF &&SYSSCAN IS &SYSSCAN
EXIT
```



```
%cvars
 THE CURRENT DATE (&SYSDATE) IS 10/10/76
 THE CURRENT TIME (&SYSTIME) IS 18:22:28
 THE CURRENT USER (&SYSUID) IS JAY01
 THE DATASET PREFIX (&SYSPREF) IS JAY01
 THE VALUE OF &SYSICMD IS CVARS
 THE VALUE OF &SYSNEST IS NO
 THE VALUE OF &SYSPCMD IS EXEC
 THE VALUE OF &SYSPROC IS IKJACCNT
 THE VALUE OF &SYSSCAN IS 16
 READY
```

## Controlling Executing Sequence

The sequence in which statements are executed in a Command List may be influenced by a couple of statements. The natural sequence of execution is to begin with the first statement contained in the CLIST and proceed one by one with each consecutive statement until the end of the CLIST is reached or an EXIT statement is encountered, terminating execution.

The value of variables may be examined using the IF statement and, based upon the outcome of the condition tested, a GOTO statement may be used to transfer control from the current statement in the CLIST to any other statement that contains a label. Current programming practice tends to frown upon this type of branching, but it is still a valid means of controlling execution sequence in a Command List and there are circumstances in which that is the only method available to implement the desired solution.

The use of a GOTO statement without a corresponding condition test made with an IF statement results in a unconditional branch.

A more structured approach to constructing execution loops is provided by a DO-WHILE-END sequence.

It is also possible to conditionally execute individual commands using the WHEN statement.

## Evaluating Variable Values - IF-THEN-ELSE Statements

The IF-THEN-ELSE structure is used to test the contents of variables during execution and is similar to equivalent statements in most high level languages. The comparative expression in an IF statement allows the comparison of any combination of literal values and symbolic variables. The syntax of the statements is:

```
[<label>:]  IF <expression> THEN <statement>
ELSE <statement>
```

If present, the <label> allows the statement to be the target of a branch from another part of the CLIST. The

<expression> is of the format:

{ literal | symbolic variable } <comparison operator> { literal | symbolic variable }

The comparison operators which are valid are:

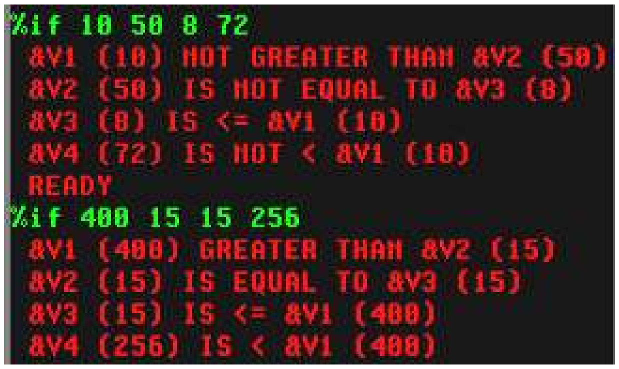
| **Comparison Performed** | **Symbol Characters** | |
|---|---|---|
| Equal | = | EQ |
| Not Equal | ¬= | NE |
| Less Than | < | LT |
| Greater Than | > | GT |
| Less Than or Equal | <= | LE |
| Greater Than or Equal | >= | GE |
| Not Greater Than | ¬> | NG |
| Not Less Than | ¬< | NL |

Complex comparisons may be constructed by joining simple comparisons with logical operators:

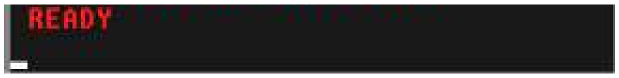
| **Connection** | **Symbol Characters** | |
|---|---|---|
| And | && | AND |
| Or | \|\| | OR |

An example CLIST showing some simple IF-THEN statements:

```
PROC 4 V1 V2 V3 V4

IF &V1 GT &V2 THEN WRITE &&V1 (&V1) GREATER THAN &&V2 (&V2)
ELSE WRITE &&V1 (&V1) NOT GREATER THAN &&V2 (&V2)

IF &V2 EQ &V3 THEN WRITE &&V2 (&V2) IS EQUAL TO &&V3 (&V3)
ELSE WRITE &&V2 (&V2) IS NOT EQUAL TO &&V3 (&V3)

IF &V3 LE &V1 THEN WRITE &&V3 (&V3) IS <= &&V1 (&V1)
ELSE WRITE &&V3 (&V3) IS NOT <= &&V1 (&V1)

IF &V4 LT &V1 THEN WRITE &&V4 (&V4) IS < &&V1 (&V1)
ELSE WRITE &&V4 (&V4) IS NOT < &&V1 (&V1)

EXIT
```



```
%if 10 50 8 72
 &V1 (10) NOT GREATER THAN &V2 (50)
 &V2 (50) IS NOT EQUAL TO &V3 (8)
 &V3 (8) IS <= &V1 (10)
 &V4 (72) IS NOT < &V1 (10)
 READY
%if 400 15 15 256
 &V1 (400) GREATER THAN &V2 (15)
 &V2 (15) IS EQUAL TO &V3 (15)
 &V3 (15) IS <= &V1 (400)
 &V4 (256) IS < &V1 (400)
```



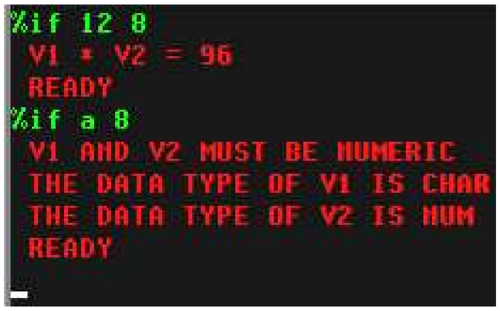
```
READY
```

Some coding rules that must be followed when writing the IF-THEN-ELSE structure are:

- The word THEN must appear on the same logical line as the word IF. Of course, the **logical** line may consist of several physical lines that have been continued using the plus (+) or minus (-) continuation characters.
- You may code only one statement to be executed if the expression is true - following the word THEN - and it must also be contained on the same logical line as the word IF.
- Likewise, you may code only one statement to be executed if the express is false, and it must be coded on the same logical like as the word ELSE. Again, continuation characters may be used to mediate this limitation.

Of course, it is most likely that you would need to execute more than one statement if the condition were true - THEN - or false - ELSE - in which case you can enclose multiple statements with a DO-END pair:

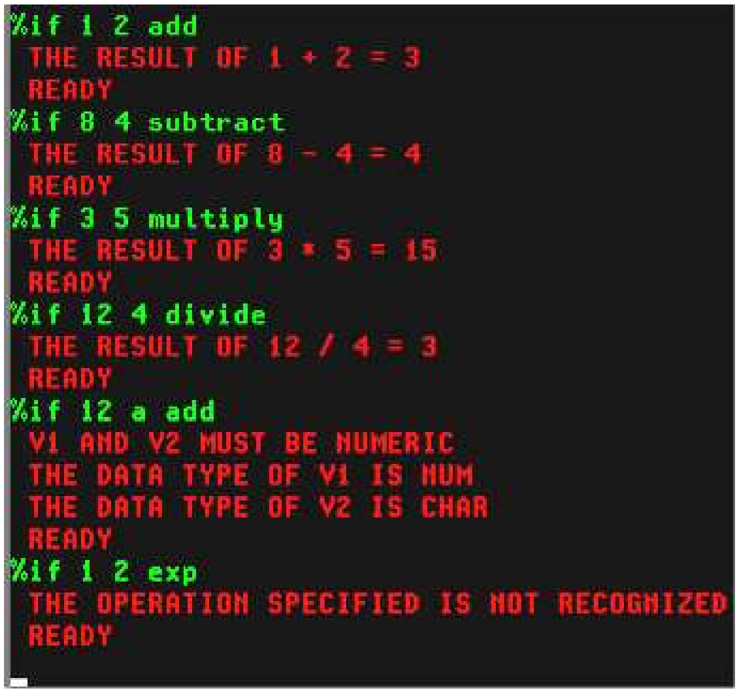
```
PROC 2 V1 V2

IF &DATATYPE(&V1) = NUM && &DATATYPE(&V2) = NUM THEN DO
   SET RESULT = &EVAL(&V1 * &V2)
   WRITE V1 * V2 = &RESULT
   END
ELSE DO
   WRITE V1 AND V2 MUST BE NUMERIC
   WRITE THE DATA TYPE OF V1 IS &DATATYPE(&V1)
   WRITE THE DATA TYPE OF V2 IS &DATATYPE(&V2)
   END

EXIT
```



```
%if 12 8
 V1 * V2 = 96
 READY
%if a 8
 V1 AND V2 MUST BE NUMERIC
 THE DATA TYPE OF V1 IS CHAR
 THE DATA TYPE OF V2 IS NUM
 READY
```

It is also possible to nest IF-THEN-ELSE structures:

```
PROC 3 V1 V2 OPERATION

IF &DATATYPE(&V1) = NUM && &DATATYPE(&V2) = NUM THEN DO
   SET OP =
   IF &OPERATION EQ ADD THEN SET OP = &STR(+)
   ELSE              /* NO ACTION ON ELSE */
   IF &OPERATION EQ SUBTRACT THEN SET OP = &STR(-)
   ELSE              /* NO ACTION ON ELSE */
   IF &OPERATION EQ MULTIPLY THEN SET OP = &STR(*)
   ELSE              /* NO ACTION ON ELSE */
   IF &OPERATION EQ DIVIDE THEN SET OP = &STR(/)
   ELSE              /* NO ACTION ON ELSE */
   IF &LENGTH(&STR(&OP)) = 1 THEN +
      WRITE THE RESULT OF &V1 &OP &V2 = &EVAL(&V1 &OP &V2)
   ELSE WRITE THE OPERATION SPECIFIED IS NOT RECOGNIZED
   END
ELSE DO
   WRITE V1 AND V2 MUST BE NUMERIC
   WRITE THE DATA TYPE OF V1 IS &DATATYPE(&V1)
   WRITE THE DATA TYPE OF V2 IS &DATATYPE(&V2)
   END

EXIT
```



```
%if 1 2 add
 THE RESULT OF 1 + 2 = 3
 READY
%if 8 4 subtract
 THE RESULT OF 8 - 4 = 4
 READY
%if 3 5 multiply
 THE RESULT OF 3 * 5 = 15
 READY
%if 12 4 divide
 THE RESULT OF 12 / 4 = 3
 READY
%if 12 a add
 V1 AND V2 MUST BE NUMERIC
 THE DATA TYPE OF V1 IS NUM
 THE DATA TYPE OF V2 IS CHAR
 READY
%if 1 2 exp
 THE OPERATION SPECIFIED IS NOT RECOGNIZED
 READY
```

## Unconditional Branching - GOTO Statement

The GOTO statement unconditional transfers control from the currently executing statement to the statement containing the label specified as an operand on the GOTO statement. The syntax of the statement is:

[<label>:] GOTO <target label>

If present, the <label> allows the statement to be the target of a branch from another part of the CLIST.

There should almost always be a way to design your CLIST so that you can avoid the use of the GOTO statement, since creating procedures containing unconditional branch instructions are frequently prone to logic errors which are difficult to locate and resolve. Nevertheless, the statement is sometimes the only way to implement the desired logic. Here is another implementation of the prior CLIST using GOTOs:

```
PROC 3 V1 V2 OPERATION

IF &DATATYPE(&V1) = NUM && +
   &DATATYPE(&V2) = NUM THEN GOTO CONT1
ELSE DO
   WRITE V1 AND V2 MUST BE NUMERIC

   WRITE THE DATA TYPE OF V1 IS &DATATYPE(&V1)
   WRITE THE DATA TYPE OF V2 IS &DATATYPE(&V2)
   GOTO ERRORS
   END

CONT1: IF &OPERATION EQ ADD THEN GOTO CADD
IF &OPERATION EQ SUBTRACT THEN GOTO CSUB
IF &OPERATION EQ MULTIPLY THEN GOTO CMUL
IF &OPERATION EQ DIVIDE THEN GOTO CDIV
WRITE THE OPERATION SPECIFIED IS NOT RECOGNIZED

ERRORS: WRITE ERRORS WERE ENCOUNTERED
GOTO CEXIT:

CADD: SET OP = &STR(+)
GOTO CEVAL:

CSUB: SET OP = &STR(-)
GOTO CEVAL:

CMUL: SET OP = &STR(*)
GOTO CEVAL:

CDIV: SET OP = &STR(*)

CEVAL: WRITE THE RESULT OF &V1 &OP &V2 IS &EVAL(&V1 &OP &V2)

CEXIT: EXIT
```

It should be obvious that it is much harder to follow the logic with GOTOs.

**Controlled Repetitive Loops - DO-WHILE-END Statements**

The DO-WHILE-END structure provides the means to construct repetitive loops in a Command List in a clear, easy to understand manner. The syntax of the statements is:

```
[<label>: ]  DO WHILE <expression>
    <one or more statements>
[<label>: ]  END
```

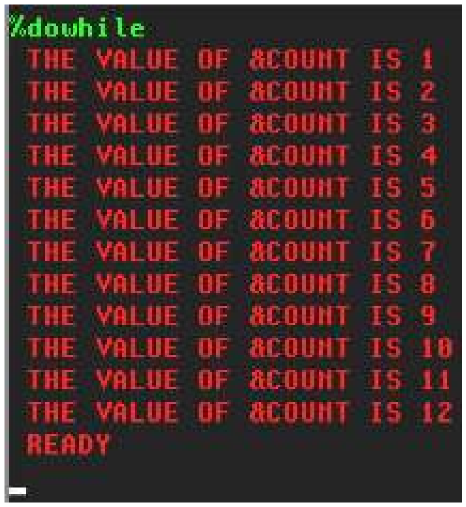
If present, the <label> allows either the DO or END statements to be the target of a branch from another part of the CLIST.

The statements enclosed by the DO and END statements are executed repeatedly as long as the <expression> coded on the WHILE statement is true. The <expression> may use logical operators to evaluate complex comparisons, in the same manner as the IF statement.

A simple example of a DO-WHILE-END loop:

A simple example of a DO-WHILE-END loop:

```
PROC 0


SET &COUNT = 0

DO WHILE &COUNT < 12
  SET &COUNT = &COUNT + 1
  WRITE THE VALUE OF &&COUNT IS &COUNT
END

EXIT
```



```
%dowhile
 THE VALUE OF &COUNT IS 1
 THE VALUE OF &COUNT IS 2
 THE VALUE OF &COUNT IS 3
 THE VALUE OF &COUNT IS 4
 THE VALUE OF &COUNT IS 5
 THE VALUE OF &COUNT IS 6
 THE VALUE OF &COUNT IS 7
 THE VALUE OF &COUNT IS 8
 THE VALUE OF &COUNT IS 9
 THE VALUE OF &COUNT IS 10
 THE VALUE OF &COUNT IS 11
 THE VALUE OF &COUNT IS 12
 READY
```

**Terminating the Command List - EXIT Statement**

In order to terminate (stop executing) your Command List, you use the EXIT statement. The syntax of the EXIT statement is:

[<label>:] EXIT [CODE(<numeric literal> | <numeric variable>)]

If present, the <label> allows the statement to be the target of a branch from another part of the CLIST.

If you just code EXIT, the CLIST simply returns control to the TSO monitor. If the CODE operand is included, the numeric value specified either with a literal or a variable, is stored in the control variable &LASTCC, where it can be examined by statements in another CLIST.

EXIT statements may appear more than once in a single Command List. In actual practice, you will probably have a single common exit point to which control flows if the CLIST terminates normally. Then you might have additional exit points that are reached when error conditions are detected by your coded statements.

**Sequential File Input/Output**

The file access statements allow records to be written to and read from sequential datasets. The operations are restricted to sequential datasets (QSAM access). In order to process a sequential dataset you must:

- use an ALLOCATE command to assign a file name to the dataset
- use an OPENFILE statement to prepare the dataset for input/output
- use one or more PUTFILE or GETFILE statements to write records to or read records from the dataset
- use a CLOSFILE statement to close the dataset
- use a FREE command to release the allocation

You should specify a CONTROL statement with the NOFLUSH operand for all CLISTs that use file input/output. If you don't and an error causes a CLIST to be terminated while files remain open, you may have to log off to recover.

#### OPENFILE Statement

Before you can issue read or write statements to a file, you must use OPENFILE to prepare the file for access. The syntax of the statement is:

[<label>:] OPENFILE <filename> { INPUT | OUTPUT | UPDATE }

If present, the <label> allows the statement to be the target of a branch from another part of the CLIST.

The <filename> operand refers to a file name which must have been ALLOCated prior to issuing the OPENFILE statement. A dataset which is opened for UPDATE may be the target of both PUTFILE and GETFILE statements. If a dataset already contains records and you wish to add more records to it or update the existing records, you must code a disposition of MOD on the ALLOCate statement.

#### PUTFILE Statement

The PUTFILE statement is used to write a single record to a dataset. The syntax of the statement is:

[<label>:] PUTFILE <filename>

If present, the <label> allows the statement to be the target of a branch from another part of the CLIST.

The <filename> operand refers to a dataset which must have been opened prior to the execution of the PUTFILE statement. PUTFILE transfers a single record to the dataset from the file variable &<filename>. The record must be initialized each time by a SET statement unless you want the same record written more than one time. If the dataset is opened in UPDATE mode, a PUTFILE statement overwrites the record read by the previous GETFILE statement. Therefore, to update the 20th record in an existing dataset, you must first issue 19 GETFILE statements to read past the records preceding the 20th record, which is the one to be updated.

**GETFILE Statement**

The GETFILE statement is used to read a single record from a dataset. The syntax of the statement is:

```
[<label>:]  GETFILE <filename>
```

If present, the <label> allows the statement to be the target of a branch from another part of the CLIST.

The <filename> operand refers to a dataset which must have been opened prior to the execution of the GETFILE statement. GETFILE transfers a single record from the dataset into the file variable &<filename>. After reading a record, the &SUBSTR built-in function may be used to extract individual fields from the file variable.

**CLOSFILE Statement**

When you are finished reading or writing records to a dataset, you should close it using the CLOSFILE statement. The syntax of the statement is:

```
[<label>:]  CLOSFILE <filename>
```

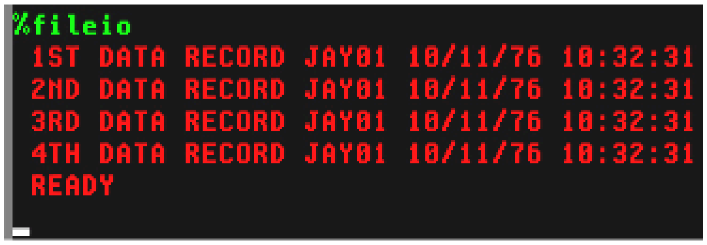
If present, the <label> allows the statement to be the target of a branch from another part of the CLIST.

The <filename> operand refers to a dataset which has been previously opened with an OPENFILE statement. In addition, you may want to code a FREE command to release the file ALLOCation.

An example CLIST using OPENFILE, PUTFILE, GETFILE, CLOSFILE:

```
PROC 0

/* ALLOCATE FILE (ALREADY EXISTS, CATALOGED) */
ALLOC F(TMPF) DA(SEQDATA) OLD

/* OPEN FILE FOR OUTPUT */
OPENFILE TMPF OUTPUT

/* WRITE SOME RECORDS TO FILE */
SET &TMPF = &STR(1ST DATA RECORD &SYSUID &SYSDATE &SYSTIME)
PUTFILE TMPF
SET &TMPF = &STR(2ND DATA RECORD &SYSUID &SYSDATE &SYSTIME)
PUTFILE TMPF
SET &TMPF = &STR(3RD DATA RECORD &SYSUID &SYSDATE &SYSTIME)
PUTFILE TMPF
SET &TMPF = &STR(4TH DATA RECORD &SYSUID &SYSDATE &SYSTIME)
PUTFILE TMPF

/* CLOSE FILE */
CLOSFILE TMPF

/* OPEN FILE FOR INPUT */
```

```
OPENFILE TMPF INPUT

/* READ RECORDS FROM FILE AND DISPLAY */

GETFILE TMPF
WRITE &TMPF
GETFILE TMPF
WRITE &TMPF
GETFILE TMPF
WRITE &TMPF
GETFILE TMPF
WRITE &TMPF

/* CLOSE FILE */
CLOSFILE TMPF

/* FREE FILE */
FREE F(TMPF)

EXIT
```



```
%fileio
 1ST DATA RECORD JAY01 10/11/76 10:32:31
 2ND DATA RECORD JAY01 10/11/76 10:32:31
 3RD DATA RECORD JAY01 10/11/76 10:32:31
 4TH DATA RECORD JAY01 10/11/76 10:32:31
 READY
```

**Extracting Fields from Records - READDVAL Statement**

The READDVAL statement is used to parse the contents of the &SYSDVAL control variable into separate variables. The syntax of the statement is:

[<label>:] READDVAL <variable1> [ <variable2> <variable3> ... <variablen> ]

If present, the <label> allows the statement to be the target of a branch from another part of the CLIST.

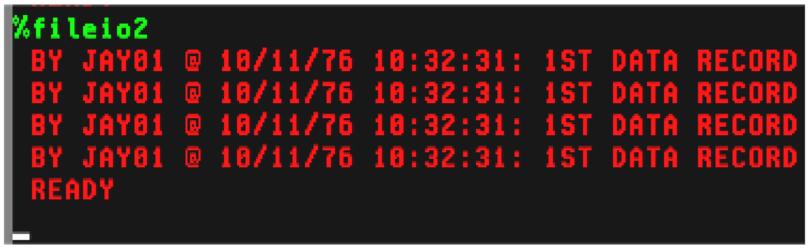
The contents of the &SYSDVAL variable are broken into separate fields delimited by standard TSO delimiters - spaces or commas. If there are more fields in &SYSDVAL than variables are provided, the extraneous fields are ignored. If there are more variables than field available in &SYSDVAL, the extra variables are assigned null values.

In order to use READDVAL/&SYSDVAL to parse the contents of records, the file name used to allocate, open, and read the dataset must be SYSDVAL.

An example CLIST using READDVAL to read the records created by the previous CLIST example:

```
PROC 0

/* ALLOCATE FILE (CREATE IN PRIOR EXAMPLE) */
```

```
        ALLOC F(SYSDVAL) DA(SEQDATA) OLD

        /* OPEN FILE FOR INPUT */
        OPENFILE SYSDVAL INPUT

        /* READ RECORDS FROM FILE AND DISPLAY */
        GETFILE SYSDVAL
        READDVAL FIELD1 FIELD2 FIELD3 FUSER FDATE FTIME
        WRITE BY &FUSER @ &FDATE &FTIME: &FIELD1 &FIELD2 &FIELD3
        READDVAL FIELD1 FIELD2 FIELD3 FUSER FDATE FTIME
        WRITE BY &FUSER @ &FDATE &FTIME: &FIELD1 &FIELD2 &FIELD3
        READDVAL FIELD1 FIELD2 FIELD3 FUSER FDATE FTIME
        WRITE BY &FUSER @ &FDATE &FTIME: &FIELD1 &FIELD2 &FIELD3
        READDVAL FIELD1 FIELD2 FIELD3 FUSER FDATE FTIME
        WRITE BY &FUSER @ &FDATE &FTIME: &FIELD1 &FIELD2 &FIELD3

        /* CLOSE FILE */
        CLOSFILE SYSDVAL

        /* FREE FILE */
        FREE F(SYSDVAL)

        EXIT
```



```
%fileio2
 BY JAY01 @ 10/11/76 10:32:31: 1ST DATA RECORD
 BY JAY01 @ 10/11/76 10:32:31: 1ST DATA RECORD
 BY JAY01 @ 10/11/76 10:32:31: 1ST DATA RECORD
 BY JAY01 @ 10/11/76 10:32:31: 1ST DATA RECORD
 READY
```

**Accepting Data from the TSO User - READ Statement**

The READ statement accepts one or more strings from the TSO user and assigns them to symbolic variables. The syntax of the statement is:

[<label>:] READ <variable1> [ <variable2> <variable3> ... <variablen> ]

If present, the <label> allows the statement to be the target of a branch from another part of the CLIST.

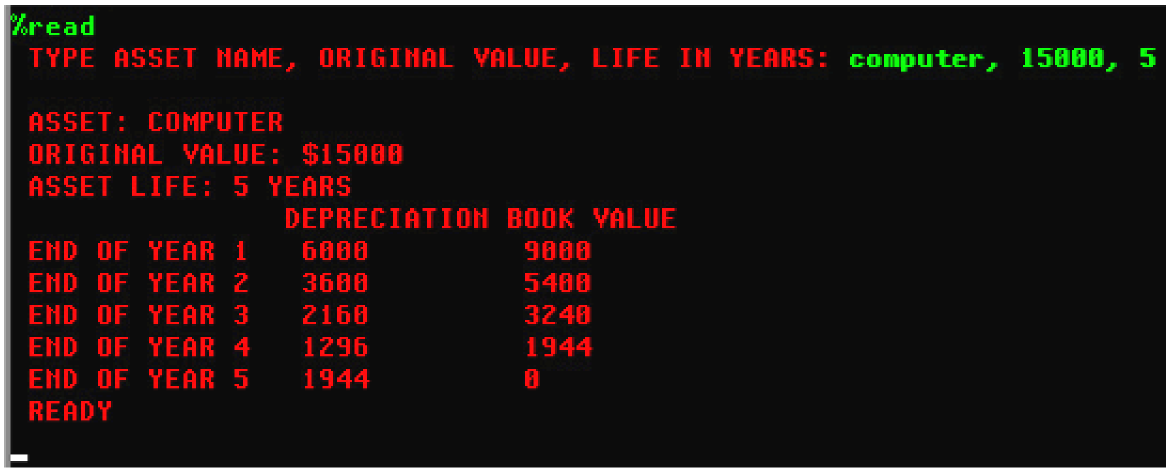
If more than one string is entered by the operator, the strings must be separated by commas. If a string contains embedded blanks or special characters, the string should be enclosed in apostrophes. If the user enter more values than the number of variables coded on the READ statement, the extraneous values are ignored. If the user enters less values than the number of variables coded on the READ statement, the unmatched variables will be assigned null values. The user can selectively omit values by entering consecutive commas with no intervening value:

15,carrots,,1.50,,10

It is generally a good practice to precede READ statements with either WRITE or WRITENR statements identifying to the user what data is to be typed in response when the READ statement is executed.

Here is an example CLIST using READ to obtain values for depreciation calculation:

```
PROC 0  /* COMPUTE DOUBLE DECLINING BALANCE DEPRECIATION */

/* READ VALUES FOR ASSET */
WRITENR TYPE ASSET NAME, ORIGINAL VALUE, LIFE IN YEARS:
READ ASSET VALUE LIFE

/* DISPLAY DEPRECIATION REPORT */
WRITE
WRITE ASSET: &ASSET
WRITE ORIGINAL VALUE: $&VALUE
WRITE ASSET LIFE: &LIFE YEARS
WRITE                DEPRECIATION BOOK VALUE
SET CYEAR = 1
DO WHILE &VALUE > 0
  IF &CYEAR = &LIFE THEN DO
    SET &DEPR = &VALUE
    SET &VALUE = 0
    END
  ELSE DO
    SET &DEPR = (2 * &VALUE) / &LIFE
    SET &VALUE = &VALUE - &DEPR
END
  WRITE END OF YEAR &CYEAR   &DEPR         &VALUE
  SET &CYEAR = &CYEAR + 1
END                              /* END DO LOOP */

EXIT
```



```
%read
 TYPE ASSET NAME, ORIGINAL VALUE, LIFE IN YEARS: computer, 15000, 5

 ASSET: COMPUTER
 ORIGINAL VALUE: $15000
 ASSET LIFE: 5 YEARS
               DEPRECIATION BOOK VALUE
 END OF YEAR 1   6000         9000
 END OF YEAR 2   3600         5400
 END OF YEAR 3   2160         3240
 END OF YEAR 4   1296         1944
 END OF YEAR 5   1944         0
 READY
```

**Embedding TSO Command Groups in CLISTs - DATA/ENDDATA Statements**

In order to include TSO commands in a Command List where the subcommands of the embedded TSO command could be interpreted by the CLIST as command statements, the DATA/ENDATA statements are required. The syntax of the statements is:

[<label>:] DATA

```
[<label>:] DATA
           ENDDATA
```

If present, the <label> allows the statement to be the target of a branch from another part of the CLIST.

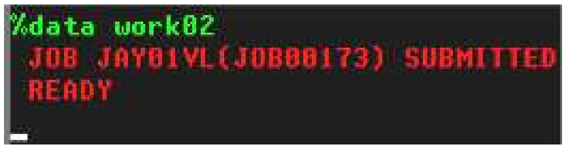
All commands and subcommands included within the DATA/ENDDATA block are regarded as data by the CLIST interpreter, but will be seen as commands/subcommands by the system. Symbolic substitution is performed on the lines with the block before execution.

An example CLIST:

```
PROC 1 SERIAL

EDIT (VTOCLIST) CNTL

DATA
C * =VTOCLIST=&SYSUID.VL=
F =SYSUT1=
C * =$$$$$$=&SERIAL=
SUBMIT
END NOSAVE
ENDDATA

EXIT
```

The CLIST invokes EDIT to tailor a generic jobstream. The purpose of the jobstream is to execute the VTOCLIST program. The EDIT subcommands contained inside the DATA/ENDDATA block perform the following functions:

- change the job name on the first record from VTOCLIST to the TSO User ID plus the characters VL
- locate the DD statement for SYSUT1
- change the marker for VOL=SER - $$$$$$ - to the serial number typed as an argument for the CLIST
- submits the modified jobstream for background execution
- ends the EDIT session without saving the modified member



```
%data work02
 JOB JAY01VL(JOB00173) SUBMITTED
 READY
```

The DATA/ENDDATA block may be used to supply subcommands to any TSO command.

## Temporarily Suspend CLIST Execution - TERMIN Statement

The TERMIN statement returns control of the session to the TSO User in order to allow the user to input any TSO commands or subcommands and then resume execution of the CLIST at the statement following the TERMIN statement. The syntax of the statement is:

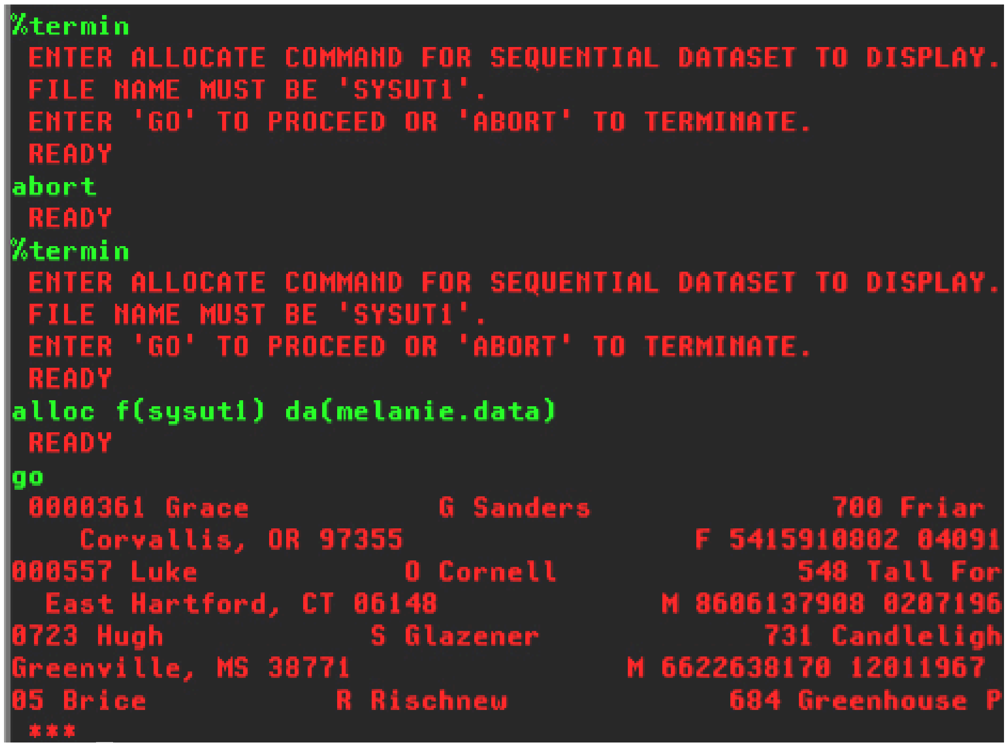
[<label>:] TERMIN [ <string1> <string2> ... <stringn> ]

If present, the <label> allows the statement to be the target of a branch from another part of the CLIST.

One or more optional character strings may be specified on the TERMIN statement. When the user enters a character string matching one of these strings, control is returned to the CLIST at the statement following the TERMIN statement. The control variable &SYSDLM contains a numeric value that corresponds to the ordinal position of the character string as it was coded on the TERMIN statement. Allowing multiple strings to resume the CLIST provides the capability for the TSO user to specify alternative execution options upon return of control to the CLIST. If no strings are to be specified on the TERMIN statement, the word TERMIN must be followed by a comma in place of the first string.

It is generally a good practice to precede TERMIN statements with WRITE statements identifying to the user what string(s) may be used to resume the execution of the CLIST.

```
PROC 0

WRITE ENTER ALLOCATE COMMAND FOR SEQUENTIAL DATASET TO DISPLAY.
WRITE FILE NAME MUST BE 'SYSUT1'.
WRITE ENTER 'GO' TO PROCEED OR 'ABORT' TO TERMINATE.

TERMIN GO ABORT

IF &SYSDLM EQ 2 THEN EXIT

ALLOC F(SYSUT2) DA(*)
ALLOC F(SYSPRINT) DUMMY
ALLOC F(SYSIN) DUMMY
CALL 'SYS1.LINKLIB(IEBGENER)'
FREE F(SYSIN SYSPRINT SYSUT1 SYSUT2)

EXIT
```



```
%termin
 ENTER ALLOCATE COMMAND FOR SEQUENTIAL DATASET TO DISPLAY.
 FILE NAME MUST BE 'SYSUT1'.
 ENTER 'GO' TO PROCEED OR 'ABORT' TO TERMINATE.
 READY
abort
 READY
%termin
 ENTER ALLOCATE COMMAND FOR SEQUENTIAL DATASET TO DISPLAY.
 FILE NAME MUST BE 'SYSUT1'.
 ENTER 'GO' TO PROCEED OR 'ABORT' TO TERMINATE.
 READY
alloc f(sysut1) da(melanie.data)
 READY
go
 0000361 Grace           G Sanders              700 Friar
    Corvallis, OR 97355                 F 5415910802 04091
000557 Luke             O Cornell              548 Tall For
  East Hartford, CT 06148             M 8606137908 0207196
0723 Hugh            S Glazener             731 Candleligh
Greenville, MS 38771                M 6622638170 12011967
05 Brice           R Rischnew             684 Greenhouse P
 ***
```

## Nested Command Lists

It is possible for one Command List to invoke a second Command List. When this occurs the CLISTs are said to be *nested*. The first CLIST is regarded as the *outer* procedure and the second CLIST is regarded as the *inner* procedure. When the second CLIST terminates, either because the last statement is executed or an EXIT statement is processed, control returns to the first CLIST and execution continues with the next statement in that procedure following the statement that invoked the second CLIST. CLISTs may be nested to many levels - the first CLIST is invoked by the TSO user, it calls a second CLIST, the second CLIST calls a third CLIST, etc. In actual practice, you should limit nesting to two or three levels. If you find that you need to nest calls more than three levels, you probably should investigate using a higher level language to satisfy your application.

### Control Options

When control passes from an outer CLIST to an inner CLIST, environment options set by any CONTROL statements are inherited by the inner CLIST. The inner CLIST may alter these options by issuing a CONTROL statement of its own. When control is returned to the outer CLIST, the options are restored to the settings they held when the inner CLIST was invoked.

### Passing Variables

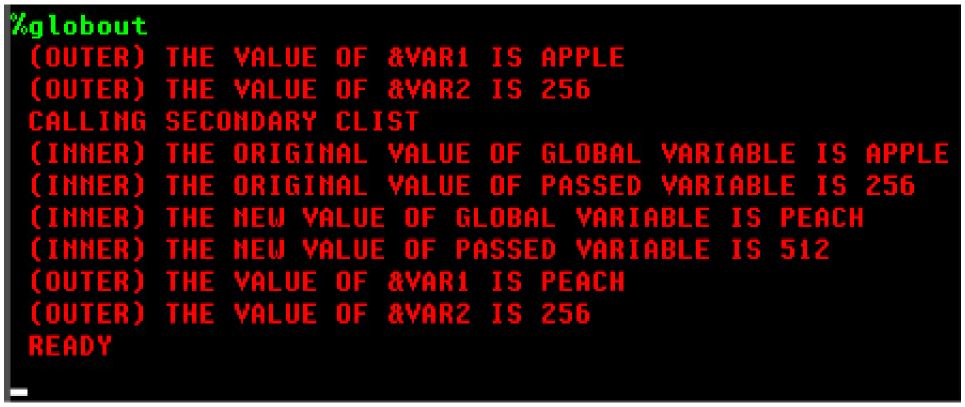
The values contained in variables in the outer CLIST may be passed to the inner CLIST in two ways, through parameters when the inner CLIST is EXECuted or through the use of the GLOBAL statement. The syntax of the statement is:

[<label>:] GLOBAL <variable1> [ <variable2> <variable3> ... <variablen> ]

If present, the <label> allows the statement to be the target of a branch from another part of the CLIST.

The GLOBAL statement is coded in both the outer and inner CLISTs. The values contained in the variables are passed by the ordinal position in the GLOBAL statements. Therefore, the names used for the variables in the inner CLIST do not have to be identical to the names used in the outer CLIST. The GLOBAL statement in the inner CLIST may list fewer variables than the GLOBAL statement in the outer CLIST; the unmatched variables will not be available to the inner CLIST. The GLOBAL statement in the inner CLIST may **not** list more variables than the GLOBAL statement in the outer CLIST. All variables listed in a GLOBAL statement must be defined before they are included on a GLOBAL statement. It is probably a good programming practice to minimize the use of GLOBAL variables in CLISTs, just as the use of global variables is discouraged in current programming methodology.

```
PROC 0                        /* OUTER NESTED CLIST */
SET VAR1 = APPLE
SET VAR2 = 256
GLOBAL VAR1
WRITE (OUTER) THE VALUE OF &&VAR1 IS &VAR1
WRITE (OUTER) THE VALUE OF &&VAR2 IS &VAR2
WRITE CALLING SECONDARY CLIST
%GLOBIN &VAR2
WRITE (OUTER) THE VALUE OF &&VAR1 IS &VAR1
WRITE (OUTER) THE VALUE OF &&VAR2 IS &VAR2
EXIT
PROC 1 PASSEDVAR              /* INNER NESTED CLIST */
GLOBAL GLOBALVAR
WRITE (INNER) THE ORIGINAL VALUE OF GLOBAL VARIABLE IS &GLOBALVAR
WRITE (INNER) THE ORIGINAL VALUE OF PASSED VARIABLE IS &PASSEDVAR
SET GLOBALVAR = PEACH
SET PASSEDVAR = 512
WRITE (INNER) THE NEW VALUE OF GLOBAL VARIABLE IS &GLOBALVAR
WRITE (INNER) THE NEW VALUE OF PASSED VARIABLE IS &PASSEDVAR
EXIT
```



```
%globout
 (OUTER) THE VALUE OF &VAR1 IS APPLE
 (OUTER) THE VALUE OF &VAR2 IS 256
 CALLING SECONDARY CLIST
 (INNER) THE ORIGINAL VALUE OF GLOBAL VARIABLE IS APPLE
 (INNER) THE ORIGINAL VALUE OF PASSED VARIABLE IS 256
 (INNER) THE NEW VALUE OF GLOBAL VARIABLE IS PEACH
 (INNER) THE NEW VALUE OF PASSED VARIABLE IS 512
 (OUTER) THE VALUE OF &VAR1 IS PEACH
 (OUTER) THE VALUE OF &VAR2 IS 256
 READY
```

As you see in the example of parameter passing between the two nested procedures above, when a variable is passed between procedures using the GLOBAL statement, the inner CLISTs is given access to the same copy of the variable. If the value of the variable is changed in the inner CLIST, the original value of the variable is lost. When a variable is passed between CLISTs as a parameter (included on the PROC statement of the called CLIST and the value passed from a symbolic variable when the inner CLIST is invoked), a copy of the value of the variable named in the outer CLIST is passed to the symbolic name used in the inner CLIST. Regardless of the changes to the passed value made by the inner CLIST, the original value of the symbolic variable remains intact.

## Restricting Use of Nested Procedures

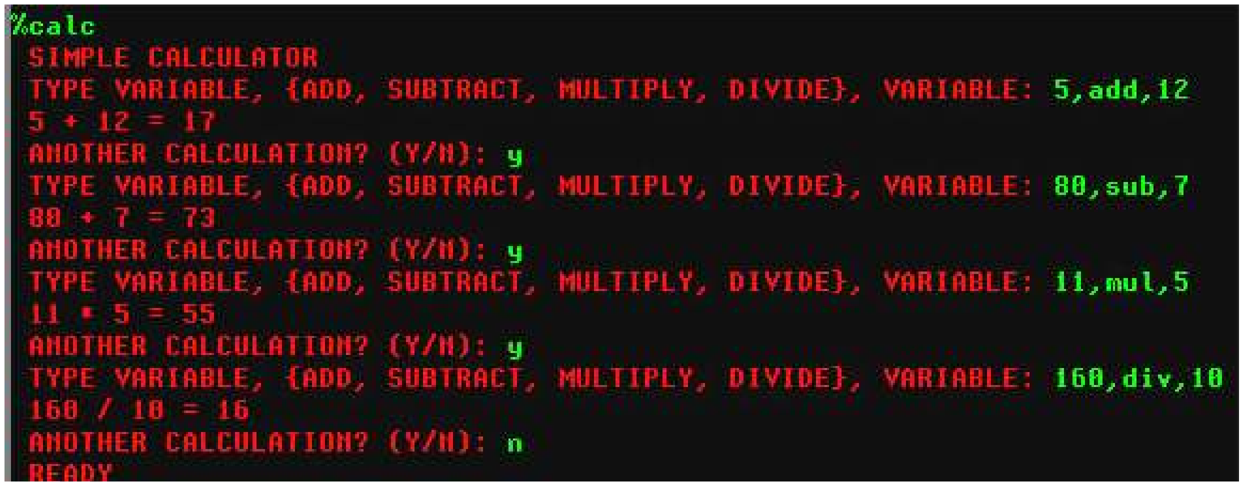
There may be times when you have designed a CLIST to be executed as a nested procedure and you wish to prevent a TSO user from explicitly invoking the CLIST from the TSO Ready prompt. You may accomplish this by using the &SYSNEST control variable:

```
IF &SYSNEST = NO THEN DO
   WRITE YOU CANNOT INVOKE THIS PROCEDURE BY ITSELF
   EXIT
   END
```

A simple calculator illustrates the use of two levels of nested CLISTs:

```
PROC 0                        /* MAIN CLIST: CALC */

WRITE SIMPLE CALCULATOR

SET AGAIN = Y

DO WHILE &AGAIN = Y
  WRITENR TYPE VARIABLE, {ADD, SUBTRACT, MULTIPLY, DIVIDE}, VARIABLE:
  READ V1, OP, V2
  IF &SUBSTR(1,&OP) = A THEN DO
    %ADD V1(&V1) V2(&V2)
    GOTO CONTINUE:
    END
  IF &SUBSTR(1,&OP) = S THEN DO
    %SUB V1(&V1) V2(&V2)
    GOTO CONTINUE:
    END
  IF &SUBSTR(1,&OP) = M THEN DO
    %MUL V1(&V1) V2(&V2)
    GOTO CONTINUE:
    END
  IF &SUBSTR(1,&OP) = D THEN DO
    %DIV V1(&V1) V2(&V2)
    GOTO CONTINUE:
    END
  WRITENR OPERATOR ENTERED (&OP) IS NOT RECOGNIZED.
CONTINUE: WRITENR ANOTHER CALCULATION? (Y/N):
  READ AGAIN
END             /* END DO WHILE */

EXIT

PROC 0 V1(0) V2(0)  /* SECOND LEVEL CLIST: ADD */
IF &SYSNEST EQ NO THEN DO
   WRITE YOU CANNOT INVOKE THIS PROCEDURE DIRECTLY
   EXIT
   END
WRITE &V1 + &V2 = &EVAL(&V1 + &V2)
EXIT
```

Note: The statements contained in the three other second level CLISTs - SUB, MUL, and DIV are not shown above.



```
%calc
 SIMPLE CALCULATOR
 TYPE VARIABLE, {ADD, SUBTRACT, MULTIPLY, DIVIDE}, VARIABLE: 5,add,12
 5 + 12 = 17
 ANOTHER CALCULATION? (Y/N): y
 TYPE VARIABLE, {ADD, SUBTRACT, MULTIPLY, DIVIDE}, VARIABLE: 80,sub,7
 80 - 7 = 73
 ANOTHER CALCULATION? (Y/N): y
 TYPE VARIABLE, {ADD, SUBTRACT, MULTIPLY, DIVIDE}, VARIABLE: 11,mul,5
 11 * 5 = 55
 ANOTHER CALCULATION? (Y/N): y
 TYPE VARIABLE, {ADD, SUBTRACT, MULTIPLY, DIVIDE}, VARIABLE: 160,div,10
 160 / 10 = 16
 ANOTHER CALCULATION? (Y/N): n
 READY
```

## Coding Exit Routines

An exit routine is a block of statements coded as a DO group that is executed either when an error occurs or when an attention key is pressed by the TSO user. The handling and coding of both types of routines is similar.

### ERROR Statement

The ERROR statement is used to set up an environment that checks for non-zero return codes from commands, subcommands, and CLIST statements in teh currently executing procedure. Detection of an error invokes a specified action, which can be an error exit routine. The syntax of the statement is:

[<label>:] ERROR { OFF | <statements> }

If present, the <label> allows the statement to be the target of a branch from another part of the CLIST.

If OFF is coded, any previously established ERROR routine is deleted. If a block of statements is coded, the statements become the current routine to invoke in the event an error occurs.

Note: A CONTROL statement with either MAIN or NOFLUSH must be in effect for a CLIST in which an ERROR routine is coded.

Normally when a TSO command or CLIST statement generates an error condition, the CLIST is terminated. Establishing an ERROR exit allows you to anticipate errors and, in some cases, recover from them while allowing the CLIST to continue execution.

### ATTN Statement

The ATTN statement is used to set up an environment that detects attention interruptions processed by the TSO monitor program. The syntax of the statement is:

[<label>:] ATTN { OFF | <statements> }

If present, the <label> allows the statement to be the target of a branch from another part of the CLIST.

If OFF is coded, any previously established ATTN routine is deleted. If a block of statements is coded, the statements become the current routine to invoke in the event an attention interruption occurs.

### RETURN Statement

The RETURN statement is used to return control following an error condition or an attention routine to the statement immediately following the one that caused the error condition to occur or the statement that was executing when the attention key was pressed. The syntax of the statement is:

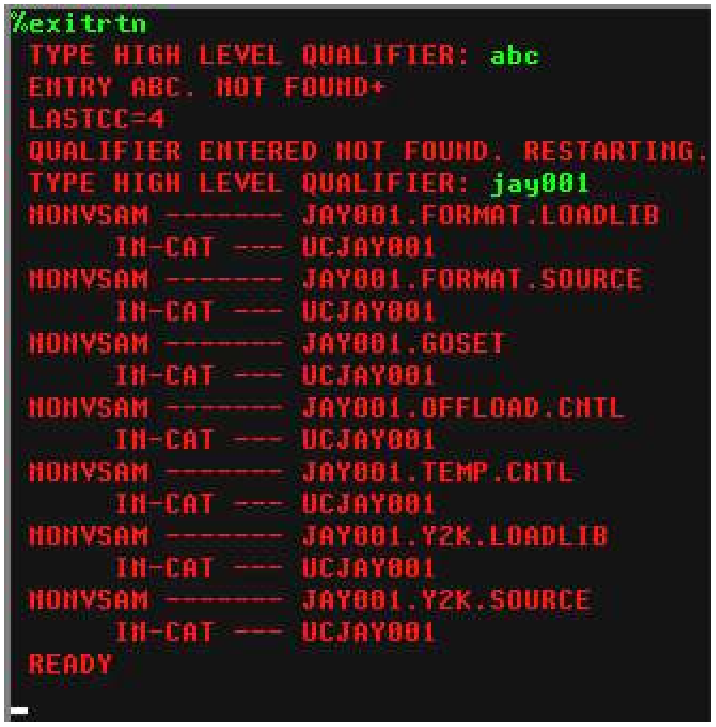
[<label>:] RETURN

If present, the <label> allows the statement to be the target of a branch from another part of the CLIST.

RETURN is valid only when issued from an activated error routine or an activated attention routine from the current CLIST. If neither condition exists, the RETURN becomes a no-operation.

An example of an ERROR exit:

```
PROC 0

CONTROL NOFLUSH

RESTART: WRITENR TYPE HIGH LEVEL QUALIFIER:
READ QUAL

ERROR DO
  IF &LASTCC = 4 THEN DO
    WRITE QUALIFIER ENTERED NOT FOUND. RESTARTING.
    GOTO RESTART
    END
  ELSE EXIT(&MAXCC)
  END

LISTCAT LVL(&QUAL.)

EXIT
```



```
%exitrtn
 TYPE HIGH LEVEL QUALIFIER: abc
 ENTRY ABC. NOT FOUND+
 LASTCC=4
 QUALIFIER ENTERED NOT FOUND. RESTARTING.
 TYPE HIGH LEVEL QUALIFIER: jay001
 NONVSAM -------- JAY001.FORMAT.LOADLIB
      IN-CAT --- UCJAY001
 NONVSAM -------- JAY001.FORMAT.SOURCE
      IN-CAT --- UCJAY001
 NONVSAM -------- JAY001.GOSET
      IN-CAT --- UCJAY001
 NONVSAM -------- JAY001.OFFLOAD.CNTL
      IN-CAT --- UCJAY001
 NONVSAM -------- JAY001.TEMP.CNTL
      IN-CAT --- UCJAY001
 NONVSAM -------- JAY001.Y2K.LOADLIB
      IN-CAT --- UCJAY001
 NONVSAM -------- JAY001.Y2K.SOURCE
      IN-CAT --- UCJAY001
 READY
```

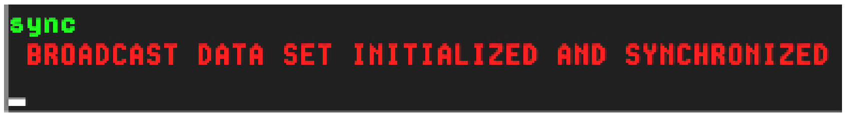
## TSO User Administration - ACCOUNT Command

The ACCOUNT command is used to perform maintenance on the dataset that controls access to the TSO system. The command invokes a conversational program that allows authorized TSO Users to perform the administrative functions of changing, deleting, adding, and listing entries in the user attribute dataset. The ACCOUNT command has no operands, since its function is to initiate a conversational process through which subcommands are used to complete tasks. Prior to entering the ACCOUNT command, the user attribute dataset and broadcast message dataset should both be allocated:

```
ALLOC F(SYSUADS) DA('SYS1.UADS')
ALLOC F(SYSLBC)  DA('SYS1.BRODCAST')
ACCOUNT
```

**SYNC Subcommand**

It is recommended that SYNC be issued as the first subcommand after ACCOUNT is started. It is only necessary to issue the SYNC subcommand once per invocation of ACCOUNT. The SYNC subcommand ensures the correct formatting is present in the broadcast dataset and synchronizes the TSO User IDs between the two datasets.



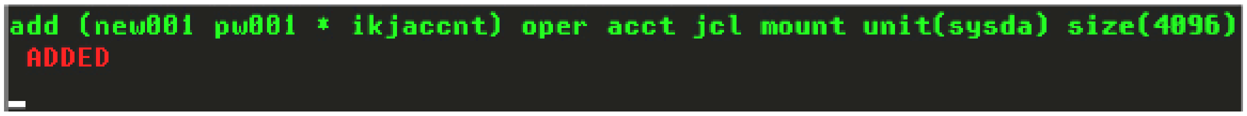
```
sync
 BROADCAST DATA SET INITIALIZED AND SYNCHRONIZED
```

**ADD Subcommand**

The ADD subcommand is used to add a new TSO User ID. The syntax of the subcommand is:

ADD ( <userid> { <password> | * } {<accounting information> | * } <logon procedure> )
  { MAXSIZE(<maximum region size>) | NOLIM }
  { OPER | NOOPER }
  { ACCT | NOACCT }
  { JCL | NOJCL }
  { MOUNT | NOMOUNT }
  USERDATA(<string>)
  { PERFORM(<performance group>) | NOPERFORM }
  SIZE(<default region size>)
  UNIT(<default DASD unit>)
  DEST(<routing destination>)

For additional help on the operands of the ADD command, you can enter HELP ADD.



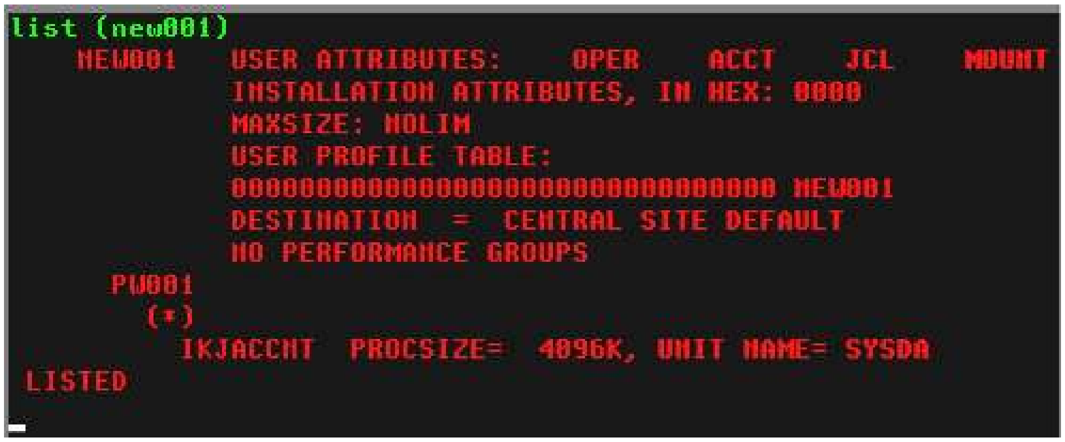
```
add (new001 pw001 * ikjaccnt) oper acct jcl mount unit(sysda) size(4096)
 ADDED
```

### LIST Subcommand

The LIST subcommand is used to list the settings for one or more TSO User IDs. The syntax of the subcommand is:

LIST ( <user id> | * )

Specifying asterisk (*) for the operand will list all User IDs.

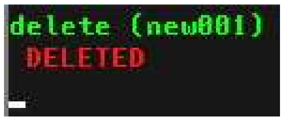


```
list (new001)
    NEW001   USER ATTRIBUTES:    OPER    ACCT    JCL    MOUNT
             INSTALLATION ATTRIBUTES, IN HEX: 0000
             MAXSIZE: NOLIM
             USER PROFILE TABLE:
             00000000000000000000000000000000 NEW001
             DESTINATION  =  CENTRAL SITE DEFAULT
             NO PERFORMANCE GROUPS
      PW001
        (*)
          IKJACCNT  PROCSIZE=  4096K, UNIT NAME= SYSDA
 LISTED
```

### DELETE Subcommand

The DELETE subcommand is used to delete a User ID. The syntax of the command is:

DELETE (<user id>)



```
delete (new001)
 DELETED
```

### CHANGE Subcommand

The CHANGE subcommand may be used to change some attributes of a User ID, however in my experience it is usually easier to simply delete the existing User ID and add it back with the desired settings.

### END Subcommand

The ACCOUNT conversational process is terminated by the END subcommand.

## MVS Console Operator Functions - OPER Command

The OPER command is used to enter a subset of the MVS console operator commands that are usually entered at the main or alternate system console. The command invokes a conversational program that allows authorized TSO Users to perform some of the functions usually carried out by the MVS system operator. The OPER command has no operands, since its function is to initiate a conversational process through which subcommands are used to complete tasks. The basic syntax of the subcommands is:

```
<subcommand> <operand1>,<operand2>,...<operandn>
```

Blanks are the only acceptable separator character between the <subcommand> and the operands. A comma is the only acceptable separator character between the operands. Details for selected subcommands is included below.

### CANCEL Subcommand

The CANCEL subcommand (which may be abbreviated C) is used to cancel TSO User sessions. The syntax of the subcommand is:

```
Cancel U=<userid>[,DUMP]
```

If the optional DUMP operand is specified, a storage dump is produced of the address space allocated for the TSO User session.

### DISPLAY Subcommand

The DISPLAY subcommand (which may be abbreviated D) is used to display information about regions, TSO User sessions, jobs, and SLIP traps. The syntax of the subcommand is:

```
Display { T | TS[,List] | R[,List] | Jobs[,List] | SLIP[,=<id>] }
```

The information displayed for each available operand is:

- T - displays the time and date
- TS - displays the number of TSO users logged on; if LIST (may be abbreviated L) is also specified a list of TSO User IDs is also displayed
- R - displays message IDs of outstanding console operator requests; if LIST (may be abbreviated L) is also specified a list of the message text is also displayed
- JOBS (may be abbreviated J) - displays the number of batch jobs and initiators currently active; if LIST (may be abbreviated L) is also specified a list of jobs (with step name and other attributes) is also displayed
- SLIP - displays a summary of all SLIP traps set; if <id> is also specified a detailed display of the trap

identified is also displayed

**Monitor Subcommand**

The MONITOR subcommand (which may be abbreviated MN) is used to request notification at your terminal when batch jobs or TSO sessions begin and end, or the disposition of datasets when they are de-allocated. The syntax of the subcommand is:

```
MoNitor  { JOBNAMES[,T] | SESS[,T] | STATUS }
```

The information displayed for each available operand is:

- JOBNAMES - requests that a message be displayed at your terminal when each batch job begins execution and ends execution; if T is also specified the system time is included in the message
- SESS - requests that a message be displayed at your terminal when each TSO User session is started and terminated; if T is also specified the system time is included in the message
- STATUS - requests that a message be displayed at your terminal showing dataset names and volume serial numbers with KEEP, CATLG, or UNCATLG disposition at step and job termination

The notification messages will only be displayed as long as the OPER command remains active.

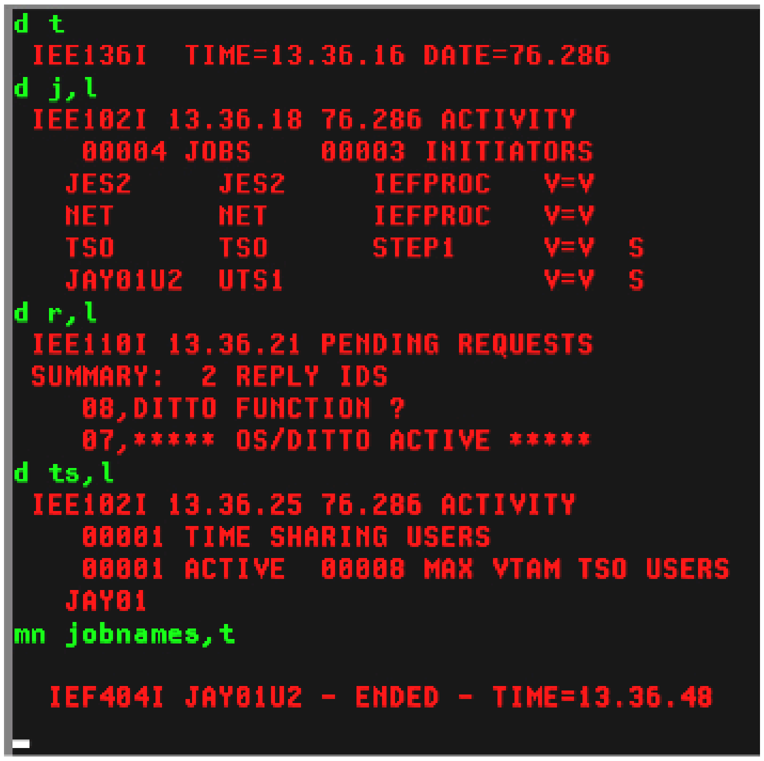
**STOPMN Subcommand**

The STOPMN subcommand (which may be abbreviated PM) is used to terminate the display of messages initiated by the MONITOR subcommand. The syntax of the subcommand is:

```
stoPMn { JOBNAMES | SESS | STATUS }
```

**END Subcommand**

The OPER conversational process is terminated by the END subcommand.



```
d t
 IEE136I  TIME=13.36.16 DATE=76.286
d j,l
 IEE102I 13.36.18 76.286 ACTIVITY
    00004 JOBS    00003 INITIATORS
   JES2      JES2      IEFPROC   V=V
   NET       NET       IEFPROC   V=V
   TSO       TSO       STEP1     V=V  S
   JAY01U2  UTS1                 V=V  S
d r,l
 IEE110I 13.36.21 PENDING REQUESTS
 SUMMARY:  2 REPLY IDS
    08,DITTO FUNCTION ?
    07,***** OS/DITTO ACTIVE *****
d ts,l
 IEE102I 13.36.25 76.286 ACTIVITY
    00001 TIME SHARING USERS
    00001 ACTIVE  00008 MAX VTAM TSO USERS
   JAY01
mn jobnames,t

  IEF404I JAY01U2 - ENDED - TIME=13.36.48

_
```

---

# TSO Tutorial Index

---

I hope that you have found my instructions useful. If you have questions that I can answer to help expand upon the information I have included here, please don't hesitate to send them to me:

dino@ jaymoseley.com

---

Back to Previous Page Return to Site Home Page

This page was last updated on September 06, 2006.