z/OS

TSO/E CLISTs

Version 2 Release 2

SA32-0978-01

Note
Before using this information and the product it supports, read the information in "Notices" on page 183.

This edition applies to Version 2 Release 2 of z/OS (5650-ZOS) and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

Figures

Tables

About this document

This document supports z/OS® (5650-ZOS).

This document describes how to use the TSO/E CLIST language to write programs called CLISTs. You can use CLISTs to perform a wide range of programming tasks on TSO/E.

Who should use this document

This document is intended for new and experienced CLIST programmers.

If you are a new user of the CLIST language, read each chapter and try coding the examples.

If you are experienced with CLISTs, review the chapters and familiarize yourself with the organization of this document. Then you'll be able to refer to the appropriate chapter when you have a question or want to refresh your memory.

To use the CLIST language effectively, you should be familiar with TSO/E commands. Familiarity with the Interactive System Productivity Facility (ISPF) is also helpful. For information about TSO/E commands, see *z/OS TSO/E Command Reference*.

How this document is organized

- Chapter 1, “Introduction,” on page 1 describes the types of functions CLISTs perform.
- Chapter 2, “Creating, editing, and executing CLISTs,” on page 3 describes how to create and edit CLIST data sets, and how to execute CLISTs.
- Chapter 3, “Writing CLISTs - Syntax and conventions,” on page 9 describes the rules for using CLIST statements, TSO/E commands, and JCL statements in CLISTs.
- Chapter 4, “Using symbolic variables,” on page 17 describes how to define symbolic variables and assign values to them.
- Chapter 6, “Using control variables,” on page 29 describes how to use CLIST control variables to obtain current information about the processing environment.
- Chapter 7, “Using built-in functions,” on page 53 describes how to use CLIST string-handling functions to process numeric data and character strings.
- Chapter 8, “Structuring CLISTs,” on page 67 describes how to use CLIST statements to make decisions and loops, and how to use CLIST subprocedures and nested CLISTs.
- Chapter 9, “Communicating with the terminal user,” on page 85 describes how to write interactive CLISTs.
- Chapter 10, “Performing file I/O,” on page 97 describes how to read and write records to and from files.
- Chapter 11, “Writing ATTN and ERROR routines,” on page 103 describes how to write routines that receive control when errors occur or when the user presses the attention key while a CLIST is running.

- Chapter 12, “Testing and debugging CLISTs,” on page 111 describes how to find and correct CLIST errors. This chapter includes a list of error codes and their meanings.
- Chapter 13, “Sample CLISTs,” on page 119 provides sample CLISTs that perform a broad range of application tasks. Each CLIST comes with a description of the concepts that it illustrates. Generally, the more advanced CLISTs expand upon concepts introduced in the simpler examples.
- Chapter 14, “Reference,” on page 145 contains complete syntax descriptions of all of the CLIST statements.

Where to find more information

Please see *z/OS Information Roadmap*for an overview of the documentation associated with z/OS, including the documentation available for z/OS TSO/E.

How to send your comments to IBM

We appreciate your input on this publication. Feel free to comment on the clarity, accuracy, and completeness of the information or provide any other feedback that you have.

Use one of the following methods to send your comments:

1. Send an email to mhvrcfs@us.ibm.com.
2. Send an email from the "Contact us" web page for z/OS (http://www.ibm.com/systems/z/os/zos/webqs.html).

Include the following information:

- Your name and address.
- Your email address.
- Your telephone or fax number.
- The publication title and order number:
  z/OS V2R2 TSO/E CLISTs
  SA32-0978-01
- The topic and page number that is related to your comment.
- The text of your comment.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute the comments in any way appropriate without incurring any obligation to you.

IBM or any other organizations use the personal information that you supply to contact you only about the issues that you submit.

If you have a technical problem

Do not use the feedback methods that are listed for sending comments. Instead, take one of the following actions:

- Contact your IBM service representative.
- Call IBM technical support.
- Visit the IBM Support Portal at z/OS Support Portal (http://www-947.ibm.com/systems/support/z/zos/).

Summary of changes for z/OS Version 2 Release 2 (V2R2)

The following changes are made for z/OS Version 2 Release 2 (V2R2).

Changed

- Modified "&SYSLRACF" on page 38 in Getting information about the system of Using control variables.
- Added SYSUSEDEXTENTS variable to "CLIST variables set by LISTDSI" on page 161.

Summary of changes for z/OS Version 2 Release 1

See the following publications for all enhancements to z/OS Version 2 Release 1 (V2R1):

- *z/OS Migration*
- *z/OS Planning for Installation*
- *z/OS Summary of Message and Interface Changes*
- *z/OS Introduction and Release Guide*

Chapter 1. Introduction

The CLIST language enables you to work more efficiently with TSO/E. You can write programs, called CLISTs, that perform given tasks or groups of tasks. From then on, you can invoke the CLISTs to do those tasks.

The term *CLIST* (pronounced "sea list") is short for *Command LIST*, because the most basic CLISTs are lists of TSO/E commands. When you invoke such a CLIST, it issues the TSO/E commands in sequence.

Besides issuing TSO/E commands, CLISTs can perform more complex programming tasks. The CLIST language includes the programming tools you need to write extensive, structured applications. CLISTs can perform any number of complex tasks, from displaying a series of full-screen panels to managing programs written in other languages.

The CLIST language is an interpretive language. Like programs in other high-level interpretive languages, CLISTs are easy to write and test. You don't have to compile and link-edit them. To test a CLIST, you execute it, correct any errors, and re-execute it.

The CLIST language is one of two command languages available in TSO/E. For information about the other command language, REXX, see *z/OS TSO/E REXX User's Guide* and *z/OS TSO/E REXX Reference*.

Features of the CLIST Language

The CLIST language provides a wide range of programming functions. Its features include:

- An extensive set of arithmetic and logical operators for processing numeric data
- String-handling functions for processing character data
- CLIST statements that let you structure your programs, perform I/O, define and modify variables, and handle errors and attention interrupts

Categories of CLISTs

A CLIST can perform a wide range of tasks. Three general categories of CLISTs are:

- CLISTs that perform routine tasks
- CLISTs that are structured applications
- CLISTs that manage applications written in other languages

CLISTs that perform routine tasks

As a user of TSO/E, you probably perform certain tasks on a regular basis. These tasks may involve entering TSO/E commands to check on the status of data sets, to allocate data sets for particular programs, and to print files.

You can write CLISTs that significantly reduce the amount of time that you have to spend on these routine tasks. By grouping together in a CLIST the instructions required to complete a task, you reduce the time, number of keystrokes, and errors involved in performing the task; thus, you increase your productivity. Such a

CLIST can consist of TSO/E commands only, or a combination of TSO/E commands, JCL statements, or CLIST statements.

Figure 1 is an example of a CLIST that consists of TSO/E commands only.

```
allocate file(ABC) dataset(name1)
allocate file(DEF) dataset(name2)
call (prog1)
free file(ABC DEF)
```

Figure 1. Sample CLIST consisting of TSO/E commands

The CLIST in Figure 1 issues TSO/E commands to allocate files for a program, call the program, and free the files when the program is finished. Whenever you wanted to perform these related tasks, you can execute the CLIST instead of retyping the commands.

If tasks require specific input from a user, you can obtain the input in a CLIST by using CLIST statements or TSO/E commands to prompt the user for the input.

CLISTs that are structured applications

The CLIST language includes the basic tools you need to write complete, structured applications. Any CLIST can invoke another CLIST, which is referred to as a nested CLIST. CLISTs can also contain separate routines called subprocedures. Nested CLISTs and subprocedures let you separate your CLISTs into logical units and put common functions in a single location. Specific CLIST statements let you:

- Define common data for subprocedures and nested CLISTs
- Restrict data to certain subprocedures and CLISTs
- Pass specific data to a subprocedure or nested CLIST

For interactive applications, CLISTs can issue commands of the Interactive System Productivity Facility (ISPF) to display full-screen panels. Conversely, ISPF panels can invoke CLISTs, based on input that a user types on the panel. When the user changes a value on a panel, the change applies to the value in the CLIST that displayed the panel. With ISPF, CLISTs can manage extensive panel-driven dialogs.

CLISTs that manage applications written in other languages

You might have access to applications that are written in other programming languages. However, the interfaces to these applications might not be easy to use or remember. Rather than write new applications, you can write CLISTs that provide easy-to-use interfaces between the user and such applications.

A CLIST can send messages to, and receive messages from, the terminal to determine what the user wants to do. Then, based on this information, the CLIST can set up the environment and issue the commands required to invoke the program that performs the requested tasks.

Chapter 2. Creating, editing, and executing CLISTs

CLIST data sets and libraries

CLISTs reside in either sequential or partitioned data sets (PDSs). A sequential CLIST data set consists of only one CLIST, while a PDS can contain one or more CLISTs. In a PDS, each CLIST is a member and has a unique member name. When a PDS consists entirely of CLISTs, it is called a CLIST library.

CLIST libraries make CLISTs easy to maintain and execute. Your installation can keep commonly used CLISTs in a system CLIST library, and you can keep your own CLISTs in a private CLIST library. If you allocate a CLIST library to the file SYSPROC, or specify the library on the ALTLIB command, you can execute the CLISTs *implicitly* by typing their member names.

Implicit execution frees you from having to code the name of the CLIST library on an EXEC command. Besides saving keystrokes, implicit execution lets you keep different versions of a CLIST in different libraries and control which version executes at a given time. For more information, see "Allocating CLIST libraries for implicit execution" on page 6.

CLISTs invoked implicitly, and command processors invoked from CLISTs, should not have names equal to reserved CLIST words. If name conflicts cannot be avoided, consider using the &STR built-in function to solve the problem. See "Defining character data - &STR" on page 57. For example, in the case of the SELECT subcommand of the RACFRW command, you can specify the subcommand name as follows, to avoid confusion with the CLIST SELECT statement:

```
&STR(SELECT) VIOLATIONS
```

Creating and editing CLIST data sets

Before coding your first CLIST, you must create a CLIST data set. There are two ways to create and edit a CLIST data set:

1. Using options 3 (UTILITIES) and 2 (EDIT) of ISPF/PDF:
   a. Create a data set using the allocate panel in ISPF (typically option 3.2 on the primary menu).
      - To simplify execution, specify CLIST as the data set type.
      - To create a data set with the same attributes as another, such as a system CLIST library, use option 3.2 to view the attributes of the existing data set and then allocate the new data set.

   b. Code your CLIST in the full-screen environment using the ISPF/PDF editor (typically option 2).
   c. Modify the CLIST by making corrections directly to the data on the screen.

   For more information about creating and editing data sets under ISPF/PDF, see *z/OS TSO/E Primer*.
2. Using the TSO/E EDIT command and its subcommands (this method includes option 6 of ISPF/PDF.):
   a. Include the CLIST keyword on the EDIT command.

b. Enter and save your CLIST statements, TSO/E commands, and TSO/E subcommands.
c. Use subcommands of EDIT to modify the CLIST.

CLISTs created with the EDIT command cannot contain characters of the double-byte character set (DBCS).

More information about creating and editing data sets under TSO/E can be found in *z/OS TSO/E Command Reference*.

CLIST data set attributes

If a CLIST data set is created by one of the previously described methods, and the CLIST keyword is specified on the EDIT command, the data set will be assigned the following default attributes (provided that your installation has not changed the default values):

- A variable-length record format

  If you specify a LINE value on the EDIT command, the data set will be of a fixed-length record format in the specified length.
- A logical record size of 255 characters
- A block size of 3120 bytes
- Line numbers are contained in the last eight bytes of all fixed-length records and in the first eight bytes of all variable-length records
- All input data and modified data are converted to uppercase characters.

Your installation may have changed these default attributes and may have established CLIST data set conventions to ease data set tasks.

If you concatenate CLIST data sets, specify the same RECFM and LRECL values for these data sets.

For a complete description of edited data sets see, the EDIT command in *z/OS TSO/E Command Reference*. For a discussion of the formats and characteristics of the RECFM subparameter of the DCB parameter, see *z/OS MVS JCL Reference*. If you want to obtain information about a data set for use in CLIST variables, see "LISTDSI statement" on page 158.

Considerations for copying CLIST data sets

When creating and editing CLISTs, you might copy an existing CLIST data set into a new data set. If you do so under ISPF/PDF, be aware of the record formats of the data sets. Variable-blocked data sets might contain line numbers in columns 1-8 that do not normally appear when you edit the data sets. If you copy a variable-blocked data set into a fixed-blocked data set, the line numbers are copied as part of the data. This data must then be removed. To find out if a data set contains line numbers, use the ISPF EDIT command PROFILE.

If you copy a fixed-blocked data set with line numbers into a variable-blocked data set, the system copies sequence numbers from columns 73-80 into the variable-blocked data set. This data must also be removed. For information about how to remove the sequence numbers from a variable-blocked data set, see *z/OS V2R2 ISPF Edit and Edit Macros*.

Executing CLISTs

To execute a CLIST, use the EXEC command. From an ISPF command line, type TSO in front of the command. In TSO/E EDIT or TEST mode, use the EXEC subcommand as you need to use the EXEC command. (CLISTs executed under EDIT or TEST can issue only EDIT or TEST subcommands and CLIST statements, but you can use the END subcommand in a CLIST to end EDIT or TEST mode and allow the CLIST to issue TSO/E commands.)

The EXEC command (or subcommand) has two forms:

1. **Explicit form:** Enter "exec" or "ex" followed by the data set name and the optional CLIST operand. By default, the EXEC command assumes that the data set type is CLIST and automatically suffixes all specified names with **.CLIST**, unless the name is in quotation marks. For example:
   - If a CLIST, LISTPGM, is a member of a PDS named PREFIX.CLISTLIB.CLIST, enter:

     ```
     {exec}  clistlib(listpgm)   [CLIST]
     {ex  }
     ```

   - If a CLIST, LISTPGM, is a member of a PDS named PREFIX.CLIST, enter:

     ```
     {exec}  (listpgm)       [CLIST]
     {ex  }
     ```

   - If the CLIST is in a sequential data set named PREFIX.LISTPGM.CLIST, enter:

     ```
     {exec}  (listpgm)       [CLIST]
     {ex  }
     ```

   - If the CLIST is in a sequential data set named PREFIX.LISTPGM, enter:

     ```
     {exec}  'prefix.listpgm'    [CLIST]
     {ex  }
     ```

   If the EXEC command is used to execute a CLIST in a sequential data set, but the data set is found to be a partitioned one, it will assume a member TEMPNAME. The system will notify you if this member is not found, otherwise it will execute it.
2. **Implicit form:** Enter only the name of the CLIST, optionally preceded by a percent sign (%). The CLIST must be a member of a PDS allocated to the file SYSPROC, or an alternative library specified with the ALTLIB command. The two implicit forms are as follows:
   a. Enter only the member name, for example:

      ```
      listpgm
      ```

      When you use this form, TSO/E first searches command libraries to ensure that the name you entered is not a TSO/E command, then searches CLIST libraries:
      - Specified with the ALTLIB command or
      - Allocated to the SYSPROC file

   b. Enter the member name prefixed with a percent sign (%), for example:

      ```
      %listpgm
      ```

      When you use this form, called the extended implicit form, TSO/E searches only the ALTLIB or SYSPROC libraries for the name, thus reducing the amount of search time.

   For information about preparing a CLIST for implicit execution, see "Allocating CLIST libraries for implicit execution" on page 6.

You can execute a CLIST in either the foreground (from your terminal) or in the background (submit it as a batch job). You can also execute a CLIST from another

CLIST (using the EXEC command) or from a program. To invoke a CLIST from a program, use the TSO/E service facility described in *z/OS TSO/E Programming Services*.

Passing parameters to CLISTs

You can pass parameters to a CLIST when you execute it. Parameters are variable input that may change from one execution to the next. To receive parameters, a CLIST must begin with a PROC statement that assigns the parameters to variables. "Using the PROC statement" on page 19 explains how to code a PROC statement to receive parameters.

To pass parameters to a CLIST, include them on the EXEC command or subcommand as follows:

- For the explicit form, pass parameters in single quotation marks:

  ```
  EX clistname 'parm1 parm2'
  ```

- For the implicit or extended implicit form, omit the quotation marks:

  ```
  %clistname parm1 parm2
  ```

For more information about the types of parameters you can pass, and how to use them in a CLIST, see "Using the PROC statement" on page 19.

For a complete syntactical definition of the EXEC command, including special considerations for passing parameters that contain single quotation marks, see Chapter 14, "Reference," on page 145.

Allocating CLIST libraries for implicit execution

After you have written CLISTs and executed them to make sure they run correctly, you can allocate them to special files to make them easier to execute.

When CLISTs are members of a partitioned data set (PDS) allocated to a special file, users and applications can execute the CLISTs implicitly by invoking the member names. How you can allocate CLIST libraries for implicit execution depends on the feature of TSO/E installed on your system.

The ALTLIB command gives you more flexibility in specifying CLIST libraries for implicit execution. With ALTLIB, a user or ISPF application can easily activate and deactivate CLIST libraries for implicit execution as the need arises. This flexibility can result in less search time when fewer CLISTs are activated for implicit execution at the same time.

In addition to CLISTs, the ALTLIB command lets you specify libraries of REXX execs for implicit execution. For information about using ALTLIB with REXX execs, see *z/OS TSO/E REXX User's Guide*.

Specifying alternative CLIST libraries with the ALTLIB command

The ALTLIB command lets you specify *alternative libraries* to contain implicitly executable CLISTs. You can specify alternative libraries on the user, application, and system levels.

- The *user level* includes CLIST libraries allocated to the file SYSUPROC. During implicit execution, these libraries are searched first.

- The *application level* includes CLIST libraries specified on the ALTLIB command by data set or file name. During implicit execution, these libraries are searched after user libraries.
- The *system level* includes CLIST libraries allocated to file SYSPROC. During implicit execution, these libraries are searched after user or application libraries.

Using the ALTLIB command

The ALTLIB command offers several functions, which you specify using the following operands:

ACTIVATE
allows implicit execution of CLISTs in a library or libraries on the specified level(s), in the order specified.

DEACTIVATE
excludes the specified level(s) from the search order.

DISPLAY
displays the current order in which CLIST libraries are searched for implicit execution.

RESET
resets searching to the system level only, for CLISTs and REXX execs.

For complete information about the syntax of the ALTLIB command, see *z/OS TSO/E Command Reference*.

Note:

1. With ALTLIB, data sets concatenated to each of the levels can have differing characteristics (logical record length and record format), but the data sets within the same level must have the same characteristics.
2. At the application and system levels, ALTLIB uses the virtual lookaside facility (VLF) to provide potential increases in library search speed.

Using ALTLIB with ISPF

ALTLIB works the same in line mode TSO/E and in ISPF. However, if you use ALTLIB under line mode TSO/E and start ISPF, the alternative libraries you specified under line mode TSO/E are unavailable until ISPF ends.

Application-level libraries that you define while running an ISPF application are in effect only while that application has control. When the application completes, the original application-level libraries are automatically reactivated.

Under ISPF, you can pass the alternative library definitions from application to application by using ISPEXEC SELECT with the PASSLIB operand. For example, to pass ALTLIB definitions to a new ISPF application (ABC), code:

```
ISPEXEC SELECT NEWAPPL(ABC) PASSLIB
```

The PASSLIB operand passes the ALTLIB definitions to the invoked application. When the invoked application completes and the invoking application regains control, the ALTLIB definitions that were passed take effect again, regardless of whether the invoked application changed them. If you omit the PASSLIB operand, ALTLIB definitions are not passed to the invoked application.

For more information about writing ISPF applications, see *z/OS V2R2 ISPF Services Guide*.

Stacking ALTLIB requests

On the application level, you can stack up to eight activate requests with the top, or current, request active.

Examples of the ALTLIB command

In the following example, an application issues the ALTLIB command to allow implicit execution of CLISTs in the data set NEW.CLIB, to be searched ahead of SYSPROC:

```
ALTLIB ACTIVATE APPLICATION(CLIST) DATASET(new.clib)
```

The application can also allow searching for any private CLISTs that the user has allocated to the file SYSUPROC, with the following command:

```
ALTLIB ACTIVATE USER(CLIST)
```

To display the active libraries in their current search order, use the DISPLAY operand as follows:

```
ALTLIB DISPLAY
```

To deactivate searching for a certain level, use the DEACTIVATE operand; for example, to deactivate searching for CLISTs on the system level (those allocated to SYSPROC), issue:

```
ALTLIB DEACTIVATE SYSTEM(CLIST)
```

And, to reset CLIST and REXX exec searching back to the system level, issue:

```
ALTLIB RESET
```

For more information about the search order EXEC uses for CLISTs and REXX execs, see *z/OS TSO/E Command Reference*.

Chapter 3. Writing CLISTs - Syntax and conventions

This chapter provides an overview of CLIST statements and describes how to use the following:

- Syntax rules of the CLIST language
- TSO/E commands and JCL statements in CLISTs
- CLIST operators and expressions
- The double-byte character set (DBCS) in CLISTs

When you are familiar with the contents of this chapter, read the following chapters for information about how to use variables and terminal input in CLISTs.

Overview of CLIST statements

CLIST statements set controls, assign values to variables, monitor the conditions under which CLISTs execute, and perform I/O. CLIST statements execute in both the command and subcommand environment (under the TSO/E EXEC command and the EXEC subcommand of TSO/E EDIT). They fall into the categories shown in Table 1.

Table 1. CLIST statement categories

Control	Assignment	Conditional	I/O
ATTN CONTROL DATA-ENDDATA DATA-PROMPT ERROR EXIT GLOBAL GOTO NGLOBAL PROC RETURN SYSCALL SYSREF TERMIN WRITE WRITENR	READ READDVAL SET LISTDSI	DO IF-THEN-ELSE SELECT	CLOSFILE GETFILE OPENFILE PUTFILE

Subsequent topics in this document describe all of the statements in detail.

Note: In addition to these CLIST statements,IBM® provides an installation exit that lets your installation add its own CLIST statements. For information about this exit, see *z/OS TSO/E Customization*.

Syntax rules

This section describes the syntax rules for CLIST statements relative to those for TSO/E commands.

Delimiters

Most CLIST statements have operands. Operands are variables or data that provide information to be used in processing the statement. Include one or more blanks between a CLIST statement and its first operand. Also, separate operands from each other by one or more blanks, a comma, or tabs.

Continuation symbols

Line continuation symbols are the same as for TSO/E commands. If used, the continuation symbol must be the last non-blank character on the line. A hyphen (-) indicates that leading blanks in the next line are not ignored. A plus sign (+) indicates that leading blanks in the next line are ignored. For example, the following command executes successfully:

```
alloc da(jcl.cntl) shr-
  reuse file(input)
```

However, if you substitute a plus sign for the hyphen in this example, the command fails because, when the lines are joined logically, there is no blank between the end of the **shr** keyword and the beginning of the **reuse** keyword. You need to insert a blank before the plus sign for correct execution.

Capitalization

All CLIST statement names must be capitalized. If you use lowercase letters for CLIST statement names, the CLIST fails. Capitalization of CLIST variable names and built-in function names is optional. Capitalization of TSO/E commands and subcommands in a CLIST is also optional.

Formatting

You can use blank lines in a CLIST as a formatting aid, to separate parts of the CLIST and make the CLIST easier to read. Blank lines do not affect CLIST processing, except that a blank line after a continuation symbol ends continuation, unless the blank line is also continued.

Length

The maximum length of a CLIST statement is 32756 bytes.

Labels

You can prefix CLIST statements and TSO/E commands with a label. Other statements can use the label to pass control to the statement or command. Labels can consist of 1-31 alphanumeric characters (A-Z, 0-9, #, $, @, _)beginning with an alphabetic character (A-Z). The label can appear on the same line as the statement or command, or on the preceding line. A colon must immediately follow the label name. For example,

label: `IF A= ...`

or

label: `+`
`• IF A= ...`

Comments

You can include a comment:
- On a line by itself
- Before, in the middle of, or after a CLIST statement or TSO/E command.

You define a comment by coding a slash-asterisk (comment delimiter) followed by the descriptive text. If you include the comment before or in the middle of a CLIST statement or TSO/E command, you must end the comment with a closing comment delimiter (asterisk-slash). The following example shows a comment included before a CLIST statement:

```
/*get return code */ SET RC = &LASTCC
```

If you include a comment after a CLIST statement or TSO/E command, or on a line by itself, the closing comment delimiter is not needed, as shown in the following example:

```
alloc file(in) da(accounts.data) shr /* Input data set
```

If a comment appears after a CLIST statement or TSO/E command that continues on the following line, the comment must end with a closing comment delimiter and the continuation character must appear *after* the comment delimiter, as shown in the following example:

```
IF &LASTCC ¬= 0 THEN /* error occurred */ +
 DO ...
```

CLISTs can begin with a comment, but the first line of a CLIST must not be a comment containing the acronym REXX; if the first line contains "REXX" in any position, the EXEC command attempts to process the CLIST as a REXX exec. Note that comments can be in both uppercase and lowercase letters. Comments are unaffected by CLIST processing.

Characters supported in CLISTs

CLIST statements can process all data characters represented by hexadecimal codes 40 through FF. It should be noted that CLISTs translate lowercase letters to uppercase letters, unless controlled by NOCAPS or ASIS, and translate lowercase numbers (B0-B9) to standard numbers (F0-F9). CLISTs also support the following control characters:

Hexadecimal code	**Control character**
05	HT (horizontal tab)
0E	Shift Out (starting delimiter for DBCS data)
0F	Shift In (ending delimiter for DBCS data)
14	RES (restore)
16	BS (backscore)
17	IL (idle)
24	BYP (bypass)
25	LF (line feed).

All other hexadecimal codes from 00 to 3F are reserved for internal processing and can cause errors if they appear in CLIST data. The use of I/O statements to process

data sets containing these codes is not supported. For example, OBJ and LOAD type data sets contain unsupported characters and must not be used for CLIST I/O.

Note: Some characters supported in CLIST, such as { (X'C0') and } (X'D0'), cannot be written to the terminal because of TSO/E output processing. To write such characters to a terminal, create TSO/VTAM translate tables and invoke the tables with the TSO/E TERMINAL command. For more information about creating translate tables, see *z/OS TSO/E Customization*. For CLISTs executed under the TSO/E Session Manager, these restrictions do not apply.

TSO/E commands and JCL statements

You can include TSO/E commands and subcommands, and JCL statements in a CLIST as needed.

TSO/E commands

You can include TSO/E commands and subcommands (and user-written commands and subcommands) in a CLIST at any point where the specific functions (for example, allocate, free, and so on) are required. For certain applications, a CLIST might consist entirely of commands and subcommands. You can also substitute CLIST variables as operands in commands and subcommands, or as commands themselves. For more information about CLIST variables, see Chapter 4, “Using symbolic variables,” on page 17.

JCL statements

From a CLIST, you might want to submit a jobstream for execution. In the CLIST, you can include the required JCL statements (EXEC, DD, and so on). However, when you include the following JCL statements in a CLIST, you must use a particular CLIST function to prevent the CLIST from modifying the statements and causing subsequent JCL errors.

1. Statements following the SYSIN statement - use the &STR built-in function to preserve leading blanks and statements that have the same names as CLIST statements.
2. A statement containing a single ampersand (&) or a double ampersand (&&) - use the &SYSNSUB or &NRSTR built-in functions.
3. JCL comments - use the &STR built-in function. Because CLIST processing detects the JCL comment as a comment for the CLIST, you must set a variable equal to &STR(/*) and use this variable in place of the JCL comment.
4. JCL imbedded in a CLIST can use the SUBMIT * form of the SUBMIT command; however, *all* JCL is converted to uppercase. If JCL conversion to uppercase is inappropriate or undesirable, use the SUBMIT (dataset_name) form of the SUBMIT command. For a description of the SUBMIT command, see *z/OS TSO/E Command Reference*.

Examples of using these built-in functions with JCL are provided in Chapter 7, “Using built-in functions,” on page 53 and in Figure 12 on page 126.

Operators and expressions

Operators cause a CLIST to perform evaluations on data; the data can be numeric or character, or can be a variable or a built-in function. Operators fall into three categories: arithmetic, comparative, and logical, as shown in Table 2 on page 13.

- Arithmetic operators perform integer arithmeticon numeric operands. The operators connect integers, variables, orbuilt-in functions to form expressions, such as *4-2*.
- Comparative operators perform comparisonsbetween two expressions, to formcomparative expressions, such as *4-2=3*. The "=" is a comparative operator.

  The comparison produces a true or false condition. Comparative expressions are often used to determine conditional branching within a CLIST.
- Logical operators perform a logical comparison between the results of two comparative expressions, to form logical expressions, such as *&A=4 AND &B=&C*.The 'AND' is a logical operator.

  Logical expressions produce true or false conditions. Logical expressions are often used to determine conditional branching within a CLIST.

In Table 2, if more than one accepted value exists for an operator, the values are separated by commas.

Table 2. Arithmetic, comparative, and logical operators

	For the function:	**Enter:**
Arithmetic	Addition Subtraction Multiplication Division Exponentiation Remainder Prioritization the order of evaluation	+ - * / ** (See note 1) // () (See note 2)
Comparative	Equal Not equal Less than Greater than Less than or equal Not greater than Not less than	=,EQ ¬=,NE <,LT >,GT <=,LE >=,GE ¬>,NG ¬<,NL
Logical	And Or	AND,&& OR,|
Notes: 1. Negative exponents are handled as exponents of zero, thus the result is always set to 1. 2. Put parentheses around operations to give them priority in the order of evaluation.		

CLISTs try to perform evaluation wherever an operator is found, including the equal sign (=) in assignment statements. If you want CLISTs to treat operators as character data instead, use the &STR built-in function. For more information, see "Defining character data - &STR" on page 57.

Order of evaluations

A CLIST evaluates operations in the following default order. (Wherever more than one operation is listed below, the CLIST performs the operations sequentially, left to right in the order in which they appear on the CLIST statement.)

1. Exponentiation remainder
2. Multiplication, division
3. Addition, subtraction
4. Comparative operators
5. Logical AND
6. Logical OR

You can override the default order by placing parentheses around the operations you want executed first. For example, without any parentheses, the following example performs multiplication, division, then addition. The statement sets *X* to the value 24.

```
SET X = 4+5*8/2
```

By placing parentheses around *4+5*, you indicate to the CLIST that it should perform addition first and then proceed with the default order (multiplication, then division). The following statement sets *X* to the value 36.

```
SET X = (4+5)*8/2
```

You can place parentheses around expressions that are themselves enclosed in parentheses. This process is called nesting parenthesized expressions. The CLIST evaluates the deepest level of nesting first and proceeds outward until all nesting has been evaluated. In the following example, *X* is set to the value 7.

```
SET X=((1+4)*2+4)/2
```

The parentheses around *1+4* indicate that the CLIST should add these numbers before performing multiplication. The parentheses around the compound expression to the left of the division operator indicate that the CLIST should evaluate the compound expression before performing division.

In the preceding example, if you omit the outer-level parentheses, the CLIST performs division as the third operation (4/2) and sets *X* to the value 12:

```
SET X=(1+4)*2+4/2
```

Valid numeric ranges

The values of numeric variables must be integers in the range from -2,147,483,647 ($-2^{31}+1$) to +2,147,483,647 ($+2^{31}-1$).

A CLIST terminates and issues an error message in the following situations:

- You explicitly code a value outside the valid range.
- The evaluation of an expression produces an intermediate or final value outside the valid range.

The double-byte character set (DBCS)

The CLIST language allows data to contain characters of the double-byte character set. The double-byte character set (DBCS) is used in national languages such as Japanese and Korean which have more than 256 characters, the maximum number that can be represented with one byte of data. As the name implies, double-byte characters are each composed of two bytes, allowing a vastly increased number of characters.

DBCS delimiters

The CLIST language uses the hexadecimal codes X'0E' and X'0F' to distinguishdouble-byte characters from EBCDIC characters. The hexadecimal code X'0E' indicates the beginning of a string of DBCS characters, and the code X'0F' indicates the end of a DBCS string. Properly delimited DBCS character strings can be passed as character data in CLIST variables, in comments, and in the operands of CLIST statements.

This document commonly refers to the beginning and ending DBCS delimiters as *shift-out* and *shift-in* characters. In examples, this document uses the convention <d1d2> to represent DBCS strings enclosed in their shift-out and shift-in characters, where d1 and d2 each represent a DBCS character, < represents X'0E', and > represents X'0F'.

When DBCS strings are joined by continuation symbols, their contiguous shift-in and shift-out characters are removed to create a single DBCS string. For example:

```
SET A = ABC<d1d2> +
 <d3d4>DEF               /* result: &A = ABC<d1d2d3d4>DEF
```

DBCS restrictions

The following restrictions apply to DBCS data in CLISTs:

- DBCS data cannot appear in any names, including the names of variables, functions, statements, data sets, or labels.
- DBCS data cannot be used in variables or operands where numeric data is expected, nor in any arithmetic operations.

This document lists further DBCS considerations and restrictions wherever they apply.

Two CLIST built-in functions, &SYSONEBYTE and &SYSTWOBYTE, convert data between the DBCS and EBCDIC character sets. These functions are described in Chapter 7, “Using built-in functions,” on page 53.

Chapter 4. Using symbolic variables

The CLIST language includes several types of variables. This chapter describes how to use symbolic CLIST variables. Later chapters discuss other types of variables, including control variables and variables set by CLIST built-in functions.

What is a symbolic variable?

A symbolic variable is a string of characters that you define as a symbol. Because the variable is a symbol, you can assign different values to it at different times. By assigning different values, you can do the same processing with different data.

For example, you can use the SET statement to assign different values to a symbolic variable named PAY_RAISE:

```
SET PAY_RAISE = 20  /* Set the value of PAY_RAISE equal to 20
```

or

```
SET PAY_RAISE = 30  /* Set the value of PAY_RAISE equal to 30
```

You can use those different values of PAY_RAISE in the following equation, to calculate your total annual raise based on various weekly raises:

```
SET ANNUAL_RAISE = &PAY_RAISE * 52
```

In CLISTs, the ampersand (&) means "the value of." In the example above, the CLIST multiplies the value of PAY_RAISE (20 or 30) by 52 and assigns the resulting value to another variable, ANNUAL_RAISE. (In a SET statement, the ampersand is required on variables to the right of the equal sign, and is optional on variables to the left of the equal sign.)

When you execute a CLIST, it scans each line and replaces the symbolic variables with their actual values. This process is called symbolic substitution.

In a CLIST, you can use symbolic variables to include variable data on TSO/E commands and subcommands, on JCL statements, and on many of the CLIST statements.

Valid names of variables

You can define symbolic variables with meaningful names. Meaningful variable names, like PAY_RAISE, describe the contents of the variable and make CLISTs easy to read and maintain. Note that an ampersand (&) is not part of a variable name; it tells the CLIST to use the value of the variable. Follow these rules when naming a symbolic variable:

1. The first character must be one of the following: A-Z, (a-z), _, #, $, @.

   Note: The system recognizes the following hexadecimal codes for these characters: _ (X'6D'), # (X'7B'), $ (X'5B'), @ (X'7C'). In countries other than the U.S., these characters on a keyboard might generate different hexadecimal codes and cause an error. For example, in some countries the $ character might generate a X'4A'.
2. The remaining characters can be any of the above, and 0 through 9.

3. The variable name can be up to 252 characters in length (not counting the ampersand).
4. Variable names must not match the character equivalents of CLIST operators, such as “EQ” and “NE” (see Table 2 on page 13 for a list).
5. Special rules apply to the PROC statement. On PROC statements:
   - All variables must begin with A-Z, and be in uppercase only.
   - Names of keyword variables cannot contain the underscore (_), or be longer than 31 characters. For more information, see “Using the PROC statement” on page 19.
6. If variables are used on ISPF panels, they cannot exceed eight characters in length.
7. Do not use the names of *statements or their keywords* as variable names. This may cause unexpected results if used in a conditional statement, as in the following sequence:

   ```
   SET WHILE = &STR(ABC)
   DO UNTIL &WHILE = &STR(ABC) WHILE (&COUNT<5)
      SET &COUNT = &COUNT + 1
   END
   ```

   The results are also unpredictable if a keyword is used within a string, as in the following:

   ```
   SET COUNT = 0
   SET VAR = ABC
   DO UNTIL &VAR = &SUBSTR(3:3,WHILE) WHILE &COUNT < 5
    SET  COUNT =  &COUNT + 1
   END
   ```

Valid values of variables

The values of CLIST variables can generally include any characters you can enter on a keyboard. See “Characters supported in CLISTs” on page 11 for information about special characters.

Values of symbolic variables can be up to 32756 bytes long, minus the length of the CLIST statement that assigns the value. For example, if the assignment statement is six bytes long (SET A=), the value can contain 32750 bytes.

Defining symbolic variables and assigning values to them

There are several ways to define symbolic variables and assign values to them in a CLIST. Here are some basic methods:

- Use the SET statement to define variables and give them specific values.
- Use the READ statement to define variables and get their values from a user.
- Use the PROC statement to define variables and get their values from parameters passed to the CLIST.

The previous statements define variables explicitly. You can also define a variable implicitly by referring to it in a CLIST statement before you explicitly define it. The CLIST assigns a null value to such a variable.

Using the SET statement

You can use the SET statement to define a symbolic variable and assign a value to it. For example, to assign the character string JOHN to the variable NAME, code:

```
SET NAME=JOHN
```

The variable NAME *contains* the value JOHN.

You can also use the SET statement to assign an initial value to a variable, then increase or decrease the value as necessary. For example, to control a loop you can initialize a counter:

```
SET COUNTER = 1
```

For each execution of the loop, you can increment the counter:

```
SET COUNTER = &COUNTER + 1
```

In the SET statement, the ampersand is required when a variable appears in the expression on the right side of the equal sign, but is optional when a variable appears on the left side of the equal sign.

In addition to symbolic variables, you can also use CLIST control variables and built-in functions in SET statements. For information about control variables and built-in functions, see Chapter 6, "Using control variables," on page 29 and Chapter 7, "Using built-in functions," on page 53.

Using the READ statement

You can use the READ statement to define a variable and give it a value provided by the CLIST user. To prompt the user for input, issue a WRITE statement before the READ statement, for example:

```
WRITE What is your name?
READ &NAME;
```

The user sees the question "What is your name?" displayed on the terminal. The user's typed response, for example, JOHN, becomes the value of the variable NAME. Your CLIST can then use this value in subsequent statements, such as:

```
WRITE HELLO &NAME!      /* (result:  HELLO JOHN!)
```

For more information about the READ and WRITE statements, see Chapter 9, "Communicating with the terminal user," on page 85.

Using the PROC statement

The PROC statement lets you pass parameters to a CLIST at invocation. The PROC statement defines symbolic variables and assigns the parameters to the variables. To do so, the PROC statement must be the first functional line of the CLIST (only comments or blank lines can precede the PROC statement).

Passing parameters to a PROC statement

When invoking a CLIST explicitly, pass parameters in single quotation marks, for example:

```
EX clistname 'parm1 parm2(value)'
```

When invoking the CLIST implicitly, omit the quotation marks:

```
%clistname parm1 parm2(value)
```

To pass parameters that contain single quotation marks, you must follow special rules that are discussed in *z/OS TSO/E Command Reference*.

The PROC statement accepts two types of parameters: positional parameters and keyword parameters. Parameter values in lowercase are changed to uppercase.

Using PROC with positional parameters

You can use the PROC statement to assign parameters to variables by position. First, type a number on the PROC statement telling how many positional parameters to expect (type `0` if none). Then specify the variables that you want to use. For example, in the following PROC statement, the number 1 tells the CLIST to assign the first parameter it receives to the variable NAME.

```
PROC 1 NAME
```

Thus, if you invoke the CLIST with the parameter JOE:

```
EX clistname 'JOE'
```

the variable NAME contains the value JOE.

Suppose you wanted the PROC statement to assign a second parameter to the variable ADDRESS. You can write the statement as follows:

```
PROC 2 NAME ADDRESS
```

The invoker must know the correct order in which to pass positional parameters, and must pass as many as you specify by number on the PROC statement. If the invoker doesn't pass a positional parameter as expected, the CLIST prompts for it. Positional parameters can have up to 252 characters (A-Z, 0-9, #, $, @, _).

Using PROC with keyword parameters

When input parameters are optional or can have default values, use the PROC statement to assign the parameters to variables by name rather than by position. Such parameters (*keyword parameters*) must match a variable name that you specify on the PROC statement. See item 5 on page 18 for special rules on naming variables specified on the PROC statement. The PROC statement can accept keyword parameters with or without values.

Keyword parameters and their matching variables have up to 31 alphanumeric characters (A-Z, 0-9, #, $, @). Keyword parameter *values* have the same length restriction as symbolic variable values: 32768 bytes.

Keywords with values

If a CLIST has a value that applies to most but not all uses of the CLIST, you can provide a default value and allow invokers to override it with a keyword parameter.

In the following example, the 0 tells the CLIST to expect no positional parameters. (If there are no positional parameters, a zero is required.) The notation STATE(NY) gives the variable STATE the default value of NY.

```
PROC 0 STATE(NY)
```

The invoker can override the default value by passing the keyword parameter with another value, for example:

```
EX clistname 'STATE(NJ)'
```

or

```
%clistname STATE(NJ)
```

Then the variable STATE takes the value NJ.

If you want a variable to have no default value but allow invokers to specify a value, use empty parentheses. The following PROC statement lets invokers pass keyword parameters such as STATE(NY) or STATE(NJ).

```
PROC 0 STATE()
```

In the example above, if an invoker passes the keyword parameter STATE without a value, the CLIST prompts for the value. If a invoker does not pass the keyword STATE at all, the variable STATE takes a null value.

Keywords without values

You can use keyword parameters without values to let invokers specify a CLIST option. For example, to let an invoker tell a CLIST to print its results, you can code the following:

```
PROC 0 PRINT
```

Then, if the invoker passes the keyword parameter PRINT:

```
EX clistname 'PRINT'
```

the variable PRINT takes the value PRINT. If the invoker does not pass the parameter PRINT, the variable PRINT takes a null value. Your CLIST can test the value to see if the invoker wants the print option. You can code this test using an IF-THEN-ELSE sequence:

```
PROC 0 PRINT
IF &PRINT = PRINT THEN (print results) /* If the value of PRINT = print ...*/
ELSE ...
```

(For more information about the IF-THEN-ELSE sequence, see "The IF-THEN-ELSE sequence" on page 67.)

Using PROC with both positional and keyword parameters

The following PROC statement receives both positional and keyword parameters:

```
PROC 2 NAME ADDRESS STATE(NY) ZIP() PRINT
```

The number 2 indicates that the invoker must pass positional parameters for the first two variables, NAME and ADDRESS. Invokers can also pass keyword parameters with values for the variables STATE (the default value is NY) and ZIP (which has no default). In addition, invokers can pass the keyword parameter PRINT without a value, to specify a print option.

Examples

The following CLIST addresses a memo based on PROC variables, displaying the address at the terminal. You can also use I/O statements (described in Chapter 10, "Performing file I/O," on page 97) to write the address to a data set.

```
/*******************************
/* Memo-addressing CLIST
/*******************************
PROC 2 NAME ADDRESS STATE(NY) ZIP()
WRITE   TO: &NAME
WRITE   AT: &ADDRESS
WRITE       &STATE &ZIP
```

Assume that the CLIST resides in the member MEMO of a partitioned data set called PROC.CLIST. If you invoked it as follows:

```
ex proc(memo) 'Perry_Gordon 22_Oak_St._Pokville ZIP(10101)'
```

You can see the following output at your terminal:

```
TO: PERRY_GORDON
AT: 22_OAK_ST._POKVILLE
    NY 10101
```

If you invoked it without parameters, for example,

```
ex proc(memo)
```

the CLIST will prompt you for a name and address. The state will default to NY, and there will be no zip code.

The following CLIST issues the LISTDS command using the PROC, READ, and SET statements to define variables and assign values to them.

```
/*********************************************************************/
/* This CLIST issues the LISTDS command, using a data set name and   */
/* any options requested by the user.  If the user enters OPTIONS    */
/* as a parameter, READ and WRITE statements prompt for the options. */
/* The CLIST gets a LISTDS return code from the &LASTCC control      */
/* variable, and writes the return code to the screen.               */
/*********************************************************************/
PROC 1 DATASET OPTIONS          /* Get a data set name            */
IF &OPTIONS = OPTIONS THEN      /* If the user wants options, */ +
  DO                            /* prompt for input               */
    WRITE Type LISTDS options (MEMBER, HISTORY, or STATUS)
    READ OPT
  END
LISTDS &DATASET &OPT            /* List data set with any options */
SET RETURN_CODE = &LASTCC       /* Get return code from LISTDS    */
WRITE RETURN CODE WAS &RETURN_CODE
```

More advanced uses of variables

The previous sections of this chapter discussed several basic ways to define and assign values to symbolic variables, using the SET, READ, and PROC statements. Other chapters describe how to use symbolic variables in more advanced applications with other CLIST statements:

- The GLOBAL, NGLOBAL, SYSCALL, and SYSREF statements let you define variables for use in nested CLISTs and CLIST subprocedures. See Chapter 8, “Structuring CLISTs,” on page 67 for information about using variables with these statements.
- The I/O statements OPENFILE, CLOSEFILE, GETFILE, and PUTFILE use symbolic variables to send and receive input between files. See Chapter 10, “Performing file I/O,” on page 97 for information about using variables with these statements.
- The LISTDSI statement uses a special set of CLIST variables for retrieving information about data set attributes. See Chapter 6, “Using control variables,” on page 29 for information about this statement and its variables.

Combining symbolic variables

You can combine one symbolic variable with another symbolic variable to form a compound variable.

Suppose a CLIST invokes programs that reside in nine data sets named PROGRAM1 through PROGRAM9. By combining &PROGRAM and &I; you can use the iterative DO loop structure to invoke PROGRAM1 through PROGRAM9 as follows:

```
SET PROGRAM = PROGRAM
DO &I = 1 to 9
 call mylib(&PROGRAM&I)
END
```

(For more information about using an iterative DO loop, see "The Iterative DO sequence" on page 73.) By increasing the value of I from one to nine in a loop, a CLIST can invoke the following set of programs without having to modify the CALL command.

```
PROGRAM1
PROGRAM2
⋮
PROGRAM9
```

You can also combine symbolic variables and character strings. When the variable precedes the character string, place a period after the symbolic variable to distinguish it from the character string:

```
&PROGRAM.A
```

No period is required when the character string precedes the symbolic variable because the ampersand distinguishes the variable from the string:

```
A&PROGRAM
```

Using a variable to preserve leading spaces in a CLIST

When TSO/E processes a job in a CLIST, statements following the *DD ** statement are left adjusted to column 1, thereby removing leading spaces. (This is unique to CLIST processing and is not a batch concern.) If you need to preserve the blanks, set a variable to a single blank or a string of blanks to provide as many blanks as required, that is *&STR( )* and precede all statements following the *DD ** with that variable. The following example shows how to include the variable within your CLIST.

```
PROC01
CONTROL
SET &A = STR(   )
SUBMIT * END(XX)
//JOBCARD
//OTHER
//JCL
//CARDS
//   DD *
&A   COPY ...
&A   ...
&A   ...
&A   ...
```

Increasing the amount of storage available for variables

You can increase the amount of storage that CLIST can use for variables by allowing the CLIST variable pool to reside in storage above the 16MB line. Use the new PROFILE option VARSTORAGE(HIGH/LOW) to indicate whether variables can use storage above the 16MB line (HIGH) or below the 16MB line (LOW). Using VARSTORAGE(HIGH) means the system can trap more variables.

See the PROFILE command in *z/OS TSO/E Command Reference* for more information.

Nesting symbolic variables

In some situations, you might want to store the name of a variable in another variable. For example, if you had to process two variables in the same way, you can assign their names to a third variable.

When you store the name of a variable in another variable, you are "nesting" variables.

To nest one variable in another variable, use an assignment statement with double ampersands. For example, to nest the variable &CAT in the variable &MAMMAL, code:

```
SET MAMMAL = &&CAT              /* result: &MAMMAL contains &CAT */
```

The double ampersands (&&) prevent the CLIST from performing symbolic substitution on the variable string &CAT. In the assignment statement, the CLIST removes only the first ampersand, setting &MAMMAL to the value &CAT.

It is most useful to nest variables when you have to process many variables that have similar names. For example, if you have to set &VARIABLE to different variables such as &LINE1, &LINE2, during processing, you can code many SET statements, or code the following sequence:

```
SET NUMBER=0
SET VARIABLE=&&LINE&NUMBER      /* Initialize &VARIABLE to &LINE0 */
DO WHILE &NUMBER<8              /* Process from &LINE1-&LINE8     */
 SET NUMBER = &NUMBER+1         /* Increase &NUMBER to create next
                                /*                  variable name */
 SET VARIABLE=&&LINE&NUMBER     /* Set &VARIABLE to next variable
                                /*                           name */
 (processing)
END
```

For more examples of using nested variables, see "&SYSOUTLINE" on page 47, and "Allocating data sets to SYSPROC - the SPROC CLIST" on page 133.

If you nest variables whose values contain double ampersands, the outermost variable contains the name of the innermost variable. For example, after the following statements execute, VARIABLE contains &LINE1 and DATA contains the value 430.

```
SET LINE1=430
SET NUMBER=1
SET VARIABLE=&&LINE&NUMBER
SET DATA=&VARIABLE
```

Combining nested variables with character strings

As previously stated, you can combine a preceding variable with a character string by placing a period between them (&PROGRAM.A). If the preceding variable is nested, place an additional period after the variable for each level of nesting. For example,

```
SET &BUDGET = June
SET &PROGRAM = &budget
call mylib(&PROGRAM.A)          /* result: call mylib(JuneA)
```

If the character string precedes the variable, no period is required:

```
SET &BUDGET = June
SET &PROGRAM = &budget
call mylib(A&PROGRAM)           /* result: call mylib(AJune)
```

Substitution of nested variables

If a CLIST encounters nested symbolic variables in a line, it normally scans the line (performs symbolic substitution) multiple times until all symbolic variables are resolved. For example:

```
SET A = 50
SET B = &&C                    /* result:  &B contains &C
SET C = &A+50                  /* result:  &C contains 100
SET D = &&A                    /* result:  &D contains &A
SET X = (&D+&B)/&D             /* result:  &X contains 3
```

To resolve the fifth expression the CLIST uses the values assigned to the symbolic variables A-D and assigns the value 3 to X.

You can limit the number of times the CLIST scans a line of nested variables, using the &SYSNSUB built-in function. For example, you can specify that the CLIST scan the fifth expression in the preceding example only once, so the variables were resolved to only one level of symbolic substitution. As a result, the CLIST needs to resolve &X from (&D+&B)/&D to (&A+&C)/&A, and go no further. See Chapter 7, “Using built-in functions,” on page 53 for a description and examples of &SYSNSUB.

Combining variables containing DBCS data

When variables containing data of the double-byte character set (DBCS) are combined with other DBCS data, contiguous DBCS delimiters are removed to create a single DBCS string. For example:

```
SET A = <d1d2>
SET B = <d3d4>&A<d5d6>         /* result:  &B = <d3d4d1d2d5d6>
```

Chapter 5. Using keyword names

Using keyword names as variables or labels within a CLIST

You can sometimes use KEYWORD names such as IF, THEN, ELSE, SELECT, WHEN, OTHERWISE, GT, LE, etc. as variables or labels within a CLIST without a problem as long as its usage context is not ambiguous and does not infer its reserved usage. However, it is strongly recommended that you not use CLIST statement names, built-in function names, or CLIST control variable names for anything other than their intended use. Doing so might be allowable in a given context, but it could be confusing to anyone else trying to maintain such a CLIST over time. For clarity, usage of CLIST instruction or statement names, CLIST built-in function names, or CLIST built-in variable names should be avoided for any usage other than its predefined, intended use for those named entities.

CLIST instructions or statements are documented in Chapter 14, “Reference,” on page 145 and include:

- ATTN
- CLOSFILE
- CONTROL
- DATA-ENDDATA
- DATA PROMPT-ENDDATA
- DO
- END
- ERROR
- EXIT
- GETFILE
- GLOBAL
- GOTO
- IF-THEN-ELSE
- LISTDSI
- NGLOBAL
- OPENFILE
- PROC
- PUTFILE
- READ
- READDVAL
- RETURN
- SELECT
- SET
- SYSCALL
- SYSREF
- TERMIN AND TERMING
- WRITE AND WRITENR

CLIST built-in function names like those documented in Chapter 7, “Using built-in functions,” on page 53 include:

- &DATATYPE
- &EVAL
- &LENGTH
- &NSTR
- &STR
- &SUBSTR

- &SYSCAPS
- &SYSCLENGTH
- &SYSCSUBSTR
- &SYSDSN
- &SYSINDEX
- &SYSLC
- &SYSNSUB
- &SYSONEBYTE
- &SYSTWOBYTE

SPECIAL variables and CLIST control variables are discussed in Chapter 6, “Using control variables,” on page 29, and listed in Table 3 on page 29, Table 4 on page 31, and Table 5 on page 31. These include variables beginning with &SYS... and the special variables &LASTCC and &MAXCC.

Note: Generally, a user should not use local defined CLIST variable or label names that begin with "SYS..." in order to avoid any possible conflict with current or future CLIST control variables or statements of the same name.

Chapter 6. Using control variables

The CLIST language includes a set of control variables. Control variables provide information about MVS™, TSO/E, and the current session, such as levels of software available, the time of day, and the date. Your CLISTs can use the control variables to obtain such current information.

You code a control variable as you need a symbolic variable. For example, to get the time of day, your CLIST can use the control variable &SYSTIME as follows:

```
WRITE It's &SYSTIME
```

If your CLIST was executing at 2:32:58 PM, the result need to be:

```
It's 14:32:58
```

You do not have to define control variables. Control variables have constant names; you refer to the variable name to obtain information.

Control variables to which you can assign values are called *modifiable* control variables. The variable &SYSOUTTRAP is an example of a modifiable control variable. &SYSOUTTRAP tells how many lines of TSO/E command output should be saved in a CLIST. If you want to save 100 lines of output from each TSO/E command in your CLIST, you can set &SYSOUTTRAP to 100, as follows:

```
SET &SYSOUTTRAP = 100
```

Your CLIST need then be able to retrieve and process up to 100 lines of output from each command in the CLIST. If you did not want to save output from some commands, you need to reset &SYSOUTTRAP to zero before issuing those commands.

Overview of using control variables

Table 3 lists the control variables in related categories, and indicates what page they are on, whether they are modifiable, and whether they are retrievable by the variable access routine, IKJCT441. For more information about IKJCT441, refer to *z/OS TSO/E Programming Services*.

Table 4 on page 31 briefly describes the modifiable control variables, and Table 5 on page 31 briefly describes the control variables you cannot modify.

Table 3. Control variable by category

Category	Variable	Modifiable	Retrievable by IKJCT441	Reference
Current date and time	&SYSDATE	No	No	&SYSDATE
	&SYSJDATE	No	No	&SYJDATE
	&SYSSDATE	No	No	&SYSSDATE
	&SYS4DATE	No	No	&SYS4DATE
	&SYS4JDATE	No	No	&SYS4JDATE
	&SYS4SDATE	No	No	&SYS4SDATE
	&SYSTIME	No	No	&SYSTIME
	&SYSSTIME	No	No	&SYSSTIME

1. Lets you test or modify the CLIST CONTROL statement values.

Table 3. Control variable by category (continued)

Category	Variable	Modifiable	Retrievable by IKJCT441	Reference
Terminal-related	&SYSLTERM &SYSWTERM	No No	No No	&SYSLTERM &SYSWTERM
User-related	&SYSUID &SYSPREF &SYSPROC	No No No	No No No	"&SYSUID" on page 35 "&SYSPREF" on page 35 "&SYSPROC" on page 36
System-related	&SYSAPPCLU &SYSCLONE &SYSCPU &SYSDFP &SYSHSM &SYSISPF &SYSJES &SYSLRACF &SYSMVS &SYSNAME &SYSNODE &SYSOPSYS &SYSRACF &SYSPLEX &SYSSECLAB &SYSSMFID &SYSSMS &SYSSRV &SYSSYMDEF &SYSTERMID &SYSTSOE	No No No No No No No No No No No No No No No No No No No No No	No No No No No No No No No No No No No No No No No No No No No	"&SYSAPPCLU" on page 39 "&SYSCLONE" on page 36 &SYSCPU "&SYSDFP" on page 37 "&SYSHSM" on page 37 "&SYSISPF" on page 38 "&SYSJES" on page 38 "&SYSLRACF" on page 38 "&SYSMVS" on page 39 "&SYSNAME" on page 39 "&SYSNODE" on page 40 "&SYSOPSYS" on page 40 "&SYSRACF" on page 40 "&SYSPLEX" on page 41 "&SYSSECLAB" on page 41 "&SYSSMFID" on page 41 "&SYSSMS" on page 41 &SYSSRV "&SYSSYMDEF" on page 42 "&SYSTERMID" on page 34 "&SYSTSOE" on page 42
CLIST-related	&SYSSCAN &SYSENV &SYSICMD &SYSPCMD &SYSSCMD &SYSNEST	Yes No No No No No	Yes No No No No No	"&SYSSCAN" on page 43 "&SYSENV" on page 42 "&SYSICMD" on page 43 "&SYSPCMD" on page 43 "&SYSSCMD" on page 43 "&SYSNEST" on page 44
CLIST-related [1]	&SYSPROMPT &SYSSYMLIST &SYSCONLIST &SYSLIST &SYSASIS &SYSMSG &SYSFLUSH	Yes Yes Yes Yes Yes Yes Yes	Yes Yes Yes Yes Yes Yes Yes	"&SYSPROMPT" on page 44 "&SYSSYMLIST" on page 44 "&SYSCONLIST" on page 44 "&SYSLIST" on page 45 "&SYSASIS" on page 45 "&SYSMSG" on page 45 "&SYSFLUSH" on page 45
Input-related	&SYSDLM &SYSDVAL	No Yes	Yes Yes	"&SYSDLM" on page 46 "&SYSDVAL" on page 46
Output-related	&SYSOUTTRAP &SYSOUTLINE	Yes Yes	Yes Yes	"&SYSOUTTRAP" on page 47 "&SYSOUTLINE" on page 47
Return codes	&LASTCC &MAXCC	Yes Yes	Yes Yes	"&LASTCC" on page 49 "&MAXCC" on page 50
	&SYSABNCD &SYSABNRC &SYSCMDRC	Yes Yes Yes	Yes Yes Yes	"Getting results of the TSOEXEC command" on page 50

Table 4. Modifiable control variables (alphabetically)

Modifiable variable	Contents
&LASTCC	Contains the return code from the last operation (TSO/E command, subcommand, or CLIST statement).
&MAXCC	Contains the highest return code issued up to this point in the CLIST or the highest passed back from a nested CLIST.
&SYSABNCD	Contains the ABEND code returned by the command most recently invoked by the TSOEXEC command.
&SYSABNRC	Contains the ABEND reason code returned by the command most recently invoked by the TSOEXEC command.
&SYSASIS	ON specifies CONTROL NOCAPS/ASIS. OFF specifies CONTROL CAPS.
&SYSCMDRC	Contains the command return code returned by the command most recently invoked by the TSOEXEC command.
&SYSCONLIST	ON specifies CONTROL CONLIST. OFF specifies CONTROL NOCONLIST.
&SYSDVAL	(1) Contains the input line supplied by the user when the user returned control to the CLIST after a TERMIN or TERMING statement. (2) Contains the input line supplied by the user after a READ statement without operands. (3) Contains the value after the execution of a SET SYSDVAL=.
&SYSFLUSH	ON specifies CONTROL FLUSH. OFF specifies CONTROL NOFLUSH.
&SYSLIST	ON specifies CONTROL LIST. OFF specifies CONTROL NOLIST.
&SYSMSG	ON specifies CONTROL MSG. OFF specifies CONTROL NOMSG.
&SYSOUTLINE	Contains the number of lines of command output produced by a TSO/E command; points to the CLIST variables containing the output.
&SYSOUTTRAP	Contains the maximum number of lines of TSO/E command output to be saved.
&SYSPROMPT	ON specifies CONTROL PROMPT. OFF specifies CONTROL NOPROMPT.
&SYSSCAN	Contains the maximum number of times a CLIST can rescan a line to evaluate variables. The default is 16 times. The maximum value is +2,147,483,647. The minimum is 0.
&SYSSYMLIST	ON specifies CONTROL SYMLIST. OFF specifies CONTROL NOSYMLIST.

Table 5. Non-modifiable control variables (alphabetically)

Non-modifiable variable	Contents
&SYSAPPCLU	Contains the APPC/MVS logical unit (LU) name.
&SYS4DATE	Contains the current date in the form: *month/day/year*, where *year* is presented as four-digit number.
&SYS4JDATE	Contains the Julian date in the form: *year.days*, where *year* is presented as four-digit number.
&SYS4SDATE	Contains the date in the form: *year/month/day*, where *year* is presented as four-digit number.
&SYSCLONE	Contains the MVS system symbol representing its system name.

Table 5. Non-modifiable control variables (alphabetically) (continued)

Non-modifiable variable	Contents
&SYSCPU	Contains the number seconds of CPU time used during the session in the form: *seconds.hundredths_of_seconds*
&SYSDATE	Contains the current date in the form: *month/day/year*
&SYSDFP	Contains the level of DFSMSdfp in the operating system.
&SYSDLM	Contains the input line supplied by the user to return control to the CLIST after a TERMIN or TERMING statement.
&SYSENV	Indicates whether the CLIST is executing in the foreground or background environment.
&SYSHSM	Indicates the level of the Data Facility Storage Management Subsystem Hierarchical Storage Manager (DFSMShsm).
&SYSICMD	Contains the name by which the invoker implicitly invoked this CLIST. (This value is null if the invoker explicitly invoked the CLIST.)
&SYSISPF	Indicates whether ISPF dialog management services are available to the CLIST.
&SYSJDATE	Contains the Julian date in the form: *year.days*
&SYSJES	Contains the name and the level of the JES installed.
&SYSLRACF	Indicates the level of RACF® available to the CLIST. (See &SYSRACF below)
&SYSLTERM	Contains the number of lines available for applications on your terminal screen.
&SYSMVS	Contains the level of the base control program (BCP) component of z/OS.
&SYSNAME	Contains the system's name your CLIST is running on, as specified on the SYSNAME statement in SYS1.PARMLIB member IEASYSxx.
&SYSNEST	Indicates whether the currently executing CLIST was invoked by another CLIST.
&SYSNODE	Contains the network node name of your installation's JES.
&SYSOPSYS	Contains the z/OS name, version, release, modification level, and FMID.
&SYSPCMD	Contains the name (or abbreviation of the name) of the most recently executed TSO/E command in this CLIST.
&SYSPLEX	Contains the MVS sysplex name as found in the COUPLExx or LOADxx member of SYS1.PARMLIB.
&SYSPREF	Contains the prefix that TSO/E uses to fully qualify data set names.
&SYSPROC	Contains the name of the logon procedure used when the TSO/E user logged on.
&SYSRACF	Indicates whether RACF is installed and available to the CLIST.
&SYSSCMD	Contains the name of the most recently executed subcommand.
&SYSSDATE	Contains the date in the form: *year/month/day*
&SYSSECLAB	Contains the security label (SECLABEL) name of the TSO/E session.
&SYSSMFID	Identifies the system on which System Management Facilities (SMF) is active.
&SYSSMS	Indicates whether DFSMS/MVS is available to your CLIST.

Table 5. Non-modifiable control variables (alphabetically) (continued)

Non-modifiable variable	Contents
&SYSSRV	Contains the number of System Resource Manager (SRM) service units used during the session.
&SYSSYMDEF	Contains the symbolic name of the MVS system.
&SYSTERMID	Contains the terminal ID of the terminal where the CLIST has been started.
&SYSSTIME	Contains the time of day in the form: *hours*:*minutes*
&SYSTIME	Contains the time of day in the form: *hours*:*minutes*:*seconds*
&SYSTSOE	Indicates the level of TSO/E installed in the form: *version release modification_number*
&SYSUID	Contains the user ID under which the current session is logged.
&SYSWTERM	Contains the width of the screen.

Getting the current date and time

The following control variables provide information related to the current date and time. You cannot modify any of them with an assignment statement.

&SYSDATE, &SYSSDATE, and &SYSJDATE

Three variables provide the current date. Note that these variables return the current year as a two-digit number. In support of dates equal or greater than 2000, another set of variables is provided that returns the current year as four-digit number.

&SYSDATE provides the date in the American standard form: *month*/*day*/*year*. If executed on June 9, 2001, the following statement displays the message "Today is 06/09/01":

```
WRITE Today is &SYSDATE
```

&SYSSDATE provides the date in a form that can be sorted: *year*/*month*/*day*. If executed on June 9, 2001, the following statement displays the message "Today is 01/06/09":

```
WRITE Today is &SYSSDATE
```

&SYSJDATE provides the date in the Julian form: *year*.*days*. If executed on June 9, 2001, the following statement displays the message "Today is 01.160":

```
WRITE Today is &SYSJDATE
```

&SYSDATE and &SYSSDATE provide data that contain slashes. As a result, when they appear in expressions on comparative and assignment statements, enclose them in &STR built-in functions. For example, in the following example &SYSDATE appears in a statement containing comparative expressions; therefore, enclose it in a &STR built-in function. However, the use of &STR is unnecessary on the WRITE statement.

```
IF &STR(&SYSDATE) = &STR(06/09/01) THEN +
 WRITE On &SYSDATE, the system was down for &TMIN minutes.
```

&SYS4DATE, &SYS4SDATE, and &SYS4JDATE

Three variables provide the current date in a format that presents years as four-digit numbers. As opposed to the variables that present the current year as two-digit numbers, these variables are capable to handle years beyond 1999.

&SYS4DATE provides the date in the American standard form: *month/day/year*. If executed on November 22, 2001, the following statement displays the message "Today is 11/22/2001":

```
WRITE Today is &SYS4DATE
```

&SYS4SDATE provides the date in a form that can be sorted: *year/month/day*. If executed on November 22, 2001, the following statement displays the message "Today is 2001/11/22":

```
WRITE Today is &SYS4SDATE
```

&SYS4JDATE provides the date in the Julian form: *year.days*. If executed on November 22, 2001, the following statement displays the message "Today is 2001.326":

```
WRITE Today is &SYS4JDATE
```

&SYS4DATE and &SYS4SDATE provide data that contain slashes. As a result, when they appear in expressions on comparative and assignment statements, enclose them in &STR built-in functions. For example, in the following example &SYS4DATE appears in a statement containing comparative expressions; therefore, enclose it in a &STR built-in function. However, the use of &STR is unnecessary on the WRITE statement.

```
IF &STR(&SYS4DATE) = &STR(11/22/2001) THEN +
 WRITE On &SYS4DATE, the system was down for &TMIN minutes.
```

&SYSTIME and &SYSSTIME

Two variables provide the current time of day.

&SYSTIME provides the time in the form: *hours*:*minutes*:*seconds*. If executed at 2:32 and 58 seconds p.m., the following statement displays the message "It's 14:32:58":

```
WRITE It's &SYSTIME
```

&SYSSTIME provides a shortened version of &SYSTIME, in the form: *hours*:*minutes*. If executed at 2:32 and 58 seconds p.m., the following statement displays the message "It's 14:32":

```
WRITE It's &SYSSTIME
```

Getting terminal characteristics

Three control variables provide information about the terminal to which the user is logged on.

&SYSTERMID

&SYSTERMID contains the terminal ID of the terminal where the CLIST has been started. For example,

```
PROC 0
WRITE &SYSTERMID
EXIT
```

may return a terminal ID of M02XA06R, with a maximum length of eight characters. Trailing blanks are removed.

If your CLIST runs in the background, the &SYSTERMID control variable returns a null string.

&SYSLTERM and &SYSWTERM

&SYSLTERM provides the number of lines available for applications on your terminal screen. &SYSWTERM provides the width of the screen.

&SYSLTERM and &SYSWTERM can be used when a CLIST reformats the screen using Session Manager commands. For example, a CLIST called HORZNTL splits the terminal screen horizontally based on the number of lines on the screen and its width. The following section of HORZNTL substitutes the control variables in the Session Manager commands that define the windows for the reformatted screen. By using &SYSLTERM and &SYSWTERM instead of explicit screen positions, HORZNTL makes optimal use of the space available on a given screen.

```
SET LINE = (&SYSLTERM-5)/2
SET TOPS = &LINE-1;
SET BOT = &LINE+1;
SET BOTS = (&SYSLTERM-1)-&BOT
SET BOTSX = (&SYSLTERM-3)-&BOT
smput /save screen;save.pfk;+
       save.win main;save.win line;save.win current;+
       del.win main;del.win line;del.win current;+
       define.window main 1 1 &TOPS &SYSWTERM;+
       define.window line &LINE 1 1 &SYSWTERM;+
       define.window current &BOT 1 &BOTS &EVAL(&SYSWTERM-18)/
```

Getting information about the user

Three control variables provide user-related information including the current user ID, logon procedure, and data set prefix.

&SYSUID

&SYSUID provides the user ID under which the current TSO/E session is logged on. Use this variable in messages and wherever logic depends on, or references, the user ID. For example, the following message displays information about how the CLIST is invoked:

```
WRITE CLIST invoked by user &SYSUID at &SYSTIME on &SYSSDATE
```

&SYSPREF

&SYSPREF provides the current data set name prefix that is prefixed to non-fully-qualified data set names. The PROFILE command controls this prefix. Use &SYSPREF when you want to allocate data sets that are unique to the user who invoked the CLIST. For example, the following ALLOCATE command allocates unique data sets for invokers of a CLIST containing the command:

```
alloc da('&SYSPREF..records.data') shr reuse
```

Two periods are required between &SYSPREF and RECORDS; the first indicates the end of the variable name, and the second is part of the text to be concatenated. After substitution, the command has the following form:

```
alloc da('prefix.records.data') shr reuse
```

&SYSPROC

&SYSPROC provides the name of the logon procedure used when the user logged on to the current TSO/E session. You can use &SYSPROC to determine whether programs, such as Session Manager, are available to the user. For example, before invoking the CLIST (HORZNTL) that reformats the screen using Session Manager commands, verify that Session Manager is active. One way to make the verification is to check the logon procedure as follows:

```
IF &STR(&SYSPROC) = SMPROC THEN +
 %horzntl
ELSE +
 DO
  WRITE Your screen cannot be reformatted.
  WRITE Log on using SMPROC as logon proc.
 END
```

&SYSPROC provides the following values:

- When the CLIST is invoked in the foreground (&SYSENV provides 'FORE'), &SYSPROC will provide the name of the current LOGON procedure.
- When the CLIST is invoked in batch (from a job submitted through the SUBMIT command), &SYSPROC will provide the value 'INIT', which is the ID for the initiator.
- When the CLIST is invoked from a Started Task (an address space that is started through the Start operator command), &SYSPROC will provide the ID of the started task. If 'S procname' is issued from the operator console, &SYSPROC will provide the value 'procname'.

Getting information about the system

The following control variables provide information about the system environment under which the CLIST is executing.

You can use these control variables in your CLISTs for different purposes. For example, the variables &SYSNAME, &SYSPLEX, &SYSCLONE, and &SYSSYMDEF allow you to write common CLISTs that are to run in a sysplex environment. You can build or identify the system-specific data set names by using the values returned by these control variables.

&SYSCLONE

&SYSCLONE returns the MVS system symbol representing its system name. It is a 1- to 2-byte shorthand notation for the system name. The value is obtained from SYS1.PARMLIB member IEASYMxx [2]. For example, if `SYSCLONE(A1)` is specified in IEASYMxx, then

```
PROC 0
WRITE &SYSCLONE
EXIT
```

returns a value of `A1`. A null string is returned if no MVS SYSCLONE ID is specified in IEASYMxx.

&SYSCPU and &SYSSRV

&SYSCPU provides the number of seconds of central processing unit (CPU) time used during the session in the form: *seconds.hundredths_of_seconds*.

2. Introduced with MVS/ESA SP 5.2; provides a mechanism to assign system substitution symbols names and values.

&SYSSRV provides the number of System Resource Manager (SRM) service units used during the session.

These variables can be used for measuring the performance of applications and reporting session duration to the user.

For example, to measure the performance of an application invoked from a CLIST, you can code the following:

```
SET CPU = &SYSCPU
SET SRV = &SYSSRV
call mylib(payroll) '50,84'
SET CPU = &STR(&SYSCPU-&CPU)
SET SRV = &STR(&SYSSRV-&SRV)
call mylib(calc) '&STR(&CPU),&STR(&SRV)' /* Measure performance */
.
.
.
                      /* Do calculations and pass back results */
.
.
.
WRITE &CPU &SRV
```

The user can then see the number of seconds of CPU time and SRM service units used by the program PAYROLL.

&SYSDFP

&SYSDFP contains the level of DFSMSdfp in the operating system. For example,

```
PROC 0
WRITE &SYSDFP
EXIT
```

might return a value of `03.01.10.00`. That represents z/OS Version 1, Release 10, Modification Level 0. The value returned is in the format *cc.vv.rr.mm*, where *cc* is a product name code, *vv* the version, *rr* the release number, and *mm* the modification level. All values are two-digit decimal numbers.

For each level of the operating system, only one value for &SYSDFP is possible.

These are the values for the *cc* code that have been used in earlier operating systems:

00 MVS/XA DFP Version 2 or MVS/DFP Version 3 running with MVS/SP on MVS/XA or MVS/ESA.

01 DFSMSdfp in DFSMS/MVS running with MVS/SP on MVS/ESA or OS/390®.

02 DFSMSdfp in OS/390 Version 2 Release 10, or in z/OS Version 1 Release 1 or z/OS Version 1 Release 2. All three releases returned "02.02.10.00".

03 DFSMSdfp in z/OS Version 1 Release 3 or later.

&SYSHSM

When DFSMShsm (Data Facility Storage Management Subsystem Hierarchical Storage Manager) is active, &SYSHSM indicates its level. When DFSMShsm is not active, &SYSHSM contains a null string.

&SYSHSM contains a character string of four two-digit decimal numbers separated by periods. The value returned is in the format *cc.vv.rr.mm*, where *cc* is a product name code, *vv* the version, *rr* the release number, and *mm* the modification level. All values are two-digit decimal numbers.

For each level of the operating system, only one value for &SYSHSM is possible.

These are the values for the *cc* code that have been used in earlier operating systems:

02 DFSMShsm in OS/390 Version 2 Release 10, or in z/OS Version 1 Release 1 or Release 2. All three releases returned "02.02.10.00".

03 DFSMShsm in z/OS Version 1 Release 3 or later.

For example, a value of 02.02.10.00 represents DFSMShsm for OS/390 V2R10. A value of 03.01.10.00 represents z/OS Version 1 Release 10 Modification Level 0.

Before OS/390 V2R10, &SYSHSM contained a character string of four decimal digits. The first digit represented the version. The second and third digits represented the release number of DFSMShsm. The fourth digit represented the modification level. For example, 1050 represented DFSMShsm 1.5.0.

&SYSISPF

&SYSISPF indicates whether ISPF dialog manager services are available. The variable can have one of two values:

ACTIVE ISPF services are available.

NOT ACTIVE
ISPF is not initialized.

&SYSJES

&SYSJES contains the name and the level of the JES installed. For example,

```
PROC 0
WRITE &SYSJES
EXIT
```

may return `JES2 OS 2.10`. In this example `JES2` is the JES name and `OS 2.10` is the JES level, representing version and release number of JES2. The JES level may contain a modification level as well.

The values returned are provided by the subsystem interface request routine (IEFSSREQ).

Both strings are separated by a blank character; any trailing blank characters are removed. If either the JES name or level returns an empty character string, then no separating blank character is inserted.

If the subsystem is not active the string `-INACTIVE-` is returned (note the string delimiters).

If the system finds that the subsystem is neither JES2 4.3 or later nor JES3 5.1.1 or later, the &SYSJES control variable contains the string `-DOWNLEVEL-` (note the string delimiters).

&SYSLRACF

&SYSLRACF indicates the level of RACF installed on the system. If RACF is not installed, &SYSLRACF contains a null value. The value of &SYSLRACF is equal to the value in RCVTVRMN.

&SYSAPPCLU

&SYSAPPCLU contains the MVS/APPC logical unit (LU) name. The LU name identifies the TSO/E address space your CLIST will be running in as the SNA addressable unit for Advanced-Program-to-Program-communication (APPC). The LU name is obtained through the APPC/MVS Advanced TP Callable Services (ATBEXAI - Information Extract Service). For example,

```
PROC 0
WRITE &SYSAPPCLU
EXIT
```

may return an LU name of `LU0001`. Trailing blanks are removed. A null string is returned if:

- There is no APPC activity in the address space the CLIST is running in, or
- No LU name is provided by the APPC/MVS Advanced TP Callable Services.

Note: CLISTs do *not* support CPI Communication (a method to let one program communicate with another program on the same or other MVS system in an SNA network). Therefore the use of the &SYSAPPCLU control variable makes sense only in a CLIST that is invoked by a program (for example, a REXX exec) that has established APPC. If the control variable is used outside this environment, a null string is returned.

&SYSMVS

&SYSMVS contains the level of the base control program (BCP) component of z/OS. For example,

```
PROC 0
WRITE &SYSMVS
EXIT
```

may return `SP7.0.1` as the level of the BCP component.

The value returned is that of the CVTPRODN field in the communications vector table (CVT).

Note: The format of the value returned by &SYSMVS may change in the future, but will remain the content of the CVTPRODN field.

&SYSNAME

&SYSNAME returns the system's name your CLIST is running on, as specified in SYS1.PARMLIB member IEASYSxx on the SYSNAME statement. For example,

```
PROC 0
WRITE &SYSNAME
EXIT
```

may return `ATQS` as the MVS system name.

You may want to use the &SYSNAME control variable to identify on which system in a multi-system global resource serialization complex your CLIST is running on. For details on how the SYSNAME value is used in a multi-system complex, see the *z/OS MVS Initialization and Tuning Reference*.

&SYSNODE

&SYSNODE contains the network node name of your installation's JES. This name identifies the local JES in a network of systems or system complexes being used for network job entry (NJE) tasks. For example,

```
PROC 0
WRITE &SYSNODE
EXIT
```

may return a value of `BOE9`, which is the network node name of your local JES.

The node name returned by the &SYSNODE control variable derives from the NODE initialization statement of JES.

If the system finds that the subsystem is not active, the &SYSNODE control variable contains the string `-INACTIVE-` (note the string delimiters).

If the system finds that the subsystem is neither JES2 4.3 or later nor JES3 5.1.1 or later, the &SYSNODE control variable contains the string `-DOWNLEVEL-` (note the string delimiters).

&SYSOPSYS

&SYSOPSYS contains the z/OS name, version, release, modification level, and FMID of the BCP portion of your installation's operating system. For example,

```
PROC 0
WRITE &SYSOPSYS
EXIT
```

may return a string of `Z/OS 01.01.00 JBB7713`, where `Z/OS` represents the product name, followed by a blank character, followed by an eight-character string representing version, release, modification number, followed by a blank character, followed by the FMID.

The &SYSOPSYS control variable was introduced after TSO/E Version 2 Release 5 with APAR OW17844. If you use this variable in a environment earlier than TSO/E 2.5, or without the PTF associated with APAR OW17844, the system returns a null string.

Note: To display the operating system product name as "z/OS" instead of "Z/OS", a CONTROL ASIS or CONTROL NOCAPS statement must be included before the &SYSOPSYS statement. Also notice that a "/" character will appear in the product name, for example, in "z/OS". CLISTs might interpret that character to be the divide operator. For example, `SET LEVEL=&SYSOPSYS` might produce an error message. To prevent a CLIST from evaluating the resulting string you should use the &STR function; for example, `SET LEVEL=&STR(&SYSOPSYS)`.

&SYSRACF

&SYSRACF indicates the status of RACF. The variable can have one of three values:

`AVAILABLE`
: RACF services are available.

`NOT AVAILABLE`
: RACF is not initialized.

NOT INSTALLED
RACF is not installed.

&SYSPLEX

&SYSPLEX returns the MVS sysplex name as found in the COUPLExx or LOADxx member of SYS1.PARMLIB. For example,

```
PROC 0
WRITE &SYSPLEX
EXIT
```

may return a value of PLEXNY02. The value has a maximum of eight characters; trailing blanks are removed. If no sysplex name is specified in SYS1.PARMLIB, &SYSPLEX returns a null string.

&SYSSECLAB

&SYSSECLAB returns the SECLABEL name that is valid for the TSO/E session where the CLIST is started. For example,

```
PROC 0
WRITE &SYSSECLAB
EXIT
```

may return a value of SYSHIGH as the current security label name. Trailing blanks are removed.

Note: The use of the &SYSSECLAB control variable requires that RACF is installed, and that security label checking has been activated. If no security information is found, the &SYSSECLAB control variable contains a null string.

&SYSSMS

&SYSSMS indicates whether SMS (storage management subsystem) is available to your CLIST. For example,

```
PROC 0
WRITE &SYSSMS
EXIT
```

returns one of the following character strings:

UNAVAILABLE
Obsolete and should no longer occur. System logic error. Contact your IBM service representative.

INACTIVE
SMS is available on your system but not active.

ACTIVE SMS is available and active, so your CLIST can depend on it.

&SYSSMFID

&SYSSMFID identifies the system on which System Management Facilities (SMF) is active. The value returned is as specified in SYS1.PARMLIB member SMFPRMxx on the SID statement. Trailing blanks are removed. For example,

```
PROC 0
WRITE &SYSSMFID
EXIT
```

returns ATQS as the SMF ID. Note that the value returned by &SYSSMFID and &SYSNAME may be the same in your installation. For details on the SYSNAME and SID statement in member SMFPRMxxee, see the *z/OS MVS Initialization and Tuning Reference*.

&SYSSYMDEF

&SYSSYMDEF*(symbol_name)* returns the value represented by the variable *"symbol_name"* as specified in SYS1.PARMLIB member IEASYMxx on the SYSDEF ... SYMDEF statement. Or, the 'string' can also be one of the system static or dynamic symbols as defined in *z/OS MVS Initialization and Tuning Reference*.

For example, if SYMDEF(&SYSTEMA = 'SA') is specified in IEASYMxx, then

```
PROC 0
WRITE &SYSSYMDEF(SYSTEMA)
EXIT
```

returns a value of SA. A null string is returned if the symbolic name is not specified in IEASYMxx, and it is not one of the MVS defined static or dynamic symbols.

Here, the symbol name SYSTEMA is assigned a name of SA on the SYMDEF statement in IEASYMxx. The &SYSSYMDEF(*symbol_name*) control variable resolves to a string of SA.

You can also retrieve the value for one of the MVS defined static or dynamic system symbols. For example:

```
WRITE &SYSSYMDEF(JOBNAME)   /*Returns JOBNAME
                                  BOB perhaps       */
```

Refer to *z/OS MVS Initialization and Tuning Reference* for a discussion and a list of the currently defined MVS static and dynamic system symbols.

For example, you can retrieve the IPL Volume Serial Name of your system using

```
WRITE &SYSSYMDEF(SYSR1)   /* may return 640S06
                             as IPL Vol. Ser. Name */
```

The SYSSYMDEF function goes through CLIST substitution first, the result of which must be a 1-8 character name specifying the symbol that has been defined in the SYMDEF statement. Any other values including CLIST delimiters may cause unpredictable results.

&SYSTSOE

&SYSTSOE indicates the level of TSO/E installed on the system. For OS/390 Version 2 Release 4 and later, &SYSTSOE returns 2060.

Getting information about the CLIST

The following control variables provide information about the CLIST.

&SYSENV

&SYSENV indicates whether the CLIST is executing in the foreground (FORE) or the background (BACK). You can use this variable when a CLIST must make logical decisions based on the environment. For example, the way a CLIST obtains its input is sensitive to background and foreground executions. You can use &SYSENV to prevent the CLIST executing READ statements in the background as follows:

```
GLOBAL LNAME /* Define global variable to be set by FETCHNAM */
:
IF &SYSENV=FORE THEN +
 DO
  WRITE Enter your last name.
  READ LNAME
 END
ELSE +
 %fetchnam
```

&SYSSCAN

&SYSSCAN contains a number that defines the maximum number of times symbolic substitution is performed on each line in a CLIST. The default number is 16. You can assign &SYSSCAN a value from 0 to +2,147,483,647 (2^{31}-1). A zero limit inhibits all scans, preventing any substitution of values for symbolic variables.

For example, to write a record containing an ampersand (&), and prevent a CLIST from performing erroneous symbolic substitution, you can code the following:

```
:
SET &SYSSCAN=0  /* Prevent symbolic substitution
WRITE Jack & Jill went up the hill
SET &SYSSCAN=16  /*Reset &SYSSCAN
```

&SYSICMD

&SYSICMD contains the name by which the user *implicitly* invoked the currently executing CLIST. If the user invoked the CLIST explicitly, this variable has a null value.

&SYSPCMD

&SYSPCMD contains the name of the TSO/E command that the CLIST most recently executed. The *initial* value of &SYSPCMD depends on the environment from which the CLIST was invoked. If the invoker used the EXEC command, the *initial* value is EXEC. If the invoker used the EXEC subcommand of EDIT, the *initial* value is EDIT.

&SYSSCMD

&SYSSCMD contains the name of the TSO/E subcommand that the CLIST most recently executed. If invoker used the EXEC command, the *initial* value of &SYSSCMD is null. If the invoker used the EXEC subcommand of EDIT, the *initial* value is EXEC.

Relationship between &SYSPCMD and &SYSSCMD

The &SYSPCMD and &SYSSCMD control variables are interdependent. Following the initial invocation, the values of &SYSPCMD and &SYSSCMD depend on the TSO/E command or subcommand most recently executed. For example, if the value of &SYSSCMD is EQUATE, a subcommand unique to the TEST command, the value of &SYSPCMD is TEST.

You can use &SYSPCMD and &SYSSCMD in error and attention exits to determine where the error or attention interrupt occurred.

&SYSNEST

&SYSNEST indicates whether the currently executing CLIST is nested. (A nested CLIST is one that was invoked by another CLIST rather than explicitly by the user.) If the CLIST is nested, &SYSNEST contains the value YES. If it is not nested, &SYSNEST contains the value NO.

Setting options of the CLIST CONTROL statement

The following control variables let you test or modify options of the CLIST CONTROL statement. For full information about the CONTROL statement and its options, see "CONTROL statement" on page 149.

&SYSPROMPT

&SYSPROMPT indicates whether the CONTROL statement's PROMPT or NOPROMPT option is active. The value `ON` indicates that CONTROL PROMPT is active, and TSO/E commands in the CLIST can prompt the terminal for input. `OFF` indicates that CONTROL NOPROMPT is active, and TSO/E commands cannot prompt the terminal.

Your CLISTs can use &SYSPROMPT to test which option is active, or change the option. For example, if you want the CLIST to allow prompting from the LISTDS command only, you can code:

```
SET &SYSPROMPT = ON
LISTDS
SET &SYSPROMPT = OFF
```

&SYSSYMLIST

&SYSSYMLIST indicates whether the CONTROL statement's SYMLIST or NOSYMLIST option is active. The value `ON` indicates that CONTROL SYMLIST is active, and CLIST statements are displayed at the terminal before being scanned for symbolic substitution. The value `OFF` indicates that CONTROL NOSYMLIST is active, and CLIST statements are not displayed at the terminal before symbolic substitution.

Your CLISTs can use &SYSSYMLIST to test which option is in effect, or to change the option. For example, if you suspect an error in part of a CLIST and you want to display certain statements before substitution, you can code:

```
SET &SYSSYMLIST = ON
⋮
  (suspected statements in error)
⋮
SET &SYSSYMLIST = OFF
```

&SYSCONLIST

&SYSCONLIST indicates whether the CONTROL statement's CONLIST or NOCONLIST option is active. The value `ON` indicates that CONTROL CONLIST is active, and CLIST statements are displayed at the terminal *after* symbolic substitution. The value `OFF` indicates that CONTROL NOCONLIST is active, and CLIST statements are not displayed at the terminal after symbolic substitution.

Your CLISTs can use &SYSCONLIST to test which option is in effect, or to change the option. For example, if you suspect an error in part of a CLIST and you want to display certain statements after substitution, you can code:

```
SET &SYSCONLIST = ON
⋮
  (suspected statements in error)
⋮
SET &SYSCONLIST = OFF
```

&SYSLIST

&SYSLIST indicates whether the CONTROL statement's LIST or NOLIST option is active. The value ON indicates that CONTROL LIST is active, and TSO/E commands and subcommands are displayed at the terminal *after* symbolic substitution. The value OFF indicates that CONTROL NOLIST is active, and commands and subcommands are not displayed at the terminal after symbolic substitution.

Your CLISTs can use &SYSLIST to test which option is in effect, or to change the option. For example, if you suspect an error in part of a CLIST and you want to display certain commands or subcommands, you can code:

```
SET &SYSLIST = ON
⋮
  (suspected commands in error)
⋮
SET &SYSLIST = OFF
```

&SYSASIS

&SYSASIS indicates whether the CONTROL statement's ASIS option is active. The value ON indicates that CONTROL ASIS is active, and lowercase characters are not converted to uppercase before processing. The value OFF indicates that CONTROL CAPS is active, and lowercase characters are converted to uppercase.

Your CLISTs can use &SYSASIS to test which option is in effect, or to change the option. For example, if you want READ and WRITE statements to preserve lowercase letters, you can code:

```
SET &SYSASIS = ON
WRITE Enter data exactly as you want it to appear.
WRITE Lowercase letters won't be changed to uppercase.
READ &Ulc_data
```

&SYSMSG

&SYSMSG indicates whether the CONTROL statement's MSG or NOMSG option is active. The value ON indicates that CONTROL MSG is active, and the CLIST can display informational messages at the terminal. The value OFF indicates that CONTROL NOMSG is active, and the CLIST cannot display informational messages at the terminal.

Your CLISTs can use &SYSMSG to test which option is in effect, or to change the option. For example, if you wanted to make sure that informational messages are displayed at the terminal, you can code:

```
SET &SYSMSG = ON
⋮
```

&SYSFLUSH

&SYSFLUSH indicates whether the CONTROL statement's FLUSH or NOFLUSH option is active. The value ON indicates that CONTROL FLUSH is active, and the system can erase (flush) any nested CLISTs when an error occurs. The value OFF

indicates that CONTROL NOFLUSH is active, and the system cannot flush nested CLISTs. When CONTROL MAIN is active, &SYSFLUSH cannot be set to ON.

Your CLISTs can use &SYSFLUSH to test which option is in effect, or to change the option. For example, if your CLIST invokes other CLISTs, you can set &SYSFLUSH to OFF to protect them from being flushed in the event of an error. You can then use an error routine to recover from the error and continue processing.

```
SET &SYSFLUSH = OFF
ERROR +
  DO
⋮
  (error routine)
⋮
  END
```

For more information about error routines and protecting nested CLISTs, see Chapter 11, "Writing ATTN and ERROR routines," on page 103.

Getting information about user input

Two control variables are related to input supplied to a CLIST.

&SYSDLM

&SYSDLM ("DLM" is for *delimiter*) contains a number that identifies the position (first, second, third, and so on) of the TERMIN or TERMING statement character string entered by the user to return control to the CLIST.

You can use this variable to determine what action should be taken when the user returns control to the CLIST, based on the string chosen. For example, the following statements inform the user what is requested (WRITE), pass control to the terminal and establish valid control character strings (TERMIN or TERMING), and determine the subsequent action based on the string entered.

```
WRITE The first phase of BUDGET has completed with
WRITE a return code of &RCODE
WRITE Enter YES if you want the results printed.
WRITE Enter NO if you do not want them printed.
TERMIN YES NO
IF &SYSDLM = 1 THEN +
⋮
  (Print results)
⋮
```

&SYSDVAL

&SYSDVAL ("DVAL" is for *default value*) contains one of the following at any given time:

- A null value
- The input the user entered when returning control to the CLIST after a TERMIN or TERMING statement
- The user's response after a READ statement without operands
- The value assigned to &SYSDVAL by an assignment statement.

Initially, &SYSDVAL contains a null value. It can also contain a null value, if:

- The user does not enter anything but a pre-defined character string or null line after a TERMIN or TERMING statement.
- The user does not enter any input after a READ statement without operands.

- You assign a null value to &SYSDVAL.

You can also use &SYSDVAL when performing I/O to a data set. You can assign the data to variables by defining SYSDVAL as the file name of the data set and naming the variables on the READVAL statement. For an example of using &SYSDVAL and READVAL in I/O, see "Using &SYSDVAL when performing I/O - the PHONE CLIST" on page 132.

Trapping TSO/E command output

Two control variables allow you to trap TSO/E command output in a CLIST: &SYSOUTTRAP and &SYSOUTLINE. These variables save output from TSO/E commands and allow a CLIST or application to process the output. You can modify the values of &SYSOUTTRAP and &SYSOUTLINE with assignment statements. For example, the assignment statement

```
SET &SYSOUTTRAP = 100
```

lets you trap and save 100 lines of output from a TSO/E command.

&SYSOUTTRAP

Use &SYSOUTTRAP to specify the maximum number of lines of TSO/E command output to be saved. If you want to save all the output from a TSO/E command, set &SYSOUTTRAP to a number greater than or equal to the number of output lines that the command produces. Any output lines produced in excess of the &SYSOUTTRAP value are not saved.

To save the output of a single command, set &SYSOUTTRAP to zero after issuing the command. Otherwise, output from subsequent commands may replace the original saved output.

&SYSOUTLINE

When you use &SYSOUTTRAP, the CLIST saves TSO/E command output in variables beginning with &SYSOUTLINE.

The CLIST uses the variable &SYSOUTLINE to record the number of output lines produced by a command. The CLIST saves the actual command output in the variables &SYSOUTLINE*nn*, where *nn* represents the positional number of the line being saved. *nn* can be any decimal number up to 21 digits in length. However, the value in &SYSOUTTRAP and the amount of storage available determine the actual number of lines saved.

The following CLIST traps output from the TSO/E LISTD command, retrieves it using nested variables, and writes each line of output.

```
PROC 0 DATASET(DEFAULT)
IF &DATASET = DEFAULT THEN +
  DO
    WRITE What data set do you want to process?
    READ DATASET
  END
SET &SYSOUTTRAP = 1000                /* Expect command produces no   */
                                      /*   more than 1000 lines       */
LISTD '&SYSPREF..&DATASET' MEMBERS    /* List data set members        */
SET B = &SYSOUTLINE                   /* Get number of lines produced */
SET &SYSOUTTRAP = 0                   /* Reset &SYSOUTTRAP            */
SET A = 1                             /* Initialize counter           */
DO WHILE                              /* Loop for the lesser of       */
  (&A <= 1000) AND                    /*   num of lines expected and  */
```

```
  (&A <= &B)                          /*   num of lines produced      */
  SET MEMBER = &STR(&&SYSOUTLINE&A)   /* Get a &SYSOUTLINEnn variable */
  WRITE &STR(&MEMBER)                 /* Write the output line        */
  SET A = &A +1                       /* Increase the line counter    */
END                                   /* End of loop on counter       */
```

For another example of using &SYSOUTTRAP and &SYSOUTLINE to process command output, see "Allocating data sets to SYSPROC - the SPROC CLIST" on page 133.

Considerations for using &SYSOUTTRAP and &SYSOUTLINE

- If you add the CONTROL LIST and SYMLIST options to a CLIST that uses &SYSOUTTRAP, more output lines are produced and you might need to adjust &SYSOUTTRAP and &SYSOUTLINE*nn* values to retrieve the desired output lines.
- To trap the output of TSO/E commands under ISPF/PDF, you must invoke a CLIST with command output trapping *after* ISPF or one of its services has been invoked.
- The output of authorized commands listed under the AUTHCMDS parameter in the active IKJTSO*xx* parmlib member cannot be trapped by a CLIST invoked under any application that builds its own ECT. For example, a CLIST must be prefixed by the TSO subcommand of IPCS to trap the output of authorized commands when invoked from IPCS under ISPF.
- If you try to display a line of output in &SYSOUTLINE*nn* where *nn* is greater than the value of &SYSOUTTRAP, the &SYSOUTLINE*nn* variable contains unreliable data.
- If you try to display a &SYSOUTLINE*nn* variable that contains no command output, the CLIST returns a null line.
- Because CLISTs use the TSO/E EXEC command to invoke nested CLISTs, &SYSOUTTRAP saves all output of nested CLISTs as TSO/E command output. Therefore, if you need to trap all of the output of a command processor that processes several subcommands, consider using a nested CLIST to do so.
- &SYSOUTTRAP does not save command output sent to the terminal by a TPUT macro, but does save output from the PUTLINE macro with DATA or INFOR keywords.
- If you run out of storage, you can use the TSO/E PROFILE option VARSTORAGE(HIGH) to allow CLIST variables to reside above the 16 MB line. See "Increasing the amount of storage available for variables" on page 23 in Chapter 4, "Using symbolic variables," on page 17.
- Whenever a CLIST starts to execute a TSO/E command or subcommand, it resets &SYSOUTLINE to zero. However, if a CLIST invokes a CLIST or a non-CLIST program containing TSO/E commands, the invoked program does not reset &SYSOUTLINE to zero for each TSO/E command. To record the number of command output lines in an invoked program, use an assignment statement to reset &SYSOUTLINE to zero before each TSO/E command. For information about assigning a value to CLIST variables in a non-CLIST environment, see *z/OS TSO/E Programming Services*.

Getting return codes and reason codes

Two control variables enable you to obtain return codes and reason codes. You can modify both &LASTCC and &MAXCC with an assignment statement.

&LASTCC

When you use &LASTCC outside an error routine, &LASTCC contains the return code from the last TSO/E command or subcommand, nested CLIST, or CLIST statement executed. Because the value of this variable is updated after the execution of each statement or command, store its value in a symbolic variable before executing code that references the value.

In an error routine, &LASTCC is not updated after the execution of each statement or command. Only the RETURN statement updates the value of &LASTCC. If you use &LASTCC in an error routine, &LASTCC contains the return code from the command or statement that was executing when the error occurred.

&LASTCC does not support negative return codes. When a negative return code is received from a REXX exec, CLIST converts it to binary, removes the first byte, and stores the remainder in &LASTCC as a positive decimal integer.

When &LASTCC receives an error return code from a TSO/E command, subcommand, nested CLIST, or CLIST statement, control passes to an error routine if present in the CLIST. However, when &LASTCC contains the return code from a subprocedure RETURN statement, control does *not* pass to an error routine.

&LASTCC can be used in error routines that handle multiple error conditions. For example, if an error routine handles arithmetic errors, it can use &LASTCC to determine what type of message to display at the terminal:

```
ERROR +
 DO
  SET RCODE = &LASTCC
/* Character data in operands? */
  IF &RCODE = 852 THEN +
   WRITE Character data was found in numbers being added.
/* Numeric value too large?    */
  IF &RCODE = 872 THEN +
   WRITE A numeric value in the addition was too large.
   (Other tests)
.
.
.
RETURN
 END
SET SUM = &VALUE1 + &VALUE2 + &VALUE3;
```

Note that &LASTCC itself does not get updated within the error routine.

When an error occurs during CLIST I/O processing, use an error routine to obtain the error code in &LASTCC. For example, to trap the error code generated by OPENFILE when attempting to open a file (BADFILE) that does not exist, code the following CLIST:

```
PROC 0
ERROR DO
SET RC=&LASTCC.
RETURN
END
OPENFILE BADFILE
WRITE LASTCC=&RC
```

See Table 8 on page 113 for a list of the CLIST error codes that &LASTCC can contain.

&MAXCC

&MAXCC contains the highest return code returned by a nested CLIST or by a TSO/E command, subcommand, or CLIST statement in the currently executing CLIST.

&MAXCC is not set when a subprocedure returns to the CLIST.

You can use &MAXCC with &LASTCC to determine error conditions. For example, error codes caused by evaluation errors are in the 800-899 range. You can modify the error routine in the example under &LASTCC to determine first whether the error was caused by an arithmetic evaluation. Insert the following IF-THEN-ELSE sequence before the check for character data in operands:

```
⋮
/* Evaluation error?           */
   IF &MAXCC <800 OR &MAXCC >899 THEN +
      GOTO ...
   ELSE +
⋮
```

Getting results of the TSOEXEC command

Three variables are related to the use of the TSOEXEC command: &SYSABNCD, &SYSABNRC, and &SYSCMDRC. You can modify any one of them with an assignment statement.

&SYSABNCD, &SYSABNRC, and &SYSCMDRC contain, the ABEND code, ABEND reason code, and command return code returned by the command most recently invoked by the TSOEXEC command. You can use these variables in situations similar to those in which you need to use &LASTCC and &MAXCC. For example, to determine if the TRANSMIT command terminated abnormally, you can code:

```
tsoexec transmit plpsc.d00abc1 dataset(letter.text)
/* Abend code non-zero?         */
IF &SYSABNCD¬=0 THEN +
 DO
  WRITE The transmission of LETTER.TEXT to
  WRITE PLPSC.D00ABC1 abended.
 END
```

Getting data set attributes

Control variables include certain predefined variables set by CLIST statements. The LISTDSI statement sets a number of variables with information about data set attributes. These LISTDSI variables cannot be modified.

The LISTDSI statement

You can use the LISTDSI (*list data set information*) statement to retrieve detailed information about a data set's attributes. The statement stores the information in CLIST variables. The CLIST can use the information to determine if the data set has enough space or the correct format for a given task. The CLIST can also use the information as input to the TSO/E ALLOCATE command to create a new data set with some attributes of the old data set while modifying others.

To retrieve a data set's allocation information, specify the data set's name on the LISTDSI statement. You can also specify that a data set migrated by the Data

Facility Storage Management Subsystem Hierarchical Storage Manager (DFSMShsm) be recalled, and that directory information be retrieved for a partitioned data set.

In response to the LISTDSI statement, the CLIST stores each of the data set's allocation attributes in a specific variable. For example, the data set's primary space allocation is stored in the variable &SYSPRIMARY, and its organization is stored in &SYSDSORG. For a complete list of the CLIST variables set by LISTDSI, see "LISTDSI statement" on page 158.

For an example of using LISTDSI, see "Allocating a data set with LISTDSI information - the EXPAND CLIST" on page 143.

Chapter 7. Using built-in functions

The CLIST language includes built-in functions that you can perform on variables, expressions, and character strings. If necessary, CLIST evaluates the variable or expression first, and then performs the requested function. The CLIST then stores the result under the name of the built-in function.

To use a built-in function, type its name, followed by the variable, expression, or character string in parentheses. The variable, expression, or character string is also called the *argument* of the built-in function. The argument must immediately follow the built-in function name, with no blanks between them. Table 6 describes each of the built-in functions briefly and gives page numbers where you can find more information.

Table 6. Built-in functions

Built-in function	Function	Reference
&DATATYPE(expression)	Indicates whether the evaluation of *expression* is a character string or a numeric value.	"Determining the data type of an expression - &DATATYPE" on page 54
&EVAL(expression)	Performs an arithmetic evaluation of *expression*.	"Forcing arithmetic evaluations - &EVAL" on page 54
&LENGTH(expression)	Evaluates *expression* if necessary and indicates the number of bytes in the result.	"Determining an expression's length in bytes - &LENGTH" on page 55
&NRSTR(string)	Preserves double ampersands, defines non-rescannable strings.	"Preserving double ampersands - &NRSTR" on page 56
&STR(string)	Defines data to be used as a character string.	"Defining character data - &STR" on page 57
&SUBSTR(exp[:exp],string)	Uses certain bytes in a character string.	"Defining a substring - &SUBSTR" on page 59
&SYSCAPS(string)	Converts the string to uppercase characters.	"Converting character strings to uppercase characters - &SYSCAPS" on page 61
&SYSCLENGTH(expression)	Evaluates *expression* if necessary and indicates the number of characters in the result.	"Determining an expression's length in characters - &SYSCLENGTH" on page 56
&SYSCSUBSTR(exp[:exp],string)	Uses certain characters in a character string.	"Defining a substring - &SYSCSUBSTR" on page 60
&SYSDSN(dsname[(member)])	Indicates whether the specified data set exists.	"Determining data set availability - &SYSDSN" on page 61
&SYSINDEX(string_1,string_2[,start])	Finds the position of a character string (string_1) within another (string_2), from a specific starting point.	"Locating one character string within another - &SYSINDEX" on page 62
&SYSLC(string)	Converts the string to lowercase characters.	"Converting character strings to lowercase characters - &SYSLC" on page 61
&SYSNSUB(level,expression)	Limits the level of symbolic substitution in the expression.	"Limiting the level of symbolic substitution - &SYSNSUB" on page 64
&SYSONEBYTE(string)	Converts a string of data from the double-byte character set (DBCS) to EBCDIC.	"Converting DBCS data to EBCDIC - &SYSONEBYTE" on page 64
&SYSTWOBYTE(string)	Converts a string of data from EBCDIC to the double-byte character set (DBCS).	"Converting EBCDIC data to DBCS - &SYSTWOBYTE" on page 65

In addition to these built-in functions, TSO/E provides an installation exit that lets your installation add its own CLIST built-in functions. For information about the exit, see *z/OS TSO/E Customization*.

Note: With the exception of &SYSNSUB, built-in functions will not resolve double ampersands (&&) that appear in an argument.

Determining the data type of an expression - &DATATYPE

Use the &DATATYPE built-in function to determine what type of data an evaluated expression contains. After evaluating the expression, a CLIST replaces this built-in function with one of the following strings: CHAR, NUM, DBCS, or MIXED. The strings indicate the following:

- CHAR -- The evaluated expression contains at least one non-numeric EBCDIC character and no double-byte character set (DBCS) characters.
- NUM -- The evaluated expression is entirely numeric.
- DBCS -- The evaluated expression is a single delimited string of DBCS data.
- MIXED -- The evaluated expression contains both DBCS and EBCDIC data.

The following examples show the evaluations of various expressions:

```
SET A = &DATATYPE(ALPHABET)            /* result: &A = CHAR
SET B = &DATATYPE(1234)                /* result: &B = NUM
SET C = &DATATYPE(SYS1;PROCLIB)        /* result: &C = CHAR
SET D = &DATATYPE(3*2/4)               /* result: &D = NUM
SET E = &DATATYPE(12.34)               /* result: &E = CHAR
```

For example, the following clause evaluates as true:

```
IF &DATATYPE(12.34)=CHAR THEN
```

The following examples use the convention *d1d2* to represent two DBCS characters and < and > to represent the shift-out and shift-in delimiters (X'0E' and X'0F') that mark the beginning and end of the DBCS string.

```
SET A = &DATATYPE(<d1d2>)              /* result: &A = DBCS
SET B = &DATATYPE(ABC<d1d2>123)        /* result: &B = MIXED
SET C = &DATATYPE(<>)                  /* result: &C = DBCS
SET D = &DATATYPE(A<>C)                /* result: &D = MIXED
SET E = &DATATYPE(<d1d2><d3d4>)        /* result: &E = MIXED
```

For example, the following clauses evaluate as true:

```
IF &DATATYPE(<d1d2d3>)=DBCS THEN
IF &DATATYPE(A<d1d2d3>B)=MIXED THEN
```

Forcing arithmetic evaluations - &EVAL

On most statements, the appearance of arithmetic expressions results in evaluations of those expressions when a CLIST executes the statements. However, on the WRITE statement, you must explicitly instruct a CLIST to evaluate an arithmetic expression by using the &EVAL built-in function. For example, to create a WRITE statement that adds two variables, &FNUM and &SNUM, and displays the results, code the following:

```
WRITE &FNUM + &SNUM = &EVAL(&FNUM+&SNUM)
```

Assuming &FNUM is four and &SNUM is three, the CLIST displays the following message:

```
4 + 3 = 7
```

Determining an expression's length in bytes - &LENGTH

Use the &LENGTH built-in function to determine the number of bytes in an expression or character string. &LENGTH performs symbolic substitution and arithmetic evaluations before determining the length. If a variable has a null value, &LENGTH returns a value of zero.

For example, after the following statement executes, &LENANSWR has the value 2 because there are two bytes in the result of the addition, 11.

```
SET LENANSWR = &LENGTH(1+1+9)
```

&LENGTH can also reference symbolic variables. Assume you want to save a value that is triple the length of the value of a variable called &CSTRING. To save the value in a variable called &NXTFIELD, code:

```
SET NXTFIELD = 3 * &LENGTH(&CSTRING)
```

If &CSTRING contains the value 100, &NXTFIELD contains the value 9.

If a string contains data of the double-byte character set (DBCS), &LENGTH counts each DBCS character as two bytes, and counts each DBCS delimiter as one byte. For example, using *d1d2* to denote two DBCS characters and using < and > to represent the DBCS delimiters X'0E' and X'0F':

```
SET A = &LENGTH(<d1d2>)    /* result: &A = 6
```

The same is true when a string contains mixed EBCDIC and DBCS characters. For example:

```
SET A = &LENGTH(ABC<d1d2>)    /* result: &A = 9
```

Suppressing arithmetic evaluations

If you do not want a CLIST to perform arithmetic evaluations of a &LENGTH expression, enclose the expression in a &STR built-in function as follows:

```
SET LENANSWR = &LENGTH(&STR(1+1+9))
```

After the previous statement executes, &LENANSWR contains the value 5.

Including leading and trailing blanks and leading zeros

If you want leading and trailing blanks and leading zeros in a &LENGTH expression included in the assignment, enclose the expression in a &STR built-in function. Otherwise, the blanks and zeros are ignored.

For example, suppose that you want to save the length of the variable &IFIELD in a variable called &SLNGTH. The contents of &IFIELD are 0 472.20. Include &IFIELD in the &STR built-in function to include the blanks and the leading zero as part of the assignment:

```
SET SLNGTH= &LENGTH(&STR(&IFIELD))
```

After the previous statement executes, &SLNGTH contains the value 8.

Determining an expression's length in characters - &SYSCLENGTH

Use &SYSCLENGTH built-in function to determine the number of characters in an expression or string that contains characters of the double-byte character set (DBCS). &SYSCLENGTH differs from &LENGTH in that &SYSCLENGTH counts each DBCS character as one character instead of two bytes, and does not count DBCS delimiters. For example:

```
SET A = &SYSCLENGTH(<d1d2>)    /* result: &A = 2
```

The same is true when a string contains mixed EBCDIC and DBCS characters. For example:

```
SET A = &SYSCLENGTH(ABC<d1d2>)    /* result: &A = 5
```

Except for the difference in counting DBCS characters, &SYSCLENGTH is identical to &LENGTH.

Preserving double ampersands - &NRSTR

You can use the &NRSTR built-in function to prevent a CLIST from:

- Removing the first ampersand when it encounters a character string with a prefix of double ampersands.
- Performing more than one level of symbolic substitution on a variable.

You can use &NRSTR with JCL statements that include the name of a temporary data set (for example, &&TEMP). Using &NRSTR prevents a CLIST from changing the name of a temporary data set (&&TEMP) to a symbolic parameter (&TEMP).

Double ampersands

To assign the character string &&DATA to the variable &FILE, code:

```
SET FILE = &NRSTR(&&DATA)
```

One level of symbolic substitution

To set two variables, &A and &C, to the value &B code:

```
⋮
SET A = &&B
SET C = &NRSTR(&A)
⋮
```

After the execution of the first SET statement, &A contains the value &B. When the second SET statement is executed, the CLIST performs symbolic substitution and substitutes &B for &A. &NRSTR prevents any further scan of the statement; therefore, &C is assigned the value &B.

Records containing JCL statements

The following paragraphs discuss the use of the &NRSTR built-in function when processing records that contain JCL statements.

Temporary data set names

If a JCL statement contains a temporary data set name (for example, &&TEMP), enclose the statement in a &NRSTR built-in function to prevent the CLIST from removing the first ampersand. The following CLIST uses &NRSTR to preserve a temporary data set name in a JCL statement.

```
submit *
//&sysuid  job 'Y2803P,?,S=C','SteveR',msgclass=r,class=j
// exec pgm=IEFBR14
//dd1   dd  dsn=&NRSTR(&&temp),disp=(,pass),unit=sysda
&null
```

Symbolic parameters

If a JCL statement contains a symbolic parameter (for example, &LIBRARY),; use the &SYSNSUB built-in function to prevent the CLIST from performing erroneous symbolic substitution. Assume that the preceding CLIST contained the JCL statement:

```
//dd2   dd  dsn=&library,disp=(,pass),unit=sysda
```

To prevent any symbolic substitution, you can enclose the symbolic parameter &library in the &SYSNSUB built-in function as follows:

```
//dd2   dd  dsn=&SYSNSUB(0,&library),disp=(,pass),unit=sysda
```

The number 0 in parentheses after &SYSNSUB tells the CLIST how many levels of symbolic substitution you want performed on the parameter (in this case, zero levels). For more information about the &SYSNSUB built-in function, see “Limiting the level of symbolic substitution - &SYSNSUB” on page 64.

Defining character data - &STR

Use the &STR built-in function to define character data and prevent the CLIST from evaluating it. The data can be any expression or statement, and can include nested variables and characters of the double-byte character set (DBCS) within DBCS delimiters.

For example, the statement SET DIMENSNS=&STR(2*4) defines 2*4 as a character string and assigns the string to the variable &DIMENSNS; Without the &STR built-in function, you can not make the desired assignment because a CLIST needs to evaluate 2*4 as an arithmetic expression and set &DIMENSNS to the value 8.

The &STR built-in function suppresses arithmetic evaluations only for the data between the parentheses. If you set &STATS to &DIMENSNS,; &STATS; will contain the value 8, not the character string 2*4. To preserve the character string, code:

```
SET STATS=&STR(&DIMENSNS)
```

Special procedures are required when defining parentheses as character data. Unlike other CLIST operators, left and right parentheses can appear at the beginning or in the middle of character data without having to be defined as character data. Only when they appear at the end of a character string do parentheses have to be defined with &STR, like the other operators.

The following examples show how to define right and left parentheses to appear as character data at the end of a character string called TEXT:

Right parenthesis:

```
SET &A = )
SET &B = TEXT&STR(&A)          /* result: B = TEXT)
```

Left parenthesis:

```
SET &C = &STR((
SET &D = TEXT&STR(&C)          /* result: D = TEXT(
```

Using &STR with &SYSDATE or &SYSSDATE

If you use &SYSDATE or &SYSSDATE on a CLIST statement other than WRITE, enclose the variable in an &STR built-in function. Otherwise, a CLIST views the slashes separating the day, month, and year as division operators and performs division.

```
SET TODAY = &STR(&SYSDATE)
```

Using &STR with leading and trailing blanks

Use the &STR built-in function to preserve leading and trailing blanks in a character string. For example, the following statement sets the variable &CMNDFLD to a blank, 2 hyphens, a greater than symbol, and four blanks:

```
SET CMNDFLD= &STR( -->    )
```

Using &STR with strings that match CLIST statement names

You can use the &STR built-in function to distinguish installation-written commands that match the names of CLIST statements. For example, if your installation had written a command named NGLOBAL, you can use &STR to issue the command from a CLIST and prevent the CLIST from misinterpreting it as the NGLOBAL statement:

```
&STR(NGLOBAL)
```

Similarly, to issue the SELECT subcommand of the RACF command RACFRW, you need to use the &STR built-in function to distinguish the subcommand from the SELECT statement. For more information, see "Distinguishing the SELECT statement from the RACF SELECT subcommand" on page 71.

Using &STR when supplying input using SYSIN JCL statements

When you submit a background job that invokes a program, you sometimes include a '//SYSIN DD *' JCL statement that supplies the input statements. If any input statement contains leading blanks or is the same as a CLIST statement, enclose that statement in a &STR built-in function. For example, suppose a hypothetical language called SES has an IF-THEN-ELSE sequence. If you were to include such a sequence in the SYSIN input statements, you need to have to enclose it in an &STR built-in function as shown in the following background invocation of a hypothetical SES program called MATRIX.

```
PROC 1 FORMAT ACCT() CLASS(A)
CONTROL MAIN
⋮
submit * end(nn)
//&SYSUID1 JOB   &ACCT,&SYSUID,CLASS=&CLASS;
//STEP1     EXEC  PGM=MATRIX
⋮
//SYSIN     DD    *
&STR(  IF &FORMAT=1 THEN OPEN DS1)
&STR(ELSE OPEN DS2)
GETFILES 1-12
&STR(SET COLUMNS=GETFILES)
⋮
nn
```

Only those input statements that contain leading blanks or are the same as CLIST statements are enclosed in &STR built-in functions. If the CLIST invoked MATRIX in the foreground, the &STR built-in functions need to be unnecessary because the

program's statements need to appear in the data set containing MATRIX. Thus, they need to be associated with the program, not the CLIST.

Defining a substring - &SUBSTR

Use the &SUBSTR built-in function to request that a CLIST use only certain bytes of an indicated string when performing substitution. You indicate the starting and ending positions of the string from which the substitution is made.

For example, assuming a variable called &ANIMALS contains the character string "DOGSCATSSEALS", to set a variable called &FELINE to the character string "CATS", code the following:

```
SET FELINE = &SUBSTR(5:8,&ANIMALS)
```

Note that the character string "CATS" begins in the fifth position of &ANIMALS and ends in the eighth position.

A &SUBSTR built-in function can contain other built-in functions. Assume your CLIST receives input from the user and assigns it to a variable called &NAME. &NAME contains a person's first and middle initial followed immediately by the family name. To add a blank between the initials and the family name, you can set a variable called &NFIELD to a character string consisting of the following:

1. The first and middle initials
2. A blank
3. The family name.

```
SET NFIELD = &STR(&SUBSTR(1:2,&NAME) &SUBSTR(3:&LENGTH(&NAME)+
             ,&NAME))
```

If you want the substring to contain only one character, you can omit the colon and end-expression. For example, if you are interested only in the first letter of the family name, code the following:

```
SET FLTRLNAME = &SUBSTR(3,&NAME)
```

You can substitute variables for starting and ending expressions. For instance, to set the section of &STRING beginning at the second position and ending at the eighth position to a variable called &WIDGET, you can create a variable and substitute it in the SET statement. Assume that the substring data represents a part number.

```
SET PART# = &STR(2:8,)
SET WIDGET = &SUBSTR(&PART#&STRING)
```

When a variable is named in &SUBSTR, arithmetic evaluation of the variable's contents is suppressed, as in &STR. For example:

```
SET DIMENSNS = &STR(2*4)
SET X = &SUBSTR(1:2,&DIMENSNS)        /result: X = 2*
```

However, when another built-in function such as &LENGTH is specified in the &SUBSTR, the variable within the built-in function is evaluated before the &SUBSTR. To protect that variable from arithmetic evaluation, use &STR.

```
SET DIMENSNS = &STR(2*4)
SET X = &SUBSTR(1:&LENGTH(&STR(&DIMENSNS)),&DIMENSNS)
/* result: X = 2*4
```

If a string contains data of the double-byte character set (DBCS), &SUBSTR counts each DBCS character as two bytes, and counts each DBCS delimiter as one byte.

For example, using *d1d2* to denote two DBCS characters and using < and > to denote the DBCS delimiters X'0E' (shift-out) and X'0F' (shift-in):

```
SET X = &SUBSTR(8:9(A<d1d2>BC)       /* result:  X = BC
```

When &SUBSTR returns DBCS data, &SUBSTR encloses the data between the DBCS delimiters X'0E' and X'0F'. &SUBSTR attempts to return the exact bytes requested. However, when the starting or ending positions of the substring are DBCS data or DBCS delimiters, &SUBSTR makes the following adjustments:

If the substring:	&SUBSTR does the following:
Starts on the first byte of a DBCS character	Replaces that byte with a single-byte blank and the right-next byte with a shift-out delimiter
Starts on the second byte of a DBCS character	Replaces that byte with a shift-out delimiter
Starts on a shift-in delimiter	Replaces that byte with a single-byte blank
Ends on shift-out delimiter	Replaces that byte with a single-byte blank
Ends on the first byte of a DBCS character	Replaces that byte with a shift-in delimiter
Ends on the second byte of a DBCS character	Replaces that byte with a single-byte blank and the left-next byte by a shift-in delimiter.

In addition, if the adjustment causes a not valid DBCS character, or a contiguous pair of DBCS delimiters, &SUBSTR replaces those by single-byte blanks. However, SUBSTR does not change any contiguous pairs of DBCS delimiters that were part of the original data string.

The following are several examples of the adjustment process. In the examples, the characters s, Dn, <, >, and b denote a single-byte character, double-byte character, shift-out delimiter, shift-in delimiter, and single-byte blank.

```
&SUBSTR(4:10,ss<D1D2D3D4>)    /* result:    b<D2D3>

&SUBSTR(5:11,ss<D1D2D3D4>)    /* result:    <D2D3>b

&SUBSTR(6:10,ss<D1><D3D4>)    /* result:    b<D3>

&SUBSTR(1:3,ss<D1D2D3D4>)     /* result:    ssb

&SUBSTR(3:5,ss<D1D2D3D4>)     /* result:    bbb
```

Because &SUBSTR may truncate data in DBCS strings, you can use &SYSCSUBSTR as an alternative to &SUBSTR for DBCS data.

Defining a substring - &SYSCSUBSTR

Use the &SYSCSUBSTR built-in function when you want a CLIST to treat double-byte character set (DBCS) characters as single characters in a substring operation. &SYSCSUBSTR differs from &SUBSTR in that &SYSCSUBSTR counts each DBCS character as one character, and does not count DBCS delimiters. If resulting substrings begin or end with DBCS characters, &SUBSTR adds DBCS delimiters as needed. For example:

```
SET X = &SUBSTR(2:3,<d1d2d3>) /* result:  X = <d2d3>
```

The same is true if a string contains both EBCDIC and DBCS characters:

```
SET Y = 1260
&SUBSTR(1:3,AB<d1d2d3>)        /* result:  X = AB<d1>
```

Except for the difference in treating DBCS characters, &SYSCSUBSTR is identical to &SUBSTR.

Converting character strings to uppercase characters - &SYSCAPS

Use &SYSCAPS to convert character strings to uppercase characters. &SYSCAPS does not modify special characters or DBCS characters included in the string. If a string begins with leading zeros, &SYSCAPS strips them off. Otherwise, &SYSCAPS does not modify numbers in the string. You can use variables containing the character strings in &SYSCAPS built-in functions.

You can use &SYSCAPS with &SYSLC to control the capitalization of text in a CLIST. For an example, see "Controlling uppercase and lowercase for READ statement input" on page 91.

Converting character strings to lowercase characters - &SYSLC

Use &SYSLC to convert character strings to lowercase characters. &SYSLC does not modify numbers, special characters, or DBCS characters included in the string. You can use variables containing the character strings in &SYSLC built-in functions. For data to be changed to lowercase, CONTROL ASIS or NOCAPS must be in effect or &SYSASIS must be set to the value ON.

Determining data set availability - &SYSDSN

Use the &SYSDSN built-in function to determine whether a specified data set or a specified data set and member exist and are available for use. If a data set has been migrated, &SYSDSN attempts to recall it. The data set name can be the name of any cataloged data set or cataloged partitioned data set with a member name. Additionally, if you specify a member of a partitioned data set, &SYSDSN checks whether you have access to the data set.

To suppress TSO/E messages issued by the &SYSDSN function, use the CONTROL NOMSG statement. For information about the CONTROL statement, see "CONTROL statement" on page 149.

&SYSDSN returns one of the following values:

```
OK   /* the data set or the data set and member exist
     /* and are available
MEMBER SPECIFIED, BUT DATASET IS NOT PARTITIONED
MEMBER NOT FOUND
DATASET NOT FOUND
ERROR PROCESSING REQUESTED DATASET
PROTECTED DATASET   /* a member was specified but the
                    /* data set is RACF-protected
VOLUME NOT ON SYSTEM
UNAVAILABLE DATASET   /* another user has an exclusive
                      /* ENQ on the specified data set
INVALID DATASET NAME, data-set-name
MISSING DATA SET NAME
```

When a data set is available for use, you may find it useful to get more detailed information. For example, if you later need to invoke a service that requires a specific data set organization, then use the LISTDSI statement. For a description of the LISTDSI statement, see "LISTDSI statement" on page 158.

For example, you can use the &SYSDSN built-in function with conditional logic (see Chapter 8, "Structuring CLISTs," on page 67) to determine which data set to allocate for use in a CLIST.

```
IF &SYSDSN('SYS1.MYLIB')=OK THEN +
  DO
    alloc f(utility) da('SYS1.MYLIB')
    call (iecompar)
  END
ELSE +
IF &SYSDSN('SYS1.INSTLIB(IECOMPAR)')=OK THEN +
  DO
    alloc f(utility) da('SYS1.INSTLIB')
    call iecompar
  END
ELSE +
⋮
```

Enclose fully-qualified data set names in single quotation marks when they appear in &SYSDSN built-in functions. You can use variables containing data set names in &SYSDSN built-in functions.

The &SYSDSN function issues message IKJ56709I if a syntactically not valid data set name is passed to the function. To prevent this message from being displayed, use CONTROL NOMSG.

```
PROC 0
SET DSNAME = ABCDEFGHIJ.XYZ       /* Syntactically not valid name,
                                  /* because a qualifier is longer
                                  /* than 8 characters
CONTROL NOMSG                     /* Set OFF to suppress any SYSDSN
                                  /* TSO/E messages
WRITE VALUE RETURNED BY SYSDSN ==>  &SYSDSN(&DSNAME)
EXIT CODE(0)
```

Locating one character string within another - &SYSINDEX

Use the &SYSINDEX built-in function to locate the position where one character string begins within another character string. In other words, &SYSINDEX returns the numeric index (or offset) of *string_1* within *string_2*. If SYSINDEX does not find *string_1* within *string_2*, &SYSINDEX returns a value of zero.

Use the following syntax:

```
&SYSINDEX(string_1,string_2[,start])
```

where:

string_1
: is the character string that you are searching for.

string_2
: is the character string to be searched in.

start
: is a numeric expression indicating where in *string_2* the search for *string_1* should begin. If omitted or zero, this value defaults to one.

In examples 1-4, assume that &X is DOG, &Y is CATDOGSDOG and &Z is 2:

```
1. SET A = &SYSINDEX(&X,&Y)          /*   result:  A = 4
```

- &SYSINDEX found DOG in the fourth position of CATDOGSDOG, thus the index is 4.

```
2. SET A = &SYSINDEX(&X,&Y,&Z)       /*   result:  A = 4
```

- &SYSINDEX started searching at the second position, and found DOG again at the fourth position.

```
3. SET A = &SYSINDEX(&X,&Y,3+&Z)     /*   result:  A = 8
```

  Because the search started in the fifth position (3+2) &SYSINDEX found the second occurrence of DOG, in the eighth position.

```
4. SET A = &SYSINDEX(&X,&Y,9)        /*   result:  A = 0
```

- The search started in the ninth position and &SYSINDEX can not find the target string DOG.

Blanks are valid in *string_1* and *string_2*. For example:

```
SET A = &SYSINDEX(is full,the car is full)  /* result:  A = 9
```

To search for a blank in *string_2*, you can set *string_1* to a variable containing the value &STR( ). For example:

```
SET BLANK = &STR( )
SET TARG = THIS IS A TEST
SET LOC = &SYSINDEX(&BLANK,&TARG)    /* result: &LOC = 5
```

If *string_1* or *string_2* might contain a comma or right parenthesis, first set the string to a variable's value using &STR, then use the variable in &SYSINDEX, again enclosed in &STR(...). For example:

```
SET ARG = &STR(,)
SET TARG = &STR((80,60))
SET &LOC = &SYSINDEX(&STR(&ARG),&STR(&TARG))   /* result: &LOC = 4

SET ARG = &STR())
SET TARG = &STR((80,60))
SET &LOC = &SYSINDEX(&STR(&ARG),&STR(&TARG))   /* result: &LOC = 7
```

Using &SYSINDEX with DBCS strings

&SYSINDEX can search for strings that contain characters of the double-byte character set (DBCS). The following considerations apply:

- Always include DBCS delimiters around DBCS characters in *string_1* and *string_2*. For example, using < and > to denote the DBCS delimiters X'0E' (shift-out) and X'0F' (shift-in):

  ```
  SET A = &SYSINDEX(<d2>,<d1d2d3>)  /* result:  A = 2
  ```

- *String_1* and *string_2* can have EBCDIC, DBCS, or mixed data. For example:

  ```
  SET X = &SYSINDEX(CD,A<d1d2>BCD)    /* result:  X = 5
  SET X = &SYSINDEX(<d2>,A<d1d2>BCD)  /* result:  X = 3
  ```

  EBCDIC and DBCS strings never match, even when they have the same hexadecimal values. For example:

  ```
  SET X = &SYSINDEX(AB,<d1d2d3>)      /* result:  X = 0
  /* where EBCDIC characters 'AB' and a DBCS character 'd2'
  /* have the same hexadecimal value.
  ```

- Contiguous shift-out/shift-in delimiters and contiguous shift-in/shift-out delimiters in *string_1* are treated as parts of the target. For example:

  ```
  SET X = &SYSINDEX(<d1><d2>,A<d1><d2>B)     /* result: X = 1
  SET X = &SYSINDEX(<d1><d2>,A<d1d2>B)       /* result: X = 0
  ```

- If *string_1* consists of DBCS delimiters only, they are searched for in *string_2*, and the result is the position of the character following the delimiters. For example:

  ```
  SET X = &SYSINDEX(<>,A<>BCD) /* result: X = 3
  ```

Limiting the level of symbolic substitution - &SYSNSUB

Use the &SYSNSUB built-in function to limit the number of times a CLIST performs symbolic substitution in a statement. With &SYSNSUB, you can limit the CLIST to from 0 to 99 levels of substitution.

&SYSNSUB has the following syntax:

```
&SYSNSUB(level,expression)
```

where:

level
is a positive whole number, or a symbolic variable that resolves to a positive whole number, from 0 to 99. This number tells the CLIST how many levels of symbolic substitution to perform on the expression. The level parameter cannot contain other built-in functions or expressions.

expression
is a CLIST expression whose level of symbolic substitution is to be controlled, and whose final value is to be frozen without further evaluation of any kind.

For example,

```
SET Y = 30             /* result: &Y contains 30
SET X = &&Y            /* result: &X contains &Y
SET Z = &&X            /* result: &Z contains &X
SET A = &SYSNSUB(2,&Z) /* result: &A contains &Y
```

As specified, the CLIST performs only two levels of substitution, substituting &X for &Z and then substituting &Y for &X. The CLIST *does not* continue and resolve &Y to 30, as it is required to be without the &SYSNSUB limit.

You can use &SYSNSUB to override the rule for double ampersands, in which the CLIST removes the first ampersand and does no substitution of the remaining variable. &SYSNSUB counts removal of the first ampersand as one level of substitution, and allows substitution to continue until the value in the *level* parameter is reached.

For example:

```
SET X = 10              /* result: &X = 10
SET Y = &&X             /* result: &Y = &X (rule for double &&)

SET Y = &SYSNSUB(2,&&X) /* result: &Y = 10 (&SYSNSUB overrides &&)
```

Note: The control variable &SYSSCAN restricts the levels of substitution that you can specify with &SYSNSUB. &SYSSCAN must contain a number greater than or equal to the number you specify in &SYSNSUB's level parameter.

Converting DBCS data to EBCDIC - &SYSONEBYTE

Use the &SYSONEBYTE built-in function to convert character strings from the double-byte character set (DBCS) to the EBCDIC character set. &SYSONEBYTE converts only DBCS characters that have EBCDIC equivalents: the DBCS blank (X'4040') and DBCS characters that begin with the value X'42'.

&SYSONEBYTE converts the DBCS characters that have EBCDIC equivalents by removing the first byte (X'40' or X'42'). The second byte, which remains, represents the character in EBCDIC.

&SYSONEBYTE places DBCS delimiters around DBCS characters that are not convertible (those that lack EBCDIC equivalents).

The following example represents a complete conversion from DBCS to EBCDIC:

```
SET X = &SYSONEBYTE(<d1d2d3d4>)   /* result: X = ABCD
```

The following example represents a partial conversion from DBCS to EBCDIC, assuming that d5 and d6 do not start with X'42' and are not the hex blank (X'4040'):

```
SET X = &SYSONEBYTE(<d3d4d5d6d7d8>)  /* result: X = CD<d5d6>EF
```

Converting EBCDIC data to DBCS - &SYSTWOBYTE

Use the &SYSTWOBYTE built-in function to convert EBCDIC characters to the double-byte character set (DBCS). The EBCDIC characters that can be converted are those with the hexadecimal equivalents X'40' and in the range from X'41' to X'FE'. Any other EBCDIC characters cause errors when used with &SYSTWOBYTE.

&SYSTWOBYTE converts the EBCDIC characters to DBCS by prefixing them with the value X'42'. In the case of the EBCDIC blank (X'40'), &SYSTWOBYTE prefixes it with the value X'40' to create the DBCS blank.

&SYSTWOBYTE encloses the resulting DBCS strings in DBCS delimiters (X'0E' and X'0F').

The following example represents a complete conversion from EBCDIC to DBCS:

```
SET X = &SYSTWOBYTE(ABCD)         /* result: X = <dAdBdCdD>
```

The following example represents a partial conversion from EBCDIC to DBCS:

```
SET X = &SYSTWOBYTE(CD<d5d6>EF)   /* result: X = <dCdDd5d6dEdF>
```

Chapter 8. Structuring CLISTs

A CLIST can be:

- A single list of commands and statements
- A series of short lists connected by statements indicating which list is to be executed next

When you create a CLIST as a series of short lists, you can connect the lists using structured programming techniques. In structured programming, you direct the flow of execution from list to list in a generally top-down sequence, from the highest to the lowest level of detail. At the lower levels of detail, the lists can be independent modules (subprocedures and nested CLISTs) containing common code that you can call from other parts of the CLIST. A structured CLIST helps you avoid repetitive code and is easier to read and maintain than an unstructured CLIST.

This chapter describes the structural elements of the CLIST language and how to use them to move from one list of commands and statements to another. Structural CLIST statements belong to the following categories:

- Selection
- Loops
- Calls to subprocedures
- Calls to other CLISTs

Making selections

To tell the CLIST which commands or statements to execute next, you can use the IF Statement or the SELECT statement. These statements combine each selection with a test; if the test proves true, the CLIST executes the instructions, if not, the CLIST can execute alternative instructions.

The IF-THEN-ELSE sequence

The IF-THEN-ELSE sequence tests a condition or set of conditions, then determines the logical path of execution (action) based on the results of the test.

The *condition* must be either a comparative expression or a variable containing a comparative expression. You may code multiple conditions, in which case the comparative expressions, variables or both must be joined by logical operators.

The *action* can be one or more instructions. If the condition or set of conditions is true, the CLIST executes the instructions in the THEN action. If the condition or set of conditions is false, the CLIST executes the instructions in the ELSE action.

The standard format

The standard format includes actions for both true and false conditions, for example:

```
IF condition THEN action ELSE action
```

If an action involves more than one statement or command, it is necessary to enclose the action in a DO-END sequence, for example:

```
IF condition THEN +
 DO
:
  (action)     /* action consists of a list of statements or commands
:
 END
ELSE action    /* action consists of a single statement or command
```

For example, assume a CLIST optionally prints a data set it has updated based on user input. Assume the CLIST has prompted the user to determine whether to print the data set and has saved the response in a variable called &PRINT; The following IF-THEN-ELSE sequence performs the desired processing:

```
/*********************************************************************/
/* If the user wants data set printed, issue a message               */
/* saying that it is being printed and issue the command             */
/* that prints it.  If user does not want data set printed           */
/* just issue a message saying that the data set is not              */
/* being printed.                                                    */
/*********************************************************************/

IF &PRINT=YES THEN +
 DO
  WRITE We are printing the data set as you requested.
  printds da(&dsn)
 END
ELSE +
 WRITE The data set will not be printed.
```

When there is only one instruction in an action, you may place the instruction on the same line as the THEN or ELSE statement. For example, you can code the ELSE statement in the previous example as follows:

```
ELSE WRITE The data set will not be printed.
```

The Null ELSE format

When a specific ELSE action is not required, you can code a null ELSE clause in one of two ways: omit the ELSE clause entirely or just code ELSE without operands (an action). The following IF-THEN-ELSE sequence omits the ELSE entirely:

```
IF &PRINT=YES THEN +
 DO
  WRITE We are printing the data set as you requested.
  printds da(&dsn)
 END
```

You can also code the following:

```
IF &PRINT=YES THEN +
 DO
  WRITE We are printing the data set as you requested.
  printds da(&dsn)
 END
ELSE
```

The Null THEN format

Assume a CLIST prints a data set itself and does not have to invoke another CLIST to do the printing. By coding a condition that is true when the data set should not be printed, you define a null THEN clause that effectively branches to the end of the ELSE clause, avoiding the code that prints the data set.

The following IF-THEN-ELSE sequence bypasses the printing action when &PRINT=NO; (If &PRINT has any other value, such as YES or null, then printing is performed.)

```
IF &PRINT=NO THEN
ELSE +
 DO
⋮
  (The rest of the CLIST, which prints the data set)
⋮
 END
```

Nesting IF-THEN-ELSE sequences

IF-THEN-ELSE sequences can contain other (*nested*) IF-THEN-ELSE sequences. For example, the following IF-THEN-ELSE sequence uses a nested IF-THEN-ELSE sequence as the action of its ELSE clause:

```
IF condition1 THEN +
  DO
   action1      /* Do if condition 1 is true
  END
ELSE +
  IF condition2 THEN +
    DO
     action2     /* Do if condition1 is false and
    END          /* condition2 is true
  ELSE +
    DO
     action3     /* Do if condition1 and condition2
    END          /* are both false
```

Nested IF-THEN-ELSE sequences allow you to control the flow of processing under very precise conditions. However, multiple nested IF-THEN-ELSE sequences can be difficult to write and maintain. As an alternative, you can use the SELECT statement in many cases.

The SELECT statement

In situations where you might want to use multiple IF-THEN-ELSE statements, you can often use a single SELECT statement instead. The SELECT statement allows a CLIST to select actions from a list of possible actions. An action consists of one or more statements or commands. The SELECT statement has the following syntax, ending with the END statement. You can use the SELECT statement with or without the initial test expression.

```
SELECT [test expression]
  WHEN expression1
⋮
  (action)
⋮
  WHEN expression2
  WHEN expression3

  [OTHERWISE]
⋮
  (action)
⋮
END
```

Using SELECT without a test expression (simple SELECT)

If you omit the test expression from the SELECT statement, the CLIST tests the WHEN expressions in sequence for a true value. If a true value is found (for example, 1 = 1) the CLIST executes the action of that WHEN clause only. Then the

CLIST passes control to the END statement. If none of the expressions evaluate to a true value, the CLIST executes the OTHERWISE action, if any.

For example, the following SELECT statement selects an action based on a return code from previous processing:

```
SELECT
  WHEN (&RTNCODE = 0) CALL 'A.B.LOAD(PGM)'
  WHEN (&RTNCODE = 1) +
    DO
      SET &X = X + 1
      SET RETRY = &STR(YES)
    END
  OTHERWISE SET &MSG = &STR(SEVERE ERROR)
END
```

For other examples of using the simple SELECT statement, see "The COPYDATA CLIST" on page 108 and "The PROFILE CLIST" on page 137.

Using SELECT with a test expression (compound SELECT)

If you include a test expression on the SELECT statement, the CLIST compares the test expression to the expressions on the WHEN clauses. On each WHEN clause, you can specify multiple expressions, or a range of values by using a colon (:) between the low and high values in the range. You can combine expressions and ranges on a WHEN clause by using the operator OR or |.

If a test expression matches a value or falls within a range of values in a WHEN expression, the CLIST executes the action for that WHEN clause, then passes control to the END statement.

For example, in the following SELECT statement, the CLIST executes the action of the first WHEN clause because the test expression (5) falls within the range of values 4:6 on that WHEN clause:

```
SELECT 5
  WHEN (3 | 7 | 4:6)       action...
  WHEN (9 | &A + &Z)       action...
END
```

If no WHEN expressions satisfy the test expression, the CLIST executes the OTHERWISE action, if any.

For example, the following CLIST uses a SELECT statement to invoke other CLISTs that print quarterly reports. The CLIST bases its selection on a test expression (the number of the month) that the invoker supplies. When the number of the month falls within a certain range, the CLIST prints the appropriate report. Otherwise, the CLIST writes an error message.

```
PROC 1 MONTH
SELECT (&MONTH)
  WHEN (1:3)       %FIRSTQTR
  WHEN (4:6)       %SECNDQTR
  WHEN (7:9)       %THIRDQTR
  WHEN (10:12)     %FORTHQTR
  OTHERWISE  WRITE The month must be a number from 1 to 12.
END
```

Distinguishing WHEN clauses from WHEN commands

The WHEN clause in a SELECT statement is syntactically distinct from the WHEN SYSRC TSO/E command. In a SELECT statement, a left parenthesis must follow a WHEN clause. If you want to use the WHEN command as part of an action in a SELECT statement, enclose the WHEN command in a DO-END sequence to

prevent the SELECT statement from interpreting the command as a not valid WHEN clause. For example, the following syntax is acceptable:

```
SELECT
  WHEN (&X=1) +
   DO                           /* The action of the WHEN clause      */
    WHEN SYSRC(= 8) TIME        /* is the WHEN SYSRC TSO command.     */
   END                          /* End of the DO group                */
END                             /* End of the SELECT statement        */
```

For more information about using the WHEN SYSRC TSO/E command, see *z/OS TSO/E Command Reference*.

Distinguishing the SELECT statement from the RACF SELECT subcommand

If, in a CLIST, you invoke the SELECT subcommand of the RACF command RACFRW, you must distinguish the subcommand from the SELECT statement. To do so, use the &STR built-in function. For example, you can specify the subcommand name as follows:

```
RACFRW
&STR(SELECT) VIOLATIONS
```

Loops

Unlike the simple DO-END sequence, the other DO-sequences in the CLIST language create loops. Loops are lists of statements or commands that can be executed one or more times or not at all, depending on conditions that you specify in the loop. A CLIST executes a loop as many times as the conditions dictate. When the conditions are satisfied or no longer true, execution continues at the instruction after the loop.

The following sections describe how to create loops with the DO statement.

The DO-WHILE-END sequence

The DO-WHILE-END sequence creates a loop that executes while a specified condition is true. If the condition is not true, the loop does not execute.

To use the DO-WHILE-END sequence, code:

```
DO WHILE condition
.
.
.
  (action)
.
.
.
END
```

The *condition* must be either a comparative expression or a variable containing a comparative expression. You can code multiple conditions by joining expressions, variables, or both with logical operators.

The *action* can be one or more instructions. The CLIST executes the instructions within the sequence repeatedly *while* the condition on the WHILE clause is true. When the condition is false, the CLIST executes the next instruction after the END statement.

For example, you can initialize a variable (typically a counter) before the sequence and include it in the conditional expression. Then, you can modify the variable in the action so that eventually the condition is false.

For example, to process a set of instructions five times, you can code the following:

```
SET &COUNTER = 5                /* Initialize counter     */
/* Perform the action while counter is greater than 0     */
DO WHILE &COUNTER > 0
.
.
.
  (Set of instructions)
.
.
.
 SET COUNTER = &COUNTER - 1      /* Decrease counter by 1  */
END
```

The variable &COUNTER is a loop counter initially set to a value of five. WHILE tests of the value of this counter each time the CLIST begins to execute the DO-WHILE-END sequence. If the value of &COUNTER is greater than zero (the test condition is true), the CLIST executes the sequence, whose last instruction decreases the counter's value by one. When the counter's value reaches zero (the test condition is false), the CLIST ends the loop, and continues processing at the instruction following the END statement.

If an error occurs in a DO-WHILE sequence, execution stops. In previous releases, a warning message was issued and execution continued, with the DO-WHILE sequence treated as a simple DO-END sequence.

The DO-UNTIL-END sequence

The DO-UNTIL-END sequence creates a loop that executes at least once and continues until a specified condition is true.

To use the DO-UNTIL-END sequence, code:

```
DO UNTIL condition
.
.
.
  (action)
.
.
.
END
```

The *condition* must be either a comparative expression or a variable containing a comparative expression. You can code multiple conditions by joining expressions, variables, or both with logical operators.

The *action* can be one or more instructions. The CLIST executes the instructions within the sequence once, then tests whether the condition on the UNTIL clause is true. If the condition is false, the CLIST repeats the loop *until* the condition is true. When the condition is true, the CLIST ends the loop and executes the next instruction after the END statement.

For example, to repeat some instructions until a condition is true, you can code the following:

```
DO UNTIL &INPUT = YES  /* Perform action until condition is YES
.
.
.
  (action)
.
.
.
  WRITE Type YES if you are finished
  READ &INPUT;
END
```

The DO UNTIL sequence is useful for requesting input from a user. Because the decision is made *after* the input is received, the loop can continue or end depending on the value of the input.

The Iterative DO sequence

The iterative DO sequence creates a loop that executes if a numeric value stays within a given range of values. The values can be variables derived from CLIST processing. The iterative DO sequence has the following structure:

```
DO variable = from_expression TO to_expression +
[BY by_expression]
.
.
  (action)
.
.
END
```

where:

variable
is the control variable for the loop. Its value changes each time the loop executes, increasing by one (the default) or by a value that you specify in the BY expression.

from_expression
is a decimal integer, or an expression that evaluates to a decimal integer, from which the control variable starts. The CLIST sets the control variable to this value when the loop begins.

to_expression
is a decimal integer, or an expression that evaluates to a decimal integer, that the control variable must increase or decrease to. The CLIST executes the loop if the value of the control variable stays within the range created by the FROM and TO expressions.

by_expression
is a decimal integer, or an expression that evaluates to a decimal integer, by which the control variable increases or decreases. The default value is one. After the loop executes, the control variable increases or decreases by this amount. If the control variable is no longer within the FROM-TO range, execution continues at the instruction after the END statement.

For example, a CLIST need to execute the following loop ten times:

```
DO &count = 1 to 10       /* using default BY, increase &count by one
                          /* each time through the loop
.
.
END                       /* &count is now equal to 11
```

And a CLIST need to execute the following loop five times:

```
DO &count = 1 TO 10 BY 2  /* increase &count by two
                          /* each time through the loop
.
.
END                       /* &count is now equal to 12
```

The FROM, TO, and BY expressions can all contain CLIST variables:

```
DO &count = &min TO &max BY &increment
.
.
END
```

Compound DO sequences

The preceding sections describe different ways to control the execution of loops. You can combine these different types of loop control in a compound DO sequence. A compound DO sequence combines an iterative DO sequence with a DO-WHILE, DO UNTIL, or both sequences.

In a compound DO sequence, the iterative DO sequence comes first, followed by either the DO-WHILE or DO-UNTIL sequence:

```
DO variable = from_exp TO to_exp BY by_exp +
WHILE condition1 +
UNTIL condition2 +
:
  (action)
:
END
```

The CLIST executes the compound DO sequence as shown in Figure 2 on page 75.

The following example demonstrates a possible compound DO sequence:

```
SET &increment = 2     /* Initialize BY condition
SET &year = 87         /* Initialize WHILE condition
DO &count = 1 TO 10 BY &increment +
WHILE &year=87 UNTIL &input=YES;
:
  (action)
:
  WRITE Type YES if you are finished
  READ &INPUT;
END
```

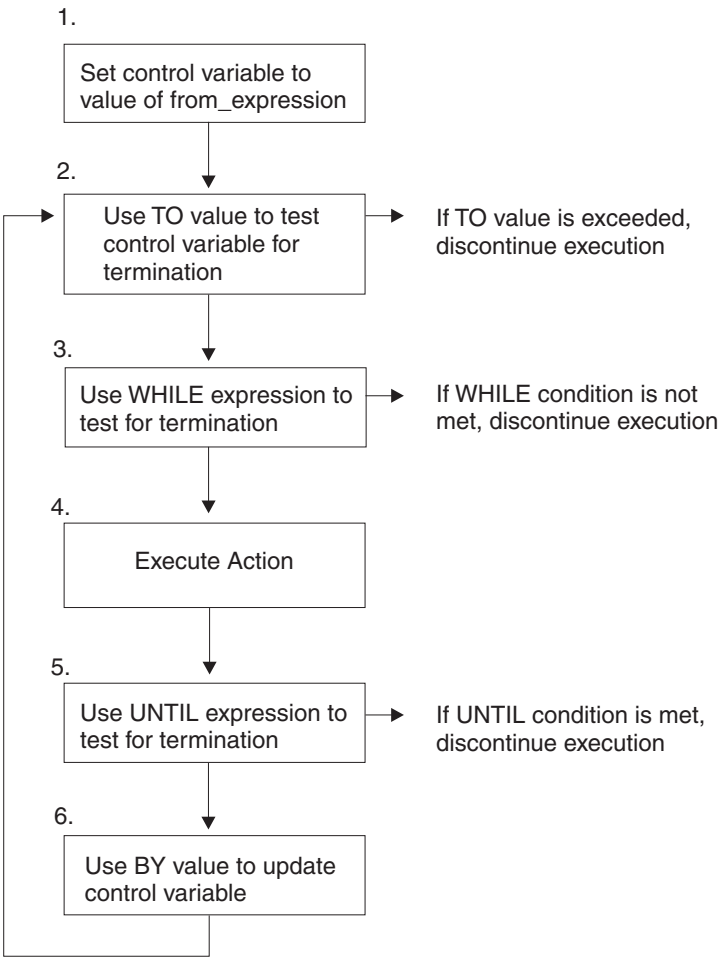


Figure 2. How a CLIST executes a compound DO sequence

If you want a WHILE or UNTIL expression to contain a return code from the action of the DO sequence, obtain the return code from &LASTCC and store it into another variable as part of the action. For example:

```
DO &I = 1 to 10 WHILE (&RCODE = 0)
⋮
 SET RCODE = &LASTCC
END
```

Nesting loops

The action of a loop can contain other loops. Loops within loops are called nested loops. Loops can contain nested loops of the same type or of a different type.

Nested loops of the same type are often iterative DO-loops within other iterative DO-loops. For example, to execute 100 CLISTs named PROC00 through PROC99, you can code:

```
DO &I = 0 to 9
 DO &J = 0 to 9
  %proc&I&J
 END
END
```

Nested loops of a different type are often DO-UNTIL loops within DO-WHILE loops, for example:

```
SET &COUNTER1 = 0               /* Initialize outer loop counter      */
SET &COUNTER2 = 3               /* Initialize nested loop counter     */
DO WHILE &COUNTER1 <  5
 /* Perform action while &counter1 is less than 5                     */
:
  (action)                      /* Executes 5 times                   */
:
 DO UNTIL &COUNTER2 = 0
  /* Perform action until &counter2 is equal to 0                     */
:
  (Subset of action)            /* Executes 3 times                   */
:
  SET COUNTER2 = &COUNTER2 - 1  /* Increase nested loop counter by 1  */
 END
:
 SET COUNTER1 = &COUNTER1 + 1   /* Increase outer loop counter by 1   */
END
```

Distinguishing END statements from END commands or subcommands

You can issue TSO/E END commands or subcommands in a CLIST. The END command terminates the CLIST, and END subcommands terminate certain commands, such as the TEST command. When you include TSO/E END commands or subcommands in the action of a DO-sequence or a SELECT statement, you must distinguish the END commands or subcommands from the END statement. You can distinguish the END statement using the CONTROL statement or the DATA-ENDDATA sequence.

Using the CONTROL statement

One way to distinguish an END statement from an END command or subcommand is by coding a CONTROL statement with the END operand. The value you code for the END operand must then be substituted for the END statement anywhere in the CLIST, unless another CONTROL END overrides the value.

For example, if you want to substitute ENDO for the END statement, you can code the following:

```
CONTROL END(ENDO)
SET COUNTER = 10
DO WHILE &COUNTER GT 0
:
  (action)
:
  test datapak(newpgm)          /* Issue TSO/E TEST command           */
:
    (TEST subcommands)
:
  end                           /* Issue END subcommand of TSO/E TEST */
:
  (more action)
:
 SET COUNTER = &COUNTER - 1     /* Decrease counter by 1              */
ENDO
```

Using the DATA-ENDDATA sequence

Another way to identify END commands or subcommands in DO-sequences or SELECT statements, is to place them in a DATA-ENDDATA sequence. For example:

```
SET COUNTER = 10
DO WHILE &COUNTER GT 0
⋮
  (action)
⋮
  DATA
  test datapak(newpgm)           /* Issue TSO/E TEST command           */
⋮
  (TEST subcommands)
⋮
  end                            /* Issue END subcommand of TSO/E TEST */
  ENDDATA
⋮
  (more action)
⋮
 SET COUNTER = &COUNTER - 1      /* Decrease counter by 1              */
END
```

Only TSO/E commands and subcommands can appear within the DATA-ENDDATA sequence. If a CLIST statement is included, TSO/E attempts to execute it as a TSO/E command, causing an error. For more information about the DATA-ENDDATA sequence, see "Coding responses to prompts - the DATA PROMPT-ENDDATA sequence" on page 87.

Subprocedures

A subprocedure is a part of a CLIST that you can call from one or more places in a CLIST. With subprocedures, you can organize a CLIST into logical units, making the CLIST easier to write and maintain. You can also keep common code in a single location and call it from other parts of the CLIST, thus avoiding repetitive code.

Subprocedures offer a variety of ways to communicate information within a CLIST. You can:

- Pass parameters to and from subprocedures, for reference or modification
- Share variables globally among subprocedures
- Isolate variables in a subprocedure from the rest of the CLIST

Calling a subprocedure

You call a subprocedure using the SYSCALL statement. On the SYSCALL statement, name the subprocedure and any parameters you want to pass to the subprocedure. The parameters can be data strings, variable values, or variable names.

For example, the following CLIST uses the SYSCALL statement to pass a data string (Jones), a variable value (&A), and a variable name (B) to a subprocedure (XYZ):

```
   SET &A = AL
   SET &B = Jr.
   SYSCALL XYZ Jones &A B        /* pass parameters to XYZ             */

XYZ: PROC 3 PARM1 PARM2 PARM3    /* receive parameters on PROC stmt    */
       SYSREF PARM3              /* indicate parm3 holds a var. name   */
       WRITE &PARM1, &PARM2 &PARM3  /* result: JONES, AL Jr.          */
     END
```

Subprocedures always begin with a labeled PROC statement. The label can consist of 1-31 characters (A-Z, 0-9, #, $, @) beginning with an alphabetic character (A-Z). In the example above, the label is XYZ; the number 3 on the PROC statement indicates that the subprocedure receives 3 positional parameters; those parameters are assigned to the variables PARM1, PARM2, and PARM3. For more information about the PROC statement, see "PROC statement" on page 169.

The SYSREF statement tells the CLIST that PARM3 contains the name of a variable (B). The SYSREF statement allows other statements in subprocedure to reference and modify the variable's value (Jr.). For more information, see "Using the SYSREF statement" on page 79.

To pass a parameter containing blanks to a subprocedure, set a variable equal to the parameter value, then refer to that variable (without the ampersand) using &STR on the SYSCALL statement. In the subprocedure, use the SYSREF statement to refer to the PROC statement parameter that corresponds to the variable name passed on the SYSCALL statement. For example,

```
SET &A = JOHN AL
SYSCALL XYZ &STR(A)  /* Pass variable to XYZ, omitting & from
                            /* the variable name
.
.
.
XYZ: PROC 1 PARM     /* Subprocedure XYZ
SYSREF &PARM         /* indicate PARM holds a variable name
WRITE &PARM          /* result: JOHN AL
```

Subprocedures must always end with the END statement. When subprocedures end, they pass control back to the statement following the SYSCALL statement.

Subprocedures can use the SYSCALL statement to:
- Call other subprocedures and pass parameters to them
- Call themselves
- Call the CLIST's main procedure, if it has a label

Returning information from a subprocedure

Subprocedures can return information to the caller using:
- Return codes
- SYSREF variables
- NGLOBAL variables

Using the RETURN CODE statement

Subprocedures can return information to the caller using the CODE option of the RETURN statement. Like return codes from TSO/E commands, return codes from subprocedures are stored in the control variable &LASTCC, but error return codes from subroutines will not cause an error routine to receive control.

In the following example, the subprocedure passes a return code to the statement following SYSCALL:

```
   SET &A = AL
   SYSCALL XYZ &A               /* pass variable &A to XYZ           */
   IF &LASTCC = 0 THEN +
     WRITE All's Well!

XYZ: PROC 1 PARM1
       WRITE &PARM1
       RETURN CODE(0)
     END
```

Using the SYSREF statement

When a SYSCALL statement passes a variable name (without the ampersand), the subprocedure can use a SYSREF statement to let following statements reference and modify the variable's value. All changes to a SYSREF variable are retroactive; that is, the new values are assigned to the original variable back in the caller.

In the following example, the subprocedure gives a new value to the variable whose name is passed (A). The new value (GEORGE) replaces the old value (AL) in the caller.

```
   SET &A = AL
   SYSCALL XYZ A                /* pass var. &A to XYZ, omitting the &*/;
   IF &LASTCC = 0 THEN +
     WRITE &A                   /* result: GEORGE

XYZ: PROC 1 &PARM1
       SYSREF &PARM1            /* refer changes to the caller        */
       SET &PARM1 = GEORGE
       RETURN CODE(0)
     END
```

Reminder: For SYSREF variables, always omit the ampersand (&) from corresponding variables on the SYSCALL statement. By omitting the ampersand on SYSCALL, you pass the name of the variable, not its value, to the subprocedure. Using the SYSREF statement, the subprocedure can then assign new values to the variable.

Sharing variables among subprocedures

In addition to passing return codes and variable values, you can define common variables to be shared among different CLISTs, or among subprocedures in a single CLIST.

Variables shared among different CLISTs are called GLOBAL variables. GLOBAL variables are defined using the GLOBAL statement, and are fully described in “Nesting CLISTs” on page 81.

Variables shared by subprocedures in one CLIST are called NGLOBAL (named global) variables. You define named global variables with the NGLOBAL statement. When you define an NGLOBAL variable, any subprocedure in the same CLIST can refer to it by name and modify its value.

The NGLOBAL variables differ from GLOBAL variables in that:
- They are not global to (shared with) other CLISTs.
- They are defined by name only (not position).
- They need to be defined only once.

Using the NGLOBAL statement

The NGLOBAL statement names variables that all the subprocedures in a CLIST can use. The following subprocedure (ABC) defines variables A, B, and C and uses the NGLOBAL statement to make them available to other subprocedures in the CLIST:

```
ABC: PROC 0                 /* In subprocedure ABC,
NGLOBAL A,B,C               /* define NGLOBAL variables
SET A = apples
SET B = bananas
SET C = cantaloup
SYSCALL XYZ                 /* call subprocedure XYZ
END
```

```
XYZ: PROC 0                 /* In subprocedure XYZ,
WRITE Mix &A, &B, and &C    /* use the NGLOBAL variables
END
```

The NGLOBAL statement must precede any statement that uses its variables. The number of variables that you can name on the NGLOBAL statement is unlimited.

For another example of using the NGLOBAL statement with subprocedures, see "Allocating a data set with LISTDSI information - the EXPAND CLIST" on page 143.

Restricting variables to a subprocedure

Variables that you define in a subprocedure are local to that subprocedure, unless you specifically name them on a GLOBAL or NGLOBAL statement. Different subprocedures in a CLIST can have variables with the same name, and each variable is local to the subprocedure that defined it. Therefore, when you define a variable, you don't have to check to see if that name has been used in the CLIST before.

Considerations for using other statements in subprocedures

Some CLIST statements require special consideration when used in subprocedures. The following sections describe these statements and considerations.

Using ATTN and ERROR statements in subprocedures

Subprocedures can have their own attention and error routines. These are routines that receive control when the CLIST user presses the attention key on a terminal keyboard, or an error occurs. See Chapter 11, "Writing ATTN and ERROR routines," on page 103 for a full description of these routines, including special considerations for using them with subprocedures. For example, a subprocedure's attention or error routine cannot contain a nested attention or error routine.

When a subprocedure receives control, the caller's attention and error routines remain in effect until the subprocedure issues an ATTN or ERROR statement. Then the subprocedure's attention or error routine prevails until the routine is turned off or replaced, or the subprocedure ends. When the subprocedure ends, the caller's attention and error routines take control again.

Using CONTROL statements in subprocedures

CLISTs can establish special conditions by issuing the CONTROL statement and certain control variables. These conditions, comprising a CONTROL environment, remain in effect when you call a subprocedure. Subprocedures can set up their own CONTROL environment, but it only applies to the subprocedure and any subprocedures it calls. When a subprocedure ends, the caller's CONTROL environment takes effect again.

Using GOTO statements in subprocedures

If you use a GOTO statement in a subprocedure, it can only branch to labels in the same subprocedure. Also, GOTO statements cannot branch to PROC statements.

Nesting CLISTs

A CLIST can invoke another CLIST, which in turn can invoke another, and so forth. CLISTs that are invoked by other CLISTs are called nested CLISTs. When a nested CLIST ends, it automatically branches back to the statement following the one that invoked it. You can define global variables that allow nested CLISTs to communicate with each other.

You can structure a series of nested CLISTs in levels. The CLIST invoked by the user is the top-level or outer-level CLIST in the nesting chain. CLISTs invoked by the outer-level CLIST are nested within it, and they may have lower-level CLISTs nested within them.

In Figure 3, PROC1 is the outer-level CLIST. It invokes PROC2 and then PROC3, which are nested within it. PROC2 invokes PROC4, and PROC4 invokes PROC5. PROC4 is nested within PROC2, and PROC5 within PROC4.

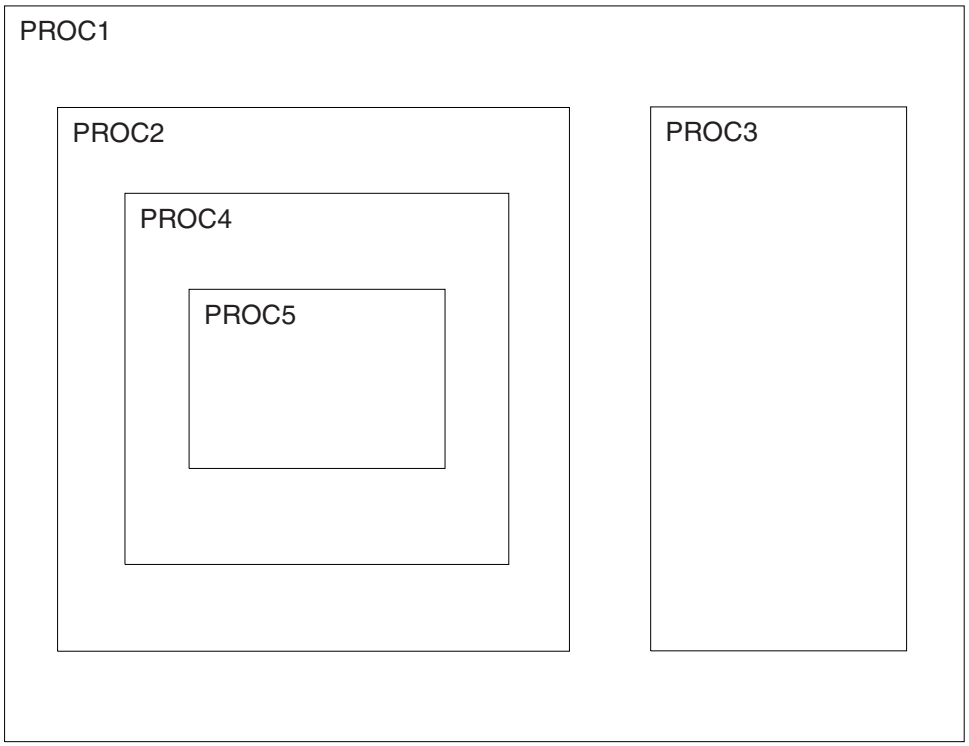


Figure 3. Nested CLISTs

Because CLISTs are executed sequentially, PROC1 cannot invoke PROC3 until PROC5, PROC4, and PROC2 finish processing.

The same CLIST can be invoked at two or more levels of a nested hierarchy because each invocation of a nested CLIST causes a new copy of it to be brought into storage. For example, PROC2 and PROC4 can both invoke PROC5.

Protecting the input stack from errors or attention interrupts

When a CLIST is executed, it translates each statement into an executable format and places it in a section of storage called the input stack. The input stack is the source from which TSO/E obtains its input (TSO/E commands and CLIST statements).

For nested CLISTs, the input stack holds the contents of the CLISTs in the order in which they are nested.

You can protect the input stack from being erased (flushed) when an error or attention interrupt occurs. To protect the input stack, code the CONTROL statement with the NOFLUSH or MAIN operand at the beginning of a CLIST that you want to receive control when an error or attention interrupt occurs.

Any options established by a nested CLIST are in effect only when that nested CLIST is executing. In particular, a nested CLIST's CONTROL statement options and attention and error routines are no longer in effect when the nested CLIST returns control to its caller.

Nested CLISTs in the subcommand environment (those invoked under the EXEC subcommand of EDIT) can execute only subcommands and CLIST statements. They cannot execute TSO/E commands, nor can any nested CLISTs that they invoke, until the END subcommand is executed.

Global variables

Global variables are variables defined on a GLOBAL statement. They allow communication between nested CLISTs. Any CLIST in the nested chain can modify or reference the value of a global variable.

All global variables in a given CLIST must have unique names. You cannot have more global variables on the GLOBAL statement in a nested CLIST than there are on the GLOBAL statement in the top-level CLIST.

To establish global variables, first determine the total number of symbolic variables that are referenced by more than one of the CLISTs in the nested chain. (Include the top-level CLIST among those in the nested chain.) Then, code GLOBAL statements in each of the CLISTs in the chain that are involved in the passing of data.

For example, in Figure 3 on page 81, assume the following global variable definitions in each of the CLISTs:

```
In PROC1:  GLOBAL  A B C D
In PROC2:  GLOBAL  X Y Z
In PROC3:  GLOBAL  F G H K
In PROC4:  GLOBAL  Q
In PROC5:  GLOBAL  R S.
```

Variables &A, &X, &F, &Q, and &R can be shared by all the CLISTs. If PROC4 sets &Q equal to D777, then &A, &X, &F, and &R are also set equal to D777.

Within nested CLISTs, global variables are positional; that is, all variables defined first refer to the same variable; all variables defined second refer to the same variable; and so on.

Exiting from a nested CLIST

There are three ways to exit from a nested CLIST:

- Let control automatically return to the calling CLIST at the end of the nested CLIST.
- Issue an END command.

- Issue an EXIT statement.

Using the END command

The END command only allows you to terminate a CLIST. Control returns to the CLIST that invoked it, but you cannot set a return code. To use the END command, code:

```
end
```

The END command just terminates a CLIST and should not be used if a return code is to be passed back to a calling CLIST. A calling CLIST may find the return code in an unpredictable state. Use the EXIT statement where proper passing of a return code to a caller is required.

Using the EXIT statement

To cause a nested CLIST to return control to the CLIST that invoked it, you can also code:

```
EXIT
```

You can specify a return code on the EXIT statement. The return code provides a way for lower-level CLISTs to pass back to their callers indications of errors or successful execution. To pass a return code when you exit, code:

```
EXIT CODE(expression)
```

The expression must be a positive integer, zero, or a symbolic variable whose value, after substitution, is an integer. The nested CLIST stores the value of the expression into the control variable &LASTCC.

If an error or attention interrupt occurs, a nested CLIST can pass control back to a CLIST that is protected from termination by the CONTROL MAIN or CONTROL NOFLUSH options. To return control to such a CLIST, code:

```
EXIT QUIT
```

or

```
EXIT CODE(expression) QUIT
```

If a CLIST in the nested chain is protected from termination, execution continues based on actions in the CLIST's active error or attention routine. For information about writing error and attention routines, see Chapter 11, “Writing ATTN and ERROR routines,” on page 103.

If no CLIST in the nested chain is protected from being terminated after an error or an attention interrupt, coding QUIT causes control to return to the environment from which the CLIST was invoked: TSO/E, TSO/E EDIT mode, or ISPF.

GOTO statements

The GOTO statement causes an unconditional branch to a label within a CLIST. The label may be a variable whose value, after symbolic substitution, is a valid label within the CLIST. Examples of using GOTO statements are:

```
    IF &A = 555 THEN GOTO A1
    IF &A NE 0 THEN GOTO A2
A1:  processing
:
:
     A2:  processing
```

```
.
.
.
    SET TARGET = B1
    IF &X = 777 THEN GOTO &TARGET
    ELSE +
      DO
       SET TARGET = B2
.
.
.
       IF LASTCC = 0 THEN +
       SET TARGET = B1
       GOTO &TARGET
      END
B1:  processing
B2:  processing

.
.
.
```

GOTO statements cannot branch:

- To another CLIST
- To a subprocedure's PROC statement
- From one subprocedure to another
- From a subprocedure to the CLIST's main procedure

Chapter 9. Communicating with the terminal user

The CLIST language offers several ways to communicate with the terminal user. These methods are:

- Prompting the user for input
- Writing messages to the user
- Receiving replies from the user
- Passing control to the user
- Using ISPF panels

Prompting the user for input

A CLIST can prompt for input by:

- Using a PROC statement with positional or keyword parameters on the first line of the CLIST
- Using WRITE and WRITENR statements
- Using TSO/E commands

Prompting with the PROC statement

When you include positional parameters on a PROC statement at the beginning of a CLIST, the CLIST user must supply a value for each of them. If the user does not specify a value at execution, the CLIST prompts until the user specifies a value. For example, the PROC statement

```
PROC 2 NAME ADDRESS
```

requires the user to pass two positional parameters at execution, for example:

```
EX clistname 'Jones Fishville'
```

If the user does not pass a parameter, the CLIST prompts for a NAME and an ADDRESS. A PROC statement at the beginning of a CLIST also prompts when a user passes a keyword parameter without a required value. For example, the following PROC statement allows the user to pass the parameter ACCT with a value in parentheses:

```
PROC 0 ACCT()
```

If the user passes ACCT without a value, for example,

```
EX clistname 'ACCT'
```

the CLIST prompts for a value.

Unlike PROC statements at the beginning of a CLIST, PROC statements on subprocedures do not prompt for missing parameters. For more information about the PROC statement, see “Using the PROC statement” on page 19.

Prompting with the WRITE and WRITENR statements

You can use either a WRITE or WRITENR statement, or a combination of both, to send a message to the terminal user and prompt for input. To obtain input after a WRITE or WRITENR, use the READ statement. For details about how to use the WRITE and WRITENR statements, see “Using the WRITE and WRITENR statements” on page 88.

Prompting with TSO/E commands

Some TSO/E commands, such as LISTDS, require more information than just the name of the command and they prompt when that information is not supplied. However, TSO/E commands included in a CLIST can prompt for input only when the CLIST allows prompting. Prompting in a CLIST is controlled by the TSO/E commands PROFILE and EXEC, and by the CLIST statement CONTROL and the control variable &SYSPROMPT.

The following table illustrates the effect on prompting using different explicit specifications of PROMPT/NOPROMPT on the PROFILE and EXEC commands and on the CONTROL statement. Note that SET &SYSPROMPT = ON has the same effect as CONTROL PROMPT and SET &SYSPROMPT = OFF has the same effect as CONTROL NOPROMPT.

Specifications	Prompting by TSO/E commands allowed in CLIST	
	YES	NO
profile prompt exec prompt CONTROL PROMPT	X	
profile prompt exec noprompt CONTROL PROMPT	X	
profile prompt CONTROL PROMPT	X	
profile prompt exec prompt	X	
profile noprompt exec prompt CONTROL PROMPT		X
profile prompt exec prompt CONTROL NOPROMPT		X
profile prompt exec noprompt		X
profile prompt		X

Note:

1. PROFILE PROMPT is the default specification and applies to a TSO/E session, not to a particular CLIST. You don't need to specify PROFILE PROMPT unless you want to override a prior PROFILE NOPROMPT command.
2. The PROFILE command can be executed either outside of, or within, a CLIST.
3. EXEC NOPROMPT is the default specification and applies only to the CLIST that it invokes.
4. The CONTROL statement applies only to the CLIST in which it appears.
5. If a CONTROL statement does not appear in a CLIST, CONTROL NOPROMPT is implied, unless &SYSPROMPT is set to a value of ON.

Coding responses to prompts - the DATA PROMPT-ENDDATA sequence

If you execute a CLIST in the background, a user cannot respond to prompts from the CLIST. To avoid this problem, use the DATA PROMPT-ENDDATA sequence. The DATA PROMPT-ENDDATA sequence lets you designate responses to prompts by TSO/E commands or subcommands.

To use the DATA PROMPT-ENDDATA sequence, code:

```
DATA PROMPT
⋮
          /* Responses */
ENDDATA
```

If the sequence is not immediately preceded by a TSO/E command or subcommand that prompts, an error occurs (error code 968 appears in control variable &LASTCC). You can ignore the error condition if a command or subcommand that *might* prompt, does not prompt.

The responses in the DATA PROMPT-ENDDATA sequence must appear exactly as if a user entered the response. Each DATA PROMPT-ENDDATA sequence can respond only to prompts issued by the immediately preceding command or subcommand. However, you can include multiple responses to satisfy multiple prompts. Excess responses can result in an error message and termination of the CLIST if an error routine is not present.

To stop TSO/E commands from prompting after a DATA PROMPT-ENDDATA sequence, code a null line after ENDDATA. To code a null line, first set a variable equal to null:

```
SET &abc =
```

Then place that variable on the line after ENDDATA:

```
ENDDATA
&abc
```

Some TSO/E commands prompt for input when you code certain operands. For example, the LINK command invokes the linkage editor. When you substitute an asterisk (*) for the data set name, TSO/E prompts for control statements. If you include such a LINK command in a CLIST that might run in the background, place the control statements within a DATA PROMPT-ENDDATA sequence. The following CLIST, when run in the background, link-edits the member *X*, which resides in the file *DD1*:

```
CONTROL PROMPT LIST
IF &SYSENV=FORE THEN /* CLIST is running in the foreground */  +
    link (*)         /* Prompt user for control statements */ +
    load('d32kds1.load') pr(*) ncal xref list let
ELSE                 /* CLIST is being run in the background */ +
  DO
    SET NULL =   /* set null line to stop prompting after ENDDATA
    link (*) +
    load('d32kds1.load') pr(*) ncal xref list let
    DATA PROMPT     /* Designate responses to prompts        */
    include dd1(x)
    entry x
    name x
    ENDDATA
    &NULL           /* null line stops prompting */
  END
```

There are additional considerations for using the DATA PROMPT-ENDDATA sequence:

- The CLIST must allow prompting.
- The CLIST performs symbolic substitution before using the responses to satisfy the prompt. (You can include variables in the responses.)

Writing messages to the terminal

CLISTs send two types of messages to the terminal user: messages that you specifically write from the CLIST, and informational messages from commands or statements in the CLIST.

Using the WRITE and WRITENR statements

Two CLIST statements are available for sending messages to the terminal and prompting for input:

- WRITE displays a message at the terminal and causes the terminal's display cursor to return to the beginning of the next line after the message is displayed.
- WRITENR displays a message at the terminal and causes the terminal's display cursor to remain at the end of the message. (The “NR” in WRITENR is for “No Return”.)

You can use either statement to send messages. You might find WRITENR preferable when the message prompts the user for input.

When prompting the user for input, include a READ statement after the WRITE or WRITENR statement. The READ statement *reads* the user input into a variable or variables. For more information, see “Using the READ statement” on page 89.

Both WRITE and WRITENR must be followed by one or more blanks and the text of the message. For example:

```
CONTROL ASIS
⋮
WRITE Your previous entry was invalid.
WRITE Do you want to continue?
WRITENR Enter yes or no.
```

As a result of these statements, the terminal user sees the following messages on the screen:

```
Your previous entry was invalid.
Do you want to continue?
Enter yes or no. __
```

The cursor stops after the period in the last line to indicate the CLIST is waiting for the user's response. Because CONTROL ASIS is specified the CLIST displays the message ‘as written’, in both uppercase and lowercase letters.

You can also use the WRITENR statement to join text. For example:

```
CONTROL CAPS
⋮
WRITENR Please enter your userid
WRITE followed by two blanks.
```

As a result of these statements, the terminal user sees the following message:

```
PLEASE ENTER YOUR USERID FOLLOWED BY TWO BLANKS.
```

Because CONTROL CAPS is specified, the message is translated to all capital letters before being displayed.

Controlling the display of informational messages

You can request that informational messages from commands or statements in a CLIST be displayed or suppressed using operands on the CONTROL statement or the &SYSMSG control variable.

- To request that they be displayed, code:

  ```
  CONTROL MSG
  ```

  or

  ```
  SET &SYSMSG = ON
  ```

- To suppress the display of informational messages, code:

  ```
  CONTROL NOMSG
  ```

  or

  ```
  SET &SYSMSG = OFF
  ```

The MSG/NOMSG option has no effect on error messages, they are always displayed.

Receiving responses from the terminal

The READ and READDVAL statements provide two ways for CLISTs to access user input from the terminal. The READ statement obtains input directly from the terminal, typically following a WRITE or WRITENR statement. The READDVAL statement obtains input from the &SYSDVAL control variable.

Using the READ statement

The READ statement makes terminal input available to a CLIST in the form of symbolic variables. You normally precede a READ statement with one or more WRITE or WRITENR statements to let the user know that the CLIST is expecting input, and what sort of input it is expecting.

You can include one or more symbolic variables on a READ statement. If a READ statement does not include any variables, the CLIST stores the information the user enters into the control variable &SYSDVAL.

Assume that a WRITE statement requests that the user enter four names. The accompanying READ statement can be coded as follows:

```
READ A,B,C,D
```

Note that variables on a READ statement do not require ampersands.

If the user's response to the previous WRITE statement is:

```
SMITH,JONES,KELLY,INGALLS,GREENE
```

The CLIST assigns the names to the symbolic variables on the READ statement as follows:

```
&A has the value SMITH.
&B has the value JONES.
&C has the value KELLY.
&D has the value INGALLS.
```

Because the READ statement only includes four variables, the CLIST ignores the fifth name (GREENE).

You can also code READ statements without variables:

```
READ
```

If the user responded with the same five names, they all need to be stored in the control variable &SYSDVAL. To preserve the input strings, the CLIST does not remove the delimiters. For example, if the user responds to the previous READ statement by entering "SMITH,JONES,KELLY,INGALLS,GREENE", &SYSDVAL has the following value:

```
SMITH,JONES,KELLY,INGALLS,GREENE
```

To assign a null value to one of the variables on a READ statement, the user can enter either a double comma or a double apostrophe (two single quotation marks). For example, assume that the CLIST sends a message to the user requesting four numbers. The READ statement to obtain these numbers is:

```
READ NUM1,NUM2,NUM3,NUM4
```

If the user responds either:

```
15,24,,73
```

or

```
'15' '24' '' '73'
```

The symbolic variables on the READ statement then have the following values:

```
&NUM1 has the value 15.
&NUM2 has the value 24.
&NUM3 has a null value.
&NUM4 has the value 73.
```

The fact that single quotation marks are valid delimiters requires that you exercise care when reading *fully-qualified* data set names into variables. Precautions are necessary because, if the user enters the data set name within single quotation marks (according to TSO/E naming conventions), the CLIST normally reads them as delimiters, not data. If a WRITE statement requests the name of a fully-qualified data set, the CLIST can obtain the data set name as entered by the user, with single quotation marks preserved, by using the READ statement with the &SYSDVAL control variable.

The following CLIST uses a READ statement and &SYSDVAL to preserve single quotation marks around a data set name. It also checks for the quotation marks to see if the user entered a fully-qualified data set name and, if not, adds the quotation marks and the user's prefix to the name.

```
PROC 0
WRITE Enter the name of a data set.
READ
SET &DSN = &SYSDVAL            /* Get name from &SYSDVAL; */
IF &SUBSTR(1:1,&DSN) ¬= &STR(') THEN +
```

```
 DO                             /* If not fully qualified, */
  SET &DSN = '&SYSPREF;.&DSN'   /* add prefix and quotation marks.  */
 END
WRITE &DSN
```

You can also use the READ statement to obtain values for PROC statement keywords that were not supplied on the invocation of the CLIST. For example, suppose a PROC statement defines &ALPHA as a keyword with a default null value. Assume &ALPHA contains the number of golf balls on the moon and that the user does not assign a value to &ALPHA when invoking the CLIST. However, a variable, &SPACEVENTS, in the CLIST results in code being executed that requires a non-null value for &ALPHA. To obtain a value for &ALPHA, the following code sends a message to the user requesting a value for &ALPHA. Then, it issues a READ statement with &ALPHA as a parameter.

```
PROC 0 ALPHA()
.
.
.
SET SPACEVENTS = &ALPHA
DO WHILE &SPACEVENTS =  /* Null */
 WRITE Enter the number of golf balls there
 WRITE are on the moon.  A null value is unacceptable.
 READ ALPHA
 SET SPACEVENTS = &ALPHA
END
```

If a user ends a line of READ input with a plus sign or hyphen, the READ statement treats it as a continuation symbol and waits for another line of input. For more information, see “Continuation symbols” on page 10.

Controlling uppercase and lowercase for READ statement input

To control uppercase and lowercase for READ statement input, use the CAPS/ASIS/NOCAPS operand on the CONTROL statement, or the &SYSASIS control variable, or the &SYSLC and &SYSCAPS built-in functions. The &SYSASIS control variable and the CAPS/ASIS/NOCAPS operand indicate whether the CLIST should translate *all* READ statement input to uppercase characters. (The CLIST does not modify numbers, national characters, special characters, or DBCS characters in such input.)

If you want the CLIST to translate all input obtained by READ statements to uppercase characters, you can use the default value (CAPS) or code:

```
CONTROL CAPS
```

or

```
SET &SYSASIS = OFF
```

To request that the CLIST leave all input obtained by READ statements in the format in which it was entered, code:

```
CONTROL ASIS
```

or

```
CONTROL NOCAPS
```

or

```
SET &SYSASIS = ON
```

The CAPS/ASIS/NOCAPS operands affect output from WRITE statements the same as they affect input from READ statements.

&SYSLC and &SYSCAPS enable you to tailor individual strings and substrings of input strings.

For example, a CLIST that prompts for first, middle, and last names, might want to guarantee that the name is properly capitalized before saving it. The following section of code shows a way to do so:

```
CONTROL ASIS /* Do not translate READ input to uppercase */
WRITENR Enter first name:
READ FNAME
WRITENR Enter middle name:
READ MNAME
WRITENR Enter last name:
READ LNAME

/*********************************************************************/
/*  Set the lengths of the first, middle, and last names to         */
/*  variables so that the substring notation is easier to read.     */
/*********************************************************************/

SET LGTHFNAME = &LENGTH(&FNAME)
SET LGTHMNAME = &LENGTH(&MNAME)
SET LGTHLNAME = &LENGTH(&LNAME)

/*********************************************************************/
/*  Capitalize the first letters in first, middle, and last names   */
/*  and make sure all other letters are in lowercase characters.    */
/*********************************************************************/

SET F = &SUBSTR(1,&SYSCAPS(&FNAME))&SUBSTR(2:&LGTHFNAME,&SYSLC(&FNAME))
SET M = &SUBSTR(1,&SYSCAPS(&MNAME))&SUBSTR(2:&LGTHMNAME,&SYSLC(&MNAME))
SET L = &SUBSTR(1,&SYSCAPS(&LNAME))&SUBSTR(2:&LGTHLNAME,&SYSLC(&LNAME))
SET NAME = &STR(&F &M &L)
```

If the input entered is CADman haVVy fisH, &NAME contains the string “Cadman Havvy Fish”.

Using the READDVAL statement

The READDVAL statement accesses the contents of the &SYSDVAL control variable. &SYSDVAL contains one of three types of information:

- Information obtained by a READ statement without operands
- The non-delimiter data on the line returning control to the CLIST after a TERMIN statement, as described in “Passing control to the terminal” on page 93
- Information that the CLIST explicitly placed into &SYSDVAL with an assignment statement

The CLIST successively places each input string in &SYSDVAL into each variable on the READDVAL statement.

Assume for the remainder of this topic that the following strings are in &SYSDVAL:

```
SMITH JONES KELLY
```

The following statement assigns the strings to symbolic variables:

```
READDVAL NAME1,NAME2,NAME3
```

Note that variables on the READDVAL statement do not require ampersands.

The preceding READDVAL statement produces the following results:

```
&NAME1; has the value SMITH.
&NAME2; has the value JONES.
&NAME3; has the value KELLY.
```

Note: The variables &NAME1, &NAME2, and &NAME3 can be set to different values during the execution of a CLIST. However, if the contents of &SYSDVAL is not modified and READDVAL is executed again, those variables are reset to the current value of SYSDVAL.

The following statement also reads all three strings from &SYSDVAL:

```
READDVAL NAME1,NAME2,NAME3,NAME4
```

The value of &NAME4 is null because there are not enough input strings in &SYSDVAL to provide a fourth value.

The following statement, however, assigns values only to the variables NAME1 and NAME2:

```
READDVAL NAME1,NAME2
```

Because there are only two variables on READDVAL to which the CLIST can assign the input strings in &SYSDVAL, the CLIST ignores the excess strings. In the previous example, the CLIST ignores KELLY.

Passing control to the terminal

Two CLIST statements are available for transferring control to the terminal and establishing a means for the user to return control to the CLIST:

1. TERMIN transfers control to the terminal and establishes a means for the user to return control to the CLIST. A CLIST executed from the TERMIN is considered to be *not* nested within the CLIST that issued the TERMIN statement, and global variables sharing between the two CLISTs is *not* allowed.
2. TERMING transfers control to the terminal and establishes a means for the user to return control to the CLIST. A CLIST executed from the TERMING is considered to be nested within the CLIST that issued the TERMING statement, and global variables sharing between the two CLISTs is allowed.

Other differences in how TERMIN and TERMING transfer control are listed in Table 7.

Table 7. TERMIN and TERMING statement comparison

Characteristic	TERMIN	TERMING
Share GLOBAL variables across the TERMIN(G) element	No	Yes
Variable access across the TERMIN(G) element through CLIST access routine IKJCT441	No	Yes
Checking Command Output Trapping - IKJCT441 and IRXEXCOM recognize CLIST and REXX execs on opposing sides of a TERMIN(G) element	No	Yes
CONTROL NOMSG statement - allow checking the NOMSG setting on opposing sides of a TERMIN(G) element	No	Yes

Because the TERMIN and TERMING elements are CLIST-generated type elements which cannot be added to the input stack through the external STACK service routine, they are considered to be of the same type. If the topmost stack element is a TERMIN or TERMING element, return code 60 (X'3C') is returned. For more information see "TERMIN and TERMING statement" on page 176.

Note: If you issue a CLIST containing a TERMIN or TERMING statement, under either ISPF or a REXX exec, or in the TSO/E background, the TERMIN or TERMING statement ends the CLIST. For CLISTs issued in the TSO/E background, TSO/E also issues message IKJ56550I to indicate that the TERMIN or TERMING statement is not supported for background processing.

The TERMIN or TERMING statement either defines character strings, one of which the user must enter to return control to the CLIST; or null lines, where the user must press the Enter key to return control to the CLIST.

The TERMIN or TERMING statement normally does not function alone. WRITE statements preceding the TERMIN or TERMING statement inform the user why control is being transferred to the terminal and how to return control to the CLIST.

Unlike the READ statement, TERMIN or TERMING enables the user to enter commands or subcommands, and invoke programs before responding to the WRITE statement prompts.

As soon as the CLIST issues the TERMIN or TERMING statement, the user receives control at the terminal. The user might receive a mode message after the TERMIN or TERMING statement is issued. If issued, the mode message might be READY or the name of the command under which the CLIST was invoked. (When READY is displayed, users might think the CLIST has terminated. You may want to avoid any confusion by telling them otherwise in the WRITE statement that precedes the TERMIN or TERMING statement.)

Returning control after a TERMIN or TERMING statement

To return control to the CLIST after a TERMIN or TERMING statement, code the TERMIN or TERMING statement and define one or more character strings that return control to the CLIST. For example:

```
TERMIN IGNORE,PROCESS,TERMINATE
```

The user then enters IGNORE, PROCESS, or TERMINATE to return control to the CLIST. The &SYSDLM control variable identifies the position of the string used. For example, if the user enters TERMINATE to return control, &SYSDLM contains a 3 because TERMINATE is the third variable on the TERMIN or TERMING statement. Multiple strings enable the user to indicate desired actions to the CLIST.

You can allow a null line as one of the valid strings but it must be the first string on the TERMIN or TERMING statement. To do so, place a comma directly before the first character string as follows:

```
TERMIN ,PROCESS,TERMINATE
```

The previous statement enables the user to return control by entering either a null line (pressing the Enter key), PROCESS, or TERMINATE.

You can issue a TERMIN or TERMING statement that lets the user return control by entering a null line (pressing the Enter key). To do so, code:

```
TERMIN
```

Exercise care in using a null line as the means for a user to return control to the CLIST, because some TSO/E command processors use null lines as function delimiters (for example, to switch between input and edit modes under EDIT).

Entering input after a TERMIN or TERMING statement

The user can optionally enter input when returning control by appending the input to the string that returns control. The CLIST stores the input in the &SYSDVAL control variable, which the CLIST can then access by executing a READDVAL statement. The READDVAL statement changes the input to uppercase, unless you code CONTROL ASIS in the CLIST.

Suppose a WRITE statement prompts the user to inform the CLIST, when returning control after a TERMIN or TERMING statement, if any data sets should be deleted. The user affirms the request by entering the following:

```
PROCESS JCL.CNTL(BUDGT) ACCOUNT.DATA
```

The following CLIST deletes the data sets in the previous statement:

```
WRITE Check your catalog and enter the names of
WRITE up to two data sets you want deleted.
WRITE They must be separated by a comma or blank and
WRITE the first name must be preceded by the word PROCESS
WRITE and a blank.  If you do not want to delete any data
WRITE sets, type in the word IGNORE.  If you want to end
WRITE the CLIST, type in TERMINATE.
TERMIN IGNORE,PROCESS,TERMINATE
/* Read the two data set names (if any) in &SYSDVAL into
/* variables called &DSN1 and &DSN2
READDVAL DSN1 DSN2
/* If the user wants to delete data sets (PROCESS),
/* delete them
IF &SYSDLM = 2 THEN +
 DO
  IF &DSN1¬=   THEN +
   delete &DSN1
  IF &DSN2¬=   THEN +
   delete &DSN2
 END
/* If the user wants the CLIST to ignore the deletion request
/* but continue processing, execute  the rest of CLIST.  The
/* null ELSE path covers the request to terminate immediately.
IF &SYSDLM = 1 THEN +
 DO
   (Rest of CLIST)
 END
```

Using ISPF panels

A CLIST can communicate with terminal users by displaying panels of the Interactive System Productivity Facility (ISPF). ISPF panels allow users to make selections and enter data; the selections and entries are then available for the CLIST to use. ISPF panels can also invoke CLISTs based on user input. With ISPF, CLISTs can conduct extensive panel-driven dialogs with users.

CLISTs use the ISPEXEC command to display ISPF panels. For complete information about using the ISPEXEC command and its operands, see *z/OS V2R2 ISPF Services Guide*.

ISPF restrictions

The names of variables used on ISPF panels can be no longer than eight characters.

Sample CLIST with ISPF panels

For an example of displaying ISPF panels from a CLIST, see “Writing full-screen applications using ISPF dialogs - the PROFILE CLIST” on page 136. The PROFILE CLIST displays any of four panels, based on input passed at invocation. On two of the panels, user input (pressing the Enter or END PF key) causes the CLIST to display another panel or end the session. The panels for the PROFILE CLIST are illustrated in their ISPF panel-definition form. Instructions for allocating the panels are included.

Chapter 10. Performing file I/O

CLISTs can perform I/O to a physical sequential data set, a member of a partitioned data set (PDS), or the terminal when allocated to a file. Four CLIST statements are available for opening, reading, writing, and closing files:

- OPENFILE opens a previously allocated file for input, output, or updating. You may have allocated the file using the TSO/E ALLOCATE command or using step allocation (JCL statements in a logon procedure).
- GETFILE reads a record from a file opened in the same CLIST.
- PUTFILE writes a record to a file opened in the same CLIST.
- CLOSFILE closes a file opened in the same CLIST.

Whenever a CLIST performs I/O, include an error routine that can handle end-of-file conditions and errors that may occur. "End-of-File processing" on page 100 shows a CLIST with an error routine that handles end-of-file conditions.

Whenever CLISTs are nested, corresponding OPENFILE, GETFILE, PUTFILE, and CLOSFILE statements must be in the same CLIST.

Characters supported in I/O

CLIST I/O statements can process all data characters represented by hexadecimal codes 40 through FF. See "Characters supported in CLISTs" on page 11 for more information and warnings for doing I/O from data sets containing special characters.

Opening a file

The OPENFILE statement has the following syntax:

```
OPENFILE filename  {INPUT }  /* to read records from the file
                   {OUTPUT}  /* to write records to the file
                   {UPDATE}  /* to update records in the file
```

To open a data set for I/O, you must allocate the data set to a file name, then use that file name on the OPENFILE statement. To preserve data integrity, after the file is opened for I/O, CLIST performs only one level of substitution against the file name variable. That is, after the file name is substituted with a file record, and to ensure the file record can be saved in its original format, CLIST does not re-scan the record.

To allocate the data set to a file name, use the ALLOCATE command with the FILE keyword. The file name is an arbitrary value; you can create it on the allocation.

For example, you can code the following:

```
⋮
allocate file(paycheks) da('d58tan1.checks.data') shr
OPENFILE PAYCHEKS
⋮
```

You can also code the file name as a symbolic variable as follows:

```
⋮
SET FILEID= PAYCHEKS
```

```
⋮
allocate file(&FILEID) da('d58tan1.checks.data') shr
OPENFILE &FILEID
⋮
```

You can open a member of a PDS after allocating the member to a file name, for example:

```
allocate file(income) da('d58tan1.receipts(july)') shr
OPENFILE INCOME
```

However, do not use OPENFILE statements to open more than one member of a PDS for output at the same time.

Closing a file

To close an open file, use a CLOSFILE statement that includes the same file name as that specified on the corresponding OPENFILE statement. For example, if you opened a file by coding:

```
OPENFILE &FILEID
```

close that file by coding:

```
CLOSFILE &FILEID
```

If you do not close an open file before the CLIST terminates, you may not be able to process that file again until you logoff and logon again.

For examples of CLOSFILE, see the examples in "Reading a record from a file" and "Writing a record to a file" on page 99.

Reading a record from a file

To read a record from an open file, use a GETFILE statement. The CLIST creates a variable of the same name as the file name and places the record into it. As long as the file remains open, successive GETFILE statements read successive records from the file. When the end of the file has been reached, &LASTCC contains the error code 400. For information about how to detect and handle end-of-file conditions, see "End-of-File processing" on page 100.

Assume a data set called D58TAN1.CHECKS.DATA contains the following records:

```
200BLACKBUY
449REFY
450YARRUM
```

To read the records into three variables, you can code the following:

```
⋮
  (error routine)
⋮
allocate file(paycheks) da('d58tan1.checks.data') shr reu
OPENFILE PAYCHEKS               /* Defaults to INPUT       */
SET COUNTER=1
DO WHILE &COUNTER ¬> 3
 GETFILE PAYCHEKS               /* Read a record           */
 SET EMPLOYEE&COUNTER=&PAYCHEKS /* Store the record        */
 SET COUNTER=&COUNTER+1         /* Increase counter by one */
END
CLOSFILE PAYCHEKS               /* Close the file          */
```

If you use GETFILE to read data from the terminal, the data is translated to uppercase, and the terminal user must end the data with a symbol that the CLIST recognizes as an end-of-file.

Writing a record to a file

To write a record to a file, do the following:

1. Open the file for output (OPENFILE filename OUTPUT).
2. Set a variable of the same name as the file name to the record you are writing to the file.
3. Specify the file name on the PUTFILE statement to write the record to the data set, for example:

```
OPENFILE PRICES OUTPUT     /* open the file for output
SET PRICES = $2590.00      /* set variable to input record
PUTFILE PRICES             /* put variable record into the file
```

Note: If you use a variable for the filename on a PUTFILE statement, use a nested variable to contain the record, for example:

```
OPENFILE &FILEID OUTPUT    /* open the file for output
SET &&FILEID = $2590.00    /* set variable to input record
PUTFILE &FILEID            /* put variable record into the file
```

As long as the file remains open, successive PUTFILE statements write successive records to the data set. For a data set with a disposition of NEW, OLD, or SHR, if you close the file and then re-open it, a subsequent PUTFILE statement overlays the first record in the data set. For a data set with a disposition of MOD, if you close the file and then re-open it, a subsequent PUTFILE statement adds a record to the end of the data set.

Assume a CLIST contains the following variables:

```
&EMPLOYEE1,; which contains the value 'BLACKBUY: $200.00'.
&EMPLOYEE2,; which contains the value 'REFY: $449.00'.
&EMPLOYEE3,; which contains the value 'YARRUM: $450.00'.
```

To place the previous values in a data set called D58TAN1.CURNTSAL.DATA, you can code the following:

```
allocate file(salaries) da('d58tan1.curntsal.data') shr reu
OPENFILE SALARIES OUTPUT   /* Open the file for output     */
SET COUNTER=1
DO WHILE &COUNTER ¬> 3
 SET EMPLOYEE=&&EMPLOYEE&COUNTER
 SET SALARIES=&EMPLOYEE    /* Set the record to be written */
 PUTFILE SALARIES          /* Write the record             */
 SET COUNTER=&COUNTER+1    /* Increase counter by one      */
END
CLOSFILE SALARIES          /* Close the file               */
```

Updating a file

To update a record in an open file, use the GETFILE and PUTFILE statements. After opening a file for updating (OPENFILE filename UPDATE), perform successive GETFILE statements until the desired record is read. After assigning the new value to a variable of the same name as the file name, perform a PUTFILE statement to update the record.

As long as the file remains open, you may update records.

Assume a data set called D58TAN1.CHECKS.DATA has a variable-blocked record format and contains the following records:

```
200BLACKBUY
449REFY
450YARRUM
```

To update the record for REFY, you can code the following:

```
⋮
  (error routine)
⋮
allocate file(paycheks) da('d58tan1.checks.data') shr reu
OPENFILE PAYCHEKS UPDATE   /* Open file for updating      */
GETFILE PAYCHEKS           /* Read first record           */
DO WHILE &SUBSTR(4:7,&PAYCHEKS)¬=REFY
 GETFILE PAYCHEKS          /* Read another record         */
END
SET PAYCHEKS = 000REFY     /* Set new value               */
PUTFILE PAYCHEKS           /* Write new value to data set */
CLOSFILE PAYCHEKS          /* Close the file              */
```

End-of-File processing

Whenever a CLIST performs I/O, include code that handles end-of-file conditions. In a CLIST, end-of-file causes an error condition (error code 400). To process this condition, provide an error routine before the code that performs the I/O.

An error routine is a block of code that gets control when an error occurs in a CLIST. The error routine can try to identify the error (such as error code 400) and take appropriate action. For a complete description of how to write an error routine, see Chapter 11, “Writing ATTN and ERROR routines,” on page 103.

The following error routine saves the value of &LASTCC, closes and frees the open file, and branches to a statement that determines whether end-of-file was reached.

```
SET RCODE=0                /* Initialize the return code variable to 0 */
SET EOF=OFF                /* Set the end-of-file indicator off        */
⋮
ERROR +
 DO
  SET RCODE = &LASTCC      /* Save the value of &LASTCC                */
  IF &RCODE=400 THEN +
   DO
    CLOSFILE PAYCHEKS      /* Close the open file
    free f(paycheks)       /* Free the open file
    WRITE No record to update because end-of-file was reached.
    SET EOF=ON
    RETURN                 /* Branch to statement that tests for
   END                     /* EOF (IF &EOF=ON THEN...)
  ELSE EXIT                /* For other errors, EXIT
 END
allocate file(paycheks) da('d58tan.checks.data') shr reu /* Allocate
/*                                                         file */
/* and establish file name of paycheks                             */
OPENFILE PAYCHEKS UPDATE   /* Open file for updating               */
SET COUNTER=1              /* Initialize counter to 1              */
DO WHILE &COUNTER <= 4
 GETFILE PAYCHEKS          /* Skip records                         */
 SET COUNTER= &COUNTER+1   /* Increase counter by 1                */
/* If EOF reached, end loop.  Null else                           */
 IF &EOF=ON THEN GOTO OUT
END
SET PAYCHEKS = 480BUZZBEE  /* Set variable to new value            */
PUTFILE PAYCHEKS           /* Update fourth record                 */
```

```
CLOSFILE PAYCHEKS          /* Close the file                          */
:
  (rest of CLIST)
:
OUT: END
```

Special considerations for performing I/O

- MOD operand

  When allocating the data set you can use the MOD operand. It allows you to append data to the end of a sequential data set. For more information about the MOD operand see *z/OS TSO/E Command Reference*, and *z/OS TSO/E REXX User's Guide*.

- Records Containing JCL Statements

  If a CLIST reads or writes records containing JCL statements, that CLIST can make unwanted modifications to the statements by symbolic substitution. To prevent the unwanted modifications, use the &NRSTR or &SYSNSUB built-in functions. See Chapter 7, "Using built-in functions," on page 53 for details and examples.

- Concatenated Data Sets

  You can perform I/O on multiple data sets that are allocated (concatenated) to a single file name. However, the first data set in the concatenation must not be empty: if a GETFILE statement is issued and the first data set in the concatenation is empty, all other data sets allocated to the file are ignored, and no records are read.

Chapter 11. Writing ATTN and ERROR routines

Two types of events cause the execution of a CLIST to halt prematurely: attention interrupts and errors. The CLIST language provides two statements that enable you to code routines to handle attention interrupts and errors. They are ATTN and ERROR. The ATTN statement is described in "Writing attention routines." The ERROR statement is described in "Writing error routines" on page 107.

An attention interrupt occurs when the user presses the attention key (typically PA1 or ATTN) on the terminal keyboard. The user may enter an attention interrupt for any number of reasons, such as to terminate an infinite loop or to end the CLIST. The user cannot enter an attention interrupt when a CLIST error routine is in execution as a result of a CLIST-invoked command processor abend or before a TSO/E command is executed within the CLIST. Any attention interruption received while a command abend is in progress is ignored.

An error can occur for any number of reasons, such as a numeric value that exceeds 2^{31}-1, an end-of-file condition, or a non-zero return code from a TSO/E command.

Writing attention routines

Use the ATTN statement to identify an action to be taken when the user enters an attention interrupt. The action can be any executable statement and is often a DO-sequence that performs operations tailored to the CLIST. You can structure an ATTN action as follows:

```
ATTN +
 DO
⋮
  (action)
⋮
 END
```

The ATTN statement and its action must precede the code to which it applies. Multiple CLIST statements may be executed in the action but only one TSO/E command, TSO/E subcommand, or null line may be executed. (A null line returns control to the statement or command that was executing when the attention interrupt occurred.) If the one TSO/E command executed is an invocation of an attention handling CLIST, you can execute as many TSO/E commands or subcommands as you want in the attention handling CLIST.

If an attention action does not execute a TSO/E command, subcommand, or null line, the action must include an EXIT or RETURN statement. The EXIT statement ends the CLIST, and the RETURN statement returns control to the CLIST statement, command, or subcommand following the one that was executing when the user entered the attention interrupt.

You should inform the user at the beginning of the attention routine that TSO/E is processing the attention interrupt. Otherwise, the user may enter another attention interrupt. For a description of how TSO/E processes multiple attention interrupts, see *z/OS TSO/E Programming Services*.

Canceling attention routines

You can cancel an attention routine at any point, letting the CLIST continue without any special attention processing. To cancel an attention routine, code:

```
ATTN OFF
```

This entry nullifies the most recently established attention routine. ATTN OFF should not be used within an attention routine itself.

You can also code attention routines that override previous ones. Each attention routine overrides all previous ones. You can initialize new attention routines as many times as you want.

Protecting the input stack from attention interrupts

When a CLIST is executed, it translates each statement into an executable format and places it in a section of storage called the input stack. The input stack is the source from which TSO/E obtains its input (TSO/E commands, CLIST statements).

If you write an attention routine that does not terminate the CLIST, protect the input stack from being erased (flushed) from storage when an attention interrupt occurs. You can protect the input stack by coding a CONTROL statement with the MAIN operand. The MAIN operand indicates that the CLIST is the main CLIST in the invoker's TSO/E environment and prevents TSO/E from flushing the input stack in the event of an attention interrupt.

Attention routine processing depends on whether CONTROL MAIN has been coded, and whether the routine executes a TSO/E command, RETURN statement, or null line.

- If CONTROL MAIN has not been coded, the CLIST terminates and the user sees the READY message, indicating that control has returned to the terminal.
- If CONTROL MAIN has been coded, and a null line executes in the attention routine, the CLIST continues at the statement or command that was executing when the user entered the attention interrupt.
- If CONTROL MAIN has been coded, and a TSO/E command or RETURN statement is issued, the CLIST continues at the statement or command *following* the one that was executing when the user entered the attention interrupt.

Also refer to *z/OS TSO/E User's Guide*, for a further explanation of attention interrupt processing.

Sample CLIST with an attention routine

The ALLOCATE CLIST shown in Figure 4 on page 105 contains an attention routine that prompts the user to indicate whether he or she wants to end the CLIST.

If the user types `YES` to end the CLIST, and data sets have been allocated, the attention routine invokes a CLIST called HOUSKPNG (see Figure 5 on page 106), which frees the allocated data sets. Then the attention routine ends the ALLOCATE CLIST.

If the user *does not* type YES to end the ALLOCATE CLIST, the attention routine issues CONTROL MAIN and a null line to return control to the point where the attention interrupt occurred.

Note that the attention routine in Figure 4 issues only one TSO/E command: %houskpng or the null line. However, the HOUSKPNG CLIST itself issues up to three commands, depending on how many data sets it has to free.

```
/*****************************************************************/
/* THE ALLOCATE CLIST ALLOCATES THREE DATA SETS REQUIRED FOR     */
/* A PROGRAM.  IT IS EQUIPPED TO HANDLE ATTENTION INTERRUPTS     */
/* ENTERED AT ANY POINT.  WHEN NECESSARY, IT INVOKES HOUSKPNG.   */
/*****************************************************************/

PROC 2 &DS1 &DS2
CONTROL END(STOP)     /* substitute "STOP" for END statement     */
CONTROL PROMPT
ATTN +
  DO
   WRITE TSO is processing your attention
   WRITENR Do you want to end?  If so, type YES ====>
   READ &END
   IF &END = YES THEN +
 /* If user wants to end, terminate the CLIST after the HOUSKPNG routine */ +
 /* frees any data sets allocated by the CLIST.                  */
     DO
       CONTROL FLUSH     /* flush the input stack after HOUSKPNG */
     STOP
   ELSE +
       CONTROL MAIN      /* return control to the CLIST          */
   IF &FOOTPRINT = YES AND &END = YES THEN +
     %houskpng &ds1 &ds2 &cleanup /* call HOUSKPNG to free data sets */
   ELSE +
     DO
       SET &NULL =
       &NULL             /* issue null line to continue at the   */
                         /* point where the attention occurred.  */
     STOP
  STOP
alloc f(input) da(&ds1.text) shr reu
SET FOOTPRINT = YES
SET CLEANUP=1
alloc f(output) da(&ds2.text) reu
SET CLEANUP=2
alloc f(temp) da(temp.text)
SET CLEANUP=3
call 'myid.myprog.load(member)'
free f(temp) da(temp.text)
SET CLEANUP=2
free f(output) da(&ds2.text)
SET CLEANUP=1
free f(input) da(&ds1.text)
SET FOOTPRINT =          /* Set FOOTPRINT back to null           */
```

Figure 4. A CLIST containing an attention routine - the ALLOCATE CLIST

```
/*****************************************************************/
/* THE HOUSKPNG CLIST IS INVOKED WHEN THE USER WANTS TO END THE  */
/* ALLOCATE CLIST AFTER AN ATTENTION AND DATA SETS ARE ALREADY   */
/* ALLOCATED.  BASED ON THE VALUE OF THE VARIABLE CLEANUP,       */
/* THE CLIST FREES FROM ONE TO THREE OF THE DATA SETS ALLOCATED  */
/* IN THE ALLOCATE CLIST.                                        */
/*****************************************************************/

PROC 3 &DS1 &DS2 &CLEANUP
CONTROL END(ENDO)
ATTN +
 EXIT QUIT
IF &CLEANUP=1 THEN +
 free f(input) da(&ds1.text)
IF &CLEANUP=2 THEN +
 DO
  free f(input) da(&ds1.text)
  free f(output) da(&ds2.text)
 ENDO
IF &CLEANUP=3 THEN +
 DO
  free f(input) da(&ds1.text)
  free f(output) da(&ds2.text)
  free f(temp) da(temp.text)
 ENDO
```

Figure 5. An attention handling CLIST - the HOUSKPNG CLIST

Subprocedures and attention routines

Attention routines can call CLIST subprocedures. TSO/E commands in called subprocedures have the same effect as TSO/E commands in the attention routine itself: when the first TSO/E command executes, attention processing ends and control passes to the line in the CLIST following the one that was executing when the attention interrupt occurred.

Subprocedures can contain attention routines. However, attention routines in subprocedures cannot contain nested attention or error routines.

CLIST attention facility

The CLIST attention facility (in TSO/E) and the CLSTATTN parameter of the STAX macro provide greater flexibility in the handling of attention interruptions. The CLSTATTN parameter of the STAX macro lets a program establish an attention routine that receives control when an attention interruption occurs during the processing of a CLIST that contains an attention routine. The program's attention routine can invoke the CLIST attention facility to process the CLIST attention routine.

Previously, the terminal monitor program (TMP) handled attention interruptions for CLISTs with attention routines. Now a program can maintain control by having its own attention routine perform that processing. For more information about using the CLIST attention facility and the STAX macro, see *z/OS TSO/E Programming Services*.

Writing error routines

Use the ERROR statement to create an error routine. The error routine defines an action to be taken when a CLIST receives a non-zero return code from something other than a CLIST subprocedure. (Table 8 on page 113 lists the CLIST error codes.) The action can be any executable statement and is often a DO-group that performs operations tailored to the indicated error. You can structure an ERROR action as follows:

```
ERROR +
 DO
⋮
  (action)
⋮
 END
```

The ERROR statement and its action must precede the code to which it applies. An action may contain TSO/E commands and subcommands, subject to the mode in which the CLIST is executing when the error occurs. Unlike attention routines, error routine actions can issue multiple TSO/E commands or subcommands.

If an error routine action does not end the CLIST, it must include a RETURN statement. The RETURN statement returns control to the CLIST statement, TSO/E command, or TSO/E subcommand following the one that was executing when the error occurred. Repeated errors which activate the same error routine may cause the CLIST to terminate.

You may also code error routines that override previous ones. Each error routine overrides all previous ones. You may initialize new error routines as many times as you want.

Canceling error routines

To cancel the most recently established error routine in a CLIST, code either:

```
ERROR OFF
```

or

```
ERROR
```

following the error routine to be cancelled.

When ERROR OFF is coded, processing continues as if an error routine had never been established. When a failure occurs, one of the following occurs depending on the type of failure:

- If the failure was because of an ABEND or non-zero return code from a TSO/E command or subcommand, the CLIST continues execution with the next sequential instruction following the failing instruction.
- If the failure was in a CLIST statement or in expression evaluation, the failing instruction and explanatory CLIST error messages are displayed, and the CLIST terminates.

When ERROR is entered with no operands, the CLIST displays the command, subcommand, or statement on the CLIST that ended in error. No explanatory CLIST error messages are displayed. &LASTCC is reset to 0 and the CLIST continues with the next sequential statement or command.

Protecting the input stack from errors

When a CLIST is executed, it translates each statement into an executable format and places it in a section of storage called the input stack. The input stack is the source from which TSO/E obtains its input (TSO/E commands, CLIST statements).

If you write a CLIST that contains an error routine, protect the input stack from being erased from storage (flushed) when an error occurs. You can protect the input stack by coding a CONTROL statement that includes the NOFLUSH or MAIN operand. The CONTROL statement must appear before any error routine, preferably at the beginning of the CLIST.

Sample CLIST with an error routine

The COPYDATA CLIST, shown in "The COPYDATA CLIST," contains an error routine that handles:

- Pre-allocation errors
- End-of-file condition
- Allocation errors

The CLIST allocates the data sets required to copy an existing data set into an output data set. If the copy is successful, the CLIST cancels the error routine by executing an ERROR statement with no operands and continues.

Subprocedures and error routines

Error routines can call CLIST subprocedures, and subprocedures can issue the RETURN statement to return control to the error routine. The error routine itself must issue RETURN to return control to the statement after the one in error. For example, the following error routine calls a subprocedure:

```
ERROR +
 DO
  SET &ECODE = 8
  SELECT
  WHEN (&FOOTPRINT=2) SYSCALL ABC ECODE
.
.
.
  END                 /* End of SELECT
  RETURN              /* return control to CLIST
 END                  /* End of error routine
.
.
.
ABC: PROC 1 CODEPARM  /* subroutine ABC
SYSREF &CODEPARM      /* refer variable back to caller's &ECODE
free f(indata)        /* free data sets
free f(outdata)
SET &CODEPARM = 12    /* set error code
RETURN                /* return control to error routine
END                   /* end of subroutine ABC
```

Subprocedures can contain error routines. However, error routines in subprocedures cannot contain nested attention or error routines.

The COPYDATA CLIST

```
/*************************************************************/
/* THE COPYDATA CLIST COPIES RECORDS FROM A DATA SET INTO AN  */
/* OUTPUT DATA SET. IT IS EQUIPPED TO HANDLE ERRORS CAUSED BY */
/* END-OF-FILE, ALLOCATION ERRORS, AND ERRORS CAUSED BY OTHER */
/* STATEMENTS AND COMMANDS IN THE CLIST.                      */
/*************************************************************/

CONTROL NOFLUSH END(ENDO) /* Protect the stack from being flushed
```

```
/* so that when error is caused by end-of-file, CLIST can continue
ERROR +
 DO
  SET RCODE=&LASTCC /* Save return code
  /* If end-of-file, branch to CLOSFILE statements
  SELECT
  WHEN (&RCODE=400) +
   DO                    /* IF End-of-file is reached,           */
    SET EOFFLAG = YES    /* Set flag and return to the           */
    RETURN               /* I/O procedure.                       */
   ENDO
  /* If error occurred before allocation, set exit code to 4
  WHEN (&FOOTPRINT=0) SET ECODE=4
  /* If allocation of file OUTDS failed, free file INDATA and set
  /* exit code to 8
  WHEN (&FOOTPRINT=1) +
   DO
    free f(indata) da(text.data)
    SET ECODE=8
   ENDO
  /* If the error was not caused by end-of-file or allocation error,
  /* free both files and set exit code to 12.  In this case, error was
  /* caused by one of the file I/O statements
  WHEN (&FOOTPRINT=2) +
   DO
    free f(indata) da(text.data)
    free f(outds)
    SET ECODE=12
   ENDO
  ENDO /* End of SELECT statement
  EXIT CODE(&ECODE) /* For all errors except end-of-file condition,
  /* exit the CLIST with the appropriate exit code
 ENDO /* End of error routine
SET FOOTPRINT=0  /* Identify pre-allocation errors
.
.
.
SET FOOTPRINT=1 /* Identify allocation error for file INDATA
alloc f(indata) da(d15rbo1.text.data) shr reu /* Allocate input data set
SET FOOTPRINT=2  /* Identify allocation error for file OUTDS
alloc f(outds) sysout(a) /* Allocate output data set
OPENFILE INDATA /* Open input data set
OPENFILE OUTDS OUTPUT  /* Open output data set
/* Copy records from input data set to output data set           */
 DO WHILE &EOFFLAG ¬= YES /* Do the following until EOF is reached*/
 GETFILE INDATA /* Read input record
 IF &EOFFLAG ¬= YES THEN +
  DO
   SET OUTDS=&INDATA /* Set output record to value of input record
   PUTFILE OUTDS /* Write output record to output data set
  ENDO
ENDO
EOF: CLOSFILE INDATA /* Close input data set
CLOSFILE OUTDS /* Close output data set
ERROR /* From this point on, display statement that causes error
/* along with any error messages
.
.
.
```

Chapter 12. Testing and debugging CLISTs

This chapter describes how to test CLISTs using diagnostic procedures to find and correct errors. The diagnostic procedures include:

- Using diagnostic options of the CONTROL statement to find errors in CLIST statements and TSO/E commands
- Getting help for CLIST messages
- Finding and understanding CLIST error codes

Using diagnostic options of the CONTROL statement

The CONTROL statement lets you define processing options for a CLIST. Some of the CONTROL statement options can help you diagnose CLIST errors. These diagnostic options, LIST, CONLIST, SYMLIST, and MSG, cause a CLIST to display its statements, commands, and any informational messages at the terminal when you execute the CLIST. From the displayed information, you can often find statements or commands that contain errors.

You can use the diagnostic options separately or together on the CONTROL statement. To obtain the most complete diagnostic information, code the options together (the order is not significant):

```
CONTROL LIST CONLIST SYMLIST MSG
```

You can place the CONTROL statement at the top of the CLIST or in any part of the CLIST that you want to test or debug. Each CONTROL statement overrides any previous CONTROL statements. To turn off the diagnostic options, type:

```
CONTROL NOLIST NOCONLIST NOSYMLIST NOMSG
```

As an alternative to retyping the CONTROL statement when you want to change options, you can use the control variables &SYSLIST, &SYSCONLIST, &SYSSYMLIST, and &SYSMSG to test or change the current settings. For more information about using these control variables, see "Setting options of the CLIST CONTROL statement" on page 44.

The diagnostic options have the following effects:

SYMLIST
The CLIST displays each TSO/E command, subcommand, or CLIST statement at the terminal before scanning it for symbolic substitution.

LIST
The CLIST displays each TSO/E command or subcommand at the terminal after symbolic substitution but before execution.

CONLIST
The CLIST displays each CLIST statement at the terminal after symbolic substitution but before execution.

MSG
The CLIST displays informational messages at the terminal.

Note: SYMLIST and CONLIST do not display the GLOBAL or NGLOBAL statements.

The CLIST in Figure 6 contains diagnostic options on the CONTROL statement. When you execute the CLIST, the commands and statements appear at the terminal as shown in Figure 7.

```
CONTROL LIST CONLIST SYMLIST MSG
SET INPUT = data.set.name
SET DSN = &INPUT;
allocate file(a) dataset('myid.&dsn')
free file(a)
```

Figure 6. Sample CLIST with diagnostic CONTROL options

```
SET INPUT = data.set.name
SET INPUT = data.set.name
SET DSN = &INPUT;
SET DSN = data.set.name
allocate file(a) dataset('myid.&dsn')
allocate file(a) dataset('myid.data.set.name')
free file(a)
free file(a)
```

Figure 7. Diagnostic output from sample CLIST

Notice that each statement and command appears twice at the terminal. The first version is caused by CONTROL SYMLIST and shows the statement or command as it appears in the CLIST. The second version shows the results of symbolic substitution on the preceding line. If a line undergoes no substitution (contains no variables), both versions are the same.

Messages in diagnostic output

The CLIST executes each statement or command after performing symbolic substitution on it. Therefore, when you use the MSG option with LIST and CONLIST, messages about execution errors appear at the terminal after the line that caused the error.

For example, the CLIST in Figure 6 fails when the input data set is not cataloged. When the input data set is not cataloged, the CLIST displays the following information at the terminal, with messages after the statement that failed to execute.

```
SET INPUT = data.set.name
SET INPUT = data.set.name
SET DSN = &INPUT;
SET DSN = data.set.name
allocate file(a) dataset('myid.&dsn')
allocate file(a) dataset('myid.data.set.name')
IKJ56228I DATA SET MYID.DATA.SET.NAME NOT FOUND IN CATALOG
OR CATALOG CANNOT BE ACCESSED
IKJ56701I MISSING DATA SET NAME+
IKJ56701I MISSING NAME OF DATA SET TO BE ALLOCATED
```

Figure 8. Error messages in diagnostic output from sample CLIST

The diagnostic output ends after the ALLOCATE command, when the CLIST detects the error. Working backwards from the last line, you can find and correct the source of the error (in this case, the value of &INPUT).

Note that the last line in Figure 8 on page 112 is a continuation of the preceding message line. When the CLIST is executed under ISPF, the continuation is displayed as shown in Figure 8 on page 112. Under line-mode TSO/E, you must type a question mark (?) after the plus sign to see the continuation.

How to make diagnostic output optional in a CLIST

You can make the diagnostic output available as an option to anyone who invokes your CLIST. To do so, code a keyword parameter such as DEBUG on the PROC statement as follows:

```
PROC 0 DEBUG
IF &DEBUG=DEBUG THEN +
CONTROL LIST CONLIST SYMLIST MSG
```

The CONTROL options take effect when you invoke the CLIST with the DEBUG parameter, for example (explicit invocation):

```
EX clistname 'DEBUG'
```

or, implicit invocation:

```
%clistname DEBUG
```

Getting help for CLIST messages

CLIST message numbers begin with the characters IKJ. For explanations of CLIST messages, look up the message number in the IKJ section of *z/OS TSO/E Messages*. The message explanations include information about the action, if any, you need to take to correct a problem.

Obtaining CLIST error codes

The CLIST control variable &LASTCC contains an error code from the last TSO/E command or CLIST statement executed. After each command or statement in a CLIST, you can retrieve the error code from &LASTCC, for example, by coding

```
SET ECODE = &LASTCC
```

You can then write the error code to the terminal or use it as a basis for further processing. For more information about using &LASTCC, see “Getting return codes and reason codes” on page 48.

Note: With the exception of the RETURN statement, CLIST statements and TSO/E commands in error routines do not update the value of &LASTCC If you use &LASTCC in an error routine, &LASTCC contains the return code from the command or statement that was executing when the error occurred.

Table 8 lists and explains the error codes that CLIST statements return in &LASTCC. Except as otherwise noted, the codes are in decimal format.

Table 8. CLIST statement error codes (decimal)

Error code	Meaning
16	Not enough virtual storage. Log on with more storage or specify VARSTORAGE(HIGH) in your TSO/E PROFILE.
300	User tried to update a control variable that can only be updated by the system.
304	Not valid keyword found on EXIT statement.

Table 8. CLIST statement error codes (decimal) (continued)

Error code	Meaning
308	CODE keyword specified, but no code given on EXIT statement.
312	Internal GLOBAL processing error.
316	TERMIN delimiter has more than 256 characters.
324	GETLINE error.
328	More than 64 delimiters on TERMIN.
332	Not valid file name syntax.
336	File already open.
340	Not valid OPEN type syntax.
344	Undefined OPEN type.
348	File specified did not open. (For example, the file name was not allocated.) Reallocate the file.
352	GETFILE - file name is not currently open.
356	GETFILE - the file has been closed by the system. (For example, the file was opened under EDIT mode and EDIT mode has been terminated.)
360	PUTFILE - file name not currently open.
364	PUTFILE - file closed by system (see code 356).
368	PUTFILE - CLOSFILE - file not opened by OPENFILE.
372	PUTFILE - issued before GETFILE on a file opened for update.
376	Unable to open the directory of a PDS using a variable record format.
380	Data sets with a logical record length greater than 32767 are not supported for CLIST I/O.
400	GETFILE - end of file. TSO/E treats this condition as an error that can be handled by an ERROR action.
404	User tried to write to a file open for INPUT.
408	User tried to read from a file open for OUTPUT.
412	User tried to update a file after end of file was reached.
416	User tried to update an empty file.
500	The TO value on a DO statement is non-numeric.
502	The FROM value on a DO statement is non-numeric.
504	The BY value on a DO statement is non-numeric.
508	A SYSCALL statement contains an undefined procedure name.
512	A RETURN statement contains an undefined keyword.
516	The name of a procedure is used as a variable.
524	Unable to establish an ESTAE routine.
528	A positional specification on the PROC statement was not valid.
532	Not valid characters were found in a symbolic parameter on the PROC statement.
536	A symbolic parameter name on the PROC statement is too long.
540	The number of positional parameters defined on the PROC statement is fewer than the number passed.

Table 8. CLIST statement error codes (decimal) (continued)

Error code	Meaning
544	No symbolic parameters were defined on the PROC statement.
548	Duplicate parameter names were found on the PROC statement.
552	A keyword parameter has a not valid default value.
556	A default keyword value was missing an ending quote on the PROC statement.
560	A PARSE error occurred while processing the PROC statement.
568	Abnormal termination
572	SYSREF variable was not passed as a parameter.
576	SYSREF variable was not defined on a PROC statement.
580	An ERROR statement was found within a subprocedure's ERROR or ATTN routine.
584	An ATTN statement was found within a subprocedure's ERROR or ATTN routine.
588	A character between DBCS delimiters was outside the range of double-byte characters.
592	A DBCS string contains an odd number of bytes, indicating that one of the characters is incomplete.
596	A beginning DBCS delimiter was found without a corresponding ending delimiter.
600	Two beginning DBCS delimiters were found without an intervening ending delimiter.
604	An error occurred while processing an installation-written CLIST built-in function in IKJCT44B.
608	An error occurred while processing an installation-written CLIST statement in IKJCT44S.
612	An error occurred in an installation exit.
620	EBCDIC &SYSTWOBYTE data is outside valid DBCS range.
624	An error occurred while processing a system variable (see note below).
708	The preceding statement has a not valid &SYSINDEX expression.
712	The preceding statement has a not valid &SYSINDEX start parameter; the start parameter must be a non-negative number.
716	The preceding statement has a not valid &SYSNSUB level parameter; the level parameter must be a number from 0 to 99.
720	The preceding statement has a missing &SYSNSUB level, expression parameter or both.
724	The preceding statement has a &SYSNSUB level parameter that uses a built-in function as a symbolic variable.
8xx	Evaluation routine error codes.
800	Data was found where operator was expected.
804	An operator was found where data was expected.
808	A comparison operator was used in a SET statement.
812	(Reserved).
816	An operator was found at the end of a statement.

Table 8. CLIST statement error codes (decimal) (continued)

Error code	Meaning
820	Operators are out of order; data may resemble operators.
824	More than one exclusive operator was found.
828	More than one exclusive comparison operator found.
832	The result of an arithmetic calculation is outside the valid range, -2,147,483,647 to +2,147,483,647.
836	(Reserved).
840	Not enough operands.
844	No valid operators.
848	An attempt was made to load data as character data, but the data was numeric (an arithmetic operation had been performed on the data).
852	Addition error - character data.
856	Subtraction error - character data.
860	Multiplication error - character data.
864	Divide error - character data or division by 0.
868	Prefix found on character data.
872	Numeric value is too large.
900	Single ampersand was found.
904	(Reserved).
908	An error occurred in an error action that received control because of another error.
912	Substring range is not valid.
916	A non-numeric value was found in a substring range.
920	Substring range value too small (zero or negative).
924	Substring syntax is not valid.
932	Substring found outside of the range of the string. (For example, an &SUBSTR variable attempted to substring the first three positions of data that contains only two characters.)
936	A built-in variable that requires a value was entered without a value.
940	Not valid symbolic variable.
944	A label was used as a symbolic variable.
948	Not valid label syntax on a GOTO statement.
952	A GOTO label was not defined.
956	A GOTO statement has no label.
960	&SYSSCAN was set to a not valid value.
964	&LASTCC was set to a not valid value and EXIT tried to use it as a default value.
968	DATA PROMPT-ENDDATA statements supplied, but no prompt occurred.
972	TERMIN statement cannot be used in background jobs.
976	READ statement cannot be used in background jobs.

Table 8. CLIST statement error codes (decimal) (continued)

Error code	Meaning
980	Maximum statement length (32756) exceeded during symbolic substitution.
984	TERMING delimiter has more than 256 characters.
988	TERMING has more than 64 delimiters.
992	TERMING statement cannot be used in background jobs.
999	Internal CLIST error.
Sxxx	A system abend code, printed in hexadecimal.
Uxxx	A user abend code, printed in hexadecimal.
Note: The underlaying error, which is summarized by error code 624, will always be shown by a more detailed error message; this message will not be suppressed when using an error routine.	

Chapter 13. Sample CLISTs

This chapter contains examples of CLISTs that illustrate the CLIST functions described in previous chapters. The examples assume that the CLISTs reside in a PDS allocated to SYSPROC.

Table 9 lists the names of the CLISTs and provides short descriptions of the functions they illustrate. Many of these CLISTs include examples of symbolic variables, control variables, built-in functions, and conditional sequences.

Table 9. Sample CLISTs and their functions

CLIST	Function	Reference
LISTER	Including TSO/E commands	Figure 9 on page 120
DELETEDS	Simplifying routine tasks	"Simplifying routine tasks - the DELETEDS CLIST" on page 120
CALC	Creating arithmetic expressions from user supplied input	Figure 11 on page 121
CALCFTND	Performing front-end prompting	"Using front-end prompting - the CALCFTND CLIST" on page 121
SCRIPTDS	Initializing and invoking system services	"Initializing and invoking system services - the SCRIPTDS CLIST" on page 122
SCRIPTN	Invoking CLISTs to perform subtasks	"Invoking CLISTs to perform subtasks - the SCRIPTN CLIST" on page 124
SUBMITDS	Including JCL; performing front-end prompting	"Including JCL statements - the SUBMITDS CLIST" on page 126
SUBMITFQ	Performing substringing; adding flexibility	"Analyzing input strings with &SUBSTR - the SUBMITFQ CLIST" on page 126
RUNPRICE	Allowing foreground or background submittal of jobs	"Allowing foreground and background execution of programs - the RUNPRICE CLIST" on page 127
TESTDYN	Providing invoker with options and performing initialization based on options specified	"Including options - the TESTDYN CLIST" on page 128
COMPRESS	Simplifying routine, system-related tasks	"Simplifying system-related tasks - the COMPRESS CLIST" on page 130
CASH	Simplifying invoker's interface to complex applications	"Simplifying interfaces to applications - the CASH CLIST" on page 131
PHONE	Performing I/O; reading records into &SYSDVAL	"Using &SYSDVAL when performing I/O - the PHONE CLIST" on page 132
SPROC	Using &SYSOUTTRAP and &SYSOUTLINE variables to manage command output	"Allocating data sets to SYSPROC - the SPROC CLIST" on page 133
PROFILE	Using ISPF dialog management services in CLISTs to create full-screen applications	"Writing full-screen applications using ISPF dialogs - the PROFILE CLIST" on page 136

Table 9. Sample CLISTs and their functions (continued)

CLIST	Function	Reference
EXPAND	Using LISTDSI statement to allocate a new data set with characteristics of an existing data set.	"Allocating a data set with LISTDSI information - the EXPAND CLIST" on page 143

Including TSO/E Commands - the LISTER CLIST

You can organize related activities so that users can invoke a CLIST to perform a given task or group of tasks. The simplest example is a CLIST that groups TSO/E commands together.

The LISTER CLIST consists of two TSO/E commands. (See Figure 9.) The LISTCAT command lists all of the entries in the invoker's catalog. The LISTALC command lists the names and status of all data sets allocated to the invoker's user ID. TSO/E displays the output produced by these commands in the same order as that in which it executes the commands. The invoker does not have to enter a command, view its output, then enter another command; all input required from the invoker is supplied at one time.

```
listcat
listalc status
```

Figure 9. The LISTER CLIST

Simplifying routine tasks - the DELETEDS CLIST

One way to simplify routine tasks is to write CLISTs that make the process as interactive as possible. For example, the syntax of the DELETE command can confuse users who want to delete some of their data sets. For those users, you can write a CLIST that simplifies the process. The DELETEDS CLIST shown in Figure 10 is an example of such a CLIST. It prompts the invoker for a data set name or a completion indicator.

```
/*********************************************************************/
/*   THIS CLIST PROMPTS THE USER FOR THE NAMES OF THE DATA           */
/*   SETS TO BE DELETED, ONE AT A TIME.                              */
/*********************************************************************/

SET DONE=NO
DO WHILE &DONE=NO
 WRITE Enter the name of the data set you want deleted.
 WRITE Omit the identification qualifier (userid).
 WRITE Do not put the name in quotation marks.
 WRITE When you are finished deleting all data sets, type an 'f'.
 READ DSN
 IF &DSN = F THEN SET DONE=YES
 ELSE delete &DSN
END
```

Figure 10. The DELETEDS CLIST

Creating arithmetic expressions from user-supplied input - the CALC CLIST

The CALC CLIST, shown in Figure 11, contains a PROC statement that requires three input strings from the invoker:

- A numeric value
- An arithmetic operator
- Another numeric value.

The CLIST creates an arithmetic expression using the positional parameter variables that represent these three values. A WRITE statement displays a message made up of the unevaluated expression, an equal sign, and the evaluated expression. CALC contains no validity-checking statements; therefore, input that does not meet the above requirements causes the &EVAL; built-in function to fail and generate an error code.

```
PROC 3 FVALUE OPER LVALUE

/*********************************************************************/
/* DISPLAY THE ENTIRE EQUATION AT THE TERMINAL, INCLUDING THE RESULT  */
/* OF THE EXPRESSION.                                                 */
/*********************************************************************/

WRITE &FVALUE&OPER&LVALUE = &EVAL(&FVALUE&OPER&LVALUE)
```

Figure 11. The CALC CLIST

Using front-end prompting - the CALCFTND CLIST

Front-end prompting verifies input data before the CLIST uses it in other statements. For example, the CALC CLIST in Figure 11 assumed that &FVALUE and &LVALUE represented valid numeric values or variables containing valid numeric values. It also assumed that &OPER represented a valid arithmetic operator.

In CALCFTND, shown in "The CALCFTND CLIST," the CLIST first ensures that &FVALUE is numeric, not character data. The WRITE statement message is tailored to address the possibility that the invoker is including decimal points in the value. The CLIST views such a value as character data, not numeric data. The DO-WHILE-END sequence executes until the invoker supplies a valid numeric value. A similar DO-WHILE-END sequence is provided for &LVALUE;

The verification of &OPER is somewhat more involved. &OPER must be a valid arithmetic operator, one of the following symbols: `+, -, *, /, **, //`. Therefore, the condition for the corresponding DO-WHILE-END sequence requires a logical ANDing of comparative expressions. Each expression is true when &OPER does not equal the operator in the expression. When all of the expressions are true, &OPER is not a valid arithmetic operator. To ensure that the CLIST views &OPER and the valid arithmetic operators as character data, enclose them in &STR built-in functions.

The CALCFTND CLIST

```
PROC 0  FVALUE( )  OPER( )  LVALUE( )

/*********************************************************************/
```

```
/* IF &FVALUE IS NOT VALID, CONTINUE PROMPTING THE USER TO ENTER        */
/* AN ACCEPTABLE VALUE.                                                 */
/************************************************************************/

CONTROL ASIS     /* Allow upper and lower case WRITE statements       */

SET &NULL =
DO WHILE &DATATYPE(&FVALUE) ¬= NUM
  IF &STR(&FVALUE) = &NULL THEN +
    WRITE Please enter a first value without decimal points &STR(-)
  ELSE +
   DO
    WRITENR Your first value is not numeric.  Reenter a number without
    WRITE decimal points &STR(-)
   END
   READ &FVALUE
END
/************************************************************************/
/* IF &OPER IS NOT VALID, CONTINUE PROMPTING THE USER TO ENTER          */
/* AN ACCEPTABLE VALUE.                                                 */
/************************************************************************/

DO WHILE &STR(&OPER)¬=&STR(+) AND &STR(&OPER)¬=&STR(-) AND +
         &STR(&OPER)¬=&STR(*) AND &STR(&OPER)¬=&STR(/) AND +
         &STR(&OPER)¬=&STR(**) AND &STR(&OPER)¬=&STR(//)
 IF &STR(&OPER) = &NULL THEN +
  DO
   WRITE Please enter a valid arithmetic operator (+,-,*,/,**,//)
   WRITE enclosed in parentheses, for example, (+) or (-).
  END
 ELSE +
  DO
   WRITE Your second value is not a valid operator (+,-,*,/,**,//).
   WRITE Reenter this value, using one of the valid arithmetic
   WRITE operators enclosed in parentheses, for example, (+) or (-).
  END
  READ &OPER
END
/************************************************************************/
/* IF &LVALUE IS NOT VALID, CONTINUE PROMPTING THE USER TO ENTER        */
/* AN ACCEPTABLE VALUE.                                                 */
/************************************************************************/

DO WHILE &DATATYPE(&LVALUE) ¬= NUM
 IF &STR(&LVALUE) = &NULL THEN +
   WRITE Please enter a second value without decimal points &STR(-)
 ELSE +
  DO
   WRITENR Your last value is not numeric.  Reenter a number without
   WRITE decimal points &STR(-).
  END
  READ LVALUE
END
/************************************************************************/
/* ONCE THE OPERANDS HAVE BEEN VERIFIED, EVALUATE THE EXPRESSION AND  */
/* DISPLAY THE RESULT AT THE TERMINAL.                                  */
/************************************************************************/
WRITE &FVALUE&OPER&LVALUE = &EVAL(&FVALUE&OPER&LVALUE)
```

Initializing and invoking system services - the SCRIPTDS CLIST

The SCRIPTDS CLIST enables a user to run the SCRIPT program against an input data set and have the output printed.

As shown in "The SCRIPTDS CLIST," SCRIPTDS requires a positional parameter, &DSN; The invoker supplies the name of a PDS member to be printed. The CLIST includes the &DSN variable as the member name of the memo.text data set on the invocation of the SCRIPT program. The invoker does not have to supply input for &SYSPREF because it is a control variable whose value is available to the CLIST. The inclusion of &SYSPREF as the identification qualifier of the input data set frees the invoker from having to enter a fully-qualified data set name. The CLIST also substitutes &SYSPREF and &DSN on the allocation of the output data set so that its name corresponds to the name of the input data set.

The SCRIPTDS CLIST

```
PROC 1   DSN LIST
/*********************************************************************/
/*  THIS CLIST (SCRIPTDS) SETS UP THE ENVIRONMENT FOR SCRIPTING A    */
/*  DATA SET, ISSUES THE SCRIPT COMMAND, AND PRINTS THE OUTPUT.      */
/*********************************************************************/
CONTROL NOFLUSH NOMSG
IF &LIST=LIST THEN +
  CONTROL LIST
/*********************************************************************/
/* DELETE THE OUTPUT DATA SET INTO WHICH THE SCRIPTED FILE WILL BE   */
/* PLACED IN CASE IT IS STILL ALLOCATED FROM A PREVIOUS INVOCATION   */
/* OF SCRIPTDS.                                                      */
/*********************************************************************/
delete '&SYSPREF.&DSN.list'
/*********************************************************************/
/* DEFINE A FILE NAME (DDNAME) FOR THE OUTPUT DATA SET SO THAT THE   */
/* SCRIPT PROGRAM CAN REFERENCE IT.  FREE THE FILE BECAUSE SCRIPT WILL*/
/* ALSO ALLOCATE THE DATA SET.                                       */
/*********************************************************************/
alloc f(a) da('&SYSPREF.&DSN.list') dsorg(ps) recfm(v,b,m) +
  blk(3156) sp(10,10) tr new release reu
free f(a)
CONTROL LIST
/*********************************************************************/
/* ISSUE THE SCRIPT COMMAND, SPECIFYING THE NAME OF THE DATA SET     */
/* MEMBER TO BE SCRIPTED: MEMO.TEXT(&DSN).                           */
/*********************************************************************/
script   '&SYSPREF.memo.text(&DSN)' +
 message(delay id trace) device(3800n6) twopass +
 profile('script.r3.maclib(ssprof)') +
 lib('script.r3.maclib') +
 sysvar(c 1 d yes) +
 bind(8 8) chars(gt12 gb12) file('&SYSPREF.&DSN.list') continue

/*********************************************************************/
/* FREE THE FILES REQUIRED TO PRINT THE SCRIPTED DATA SET.           */
/* THEN ALLOCATE THEM, REQUESTING TWO COPIES ON THE 3800 PRINTER.    */
/*********************************************************************/
SET RC=&LASTCC            /* Get SCRIPT return code                  */
IF RC<=4 THEN +
DO
 CONTROL NOMSG
 CONTROL MSG
 alloc f(sysprint) dummy reuse
 alloc f(sysut1) da('&SYSPREF.&DSN.list') shr reuse
 alloc f(sysut2) sysout(n) fcb(std4) chars(gt12,gb12) +
   copies(2) optcd(j) reuse
 alloc f(sysin) dummy reuse

/*********************************************************************/
/* INVOKE THE UTILITY TO HAVE THE DATA SET PRINTED AND FREE THE      */
/* FILES.                                                            */
```

```
/*********************************************************************/
 call 'sys1.linklib(iebgener)'
 free f(sysut1,sysut2,sysprint,sysin)
END
```

Invoking CLISTs to perform subtasks - the SCRIPTN CLIST

While you can write CLISTs that perform application tasks directly, you can also write CLISTs that subdivide application tasks among nested CLISTs and control their execution. For example, you can write a CLIST that invokes two other CLISTs to perform the same tasks as those performed by SCRIPTDS in "The SCRIPTDS CLIST" on page 123.

SCRIPTN, shown in "The SCRIPTN CLIST," produces the same results as SCRIPTDS. The invoker provides a data set name qualifier as done for SCRIPTDS. SCRIPTN defines &DSNAM as a global variable because SCRIPTN invokes two CLISTs that refer to the variable. SCRIPTN invokes a CLIST called SCRIPTD, which includes the &DSNAM variable as the member name of the memo.text data set on the invocation of the SCRIPT command (See "The SCRIPTD CLIST"). When finished with these tasks, SCRIPTD returns control to SCRIPTN and execution continues at the command following the invocation of SCRIPTD. This command is the invocation of a CLIST called OUTPUT (See "The OUTPUT CLIST" on page 125). OUTPUT performs the required allocations to invoke the IEBGENER utility to print the output data set.

The SCRIPTN CLIST

```
PROC 1 DSN
GLOBAL DSNAM
SET DSNAM=&DSN
IF &LENGTH(&DSN) LE 8 AND /* ENSURE VALID NAME AND */ +
  &DATATYPE(&SUBSTR(1,&DSN))=CHAR THEN /* VALID FIRST CHARACTER */ +
 DO

/*********************************************************************/
/*   INVOKE THE SCRIPTD CLIST TO SET UP THE ENVIRONMENT REQUIRED TO   */
/*   SCRIPT THE INPUT DATA SET AND THEN RUN THE SCRIPT COMMAND.   */

/*********************************************************************/
 %scriptd
/*********************************************************************/

/*   INVOKE THE OUTPUT CLIST TO PRINT 2 COPIES OF THE SCRIPTED        */
/*   DATA SET ON THE 3800 PRINTER.                                    */
/*********************************************************************/

 IF &LASTCC<=4 THEN  /* Test return code from SCRIPTD */ +
  DO
 %output
  END
 ELSE WRITE SCRIPTD FAILED
 END
ELSE +
 WRITE The name entered must be less than 9 characters long and +
         the first character must not be numeric.
```

The SCRIPTD CLIST

```
GLOBAL DSNAM

/*********************************************************************/
/*  THIS CLIST (SCRIPTD) SETS UP THE ENVIRONMENT FOR SCRIPTING A      */
/*  DATA SET PROVIDED BY THE USER AND ISSUES THE SCRIPT COMMAND.      */
```

```
/*********************************************************************/

CONTROL NOFLUSH NOMSG
ERROR +
 DO     /* If an error occurs,
  SET RC=&LASTCC       /* get return code
  EXIT CODE(&RC)/* and pass control back to SCRIPTN
 END
/*********************************************************************/
/* DELETE THE OUTPUT DATA SET INTO WHICH THE SCRIPTED FILE WILL BE    */
/* PLACED IN CASE IT IS STILL ALLOCATED FROM A PREVIOUS INVOCATION    */
/* OF SCRIPTN.                                                        */
/*********************************************************************/

delete '&SYSPREF.&DSNAM.list'
/*********************************************************************/
/* DEFINE THE OUTPUT DATA SET SO THAT THE SCRIPT PROGRAM CAN REFERENCE*/
/* IT.  FREE THE FILE BECAUSE SCRIPT WILL ALSO ALLOCATE THE DATA SET  */
/*********************************************************************/

alloc f(a) da('&SYSPREF.&DSNAM.list') dsorg(ps) recfm(v,b,m) +
  blk(3156) sp(50,30) tr new release reu
free f(a)
CONTROL LIST
/*********************************************************************/
/* ISSUE THE SCRIPT COMMAND, SPECIFYING THE NAME OF THE DATA SET      */
/* MEMBER TO BE SCRIPTED: MEMO.TEXT(&DSNAM).                          */
/* THEN RETURN CONTROL TO SCRIPTN.                                    */
/*********************************************************************/

script   '&SYSPREF.memo.text(&DSNAM)' +
 message(delay id trace) device(3800n6) twopass +
 profile('script.r3.maclib(ssprof)') +
 lib('script.r3.maclib') +
 sysvar(c 1 d yes) +
 bind(8 8) chars(gt12 gb12) file('&SYSPREF.&DSNAM.list') continue
```

The OUTPUT CLIST

```
GLOBAL DSNAM

/*********************************************************************/
/* THIS CLIST (OUTPUT) FREES FILES REQUIRED TO PRINT THE SCRIPTED     */
/* DATASET, ALLOCATES THEM REQUESTING TWO COPIES ON THE 3800          */
/* PRINTER, AND INVOKES IEBGENER TO HAVE THEM PRINTED.                */
/*********************************************************************/

CONTROL NOMSG
CONTROL MSG
alloc f(sysprint) dummy reuse
alloc f(sysut1) da('&SYSPREF.&DSNAM.LIST') shr reuse
alloc f(sysut2) sysout(n) fcb(std4) chars(gt12,gb12) +
  copies(2) optcd(j) reuse
alloc f(sysin) dummy reuse

/*********************************************************************/
/* INVOKE THE UTILITY TO HAVE THE DATA SET PRINTED AND FREE THE       */
/* FILES.  THEN RETURN CONTROL TO SCRIPTN.                            */
/*********************************************************************/

call 'sys1.linklib(iebgener)'
free f(sysut1,sysut2,sysprint,sysin)
```

Including JCL statements - the SUBMITDS CLIST

You can include job control language (JCL) statements in CLISTs. The SUBMITDS CLIST, shown in Figure 12, uses the SUBMIT * command to indicate that the JCL statements immediately follow the command.

SUBMITDS verifies job card information using front-end prompting and then submits a job that copies one data set into another. The validity-checking does not go beyond verifying that the account number is a four-digit number.

Because an account number may contain leading zeros that are ignored by the &LENGTH built-in function, the CLIST uses the &STR built-in function in the evaluation of the length of &ACCT.

The SUBMITDS CLIST assumes that:
- The account number is required and must be a four-digit number.
- The account number may contain leading zeros.
- The default CLASS for the job is C.

```
PROC 2 DSN ACCT CLASS(C)

/*********************************************************************/
/* IF &ACCT IS NOT VALID, CONTINUE PROMPTING UNTIL THE USER ENTERS      */
/* AN ACCEPTABLE VALUE.                                              */
/*********************************************************************/

DO WHILE &LENGTH(&STR(&ACCT)) ¬= 4 OR &DATATYPE(&ACCT) ¬= NUM
 WRITE Your account number is invalid.
 WRITE Reenter a four-digit number.
 READ ACCT
END

/*********************************************************************/
/* ONCE ACCOUNT NUMBER HAS BEEN VERIFIED, SUBMIT THE JOB.            */
/*********************************************************************/

SET SLSHASK=&STR(/*) /* Set the /* required for jcl comment statement */
SUBMIT *   END($$)
//&SYSUID1 JOB   &ACCT,&SYSUID,CLASS=&CLASS,NOTIFY=&SYSUID
/&SLSHASK  THIS STEP COPIES THE INPUT DATASET TO SYSOUT=A
//COPY      EXEC  PGM=COPYDS
//SYSUT1    DD    DSN=&SYSUID.&DSN,DISP=SHR;
//SYSUT2    DD    SYSOUT=A
$$
```

Figure 12. The SUBMITDS CLIST

Analyzing input strings with &SUBSTR - the SUBMITFQ CLIST

You can use the &SUBSTR built-in function to analyze input from the invoker and to modify the input if necessary.

The SUBMITFQ CLIST, shown Figure 13 on page 127, determines whether the data set name supplied by the invoker is a fully-qualified name or not. If the data set name is not fully qualified (does not include a user ID), the SUBMITFQ adds the user ID and submits the data set name in the correct form on a JCL statement.

SUBMITFQ determines whether the data set name is fully qualified by comparing the first character in &DSN to a single quote ('). If the logical comparison is true, the CLIST assumes a fully-qualified data set name and removes the quotation marks. (Unlike on the ALLOCATE command, fully-qualified data set names are not enclosed in singquotation marksle quotes on JCL statements.) If the first character of &DSN is not a single quote, the CLIST assumes the data set name is not fully qualified and prefixes the character string "&SYSUID.." to the value of &DSN. In either case, &DSN contains a fully-qualified data set name when referred to in the SYSUT1 JCL statement.

```
PROC 2 DSN ACCT CLASS(C)

/*********************************************************************/
/* IF &ACCT IS NOT VALID, CONTINUE PROMPTING UNTIL THE USER ENTERS  */
/* AN ACCEPTABLE VALUE.                                              */
/*********************************************************************/

DO WHILE &LENGTH(&STR(&ACCT)) ¬= 4 OR &DATATYPE(&ACCT) ¬= NUM
 WRITE Your account number is invalid.
 WRITE Reenter a four-digit number.
 READ ACCT
END
/*********************************************************************/
/* IF THE DATA SET IS FULLY QUALIFIED, REMOVE THE QUOTATION MARKS.  OTHERWISE, */
/* PREFIX THE CURRENT USERID.                                        */
/*********************************************************************/

IF &STR(&SUBSTR(1,&DSN)) = ' THEN +
 SET DSN = &STR(&SUBSTR(2:&LENGTH(&DSN)-1,&DSN))
ELSE SET DSN=&STR(&SYSUID.&DSN)
WRITE &DSN
/*********************************************************************/
/* ONCE ACCOUNT NUMBER HAS BEEN VERIFIED, SUBMIT THE JOB.            */
/*********************************************************************/

SET SLSHASK=&STR(/*) /* Set the /* req. for the jcl comment statement */
SUBMIT *   END($$)
//&SYSUID1 JOB   &ACCT,&SYSUID,CLASS=&CLASS
/&SLSHASK   THIS STEP COPIES THE INPUT DATASET TO SYSOUT=A
//COPY      EXEC  PGM=COPYDS
//SYSUT1    DD    DSN=&DSN,DISP=SHR;
//SYSUT2    DD    SYSOUT=A
$$
```

Figure 13. The SUBMITFQ CLIST

Allowing foreground and background execution of programs - the RUNPRICE CLIST

You can write CLISTs that invoke programs in either the foreground or the background. By creating a background job, the CLIST can have the job invoke any program, including itself, in the background. You can use this type of a CLIST to enable users who are unfamiliar with JCL to submit programs. By placing the JCL in a CLIST, you simplify the user's work, while adding greater range to the tasks the user can perform. The RUNPRICE CLIST, shown in Figure 14 on page 128, illustrates this type of a CLIST.

RUNPRICE either executes a COBOL program called APRICE in the foreground or submits a job that executes APRICE in the background. The CLIST determines

which type of invocation to perform based on whether the invoker includes the BATCH keyword on the invocation of RUNPRICE.

```
PROC 0 M(R) BATCH

/*********************************************************************/
/*   THIS CLIST (RUNPRICE) SUBMITS A JOB FOR EXECUTION EITHER IN THE  */
/*   FOREGROUND OR BACKGROUND, BASED ON WHETHER THE INVOKER INDICATES */
/*   'BATCH' ON THE INVOCATION.  THE MESSAGE CLASS DEFAULTS TO 'R',   */
/*   THOUGH THE INVOKER MAY CHANGE IT.                               */
/*********************************************************************/
CONTROL  END(ENDO)
/*********************************************************************/
/*   IF &BATCH DOES NOT EQUAL A NULL, THIS INDICATES THAT THE INVOKER */
/*   INCLUDED THE KEYWORD ON THE INVOCATION. IN THIS CASE, THE INVOKER*/
/*   WANTS THE JOB SUBMITTED IN THE BACKGROUND, SO CREATE A JOB THAT  */
/*   EXECUTES THE TMP AND THEN INVOKES RUNPRICE WITHOUT THE 'BATCH'   */
/*   KEYWORD.  ON THIS SECOND INVOCATION OF RUNPRICE, ONLY THE        */
/*   APRICE PROGRAM IS EXECUTED.                                     */
/*   IF &BATCH EQUALS A NULL, THIS INDICATES THAT THE INVOKER WANTS   */
/*   TO START THE PROGRAM IN THE FOREGROUND.  IN THIS CASE, SIMPLY  */
/*   INVOKE THE APRICE PROGRAM DIRECTLY.                             */
/*********************************************************************/
SET SLSHASK=&STR(/*) /* Set the /* for JOBPARM to a variable          */
IF &BATCH=BATCH THEN +
 DO
  CONTROL NOMSG
  SUBMIT * END(NN)
  //STEVE1   JOB 'accounting info','STEVE',
  //         MSGLEVEL=(1,1),CLASS=T,NOTIFY=&SYSUID,MSGCLASS=&M,
  //         USER=????????,PASSWORD=????????
  &SLSHASK JOBPARM COPIES=1
  //BACKTMP EXEC  PGM=IKJEFT01,REGION=4096K,DYNAMNBR=10
  //SYSPRINT DD DUMMY
  //SYSTSPRT DD SYSOUT=*
  //SYSTSIN  DD *
  ex 'd84rlh1.tsoer2.pubs.clist(runprice)'
  NN
 ENDO
ELSE  call 'd60fot1.allot.cobol(aprice)'
```

Figure 14. The RUNPRICE CLIST

Including options - the TESTDYN CLIST

You can code options in a CLIST so that the CLIST performs different functions depending on what the invoker specifies.

The TESTDYN CLIST, shown in "The TESTDYN CLIST" on page 129, sets up the environment needed to execute a program called PARMTEST, which tests dynamic allocation input parameters entered from the terminal. In TESTDYN, conditional IF-THEN-ELSE sequences and optional keywords on the PROC statement enable the invoker to select a number of options when invoking the CLIST. For example, one option is whether the invoker wants the system messages that PARMTEST produces sent to a data set rather than to the terminal. TESTDYN includes a keyword parameter, SYSPRINT, on its PROC statement and assigns it a default value of *, which sends system messages to the terminal. The invoker can override that default value and have system messages sent to a system output data set.

Note that special considerations are taken in the processing that determines whether SYSOUT has been coded for SYSPRINT. On the IF statement, the variable &SYSPRINT is enclosed in a &STR built-in function because &SYSPRINT defaults to an asterisk, which the CLIST views as a multiplication operator. The &STR built-in function defines the asterisk as character data and prevents the CLIST from using it arithmetically.

The TESTDYN CLIST

```
PROC 0 MBR(PARMTEST) SYSPRINT(*) SYSLIB(LOAD) OUTFILE(VLDPARMS) LISTDSETS

/*********************************************************************/
/*   THIS CLIST SETS UP THE ENVIRONMENT NEEDED FOR EXECUTION OF      */
/*   A PROGRAM NAMED 'PARMTEST' WHICH TESTS DYNAMIC ALLOCATION       */
/*   INPUT PARAMETERS ENTERED FROM THE TERMINAL.                     */
/*********************************************************************/

/*********************************************************************/
/*    IF THE USER REQUESTED THAT DATA SETS BE LISTED, LIST THEM.     */
/*********************************************************************/

IF &LISTDSETS = LISTDSETS THEN +
 DO
  WRITE PROGRAM:   &MBR
  WRITE SYSPRINT:  &SYSPRINT
  WRITE SYSLIB:    &SYSLIB
  WRITE OUTFILE:   &OUTFILE
END
/*********************************************************************/
/*    IF THE USER REQUESTED THAT SYSTEM MESSAGES BE SENT TO A SYSTEM  */
/*    OUTPUT DATA SET, ALLOCATE SYSPRINT TO SYSOUT.  OTHERWISE,      */
/*    ALLOCATE SYSPRINT TO THE DATA SET NAME (OR TERMINAL) AS        */
/*    INDICATED BY THE USER.                                         */
/*********************************************************************/

IF &STR(&SYSPRINT) = SYSOUT THEN +
 alloc f(sysprint) sysout reu
ELSE alloc f(sysprint) da(&SYSPRINT) reu

/*********************************************************************/
/*    ALLOCATE THE SYSTEM LIBRARY, WHETHER IT BE THE DEFAULT (LOAD)  */
/*    OR ANOTHER LIBRARY.                                            */
/*********************************************************************/

alloc f(syslib) da(&SYSLIB) reu shr

/*********************************************************************/
/*   ALLOCATE THE OUTPUT DATA SET FOR THE PROGRAM.  ALLOCATE THE     */
/*   INPUT DATA SET TO THE TERMINAL.                                 */
/*********************************************************************/

alloc f(outfile) da(&OUTFILE) lrecl(121) blksize(1210) recfm(f,b) reu
alloc f(sysin) da(*) reu
/*********************************************************************/
/*   CALL PARMTEST AND NOTIFY THE USER THAT THE INVOCATION WAS       */
/*   SUCCESSFUL OR UNSUCCESSFUL.                                     */
/*********************************************************************/

CONTROL NOFLUSH
call 'steve.lib.load(&MBR)'
IF &LASTCC = 0 THEN +
 WRITE &MBR invoked successfully at &SYSTIME on &SYSDATE
ELSE +
 WRITE &MBR invoked unsuccessfully at &SYSTIME on &SYSDATE
```

Simplifying system-related tasks - the COMPRESS CLIST

From time to time, users must compress a data set they have updated multiple times to free some space for additional members. The process involves allocating the data sets required by the IEBCOPY utility, which performs the copying involved in compressing the data set, and invoking the utility.

The COMPRESS CLIST, shown in "The COMPRESS CLIST," performs all of the functions required to compress a data set.

The COMPRESS CLIST includes special procedures to make the best use of storage space. For example, COMPRESS can allocate a data set to contain the input required by the IEBCOPY utility. However, IEBCOPY requires only the following command for input:

```
copy indd=output,outdd=output
```

Rather than waste permanent storage for the one command, COMPRESS creates a virtual I/O (VIO) data set for the SYSIN file using an ALLOCATE command that does not specify a data set name. The ALLOCATE command assigns the file name SYSIN to the VIO data set and then writes a record containing the COPY command to the SYSIN file.

The COMPRESS CLIST

```
PROC 1 DSNAME DISP(OLD) LIST
CONTROL NOFLUSH /* Preserve the input stack for errors                */

/*********************************************************************/
/* THIS CLIST (COMPRESS) COMPRESSES A DATA SET AND INFORMS THE USER   */
/* WHETHER  THE COMPRESS WAS SUCCESSFUL.                              */
/*********************************************************************/
/* SET UP AN ERROR ROUTINE TO FREE ALLOCATED FILES WHEN AN ERROR OCCURS.
/*********************************************************************/

ERROR +
 DO
  ERROR OFF
  WRITE An error has occurred prior to the actual compress.
  free file(sysin,sysprint,sysut3,sysut4,output)
  GOTO FINISH
 END
/*********************************************************************/
/* IF THE USER WANTS TO VIEW THE TSO COMMANDS AS THEY START, ISSUE  */
/* THE CONTROL LIST STATEMENT.                                       */
/*********************************************************************/

IF &LIST=LIST THEN +
 CONTROL LIST
/*********************************************************************/
/*           ESTABLISH ENVIRONMENT NEEDED BY IEBCOPY UTILITY.        */
/*********************************************************************/

allocate file(sysin) space(1,1) track lrecl(80) recfm(f) blksize(80) reuse
IF &SYSDSN(COMPRESS;LIST) ¬= OK THEN +
allocate file(sysprint) dataset(compress.list) recfm(f,b,a) +
  lrecl(121) blksize(12947) space(1,1) track reuse
ELSE +
allocate file(sysprint) dataset(compress.list) shr reuse
allocate file(sysut3) unit(sysda) space(1,1) cylinders reu
allocate file(sysut4) unit(sysda) space(1,1) cylinders reu
allocate file(output) dataset(&DSNAME) &DISP reu
/*********************************************************************/
/*  PLACE THE COPY COMMAND INTO THE SYSIN FILE REQUIRED BY IEBCOPY.   */
```

```
/*********************************************************************/

OPENFILE SYSIN OUTPUT
SET SYSIN = &STR(   COPY INDD=OUTPUT,OUTDD=OUTPUT)
PUTFILE SYSIN
CLOSFILE SYSIN
/*********************************************************************/
/* Set up an error routine to notify user of compress errors.        */
/*********************************************************************/

ERROR +
 DO
  WRITE Compress error--Details in '&SYSPREF compress.list'
  GOTO FINISH
 END
/*********************************************************************/
/*        INVOKE IEBCOPY UTILITY TO PERFORM THE COMPRESS.            */
/*********************************************************************/

tsoexec call 'sys1.linklib(iebcopy)' 'size=512k'
  WRITE &DSNAME compressed at &SYSTIME
FINISH: end  /* End the CLIST                                        */
```

Simplifying interfaces to applications - the CASH CLIST

You may have access to applications written in other programming languages. However, the interfaces required to invoke these programs may not be easily mastered by users who use the system infrequently. Rather than write new applications, you can write CLISTs that act as intermediaries between users and such programs.

For example, a program called CASHFLOW creates and prints weekly and monthly reports. If the invoker wants a weekly report, the invocation is:

```
call 'sys1.plib(cashflow)' 'a,,,38,ccfdacr'
```

If the invoker wants a monthly report, the invocation is:

```
call 'sys1.plib(cashflow)' 'x,,,49,ccfmacr'
```

Not only are the preceding invocations quite technical, they are difficult to remember.

CASHFLOW also requires the allocation of a file. For weekly reports, it requires:

```
alloc f(projwkly) da(weekly) shr
```

For monthly reports, it requires:

```
alloc f(projmtly) da(monthly) shr
```

To simplify the process of invoking CASHFLOW, the CASH CLIST, shown in "Simplifying interfaces to applications - the CASH CLIST," performs the following intermediary tasks:

1. It determines whether the invoker wants a weekly or monthly report.
2. It assigns values to the variables substituted in the parameter string on the CALL command that invokes CASHFLOW. The values correspond to the parameters required for the type of report requested.
3. It allocates the appropriate data set.

```
/* PROMPT THE USER FOR THE WORD 'WEEKLY' or 'MONTHLY'                */

DO WHILE &TYPE¬=WEEKLY AND &TYPE¬=MONTHLY
 WRITE Enter the word WEEKLY or MONTHLY to indicate the
 WRITE type of report you want to create.
 READ TYPE
END
/*********************************************************************/
/* NOW THAT A VALID REQUEST HAS BEEN ESTABLISHED, ALLOCATE THE       */
/* APPROPRIATE DATA SET, ASSIGN THE APPROPRIATE VALUES TO CALL       */
/* COMMAND PARAMETER VARIABLES, AND INVOKE CASHFLOW.                 */
/*********************************************************************/

IF &TYPE=WEEKLY THEN +
 DO
  alloc f(projwkly) da(weekly) shr
  SET INVOKE=38
  SET CHAR=a
  SET OPT=ccfdacr
 END
ELSE +
 DO
  alloc f(projmtly) da(monthly) shr
  SET INVOKE=49
  SET CHAR=x
  SET OPT=ccfmacr
 END
call 'sys1.plib(cashflow)' '&CHAR,,,&INVOKE,&OPT'
```

Figure 15. The CASH CLIST

Using &SYSDVAL when performing I/O - the PHONE CLIST

Data records often contain related pieces or blocks of information. For instance, a sequential record can contain a person's name and telephone number. When you read records of this type, you may want to separate the blocks of information. By defining SYSDVAL as the file name of the data set containing the records, you read each record into SYSDVAL, which the CLIST equates with the &SYSDVAL control variable. Then you can issue a READDVAL statement that contains the names of the variables into which you want the blocks of information stored.

The PHONE CLIST, shown in Figure 16 on page 133, takes advantage of this technique. PHONE receives a family name as input using a positional parameter called NAME. PHONE then allocates a data set called SYS1.STAFF.DIRECTRY and assigns it the file name SYSDVAL. Each record in SYS1.STAFF.DIRECTRY contains a family name, followed by a blank and a telephone number. Sample records are:

```
PICKERELL 555-5555
GORGEN 555-4444
```

PHONE sets the first character string in the record to a variable called &LNAME and sets the second string to a variable called &PHONUMBR. Then, it compares &NAME to &LNAME and, if they are equal, displays the corresponding telephone number (contained in &PHONUMBR) at the terminal. If the names are not equal, PHONE reads another record and performs the same test.

If none of the names in the directory match the name supplied by the invoker, the CLIST branches to the end-of-file error routine. The end-of-file routine informs the invoker that a name was not found, and sets the variable DONE=YES to cause the

loop to terminate.

```
PROC 1 NAME

/*********************************************************************/
/* THIS CLIST (PHONE) SEARCHES A DATA SET FOR A NAME THAT MATCHES THE */
/* NAME SUPPLIED TO THE CLIST. IF A MATCH IS FOUND, THE CORRESPONDING */
/* TELEPHONE NUMBER IS DISPLAYED AT THE TERMINAL. OTHERWISE, A MESSAGE IS */
/* ISSUED INFORMING THE USER THAT A MATCH WAS NOT FOUND.             */
/*********************************************************************/

/*********************************************************************/
/* ALLOCATE THE INPUT DATA SET FOR THE CLIST.                        */
/*********************************************************************/

alloc f(sysdval) da('sys1.staff.directry') shr reu
/*********************************************************************/
/* OPEN THE FILE, AND SET UP AN ERROR ROUTINE TO HANDLE END-OF-FILE.  */
/*********************************************************************/

CONTROL NOMSG NOFLUSH
ERROR +
 DO
  IF &LASTCC = 400 THEN +
   DO
    WRITENR The name requested, &NAME, was not found in the staff
    WRITE directory.
    SET DONE=YES
   END
  RETURN
 END /* END OF END-OF-FILE ROUTINE                                  */
SET DONE=NO
OPENFILE SYSDVAL
/*********************************************************************/
/* THIS LOOP RETRIEVES RECORDS FROM THE INPUT DATA SET UNTIL A MATCH  */
/* IS FOUND OR END OF FILE IS REACHED.  IF A MATCH IS FOUND, THE      */
/* SECOND VARIABLE ON THE READDVAL STATEMENT (THE ONE CONTAINING      */
/* THE TELEPHONE NUMBER) IS DISPLAYED.                                  */
/*********************************************************************/

DO WHILE &DONE=NO
 GETFILE SYSDVAL
 READDVAL LNAME PHONUMBR
 IF &STR(&NAME) = &STR(&LNAME) THEN +
  DO
   WRITE &PHONUMBR
   SET DONE=YES
  END
END
CLOSFILE SYSDVAL
free file(sysdval)
```

Figure 16. The PHONE CLIST

Allocating data sets to SYSPROC - the SPROC CLIST

The SPROC CLIST allocates a CLIST data set to the file SYSPROC, so users can implicitly execute CLISTs that are in that data set. The SPROC CLIST allocates the CLIST data set as the first in the list of data sets allocated to SYSPROC, so TSO/E searches that data set for CLISTs before searching any of the others.

Note: You can also use the ALTLIB command to define CLIST data sets and establish their search order for implicit execution.

SPROC performs the following steps: finds all data sets currently allocated to SYSPROC and concatenates them; then adds the invoker's data set to the beginning of the concatenation and allocates the concatenation to SYSPROC.

The CLIST, shown in "Allocating data sets to SYSPROC - the SPROC CLIST" on page 133, uses &SYSOUTTRAP to intercept the output from the LISTALC STATUS command and saves the command output in &SYSOUTLINE*nn* variables. The output produced by the LISTALC STATUS command is formatted as follows:

```
--DDNAME---DISP--
DATA-SET-NAME1
  FILE-NAME1    DISPOSITION
DATA-SET-NAME2
  FILE-NAME2    DISPOSITION
DATA-SET-NAME3
                DISPOSITION
DATA-SET-NAME4
  FILE-NAME3    DISPOSITION
```

In the previous format, DATA-SET-NAME1 is allocated to FILE-NAME1; DATA-SET-NAME2 and DATA-SET-NAME3 are allocated to FILE-NAME2; and DATA-SET-NAME4 is allocated to FILE-NAME3. The name of a file always begins in the third position, whereas a data set name begins in the first position of the output line. SPROC does the following:

1. Loops through &SYSOUTLINE*nn* variables until either the string SYSPROC is found or until all output has been searched. (It is possible no data sets are allocated to SYSPROC.)
2. If SYSPROC is found, SPROC sets a variable to the name of the previous data set in the list and encloses it in single quotation marks.
3. Begins with the &SYSOUTLINE*nn* variable three lines after the one containing the name of the first data set allocated to SYSPROC. This line either contains a new file name, in which case SPROC has found all data sets allocated to SYSPROC, or it contains the disposition of the next data set in the concatenation. By setting a variable to three blanks, SPROC determines the contents of the line.

   If the line contains a disposition, SPROC decreases &SYSOUTLINE*nn* by one to get the data set name and add it to the variable (&CONCAT) representing the data sets in the new concatenation. SPROC repeats this procedure until another file name is encountered or until all command output has been searched. After all data sets have been added to the concatenation list, SPROC issues the ALLOCATE command, adding the user's data set name to the beginning of the concatenation list.

SPROC contains an error routine to handle allocation errors should they occur. SPROC may itself be allocated to SYSPROC, in which case the user can invoke SPROC implicitly. However, if the CLIST fails after it frees the SYSPROC file, but before it is able to re-establish the concatenation, the user cannot re-invoke SPROC implicitly without first logging off and logging on again.

The SPROC CLIST

```
PROC 0 LIST
  IF &LIST=LIST THEN +
  CONTROL LIST CONLIST
/*********************************************************************/
/*  THIS CLIST (SPROC) CONCATENATES DATA SETS AND ALLOCATES THEM      */
```

```
/*  TO THE FILE SYSPROC.                                              */
/*  THE USER IS PROMPTED TO SUPPLY THE NAME OF THE DATA               */
/*  SET TO BE ADDED TO THE BEGINNING OF THE CONCATENATION.            */
/*********************************************************************/
/*********************************************************************/
/*  IF ALLOCATION FAILS, TELL THE USER TO LOG OFF, LOG ON, AND, IF    */
/*  DESIRED, TRY EXECUTING SPROC AGAIN.                               */
/*********************************************************************/
CONTROL NOFLUSH
ERROR +
 DO
  WRITE An error has been encountered in the SYSPROC concatenation.
  WRITE Please log off, then log on again, and, if desired, re-invoke
  WRITE SPROC.  If the problem persists, see your system programmer.
  GOTO OUT
 END
/*********************************************************************/
/*  PROMPT THE USER FOR THE NAME OF THE DATA SET TO BE ADDED TO THE   */
/*  BEGINNING OF THE SYSPROC CONCATENATION.                           */
/*********************************************************************/
WRITE Enter the fully-qualified data set name you want
WRITE added to the beginning of the SYSPROC concatenation.
WRITE Do  N O T  place quotation marks around the dataset name.
READ ADD
/*********************************************************************/
/*  SET A VARIABLE TO THREE BLANKS.  THIS VARIABLE IS USED TO CHECK   */
/*  THE LISTALC COMMAND OUTPUT FOR THE BEGINNING OF A DIFFERENT       */
/*  FILENAME AFTER SYSPROC DATA SETS HAVE BEEN LISTED.                */
/*********************************************************************/
SET BLANKS = &STR(   )
/*********************************************************************/
/*  SET &SYSOUTTRAP TO A LARGE ENOUGH VALUE TO ENSURE THAT ALL OF     */
/*  THE LINES OF OUTPUT FROM THE LISTALC COMMAND CAN BE VIEWED.       */
/*********************************************************************/

SET &SYSOUTTRAP = 300
/*********************************************************************/
/*  ISSUE THE LISTALC STATUS COMMAND AND LOOP THROUGH THE VARIABLES   */
/*  CONTAINING THE OUTPUT LINES UNTIL THE LINE CONTAINING             */
/*  THE FILENAME                                                      */
/*  SYSPROC IS FOUND OR UNTIL ALL LINES HAVE BEEN VIEWED.             */
/*  (ALL LINES HAVE BEEN VIEWED WHEN A NULL LINE IS RETURNED.)        */
/*  AN AUXILIARY VARIABLE MUST BE CREATED (&DSN) TO LOOP THROUGH      */
/*  &SYSOUTLINEnn &I REPRESENTS THE VALUE OF nn.                     */
/*  NOTE THAT, TO SET &DSN TO &SYSOUTLINE, TWO AMPERSANDS             */
/*  MUST BE PLACED BEFORE SYSOUTLINE TO AVOID SYMBOLIC SUBSTITUTION   */
/*  OF &SYSOUTLINE                                                    */
/*  IF SYSPROC IS FOUND, SET THE VARIABLE &CONCAT EQUAL TO            */
/*  THE PREVIOUS LINE (CONTAINING THE NAME                            */
/*  OF THE FIRST DATA SET ALLOCATED TO SYSPROC).                      */
/*********************************************************************/
/*********************************************************************/
/*  IF SYSPROC WAS FOUND, LOOP THROUGH DATA SETS UNTIL ANOTHER        */
/*  FILENAME IS ENCOUNTERED OR UNTIL THE REST OF THE OUTPUT HAS       */
/*  BEEN PROCESSED.  SETTING &I = &I+3 MAPS &DSN TO THE LINE AFTER    */
/*  THE NEXT DATA SET NAME, WHICH WILL CONTAIN ANOTHER FILENAME IF    */
/*  WE HAVE ALREADY PROCESSED THE LAST DATA SET ALLOCATED TO SYSPROC  */
/*  AND WE HAVE NOT REACHED THE END OF THE COMMAND OUTPUT.            */
/*********************************************************************/
IF &FOUND=YES THEN  +
  DO WHILE &I+3 <= &SYSOUTLINE
   SET I = &I+3;
   SET DSN = &&SYSOUTLINE&I
    IF &STR(&SUBSTR(1:3,&DSN)) = &BLANKS THEN +
     DO
      SET I = &I-1
      SET DSN = &&SYSOUTLINE&I
```

```
      SET CONCAT = &CONCAT&STR( ')&DSN'
     END
    ELSE +
     SET I=&SYSOUTLINE
  END
/*********************************************************************/
/*  WHEN ALL DATA SETS ALLOCATED TO SYSPROC HAVE BEEN ADDED TO THE    */
/*  VARIABLE &CONCAT, ADD THE USER'S DATA SET TO THE BEGINNING OF     */
/*  THE CONCATENATION.  (INSERT THE VARIABLE &ADD BEFORE &CONCAT)    */
/*  THIS CLIST ASSUMES THAT THE DATA SET HAS BEEN ENTERED CORRECTLY   */
/*  BY THE USER.                                                      */
/*********************************************************************/
alloc f(sysproc) da('&ADD'  &CONCAT) shr reu
OUT: end
```

Writing full-screen applications using ISPF dialogs - the PROFILE CLIST

The CLIST language is well-suited for applications that invoke ISPF dialog management services to display full-screen panels. For more information about ISPF, see *z/OS V2R2 ISPF Services Guide*.

The PROFILE CLIST is an example of a CLIST that displays entry panels on which the user can modify information. The PROFILE CLIST allows the user to perform any of the following functions to modify his or her profiles:

- Set terminal characteristics.
- Set LOG/LIST parameters.
- Set PF keys (1-12).
- Set PF keys (13-24).

The PROFILE CLIST receives control from a CLIST that displays the primary selection panel. The primary selection panel prompts the user to indicate which function is being requested (QCMD); and if the function is setting PF keys, which PF keys are to be viewed (QKEYS). Then, the CLIST invokes PROFILE, passing the values for QCMD and QKEYS.

PROFILE determines which selection was requested by referencing PROC statement keywords called QCMD and QKEYS.

If &QCMD is 1, PROFILE displays the terminal characteristics panel definition.

If &QCMD is 2, PROFILE displays the LOG/LIST parameters panel definition.

If &QCMD is 3 and &QKEYS is 12, PROFILE displays the PF keys 1-12 panel definition.

If &QCMD is 3 and &QKEYS is 24, PROFILE displays the PF keys 13-24 panel definition.

Panels are displayed using the ISPEXEC command.

When the user presses the END key after viewing, modifying, or viewing and modifying a particular panel, the value of &LASTCC is 8. By testing the value of &LASTCC, PROFILE can determine when the user is finished with the selection.

When the user is viewing one of the two PF key panels, the user can switch to the other panel by pressing the Enter key. PROFILE sets &QKEYS to the PF key (12 or 24) that represents the other panel so that the user can continue to switch back and

forth if desired. Pressing Enter re-executes the DO-UNTIL-END sequence, causing PROFILE to test the value of &QKEYS to determine which panel to display. As with the other selection sequences, the PF key sequence ends when the user presses the END key.

Values set or changed on any of the four panels displayed by PROFILE are stored in the corresponding variables on the panel definitions.

Table 10 contains the purpose of, and figures containing, the PROFILE CLIST and its supporting four panel definitions.

Table 10. Purpose of, and figures containing, PROFILE CLIST and supporting panels

CLIST/Panel	Purpose	Figure
PROFILE	Manage user profile panels	26
XYZABC10	Terminal characteristics panel	27
XYZABC20	LOG/LIST parameters panel	28
XYZABC30	PF keys 1-12 panel	29
XYZABC40	PF keys 13-24 panel	30

The PROFILE CLIST

```
PROC 0 QCMD(1) QKEYS(12)

/*********************************************************************/
/*  THIS CLIST (PROFILE) DISPLAYS THE PANEL THAT CONTAINS THE PROFILE */
/*  DATA THE USER WANTS TO UPDATE.  IT SETS THE FINISH FLAG TO NO AND */
/*  THEN DETERMINES WHICH OF THE FOUR POSSIBLE PANELS THE USER NEEDS  */
/*  DISPLAYED.                                                       */
/*********************************************************************/

CONTROL MSG END(ENDO)
SET FINISH = NO
/*********************************************************************/
/*  IF THE USER WANTS TO UPDATE TERMINAL CHARACTERISTICS, DISPLAY     */
/*  THE ASSOCIATED PANEL.                                            */
/*********************************************************************/

SELECT
WHEN (&QCMD = 1) +
 DO UNTIL (&FINISH = YES)
  ISPEXEC DISPLAY PANEL(XYZABC10)   /* Display first panel           */
  IF &LASTCC = 8 THEN               /* If user presses END, */ +
   SET FINISH = YES                 /* end panel display             */
 ENDO
/*********************************************************************/
/*  IF THE USER WANTS TO UPDATE LOG/LIST PARAMETERS, DISPLAY         */
/*  THE ASSOCIATED PANEL.                                            */
/*********************************************************************/

WHEN (&QCMD = 2) +
  DO UNTIL (&FINISH = YES)
  ISPEXEC DISPLAY PANEL(XYZABC20)   /* Display 2nd panel             */
  IF &LASTCC = 8 THEN               /* If user presses END, */ +
   SET FINISH = YES                 /* end panel display.            */
  ENDO
/*********************************************************************/
/* IF THE USER WANTS TO UPDATE PF KEYS, DETERMINE WHICH GROUP THE     */
/* USER WANTS TO UPDATE: 1-12 or 13-24.  DISPLAY THE ASSOCIATED PANEL.*/
/*********************************************************************/
WHEN (&QCMD = 3) +
  DO UNTIL (&FINISH = YES)
```

```
    IF &QKEYS = 12 THEN +
     DO
      ISPEXEC DISPLAY PANEL(XYZABC30)  /* Display PF keys 1-12         */
      IF &LASTCC = 8 THEN              /* If user presses END,  */ +
       SET FINISH = YES                /* end panel display.           */
      ELSE +
       SET QKEYS = 24                  /* If user presses ENTER,       */
     ENDO                              /* display next panel.          */
    ELSE +
     DO
      ISPEXEC DISPLAY PANEL(XYZABC40)  /* Display PF keys 13-24        */
      IF &LASTCC = 8 THEN              /* If user presses END,  */ +
       SET FINISH = YES                /* end panel display.           */
      ELSE +
       SET QKEYS = 24                  /* If user presses ENTER,       */
     ENDO                              /* display previous panel.      */
   ENDO
 ENDO         /* End of SELECT statement
 /*
 /*   EXIT ROUTINE
 /*
 FINAL: +
 SET FCODE = 0
 EXIT CODE(&FCODE)
```

The panels displayed by the PROFILE CLIST appear on the following pages in panel definition form. In order for the PROFILE CLIST to display them, the panels must be members of a partitioned data set allocated to the file ISPPLIB, for example:

```
allocate file(ispplib) dataset(test.panels)
```

For more information about how to create and allocate ISPF panel definitions, see *z/OS V2R2 ISPF Services Guide*.

The Terminal characteristics panel definition (XYZABC10)

```
 )ATTR DEFAULT(%+_)
       /*  % TYPE(TEXT) INTENS(HIGH)      defaults displayed for       */
       /*  + TYPE(TEXT) INTENS(LOW)       information only             */
       /*  _ TYPE(INPUT) INTENS(HIGH) CAPS(ON) JUST(LEFT)              */
   @ TYPE(INPUT) INTENS(HIGH) PAD(_) CAPS(ON)
 )BODY
 +             SAMPLE - SET THE TERMINAL CHARACTERISTICS
 %COMMAND === _ZCMD                                                    +
 %
+Type the information where requested, or change the information shown
 +by typing over it:
 +
 +   TERMINAL TYPE    %=== @Z    +    3277, 3277A, 3278, 3278A, or 3278T   +
 +   NUMBER OF PF KEYS%===>@Z +       12 or 24
 +   INPUT FIELD PAD  %===>@Z+        Nulls (N) or Blanks (B)
 +   SCREEN FORMAT    %===>@Z   +     (3278 Model 5 only) DATA, STD, or MAX
 +   COMMAND DELIMITER%===>@Z+        Any special character
 +
 +
 +
 +
 +
 +
 +
 +
 +
 +
 +
 +
 +
```

```
+
)INIT
  .ZVARS = '(ZTERM ZKEYS ZPADC ZSF ZDEL)'
  &ZSF  = TRANS (&ZFMT D,DATA S,STD M,MAX *,' ')
)PROC
 IF (&ZCMD ¬= ' ') .MSG = ISPZ001      /* NOT VALID COMMAND      */
 VER (&ZTERM NB LIST 3277,3277A,3278,3278A,3278T)
 &ZCHARLM = TRANS(&ZTERM
                  3277  ,  ISP3277
                  3277A ,  ISP3277A
                  3278  ,  ISP3278
                  3278A ,  ISP3278A
                  3278T ,  ISP3278T)
 VER (&ZKEYS NB LIST 12,24)
 IF  (&ZKEYS = 24)
   VER (&ZTERM LIST 3278 MSG=ISPO002)
 VER (&ZPADC NB LIST N,B)
 VER (&ZSF,NONBLANK)
 &ZFMT = TRUNC (&ZSF,1)
 VER (&ZFMT,LIST  D,S,M)
 VER (&ZDEL NB PICT C)
 IF (.MSG ¬= ' ')
   .RESP = ENTER
)END
```

The LOG/LIST characteristics panel definition (XYZABC20)

```
)ATTR DEFAULT(%+_)
      /*  % TYPE(TEXT) INTENS(HIGH)      defaults displayed for       */
      /*  + TYPE(TEXT) INTENS(LOW)       information only             */
      /*  _ TYPE(INPUT) INTENS(HIGH) CAPS(ON) JUST(LEFT)              */
  @ TYPE(INPUT) INTENS(HIGH) PAD(_) CAPS(ON)
)BODY
+              SAMPLE - SET THE LOG/LIST PARAMETERS
%COMMAND  ===>_ZCMD                                                  +
%
+Type the information where requested, or change the information shown
 +by typing over it:
 +
 +                         %LOG                    %LIST                   +
 +
 +   PROCESS OPTION   %===>@Z+                     @Z+
 +   SYSOUT CLASS     %===>@Z              +        @Z              +
 +   LOCAL PRINTER ID %===>@Z       +              @Z       +
 +   LINES PER PAGE   %===>@Z +                    @Z +
 +   PRIMARY PAGES    %===>@Z   +                   @Z   +
 +   SECONDARY PAGES  %===>@Z   +                   @Z   +
 +
 +
 +
 +
 +
 +
 +
 +
 +
 +
 +
)INIT
  .ZVARS = '(ZLOGFDSP,ZLSTFDSP,ZLOGCLA,ZLSTCLA,ZLOGPID,ZLSTPID, +
             ZLOGLIN,ZLSTLIN,ZLOG1PG,ZLST1PG,ZLOG2PG,ZLST2PG)'

)PROC
  IF (&ZCMD ¬= ' ') .MSG = ISPZ001      /* NOT VALID COMMAND     */
  VER (&ZLOGFDSP  LIST J,L,K,D,' ')
  VER (&ZLSTFDSP  LIST J,L,K,D,' ')
  IF (&ZLOGFDSP = J)
    VER (&ZLOGCLA,NB)
```

```
  IF (&ZLOGFDSP = L)
    VER (&ZLOGPID,NB)
  IF (&ZLSTFDSP = J)
    VER (&ZLSTCLA,NB)
  IF (&ZLSTFDSP = L)
    VER (&ZLSTPID,NB)
  VER (&ZLOGLIN   NB NUM)
  VER (&ZLOGLIN   RANGE 1,99)
  VER (&ZLSTLIN   NB NUM)
  VER (&ZLSTLIN   RANGE 1,99)
  VER (&ZLOG1PG   NB NUM)
  VER (&ZLOG1PG   RANGE 0,9999)
  VER (&ZLST1PG   NB NUM)
  VER (&ZLST1PG   RANGE 1,9999)
  VER (&ZLOG2PG   NB NUM)
  VER (&ZLOG2PG   RANGE 0,9999)
  VER (&ZLST2PG   NB NUM)
  VER (&ZLST2PG   RANGE 1,9999)
  IF (&ZLOG1PG = 0)
    VER (&ZLOG2PG,NB)
    VER (&ZLOG2PG,RANGE,0,0)
  IF (&ZLOG1PG ¬= 0)
    VER (&ZLOG2PG,NB NUM)
    VER (&ZLOG2PG,RANGE,1,9999)
  IF (.MSG ¬= ' ' )
    .RESP = ENTER
)END
```

The PF keys 1-12 panel definition (XYZABC30)

```
)ATTR DEFAULT(%+_)
      /*  % TYPE(TEXT) INTENS(HIGH)      defaults displayed for       */
      /*  + TYPE(TEXT) INTENS(LOW)       information only             */
      /*  _ TYPE(INPUT) INTENS(HIGH) CAPS(ON) JUST(LEFT)              */
  @ TYPE(INPUT) INTENS(HIGH) PAD(_) CAPS(ON)
)BODY
+                   SAMPLE - SET PF KEYS 1-12
%COMMAND ===>_ZCMD  +
%
+Type the information where requested, or change the information shown
+by typing over it:
+
+   PF1   %===>@QPF01 +
+   PF2   %===>@QPF02 +
+   PF3   %===>@QPF03 +
+   PF4   %===>@QPF04 +
+   PF5   %===>@QPF05 +
+   PF6   %===>@QPF06 +
+   PF7   %===>@QPF07 +
+   PF8   %===>@QPF08 +
+   PF9   %===>@QPF09 +
+   PF10  %===>@QPF10 +
+   PF11  %===>@QPF11 +
+   PF12  %===>@QPF12 +
+
+
+
+
+
+
)INIT
 IF (&QPF01 = ' ')
   &QPF01 = HELP
 IF (&QPF02 = ' ')
   &QPF02 = SPLIT
 IF (&QPF03 = ' ')
   &QPF03 = END
```

```
 IF (&QPF04 = ' ')
   &QPF04 = RETURN
 IF (&QPF05 = ' ')
   &QPF05 = RFIND
 IF (&QPF06 = ' ')
   &QPF06 = RCHANGE
 IF (&QPF07 = ' ')
   &QPF07 = UP
 IF (&QPF08 = ' ')
   &QPF08 = DOWN
 IF (&QPF09 = ' ')
   &QPF09 = SWAP
 IF (&QPF10 = ' ')
   &QPF10 = LEFT
 IF (&QPF11 = ' ')
   &QPF11 = RIGHT
 IF (&QPF12 = ' ')
   &QPF12 = CURSOR
)PROC
 IF (&ZCMD ¬= ' ') .MSG = ISPZ001
 IF (&QPF01 = ' ')
   &QPF01 = HELP
 IF (&QPF02 = ' ')
   &QPF02 = SPLIT
 IF (&QPF03 = ' ')
   &QPF03 = END
 IF (&QPF04 = ' ')
   &QPF04 = RETURN
 IF (&QPF05 = ' ')
   &QPF05 = RFIND
 IF (&QPF06 = ' ')
   &QPF06 = RCHANGE
 IF (&QPF07 = ' ')
   &QPF07 = UP
 IF (&QPF08 = ' ')
   &QPF08 = DOWN
 IF (&QPF09 = ' ')
   &QPF09 = SWAP
 IF (&QPF10 = ' ')
   &QPF10 = LEFT
 IF (&QPF11 = ' ')
   &QPF11 = RIGHT
 IF (&QPF12 = ' ')
   &QPF12 = CURSOR
 IF (.MSG ¬= ' ')
   .RESP = ENTER
)END
```

The PF keys 13-24 panel definition (XYZABC40)

```
)ATTR DEFAULT(%+_)
      /*  % TYPE(TEXT) INTENS(HIGH)      defaults displayed for       */
      /*  + TYPE(TEXT) INTENS(LOW)       information only             */
      /*  _ TYPE(INPUT) INTENS(HIGH) CAPS(ON) JUST(LEFT)              */
  @ TYPE(INPUT) INTENS(HIGH) PAD(_) CAPS(ON)
)BODY
+                   SAMPLE - SET PF KEYS 13-24
%COMMAND ===>_ZCMD                                                    +
%
+Type the information where requested, or change the information shown
 +by typing over it; then, to set PF keys 1-12, press ENTER.
 +
 +   PF13  %===>@QPF13                                                +
 +   PF14  %===>@QPF14                                                +
 +   PF15  %===>@QPF15                                                +
 +   PF16  %===>@QPF16                                                +
 +   PF17  %===>@QPF17                                                +
```

```
+   PF18  %===>@QPF18                                                +
+   PF19  %===>@QPF19                                                +
+   PF20  %===>@QPF20                                                +
+   PF21  %===>@QPF21                                                +
+   PF22  %===>@QPF22                                                +
+   PF23  %===>@QPF23                                                +
+   PF24  %===>@QPF24                                                +
+
+
+
+
+
+
)INIT
 IF (&QPF13 = ' ')
   &QPF13 = HELP
 IF (&QPF14 = ' ')
   &QPF14 = SPLIT
 IF (&QPF15 = ' ')
   &QPF15 = END
 IF (&QPF16 = ' ')
   &QPF16 = RETURN
 IF (&QPF17 = ' ')
   &QPF17 = RFIND
 IF (&QPF18 = ' ')
   &QPF18 = RCHANGE
 IF (&QPF19 = ' ')
   &QPF19 = UP
 IF (&QPF20 = ' ')
   &QPF20 = DOWN
 IF (&QPF21 = ' ')
   &QPF21 = SWAP
 IF (&QPF22 = ' ')
   &QPF22 = LEFT
 IF (&QPF23 = ' ')
   &QPF23 = RIGHT
 IF (&QPF24 = ' ')
   &QPF24 = CURSOR
)PROC
 IF (&ZCMD ¬= ' ') .MSG = ISPZ001
 IF (&QPF13 = ' ')
   &QPF13 = HELP
 IF (&QPF14 = ' ')
   &QPF14 = SPLIT
 IF (&QPF15 = ' ')
   &QPF15 = END
 IF (&QPF16 = ' ')
   &QPF16 = RETURN
 IF (&QPF17 = ' ')
   &QPF17 = RFIND
 IF (&QPF18 = ' ')
   &QPF18 = RCHANGE
 IF (&QPF19 = ' ')
   &QPF19 = UP
 IF (&QPF20 = ' ')
   &QPF20 = DOWN
 IF (&QPF21 = ' ')
   &QPF21 = SWAP
 IF (&QPF22 = ' ')
   &QPF22 = LEFT
 IF (&QPF23 = ' ')
   &QPF23 = RIGHT
 IF (&QPF24 = ' ')
   &QPF24 = CURSOR
 IF (.MSG ¬= ' ')
   .RESP = ENTER
)END
```

Allocating a data set with LISTDSI information - the EXPAND CLIST

The EXPAND CLIST, shown in Figure 17, reallocates a data set with more space to prevent the data set from running out of space.

The EXPAND CLIST uses the LISTDSI statement to retrieve information about a base data set's allocation. The information is stored in CLIST variables. The CLIST then uses the information as input to a subprocedure. The subprocedure issues the TSO/E ALLOCATE command to create a new data set using the same attributes as the base data set, but doubling the primary space.

For more information about the CLIST variables set by LISTDSI, see "LISTDSI statement" on page 158.

```
/*********************************************************************/
/* PROCEDURE: EXPAND                                                 */
/*                                                                   */
/* INPUT:       BASEDS - NAME OF DATA SET WITH THE ALLOCATION        */
/*                       ATTRIBUTES YOU WANT THE NEW DATA SET        */
/*                       TO HAVE.                                    */
/*              NEWDS  - NAME OF NEW DATA SET TO BE ALLOCATED.       */
/*                                                                   */
/* OUTPUT:      NEW DATA SET ALLOCATED WITH THE SAME ATTRIBUTES AS    */
/*              THE BASE DATA SET BUT WITH A PRIMARY ALLOCATION      */
/*              TWICE THE SIZE OF THE BASE DATA SET.                 */
/*                                                                   */
/* DESCRIPTION: ISSUE BUILT-IN FUNCTION &SYSDSN TO ENSURE THE BASE   */
/*              DATA SET EXISTS. ISSUE LISTDSI STATEMENT TO SET      */
/*              CLIST VARIABLES WITH ATTRIBUTES OF THE BASE DATA     */
/*              SET. DOUBLE THE CONTENTS OF THE PRIMARY SPACE        */
/*              VARIABLE, THEN USE THE VARIABLES AS INPUT TO         */
/*              THE ALLOCATE COMMAND TO ALLOCATE A NEW DATA SET.     */
/*********************************************************************/
PROC 2 BASEDS NEWDS
IF &SYSDSN(&BASEDS) = OK THEN   +
 DO                                 /* If the base data set exists    */
  LISTDSI &BASEDS                   /* Issue LISTDSI statement        */
  NGLOBAL &SYSPRIMARY,&SYSSECONDS   /* Make LISTDSI variables avail-  */
  SET &RC = &LASTCC                 /* able to subprocedures          */
  IF &RC = 0 THEN +
   SYSCALL ALC &BASEDS &NEWDS       /* Call subprocedure ALC          */
  ELSE  +
   DO                               /* If LISTDSI failed              */
    WRITE &SYSMSGLVL1               /* First-level message            */
    WRITE &SYSMSGLVL2               /* Second-level message           */
    WRITE RETURN CODE = &RC         /* Return code                    */
    WRITE REASON CODE = &SYSREASON  /* LISTDSI reason code            */
   END
 END
ELSE  +
 WRITE DATA SET &BASEDS NOT FOUND

ALC: PROC 2 BASE NEW                    /* Subprocedure ALC           */
    SET NEWPRIMARY = 2 * &SYSPRIMARY    /* Compute new primary space  */
    ALLOCATE DA(&NEW) NEW SPACE(&NEWPRIMARY,&SYSSECONDS)  +
      LIKE(&BASE) CATALOG               /* Allocate the new data set  */
    WRITE DATA SET &NEW HAS BEEN ALLOCATED
END
```

Figure 17. The EXPAND CLIST

Chapter 14. Reference

This section describes the syntax and KEYWORD names of the CLIST statements. For information about the two TSO/E commands—EXEC and END—that you use to start and end CLIST execution, see *z/OS TSO/E Command Reference*.

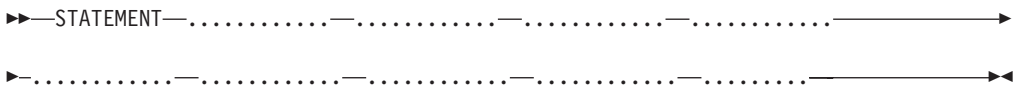
How to read the CLIST statement syntax

Throughout this chapter, syntax is described using the structure defined below.

Read the syntax diagrams from left to right, from top to bottom, following the path of the line.

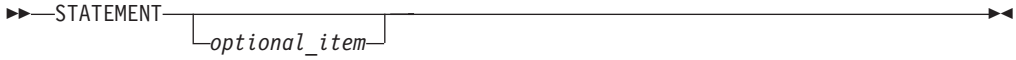
Double arrows indicate the beginning and ending of a statement.

If a statement syntax requires more than one line to be shown, single arrows indicate their continuation.



Required items appear on the horizontal line (the main path).

Optional items appear below the main path.



If you can choose from two or more items, they are stacked vertically.

- If you *must* choose one of the items, an item of the stack appears on the main path.

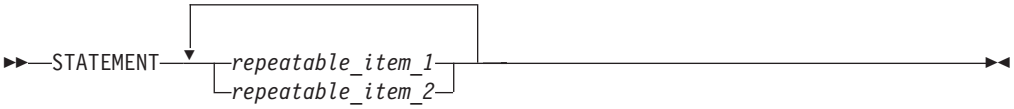
  

- If choosing one of the items is optional, the entire stack appears below the main path.

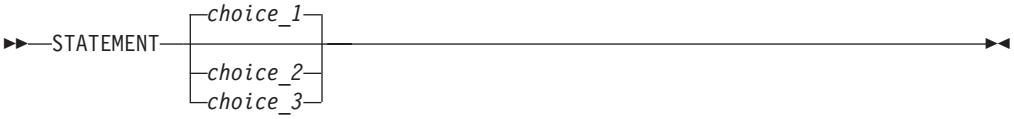
  

An arrow returning to the left above the main line indicates an item that can be repeated.

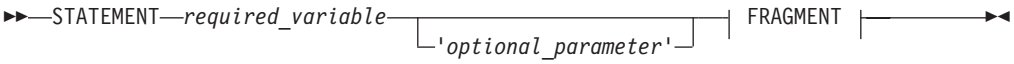
A repeat arrow above a stack indicates that you can make more than one choice from the stacked items, or repeat a single choice.



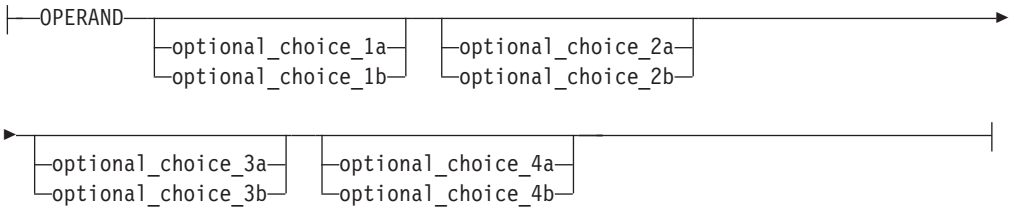
Default values appear above the main path. For example, if you choose neither *choice_2* nor *choice_3*, *choice_1* is assumed. (Defaults *can* be coded for clarity reasons.)



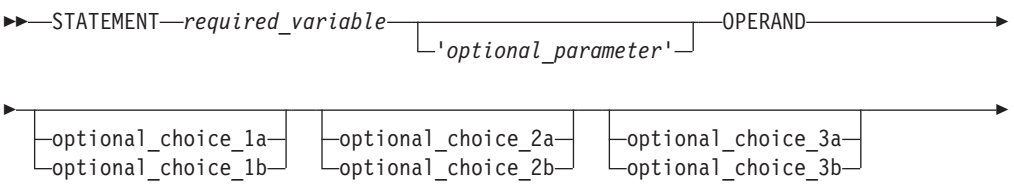
If a syntax diagram becomes too large or too complex to be printed or shown, fragments of it are shown below the main diagram as details.

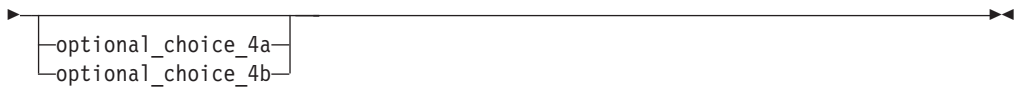


FRAGMENT:



The previous syntax diagram is equivalent to the following diagram:





In a CLIST statement, use uppercase letters, numbers, and the set of symbols listed below exactly as shown in the syntax.

apostrophe or single quote
'

asterisk
*

comma
,

equal sign
=

parentheses
( )

period .

ampersand
&

percent
%

colon :

Lowercase italic letters and symbols appearing in the syntax represent variable information for which you substitute specific information in the statement. For example, if *name* appears in the syntax, substitute a specific value (for example, ALPHA) for the variable when you enter the statement.

Hyphens join lowercase words and symbols to form a single variable. For example, if *member-name* appears in the syntax, substitute a specific value (for example, BETA) for the variable in the statement.

Alphanumeric characters: unless otherwise indicated, an alphanumeric character is one of the following:

Alphabetic:
A-Z

Numeric:
0-9

Special:
$ # @.

CLIST statements may be prefixed with a label consisting of 1-31 alphanumeric characters, beginning with an alphabetic character. The label may appear on a separate line. A colon must immediately follow the label name. For example,

label: +
- IF A= ...

ATTN statement

Use the ATTN statement to define a routine that TSO/E executes when the user causes an attention interrupt. The attention interrupt halts execution of a CLIST so that the user can terminate or alter its processing.

label
A name the CLIST can reference in a GOTO statement to branch to this ATTN statement. *label* is one-to-31 alphanumeric characters, beginning with an alphabetic character.

OFF
Any previous attention action is nullified. Do not use ATTN OFF within an attention routine.

action
specifies either:

1. One TSO/E command, commonly an EXEC command that invokes an attention processing CLIST, or a null (blank) line. An attention processing CLIST can execute multiple TSO/E commands, while the action can execute only one.
2. A DO-END sequence constituting an attention exit routine. This routine can contain CLIST statements, including the RETURN statement or EXIT statement, and one TSO/E command, or a null line.

If a null line is executed, TSO/E ignores the attention and execution continues at the point where the interruption occurred.

If an EXIT statement is executed, the attention is ignored and the CLIST is terminated.

If a TSO/E command is executed, control is given to the command.

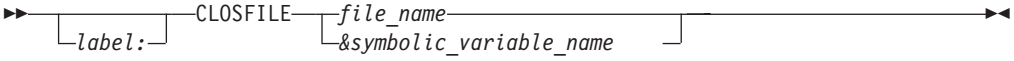
When a TSO/E command, an EXIT statement, or a null line is executed, TSO/E ignores all other.

If the attention routine does anything other than terminate the CLIST, use the MAIN operand of the CONTROL statement to protect the CLIST from being flushed from the input stack when an attention interrupt occurs. For more information, see “CONTROL statement” on page 149.

CLOSFILE statement

Use the CLOSFILE statement to close a QSAM file that has been previously opened by an OPENFILE statement. Only one file can be closed with each CLOSFILE statement.

Note: The CLOSFILE statement must be issued in the same CLIST as the corresponding OPENFILE statement.



label
A name the CLIST can reference in a GOTO statement to branch to this CLOSFILE statement. *label* is one-to-31 alphanumeric characters, beginning with an alphabetic character.

file_name | *symbolic_variable_name*

file_name
is the file name (ddname) assigned to the file (data set) when it was allocated in the current session.

symbolic_variable_name
is the symbolic variable to which you assigned *file_name*.

CONTROL statement

Use the CONTROL statement to define processing options for a CLIST. The options are in effect from the time CONTROL executes until either the CLIST terminates or it issues another CONTROL statement.

You can also set CONTROL options on or off in the following variables:

&SYSPROMPT
ON equals PROMPT, OFF equals NOPROMPT

&SYSSYMLIST
ON equals SYMLIST, OFF equals NOSYMLIST

&SYSCONLIST
ON equals CONLIST, OFF equals NOCONLIST

&SYSLIST
ON equals LIST, OFF equals NOLIST

&SYSASIS
ON equals ASIS, OFF equals CAPS

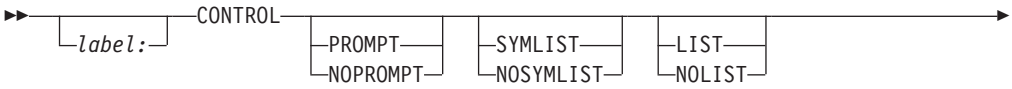
&SYSMSG
ON equals MSG, OFF equals NOMSG

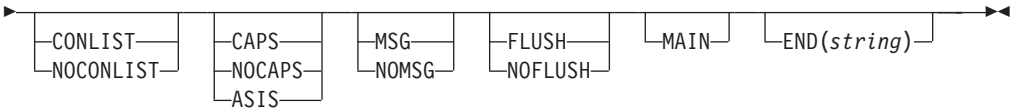
&SYSFLUSH
ON equals FLUSH, OFF equals NOFLUSH.

CLISTs that do not issue CONTROL statements or one of the above variables execute with the following options: NOPROMPT, NOSYMLIST, NOLIST, NOCONLIST, CAPS, MSG, and FLUSH. The user can set PROMPT and LIST by entering them as keywords on the EXEC command or subcommand issued to invoke the CLIST.

CONTROL has no default operands. If you enter CONTROL with no operands, the system uses options already defined by system default, the EXEC command, or a previous CONTROL statement. In addition, when there are no operands specified, the system displays those options currently in effect.

Note: CONTROL operands cannot be entered as symbolic variables.





label
A name the CLIST can reference in a GOTO statement to branch to this CONTROL statement. *label* is one-to-31 alphanumeric characters, beginning with an alphabetic character.

PROMPT |NOPROMPT

PROMPT
TSO/E commands in the CLIST may prompt the terminal for input. (The PROMPT operand on the PROFILE command must also be in effect.)

NOPROMPT
TSO/E commands in the CLIST may not prompt the terminal for input.

SYMLIST | NOSYMLIST

SYMLIST
Each executable statement is displayed at the terminal before it is scanned for symbolic substitution. Executable statements include commands, subcommands, and CLIST statements.

NOSYMLIST
Executable statements are not displayed at the terminal before symbolic substitution.

LIST | NOLIST

LIST
Commands and subcommands are displayed at the terminal after symbolic substitution but before execution.

NOLIST
Commands and subcommands are not displayed at the terminal.

CONLIST | NOCONLIST

CONLIST
CLIST statements are displayed at the terminal after symbolic substitution but before execution.

NOCONLIST
CLIST statements are not displayed at the terminal after symbolic substitution.

CAPS | NOCAPS | ASIS

CAPS
Character strings are converted to uppercase letters before being processed.

NOCAPS or ASIS
Character strings are not converted to uppercase before being processed.

MSG | NOMSG

MSG
Informational messages from commands and statements in the CLIST are displayed at the terminal.

NOMSG
Informational messages from commands and statements in the CLIST are not displayed at the terminal.

FLUSH | NOFLUSH

FLUSH
The system can erase (flush) the queue of nested CLISTs called the input stack unless NOFLUSH or MAIN is encountered. The system normally flushes the stack when an execution error occurs.

NOFLUSH
The system cannot flush the CLIST when an error occurs.

Note: To protect a CLIST from being flushed, the CLIST must contain an error routine.

MAIN
This is the main CLIST in your TSO/E environment and cannot be deleted by a stack flush request from the system. When MAIN is specified, the NOFLUSH condition is assumed for this CLIST, regardless of whether FLUSH was in effect. This operand is required for CLISTs containing attention routines that do anything other than terminate the CLIST.

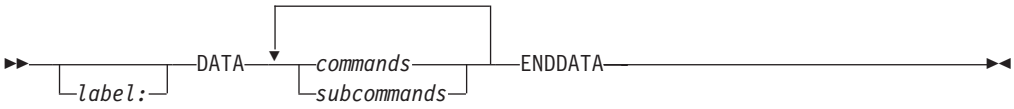
END(*string***)**
A character string recognized by the CLIST as a replacement for an END statement that concludes a DO or SELECT statement, or a subprocedure. *string* is 1-4 alphanumeric characters, beginning with an alphabetic character.

DATA-ENDDATA sequence

Use the DATA-ENDDATA sequence when you do not want a command or subcommand to be interpreted as a CLIST statement. The CLIST views the group of commands and subcommands in the DATA-ENDDATA sequence as data to be ignored and passed to TSO/E for execution.

Do not include CLIST statements in a DATA-ENDDATA sequence because TSO/E attempts to execute them as commands or subcommands.

Symbolic substitution is performed before execution of the group.



label
A name the CLIST can reference in a GOTO statement to branch to this DATA-ENDDATA sequence. *label* is one-to-31 alphanumeric characters, beginning with an alphabetic character.

commands | *subcommands*
The data to be ignored and passed to TSO/E for execution.

DATA PROMPT-ENDDATA sequence

Use the DATA PROMPT-ENDDATA sequence to designate responses to prompts by TSO/E commands or subcommands. An error condition (error code 968) occurs unless the sequence is immediately preceded by a command or subcommand issuing a prompt.

Note: When using the DATA PROMPT-ENDDATA sequence, the following rules apply:
- The CLIST must allow prompting.
- Symbolic substitution is performed before a reply is sent.

DO statement

Use the DO statement to execute sequences of commands, subcommands, and statements (DO-sequences). You can use the DO statement to execute DO-sequences once, repeatedly, and when certain conditions are true.

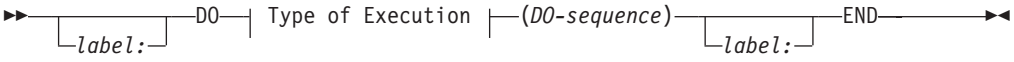
To execute a DO-sequence *once*, include only the DO and END statements.

To execute a DO-sequence *repeatedly*, include a variable with a starting value, a TO value, and, optionally, a BY value.

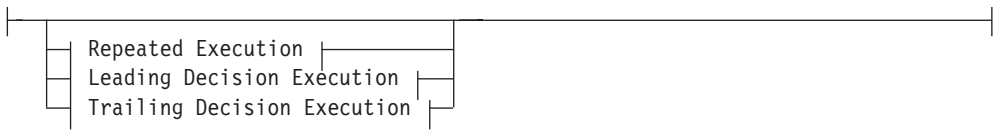
To execute a DO-sequence *conditionally*, include a WHILE or UNTIL clause. The WHILE clause contains a leading decision and executes while a comparative expression is true, and the UNTIL clause contains a trailing decision and executes until a comparative expression is true.

To execute a DO-sequence repeatedly *and* conditionally (compound DO), the WHILE, UNTIL, or both clauses must follow the from, TO, and optional BY clauses.

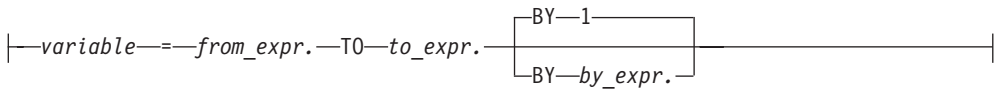
The DO statement indicates the beginning of a DO-sequence. The END statement concludes the DO-sequence. If you want to use the TSO/E END command in a DO-sequence, you must redefine the END statement, using the END operand of the CONTROL statement.



Type of Execution



Repeated Execution



Leading Decision Execution

Trailing Decision Execution

label
: A name the CLIST can reference in a GOTO statement to branch to this DO statement. *label* is one-to-31 alphanumeric characters, beginning with an alphabetic character.

variable
: A symbolic variable that controls execution of the DO-sequence. With each execution, the variable value increases or decreases by a certain amount. When the value passes a certain limit, the CLIST stops executing the DO-sequence and executes the next instruction after the END statement.

from_expression
: A decimal integer, or an expression that evaluates to a decimal integer, forming the initial value of the DO variable.

to_expression
: A decimal integer, or an expression that evaluates to a decimal integer, forming the terminal value of the DO variable.

by_expression
: A decimal integer, or an expression that evaluates to a decimal integer, by which the DO variable increases or decreases each time the DO-sequence executes.

condition
: A comparative expression or a sequence of comparative expressions sequenced by logical operators. The expression or expressions can include character data, including characters of the double-byte character set.

In the absence of a BY clause, the value of the DO variable increases by 1 with each execution of the DO sequence.

If the *by-expression* evaluates to a negative number or consists of a number beginning with a minus sign, the DO variable decreases by that amount.

If the statements in a DO-sequence modify a DO variable, the CLIST uses the new value in determining whether to repeat the DO-sequence.

DO-sequences can contain nested DO statements.

END statement

Use the END statement to mark the end of a DO-sequence, a SELECT statement, or a subprocedure. The END statement must appear on a line by itself following the DO-sequence, SELECT statement, or subprocedure.

The END statement is distinct from the TSO/E END command. If you use both the END statement and END command in a CLIST, you must distinguish them by redefining the END statement. Using the CONTROL statement, you can redefine the END statement as follows:

```
CONTROL END(string)
```

where *string* is 1-4 alphanumeric characters, beginning with an alphabetic character. You then use the string in place of END statements in the CLIST.

ERROR statement

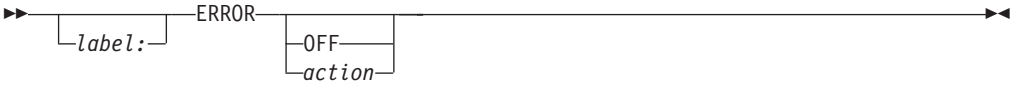
Use the ERROR statement to set up an environment that checks for non-zero return codes from commands, subcommands, and CLIST statements in the currently executing CLIST. When an error code is detected, processing continues at the ERROR routine active for the command, subcommand, or CLIST statement that registered the error. If an ERROR routine is not active, the CLIST either terminates or continues, depending on the severity of the error.

The error exit must be protected from being flushed from the input stack by the system. Stack flushing makes the error return codes unavailable. Use the MAIN or NOFLUSH operands of the CONTROL statement to prevent stack flushing.

When ERROR is entered with no operands, the CLIST displays the command, subcommand, or statement in the CLIST that ended in error. No explanatory CLIST ERROR messages are displayed. &LASTCC is reset to 0 and the CLIST continues with the next sequential statement or command.

If the LIST option was requested for the CLIST, the null ERROR statement is ignored.

The ERROR statement must precede any statements that might cause a branch to it.



label
A name the CLIST can reference in a GOTO statement to branch to this ERROR statement. *label* is one-to-31 alphanumeric characters, beginning with an alphabetic character.

OFF | *action*

OFF
Any action previously set up by an ERROR statement is nullified.

action
Any executable statement, commonly a DO-sequence constituting a routine. The action can execute TSO/E commands, subcommands, and CLIST statements.

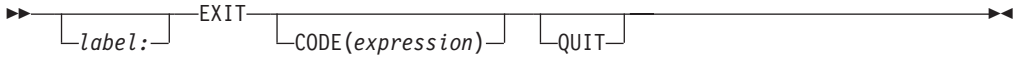
Note: Coding ERROR OFF within the DO-sequence routine itself prevents the routine from returning control to the CLIST.

EXIT statement

Use the EXIT statement to return control to the program that called the currently executing CLIST. The return code associated with this exit can be specified by the user or allowed to default to 0.

A CLIST that is called by another CLIST is said to be nested. Multiple levels of nesting are allowed. The structure of the nesting is called the hierarchy. You go "up" in the hierarchy when control passes back to the calling CLIST. TSO/E itself is at the top of the hierarchy.

Entering EXIT causes control to go up one level. When EXIT is entered with the QUIT operand, the system attempts to pass control upward to the first CLIST encountered that has MAIN or NOFLUSH in effect (see "CONTROL statement" on page 149). If no such CLIST is found, control passes to TSO/E, which flushes all CLISTs from the input stack and passes control to the terminal.



label
A name the CLIST can reference in a GOTO statement to branch to this EXIT statement. *label* is one-to-31 alphanumeric characters, beginning with an alphabetic character.

CODE (*expression***)**
A CLIST-defined return code. *expression* must be a positive integer, zero, or an expression that evaluates to a decimal integer. When CODE is not specified, the system uses 0 as the default return code.

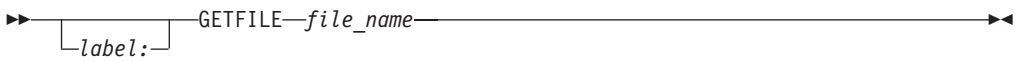
QUIT
Control is passed up the nested hierarchy until either a CLIST is found with the MAIN or NOFLUSH option active or TSO/E receives control.

GETFILE statement

Use the GETFILE statement to read a record from a QSAM file opened by the OPENFILE statement. One record is obtained by each execution of GETFILE.

After GETFILE executes, the file name variable contains the record obtained. If you use GETFILE to read data from the terminal, the data is translated to uppercase.

Note: The GETFILE statement must be issued in the same CLIST as the corresponding OPENFILE statement.



label
A name the CLIST can reference in a GOTO statement to branch to this GETFILE statement. *label* is one-to-31 alphanumeric characters, beginning with an alphabetic character.

file_name
The file name (ddname) assigned to the file (data set) when it was allocated in the current session. Do not specify a symbolic variable containing the file name.

GLOBAL statement

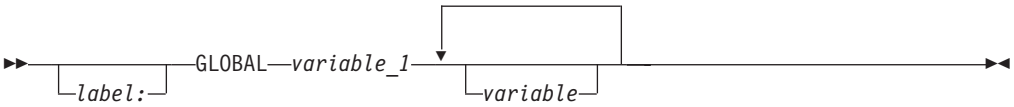
Use the GLOBAL statement to share values between nested CLISTs. In the hierarchy of nested CLISTs, the highest-level CLIST must contain a GLOBAL statement with the maximum number of variables used throughout the nested chain. Lower-level CLISTs must include a GLOBAL statement if they intend to refer to the global variables defined in the highest-level CLIST.

Note: The GLOBAL statement cannot be used to give a REXX exec access to a CLIST's global variables. CLIST variables cannot be accessed by REXX execs.

The global variables are positional, and the order is defined by the GLOBAL statement in the highest-level CLIST. All lower-level CLISTs that reference this same set of variables must follow this order to reference the same values. The variable names may be unique to the lower-level CLISTs. This means that the Nth name on any level GLOBAL statement refers to the same value, even though the symbolic name at each level may be different. For example, if a nested CLIST references the fifth global variable, then it must define five global variables. If it references the second global variable, then it needs to define only two global variables.

Multiple GLOBAL statements are cumulative. For example, if a CLIST has a GLOBAL statement that defines three variables followed by another GLOBAL statement that defines two variables, then five variables have been defined. The second GLOBAL statement defines the fourth and fifth variables.

The GLOBAL statement must precede any statement that uses or defines its variables.

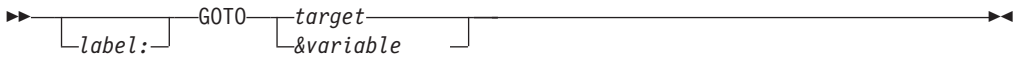


label
A name the CLIST can reference in a GOTO statement to branch to this GLOBAL statement. *label* is one-to-31 alphanumeric characters, beginning with an alphabetic character.

variable_1 / variable
A symbolic variable name for this CLIST. The name refers to a variable that is either being created by this GLOBAL statement or that was created by a GLOBAL statement in the highest-level CLIST.

GOTO statement

Use the GOTO statement to cause an unconditional branch within a CLIST. Branching to another CLIST is not allowed.



label
A name the CLIST can reference in a GOTO statement to branch to this GOTO statement. *label* is one-to-31 alphanumeric characters, beginning with an alphabetic character.

target | *variable*

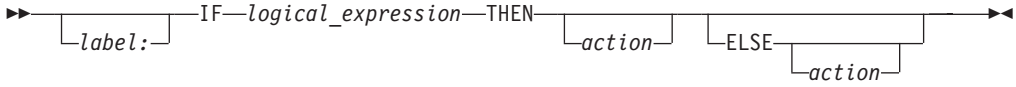
target
A label on a statement or command.

variable
A symbolic variable that contains a valid label.

GOTO statements cannot branch:
- To another CLIST
- To a subprocedure's PROC statement
- From one subprocedure to another
- From a subprocedure to the CLIST's main procedure

IF-THEN-ELSE sequence

Use the IF-THEN-ELSE sequence to define a condition, test the truth of that condition, and initiate an action based on the test results. Do not code THEN and ELSE on the same logical line.



label
A name the CLIST can reference in a GOTO statement to branch to this IF-THEN-ELSE sequence. *label* is one-to-31 alphanumeric characters, beginning with an alphabetic character.

logical_expression
A comparative expression or a sequence of comparative expressions sequenced by logical operators. The expression or expressions can include character data, including characters of the double-byte character set.

action
An executable command, subcommand, or CLIST statements. (Enclose an action consisting of more than one statement in a DO-sequence.) The THEN action is invoked if the logical expression is true. The ELSE action is invoked if the logical expression is false. If a null THEN or null ELSE statement is executed, control passes to the next sequential statement after the IF-THEN-ELSE sequence.

The action must be on the same line as a THEN or ELSE clause, or be joined to the line by a continuation character. For example, the following are correct:

```
IF &FOOTPRINT = 0 THEN SET ECODE = 4

IF &FOOTPRINT = 0 THEN +
  SET ECODE = 4

IF &FOOTPRINT = 0 THEN +
  DO
    SET ECODE = 4
⋮
  END
```

LISTDSI statement

Use the LISTDSI statement to obtain information about a data set that is available on DASD. The LISTDSI statement can retrieve information about a data set's allocation, protection, and directory, and store the information in CLIST variables.

The LISTDSI statement supports generation data group (GDG) data sets, but it does not support the following types of data or data sets:

- data that is on tape
- relative GDG names
- UNIX file system data sets

Otherwise, unpredictable results might occur.

The CLIST can use the LISTDSI information to determine whether the data set is the right size or has the right organization or format for a given task. It can also use the LISTDSI information as input to the ALLOCATE command, to create a new data set using some attributes from the old data set while modifying others.

If you use LISTDSI to retrieve information about a VSAM data set, the CLIST stores only the volume serial ID (in variable &SYSVOLUME), the generic device type (in variable &SYSUNIT), and the data set organization (in variable &SYSDSORG). The CLIST sets all other LISTDSI variables to question marks.

If you use LISTDSI to retrieve information about a multiple volume data set, the CLIST stores information for the first volume only. Similarly, if you specify a file name or the PREALLOC parameter and you have other data sets allocated to the same file name, then the system might not retrieve information for the data set you wanted.

When you use LISTDSI to obtain information about a concatenation of more than one data set, LISTDSI returns information only about the first data set in the concatenation. Likewise, if the file name identifies a multi-volume data set, LISTDSI can return information only about the first volume, and is not able to detect that the data set is multi-volume.

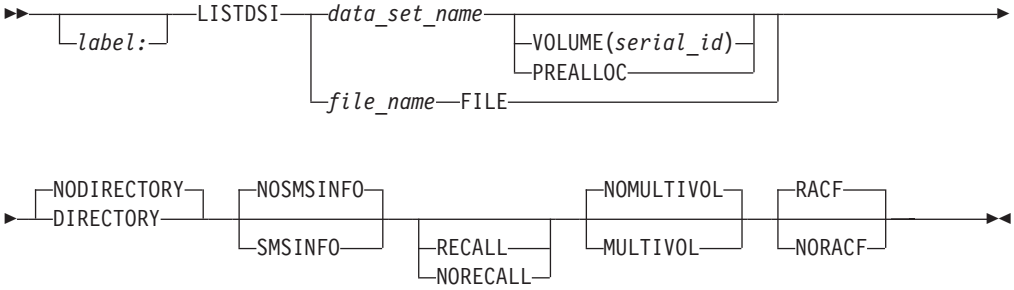
If the data set is SMS managed and is capable of expanding to multiple volumes, but has not yet done so, it is considered a single volume data set by LISTDSI until it has expanded to the second volume. In any case, LISTDSI will only retrieve information for the first volume referenced by the request.

As part of its processing, LISTDSI issues a RACF authority check against the provided data set which will cause a security violation to occur if the user does not have at least READ access to the data set. RACF does not issue an ICH408I message due to message suppression requested by LISTDSI, and therefore LISTDSI

issues a return code of 0. The only indication a security violation has occurred is that an SMF type-80 RACF audit record is created.

LISTDSI considers file names in the form SYSnnnnn as system-generated file names. If LISTDSI is used to obtain information about a data set that was pre-allocated multiple times using a file name of the form SYSnnnnn, an existing file may be unintentionally freed.

To suppress TSO/E messages issued by the LISTDSI statement, use the CONTROL NOMSG statement. For information about the CONTROL statement, see "CONTROL statement" on page 149.



label
A name the CLIST can reference in a GOTO statement to branch to this LISTDSI statement. *label* is one-to-31 alphanumeric characters, beginning with an alphabetic character.

data_set_name | *file_name*

data_set_name
The name of the data set about which you want to retrieve information.

file_name
The name of an allocated file (ddname) about which you want to retrieve information.

VOLUME(*serial_id***)** | **PREALLOC**

VOLUME(*serial_id***)**
specifies the serial number of the volume where the data set is located.

PREALLOC
specifies that the location of the specified data set is determined by allocating the data set, rather than through a catalog search. PREALLOC allows data sets that have been previously allocated to be located without searching a catalog and allows unmounted volumes to be mounted.

If you do not specify either VOLUME or PREALLOC, the system locates the data set through catalog search.

If you specify a *file_name*, LISTDSI ignores the VOLUME and PREALLOC parameters.

FILE
specifies that you provided a *file_name* instead of a *data_set_name*. If you do not specify FILE, LISTDSI assumes that you provided a data set name.

DIRECTORY | **NODIRECTORY**

DIRECTORY
indicates that you want directory information for a partitioned data set (PDS or PDSE).

Requesting DIRECTORY information for a PDS may cause the date last referenced (&SYSREFDATE) to be updated by LISTDSI. Refer to the description of the &SYSREFDATE variable for more information about when &SYSREFDATE might be updated by LISTDSI.

NODIRECTORY
indicates that you do not want directory information for a partitioned data set. If you do not require directory information, NODIRECTORY can significantly speed up processing. NODIRECTORY is the default.

SMSINFO | NOSMSINFO
indicates whether you want SMS information about an SMS-managed data set, like the type of data set, the used space, the data -, storage -, and management class names. See also Table 11 on page 161.

SMSINFO
indicates that you want SMS information about *data_set_name* or *file_name*. Neither *data_set_name* nor *file_name* may refer to a VSAM index or a data component.

If the specified data set is not managed by SMS, LISTDSI continues, but no SMS information is provided in the corresponding CLIST variables.

Specify SMSINFO only if you want SMS information about a data set. NOSMSINFO (the default) may significantly reduce the execution time of the LISTDSI statement.

Requesting SMSINFO for a PDSE data set may cause the date last referenced (&SYSREFDATE) to be updated by LISTDSI. Refer to the description of the &SYSREFDATE variable for more information about when &SYSREFDATE might be updated by LISTDSI.

NOSMSINFO
indicates that you do not want SMS information about the specified data set. NOSMSINFO is the default.

RECALL | NORECALL

RECALL
indicates that you want to recall a data set migrated by HSM. The system recalls the data set regardless of its level of migration or the type of device it has been migrated to.

NORECALL
indicates that you do not want to recall a data set. If the data set has been migrated, the system displays an error message.

If you do not specify either RECALL or NORECALL, the system recalls the data set only if it has been migrated to a direct access storage device (DASD).

MULTIVOL | NOMULTIVOL
indicates whether data size calculations should include all volumes, when a data set resides on more than one volume. The new SYSNUMVOLS and SYSVOLUMES variables are not affected by this operand, as these are always set.

If the VOLUME keyword and the MULTIVOL keyword are both specified, the MULTIVOL keyword is ignored. In this case, data set size information is returned just for the specified volume.

RACF | NORACF
indicates whether a check for RACF authority is done or not. If not, the data set will not be opened by LISTDSI, for example, to read directory information.

The LISTDSI function issues message IKJ56709I if a syntactically incorrect data set name is passed to the function. To prevent this message from being displayed, use CONTROL NOMSG.

```
PROC 0
SET DSNAME = ABCDEFGHIJ.XYZ       /* Syntactically not valid name,
                                  /* because a qualifier is longer
                                  /* than 8 characters
CONTROL NOMSG                     /* Set OFF to suppress any LISTDSI
                                  /* TSO/E messages
LISTDSI &DSNAME                   /* Obtain data set information
WRITE Return code from LISTDSI is ==> &LASTCC
EXIT CODE(0)
```

CLIST variables set by LISTDSI

Table 11 describes the contents of the CLIST variables set by LISTDSI. For VSAM data sets, only the variables &SYSDSNAME, &SYSEATTR, &SYSEADSCB, &SYSVOLUME, &SYSUNIT, and &SYSDSORG are accurate; all other variables are set to question marks.

Table 11. Variables set by LISTDSI

Variable	Contents
&SYSDSNAME	Data set name
&SYSVOLUME	Volume serial ID
&SYSUNIT	Generic device type on which volume resides, for example, “3390”.
&SYSDSORG	Data set organization: **PS** Physical sequential **PSU** Physical sequential unmovable **DA** Direct organization **DAU** Direct organization unmovable **IS** Indexed sequential **ISU** Indexed sequential unmovable **PO** Partitioned organization **POU** Partitioned organization unmovable **VS** VSAM **???** Unknown

Table 11. Variables set by LISTDSI (continued)

Variable	Contents
&SYSRECFM	Record format; 1- to 6-character combination of the following: **U** Records of undefined length **F** Records of fixed length **V** Records of variable length **T** Records written with the track overflow feature of the device (no currently supported device supports the track overflow feature) **B** Records blocked **S** Records written as standard or spanned variable-length blocks **A** Records contain ANSI control characters **M** Records contain machine code control characters **??????** Unknown
&SYSLRECL	Logical record length
&SYSBLKSIZE	Block size
&SYSKEYLEN	Key length
&SYSALLOC	Allocation, in space units
&SYSUSED	Allocation used, in space units. For a partitioned data set extended (PDSE) "N/A" will be returned; see the description of the &SYSUSEDPAGES for used space of a PDSE.
&SYSUSEDEXTENTS	Indicates the number of extents used. For a partitioned data set extended (PDSE), this variable returns 'N/A'. See the descriptions of variables SYSUSEDPAGES and SYSUSEDPERCENT for more information about used space of a PDSE.
&SYSUSEDPAGES	The used space of a partitioned data set extended (PDSE) in 4K pages.
&SYSPRIMARY	Primary allocation in space units
&SYSSECONDS	Secondary allocation in space units
&SYSUNITS	Space units: **CYLINDER** Space units in cylinders **TRACK** Space units in tracks **BLOCK** Space units in blocks **????????** Space units are unknown
&SYSEXTENTS	Number of extents allocated

Table 11. Variables set by LISTDSI (continued)

Variable	Contents
&SYSUSEDEXTENTS	Indicates the number of extents used. For a partitioned data set extended (PDSE), this variable returns 'N/A'; see the descriptions of variables SYSUSEDPAGES and SYSUSEDPERCENT for more information about used space of a PDSE.
&SYSCREATE	Creation date in `Year/day` format, for example: `1985/102`.
&SYSREFDATE	Last referenced date in `Year/day` format, for example: `2010/107`. Specifying DIRECTORY or SMSINFO may cause the last referenced date to be updated to the current date under the following circumstances: • Specifying DIRECTORY causes the date to be updated only if the data set is a PDS and the user running LISTDSI has RACF READ authority to the data set. In all other cases, including when the data set is a PDSE, DIRECTORY has no effect on this date. • Specifying SMSINFO causes the date to be updated only if the data set is a PDSE and the user running LISTDSI has RACF READ authority to the data set. In all other cases, SMSINFO has no effect on this date.
&SYSEXDATE	Expiration date in `Year/day` format, for example: `1995/365`.
&SYSPASSWORD	Password indication: **NONE** No password protection **READ** Password required to read **WRITE** Password required to write
&SYSRACFA	RACF indication: **NONE** No RACF protection **GENERIC** Generic profile covers this data set **DISCRETE** Discrete profile covers this data set
&SYSUPDATED	Backup change indicator: **YES** Data set has been updated since its last backup by DFSMShsm (or its equivalent). **NO** Data set has not been updated since its last backup.
&SYSTRKSCYL	Tracks per cylinder for the unit identified in the &SYSUNIT variable
&SYSBLKSTRK	Blocks of &SYSBLKSIZE per track for the unit identified in the &SYSUNIT variable. For a PDSE, the value “N/A” is returned because a block of size &SYSBLKSIZE can 'span' a track in a PDSE. The value contained in &SYSUSEDPAGES is a more meaningful measurement of space usage for a PDSE.

Table 11. Variables set by LISTDSI (continued)

Variable	Contents
&SYSADIRBLK	For a partitioned data set (PDS), the number of directory blocks allocated will be returned. For a partitioned data set extended (PDSE), "NO_LIM" will be returned because there is no static allocation for its directory. A value is returned only if DIRECTORY is specified on the LISTDSI statement.
&SYSUDIRBLK	For a partitioned data set (PDS), the number of directory blocks used will be returned. For a partitioned data set extended (PDSE), "N/A" will be returned because it is not a static value. A value is returned only if DIRECTORY is specified on the LISTDSI statement.
&SYSMEMBERS	Number of members - returned only for partitioned data sets when DIRECTORY is specified
&LASTCC	LISTDSI return code
&SYSREASON	LISTDSI reason code
&SYSMSGLVL1	First-level message if an error occurred
&SYSMSGLVL2	Second-level message if an error occurred
&SYSDSSMS	Contains information about the type of a data set, provided by DFSMS/MVS. If the SMS data set type information could not be retrieved, the SYSDSSMS variable contains: **SEQ** for a sequential data set **PDS** for a partitioned data set **PDSE** for a partitioned data set extended If the data set is a PDSE and the and the SMSINFO operand was specified on the LISTDSI call and SMS data set type information could be retrieved, the SYSDSSMS variable contains: **LIBRARY** for an empty PDSE **PROGRAM_LIBRARY** for a partitioned data set extended program library **DATA_LIBRARY** for a partitioned data set extended data library **Note:** This detailed data set type information for a PDSE is not returned if the user issuing the LISTDSI call does not have RACF READ authority to the data set.
&SYSDATACLASS([1])	The SMS data class name - returned only if SMSINFO is specified on the LISTDSI statement and the data set is managed by SMS.
&SYSSTORCLASS([1])	The SMS storage class name - returned only if SMSINFO is specified on the LISTDSI statement and the data set is managed by SMS.
&SYSMGMTCLASS([1])	The SMS management class name - returned only if SMSINFO is specified on the LISTDSI statement and the data set is managed by SMS.

Table 11. Variables set by LISTDSI (continued)

Variable	Contents
&SYSSEQDSNTYPE	Indicates the type of sequential data set: BASIC - regular sequential data set LARGE - large sequential data set EXTENDED - an extended sequential data set If the data set is not sequential, this variable returns a null string.
&SYSEATTR	Indicates the current status of the EATTR bits in the DSCB that describe the EAS eligibility status of a data set. A data set can reside in the EAS only when it is EAS-eligible. Default blank indicates that the EATTR bits are '00'b. The defaults for EAS eligibility apply: • VSAM data sets are EAS-eligible, and can have extended attributes (format 8 and 9 DSCBs). • Non-VSAM data sets are not EAS-eligible, and cannot have extended attributes (format 8 and 9 DSCBs). **NO** Indicates that '01'b is specified for the EATTR bits. The data set is not EAS-eligible, and cannot have extended attributes (format 8 and 9 DSCBs). **OPT** Indicates that '10'b is specified for the EATTR bits. The data set is ESA-eligible, and can have extended attributes (format 8 and 9 DSCBs).
&SYSEADSCB	Indicates whether the data set has extended attributes: **YES** The data set has extended attributes (format 8 and 9 DSCBs) and can reside in the EAS. **NO** The data set does not have extended attributes (format 8 and 9 DSCBs) and can not reside in the EAS.
&SYSALLOCPAGES	Indicates number of pages allocated to a PDSE.
&SYSUSEDPERCENT	Indicates percentage of pages used out of pages allocated for a PDSE. This is a number from 0 to 100, rounded down to the nearest integer value.
&SYSNUMVOLS	Indicates a number from 1 to 59 as tape is not supported but will always return 1 if a volume is specified, instead of trying LOCATE all volumes for the data set.
&SYSVOLUMES	Indicates up to 412 characters with a list of volumes separated by spaces where the first six characters will always match SYSVOLUME and each volume name takes up six spaces padded with blanks to help simplify parsing. If a volume is specified on the LISTDSI call, just that volume name is returned.

Table 11. Variables set by LISTDSI (continued)

Variable	Contents
&SYSCREATETIME	Indicates the time a data set was created in the format *hh:mm:ss* where *hh* is hours since midnight, *mm* is minutes since midnight, and *ss* is seconds since midnight. This variable is only set for EAV data sets and can be used together with the SYSCREATE variable to determine the date and time when a data set was created.
&SYSCREATESTEP	Indicates the name of the job step that created the data set.
&SYSCREATEJOB	Indicates the name of the job that created the data set.
Note: For data sets not managed by SMS, these variables return a null string.	

Return codes

Return codes from the LISTDSI statement appear in CLIST variable &LASTCC. Error routines do not receive control when a CLIST receives a non-zero return code from LISTDSI. Table 12 lists the LISTDSI return codes and their meanings.

Table 12. LISTDSI return codes

Return code	Meaning
0	Processing successful
4	Some data set information is unavailable. Review the reason code in the returned variable &SYSREASON and check the messages returned in &SYSMSGLVL1 and &SYSMSGLVL2 to determine which information is unavailable.
16	Processing unsuccessful. None of the CLIST variables can be considered valid.

Reason codes

Reason codes from the LISTDSI statement appear in CLIST variable &SYSREASON. Table 13 lists the LISTDSI reason codes and their meanings. With each reason code the CLIST variable &SYSMSGLVL2 is set to message IKJ584*nn*I, where *nn* is the reason code. These messages are described in *z/OS TSO/E Messages*.

Table 13. LISTDSI reason codes

Reason code	Meaning
0	Normal completion
1	Error parsing the statement.
2	Dynamic allocation processing error (SVC 99 error).
3	The data set is a type that cannot be processed.
4	Error determining UNIT name (IEFEB4UV error).
5	Data set not cataloged (LOCATE macro error).
6	Error obtaining the data set attributes (OBTAIN macro error).
7	Error finding device type (DEVTYPE macro error).
8	The data set does not reside on a direct access device.
9	DFSMShsm migrated the data set, NORECALL prevents retrieval.

Table 13. LISTDSI reason codes (continued)

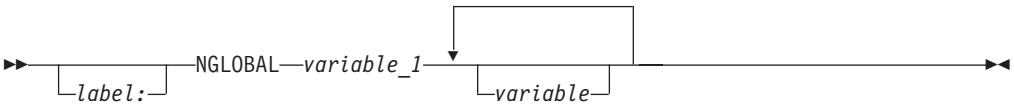
Reason code	Meaning
11	Directory information was requested, but you lack authority to access the data set.
12	VSAM data sets are not supported.
13	The data set can not be opened.
14	Device type not found in unit control block (UCB) tables.
17	System or user abend occurred.
18	Partial data set information was obtained.
19	Data set resides on multiple volumes. Consider using the MULTIVOL keyword to obtain data set size information totaled across all volumes.
20	Device type not found in eligible device table (EDT).
21	Catalog error trying to locate the data set.
22	Volume not mounted (OBTAIN macro error).
23	Permanent I/O error on volume (OBTAIN macro error).
24	Data set not found by OBTAIN macro.
25	Data set migrated to non-DASD device.
26	Data set on MSS (Mass Storage) device.
27	No volume serial is allocated to the data set.
28	ddname must be one to eight characters.
29	Data set name or ddname must be specified.
30	Data set is not SMS-managed.
31	ISITMGD macro returned with bad return code and reason code. Return code and reason code can be found in message IKJ58431I, which is returned in variable &SYSMSGLVL2.
32	Unable to retrieve SMS information. SMS has incorrect level.
33	Unable to retrieve SMS information. SMS is not active.
34	Unable to retrieve SMS information. OPEN error.
35	Unexpected error from DFSMSdfp internal service IGWFAMS.
36	Unexpected error from the SMS service module.
37	Unexpected error from DFSMSdfp service IGGCSI00.

NGLOBAL statement

Use the NGLOBAL statement to share values between subprocedures in a CLIST.

The NGLOBAL (named global) statement defines variables by name. When you define an NGLOBAL variable, other subprocedures in the same CLIST can refer to it by name and modify its value. Other CLISTs cannot access or modify an NGLOBAL variable.

There is no limit to the number of variables that can be defined on an NGLOBAL statement. The NGLOBAL statement must precede any statement that uses its variables.



label
A name the CLIST can reference in a GOTO statement to branch to this NGLOBAL statement. *label* is one-to-31 alphanumeric characters, beginning with an alphabetic character.

variable_1 **|** *variable*
A symbolic variable name for this CLIST. The name refers to a variable that is being defined by this NGLOBAL statement.

Note: Variables named on an NGLOBAL statement cannot appear on a PROC statement.

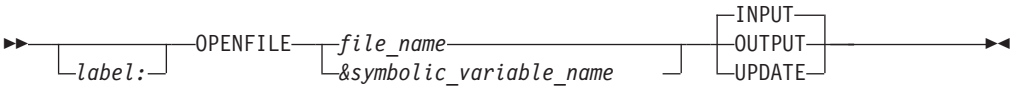
OPENFILE statement

Use the OPENFILE statement to open a QSAM file for I/O. The file must have been allocated during the session and assigned a file name. Each execution of OPENFILE can open only one file, and files cannot be open for different members of the same PDS at the same time. The files must represent data sets with logical record lengths no greater than 32767 bytes.

Note: The OPENFILE statement sets any I/O variables to nulls. Always execute the OPENFILE statement before using any SET statements to create I/O records.

Complete your file I/O on a specific file before changing from command to subcommand mode and vice versa. Cross-mode file I/O is not supported and causes unpredictable abnormal terminations.

Specify NOFLUSH for a CLIST that uses file I/O. (See the CONTROL statement.) If a system action causes TSO/E to flush the input stack because you did not specify NOFLUSH, a user may have to log off the system to recover. The user will recognize the condition by receiving a message similar to “FILE NOT FREED, DATA SET IS OPEN.”.



label
A name the CLIST can reference in a GOTO statement to branch to this OPENFILE statement. *label* is one-to-31 alphanumeric characters, beginning with an alphabetic character.

file_name **|** *symbolic_variable_name*

file_name
The file name (ddname) you assigned to the file (data set) when allocating it in the current session.

symbolic_variable_name
The symbolic variable to which you assigned *file_name*.

INPUT | OUTPUT | UPDATE

INPUT
Open the file for input.

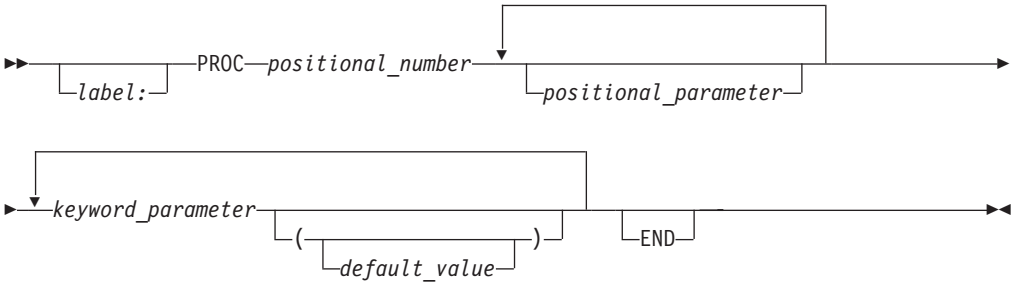
OUTPUT
Open the file for output.

UPDATE
Open the file for updating in place; that is, the CLIST can execute GETFILE and PUTFILE statements before closing the file.

PROC statement

Use the PROC statement to:

- Define parameters to be passed on the EXEC command to the CLIST. In this case PROC is optional, but if you use it, it must be the first executable statement in the CLIST.
- Define a subprocedure and any parameters passed on the SYSCALL statement to the subprocedure. A subprocedure must begin with a PROC statement. In this case the PROC statement must have a label, and a corresponding END statement to mark the end of the subprocedure.



label
A name the CLIST can reference in a SYSCALL statement to pass control to this PROC statement. Required on PROC statements that begin subprocedures, *label* is one-to-31 alphanumeric characters, beginning with an alphabetic character.

positional_number
The number of required positional parameters to be passed. Enter 1-5 decimal digits. If none, enter 0.

positional_parameter
A positional parameter passed to the CLIST or subprocedure.

A positional parameter name may be 1-252 alphanumeric characters in length, beginning with an alphabetic character. Lowercase values are changed to uppercase.

If the name of a positional parameter on the PROC statement is the same as the name of a GLOBAL variable, an error occurs.

keyword_parameter
A keyword parameter passed to the CLIST or subprocedure.

A keyword parameter name can be 1-31 alphanumeric characters in length, beginning with an alphabetic character, and cannot contain the character underscore (_). Lowercase values are changed to uppercase.

default_value
The value assigned to the corresponding variable in the CLIST or subprocedure if the user does not specify a value on the associated keyword on the EXEC command or SYSCALL statement.

If the value is omitted (empty parentheses) the user may supply a value on the associated keyword on the EXEC command or SYSCALL statement.

Note: Symbolic substitution does not occur for default values of a keyword parameter.

All parameters have an initial value at the time the CLIST or subprocedure begins execution. Each parameter name becomes the name of a symbolic variable that has the initial value of the associated parameter. The values of passed parameters are in effect only while the CLIST or subprocedure is active. Values passed in lowercase are converted to uppercase by the exec command.

PUTFILE statement

Use the PUTFILE statement to write a record to an open QSAM file. Each execution of PUTFILE writes one record. Unless the user wants the same record sent more than once, the file name variable must be assigned a different record using an assignment statement before the next PUTFILE statement is issued.

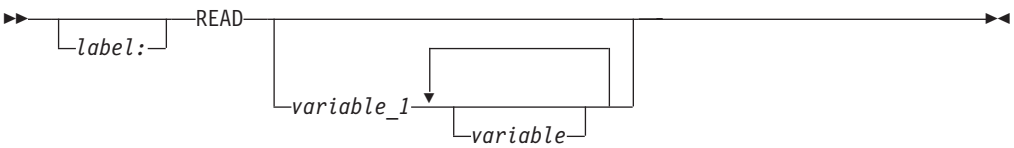
Note: The PUTFILE statement must be issued in the same CLIST as the corresponding OPENFILE statement.

label
A name the CLIST can reference in a GOTO statement to branch to this PUTFILE statement. *label* is one-to-31 alphanumeric characters, beginning with an alphabetic character.

file_name
The file name (ddname) assigned to the file (data set) when it was allocated in the current session. Do not specify a symbolic variable containing the file name.

READ statement

Use the READ statement to read input from the terminal and store it in symbolic variables. These variables may be defined on the READ statement or elsewhere in the CLIST. The READ statement is typically preceded by a WRITE or WRITENR statement that requests the user to enter the expected input at the terminal.



label
A name the CLIST can reference in a GOTO statement to branch to this READ statement. *label* is one-to-31 alphanumeric characters, beginning with an alphabetic character.

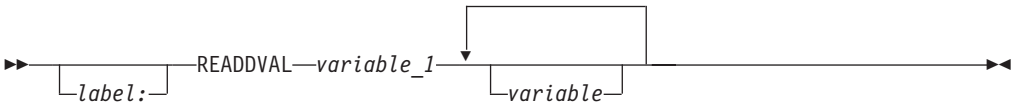
variable_1 / variable
Any valid variable name. The variables are positional in that values in the input data entered by the terminal user are stored sequentially into the specified variables.

If the operand is omitted the input is stored in the &SYSDVAL control variable.

READDVAL statement

Use the READDVAL statement to assign the current contents of the &SYSDVAL control variable to one or more specified symbolic variables.

The assignment is done sequentially to the variables in the order specified; variables not assigned values default to null values. If there are more values than variables, the excess values from &SYSDVAL are not assigned.



label
provides a name the CLIST can reference in a GOTO statement to branch to this READDVAL statement. *label* is one-to-31 alphanumeric characters, beginning with an alphabetic character.

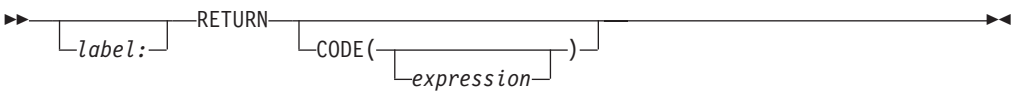
variable_1 / variable
Any valid variable name. A variable need not have been previously defined.

RETURN statement

Use the RETURN statement to:

- Return control from an error routine or an attention routine to the statement following the one that ended in error or the one that was interrupted by an attention.
- Provide a return code from a subprocedure. Control will pass to the statement following the SYSCALL statement that called the subprocedure. The return code is stored in the control variable &LASTCC (Note, however, that return codes from CLIST subprocedures do *not* cause an error routine to receive control.)

RETURN is valid only when issued from a subprocedure, an activated error routine, or an activated attention routine. If issued from any other place, RETURN is treated as a no-operation.



label
A name the CLIST can reference in a GOTO statement to branch to this RETURN statement. *label* is one-to-31 alphanumeric characters, beginning with an alphabetic character.

CODE
Subprocedures can issue a return code. Control will pass to the statement following the SYSCALL statement that called the subprocedure.

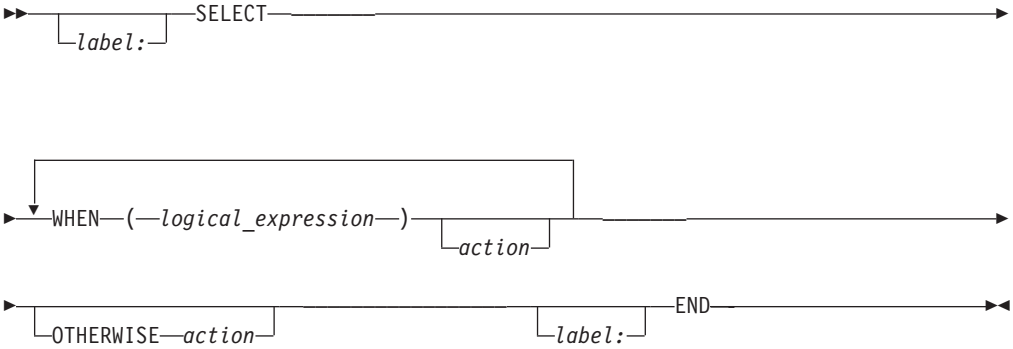
expression
A CLIST-defined return code. *expression* can be a character string, a decimal integer, or an expression that evaluates to a decimal integer. The expression is stored in the control variable &LASTCC. If CODE appears without an expression, &LASTCC takes a null value.

SELECT statement

Use the SELECT statement to conditionally perform one of several alternative actions. There are two forms of the SELECT statement: the simple SELECT and the compound SELECT.

Simple SELECT

In the simple SELECT statement, the CLIST tests one or more expressions. When the CLIST finds an expression that evaluates to a true value, the CLIST performs the associated action, then passes control to the END statement. If none of the expressions are true, the CLIST performs the action on the OTHERWISE clause, if any, or passes control to the END statement.



Note: Each WHEN and OTHERWISE statement must be on a separate line, and must be separated from the SELECT and END. See syntax diagram above.

label
A name the CLIST can reference in a GOTO statement to branch to this SELECT statement. *label* is one-to-31 alphanumeric characters, beginning with an alphabetic character.

logical_expression
A comparative expression, such as `&A = 3` or `&B ¬> 10`, that evaluates to a true or false condition.

action
Any CLIST statement, TSO/E command, or DO sequence. A null action passes

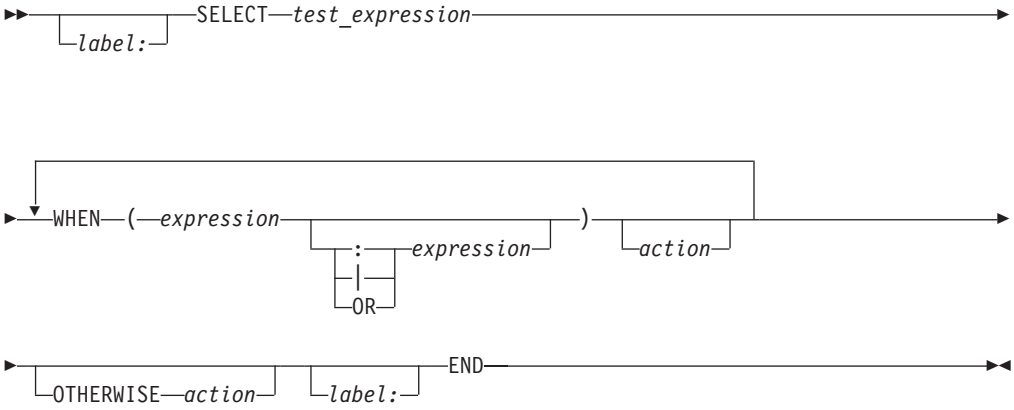
control to the END statement. The action can include nested IF, DO, and SELECT statements. Any statements in the action can have labels, allowing GOTO statements to branch to them.

Compound SELECT

A compound SELECT statement includes an initial test expression. The CLIST evaluates the test expression and compares its value to those of the WHEN expressions.

In a compound SELECT statement, a WHEN expression can contain multiple expressions separated by the logical operator | (OR). WHEN expressions can also include ranges of values, represented by a colon (:) between the lowest and highest values of the range. For example, 3:5 represents 3, 4, and 5.

When a test expression matches a value or falls within a range of values in a WHEN expression, the CLIST performs the associated action and passes control to the END statement. If no matches are found, the CLIST performs the action on the OTHERWISE clause, if any, or passes control to the END statement.



label
A name the CLIST can reference in a GOTO statement to branch to this SELECT statement. *label* is one-to-31 alphanumeric characters, beginning with an alphabetic character.

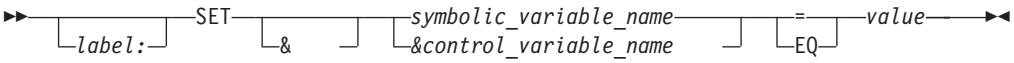
test_expression
A character string or a logical expression that results in a value to be compared to the expressions in the WHEN clauses.

expression
A character string, a single logical expression, or a range such as 1:5. Values and ranges can be combined, for example: WHEN (&A-3 | &B | 4:6)

action
Any CLIST statement, TSO/E command, or DO sequence. A null action passes control to the END statement. The action can include nested IF, DO, and SELECT statements. Any statements in the action can have labels of their own.

SET statement

Use the SET statement to assign a value to a symbolic variable or a control variable.



label
A name the CLIST can reference in a GOTO statement to branch to this SET statement. *label* is one-to-31 alphanumeric characters, beginning with an alphabetic character.

symbolic_variable_name | *control_variable_name*

symbolic_variable_name
The symbolic variable to which you are assigning a value.

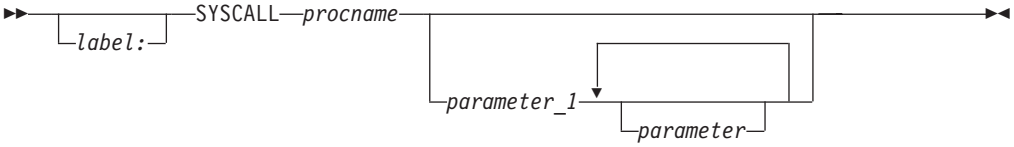
control_variable_name
The control variable to which you are assigning a value. (See Table 4 on page 31 for those control variables that you can modify.)

EQ | **=**
The operator 'equal'.

value
Any valid numeric value or character string.

SYSCALL statement

Use the SYSCALL statement to pass control to a subprocedure. The SYSCALL statement contains the name of the subprocedure and any parameters to be passed. The name of the subprocedure must match the label on the PROC statement that begins the subprocedure.



label
A name the CLIST can reference in a GOTO statement to branch to this SYSCALL statement. *label* is one-to-31 alphanumeric characters, beginning with an alphabetic character.

procname
The label of the PROC statement that begins the subprocedure.

parameter_1 / *parameter*
Any valid CLIST expression, including constants, symbolic variables, built-in functions, and arithmetic expressions. All parameters are separated by CLIST delimiters (blanks, commas, or tabs). For information about how to pass a parameter that contains blanks, see "Calling a subprocedure" on page 77.

If the parameter is the name of a variable that is referred to in a SYSREF statement in the subprocedure, the variable name must not include an ampersand on the SYSCALL statement.

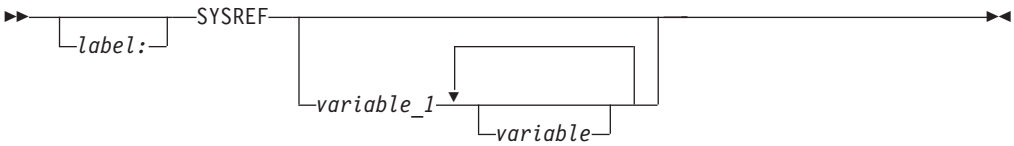
The PROC statement of the subprocedure is responsible for defining variables to receive the parameters.

```
SET &A = John
SET &B = AL
SYSCALL  XYZ  &A  B          /* pass variables to XYZ, omitting & from
                             /* the variable name referenced on SYSREF
    .                        /* result:  GEORGE
    WRITE  &B
    .
    .
XYZ:  PROC 2  PARM1  PARM2   /*  Subprocedure XYZ  */
        .
        SYSREF &PARM2        /*  Indicate PARM2 holds a variable name
        WRITE &PARM2         /*  result:  AL
        SET &PARM2 = GEORGE
    END
```

SYSREF statement

Use the SYSREF statement in a subprocedure to identify the names of variables, passed from the caller, whose values the subprocedure can reference and modify. When you assign a new value to a SYSREF variable, the new value is retroactive; that is, the new value takes effect both in the caller and in the subprocedure.

On the SYSREF statement in the subprocedure, list the PROC statement parameter that corresponds to the variable name that the caller passed. The SYSREF statement must precede any subprocedure statement that uses its variables.



label
A name the CLIST can reference in a GOTO statement to branch to this SYSREF statement. *label* is one-to-31 alphanumeric characters, beginning with an alphabetic character.

variable_1 / variable
The name of a parameter from the PROC statement. The parameters correspond to variable names that were passed to the PROC statement. Ampersands (&) are optional on the variable name.

In the following example, the subprocedure assigns a new value to the variable whose name was passed (B). The new value (GEORGE) replaces the variable's old

value (AL) in the caller.

```
    SET &A = John
    SET &B = AL
    SYSCALL  XYZ  &A  B          /*  pass variables to XYZ, omitting & from
        .                        /*  the variable name referenced on SYSREF
        WRITE  &A                /*  result:  JOHN (original value)
        WRITE  &B                /*  result:  GEORGE (changed value)
        .
XYZ:  PROC 2  PARM1  PARM2       /*  Subprocedure XYZ  */
            .
            SET &parm1 = Joe     /*  change value of &parm1
            WRITE &parm2         /*  result:  JOE
            SYSREF &PARM2        /*  indicate PARM2 holds a variable name
            WRITE &PARM2         /*  result:  AL
            SET &PARM2 = GEORGE  /*  change value of SYSREF variable
        END
```

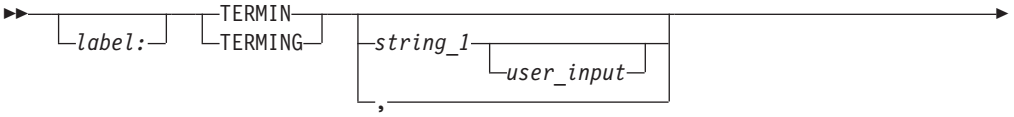
TERMIN and TERMING statement

Use the TERMIN or TERMING statement to pass control from the CLIST to the terminal user. You can also use TERMIN or TERMING to define the character strings, including a null line, that a user enters to return control to the CLIST. TERMIN is typically preceded by a WRITE statement that requests the expected response from the terminal user.

The TERMIN or TERMING statement ends a CLIST when you issue a CLIST in any of the following ways:

- Under ISPF
- In the background
- From a REXX exec (a nested CLIST)

Control returns to the CLIST at the statement after TERMIN or TERMING. When control returns, &SYSDLM and &SYSDVAL have been set.



label
A name the CLIST can reference in a GOTO statement to branch to this TERMIN statement. *label* is one-to-31 alphanumeric characters, beginning with an alphabetic character.

TERMIN | TERMING
transfers control to the terminal and establishes a means for the user to return control to the CLIST.

TERMIN
A CLIST executed from the TERMIN is *not* considered nested within the CLIST that issued the TERMIN statement, which has the following effects:

- Sharing GLOBAL variables - GLOBAL variables *cannot* be shared across the TERMIN. Global variable sharing between the CLIST executed from the TERMIN and the CLIST that issued the TERMIN is *not* allowed.
- Variable access - variable access across the TERMIN *cannot* be communicated through the CLIST variable access routine IKJCT441.
- Checking command output trapping (&SYSOUTTRAP) - IKJCT441 and IRXEXCOM *do not* recognize CLISTs or REXX execs on opposing sides of a TERMIN element.
- CONTROL NOMSG statement - checking the NOMSG setting on opposing sides of a TERMIN element is *not* allowed.

TERMING
A CLIST executed from the TERMING is considered nested within the CLIST that issued the TERMING statement, which has the following effects:

- Sharing GLOBAL variables - GLOBAL variables can be shared across the TERMING. Global variable sharing between the CLIST executed from the TERMING and the CLIST that issued the TERMING is allowed.
- Variable access - variable access across the TERMING can be communicated through the CLIST variable access routine IKJCT441.
- Checking command output trapping (&SYSOUTTRAP) - IKJCT441 and IRXEXCOM recognize CLISTs or REXX execs on opposing sides of a TERMING element.
- CONTROL NOMSG statement - checking the NOMSG setting on opposing sides of a TERMING element is allowed.

string_1 / string
A character string that the terminal user enters to return control to the CLIST. The &SYSDLM control variable contains a number corresponding to the position of the string that the user entered (1 for $string_1$, 2 for $string_2$, and so on).

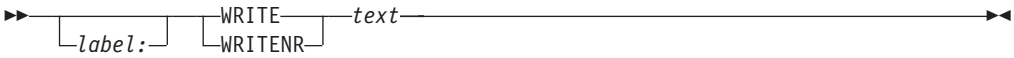
user_input
Additional input entered by the terminal user. The input is stored in the &SYSDVAL control variable.

, If you specify a comma in place of a string, the terminal user can enter a null line (press the Enter key) to return control to the CLIST.

If no operands are specified the terminal user enters a null line to return control to the CLIST.

WRITE and WRITENR statements

Use the WRITE and WRITENR statements to define text and have it displayed at the terminal. This text can be used for messages, information, or prompting.



label
A name the CLIST can reference in a GOTO statement to branch to this WRITE/WRITENR statement.*label* is one-to-31 alphanumeric characters, beginning with an alphabetic character.

WRITE | WRITENR

WRITE
The cursor moves to a new line after the text is displayed.

WRITENR
The cursor does not move to a new line after the text is displayed.

text
What is displayed at the terminal. You can enter any character string, including symbolic variables. Unless you enclose an arithmetic expression in an &EVAL built-in function, the WRITE/WRITENR statement does not perform evaluation on the expression. The CLIST also displays any comments on the same line as the WRITE/WRITENR statement.

END command

For information about the END command, see *z/OS TSO/E Command Reference*.

EXEC command

For a description of the EXEC command, see *z/OS TSO/E Command Reference*.

Appendix. Accessibility

Accessible publications for this product are offered through IBM Knowledge Center (http://www.ibm.com/support/knowledgecenter/SSLTBW/welcome).

If you experience difficulty with the accessibility of any z/OS information, send a detailed message to the "Contact us" web page for z/OS (http://www.ibm.com/systems/z/os/zos/webqs.html) or use the following mailing address.

IBM Corporation
Attention: MHVRCFS Reader Comments
Department H6MA, Building 707
2455 South Road
Poughkeepsie, NY 12601-5400
United States

Accessibility features

Accessibility features help users who have physical disabilities such as restricted mobility or limited vision use software products successfully. The accessibility features in z/OS can help users do the following tasks:

- Run assistive technology such as screen readers and screen magnifier software.
- Operate specific or equivalent features by using the keyboard.
- Customize display attributes such as color, contrast, and font size.

Consult assistive technologies

Assistive technology products such as screen readers function with the user interfaces found in z/OS. Consult the product information for the specific assistive technology product that is used to access z/OS interfaces.

Keyboard navigation of the user interface

You can access z/OS user interfaces with TSO/E or ISPF. The following information describes how to use TSO/E and ISPF, including the use of keyboard shortcuts and function keys (PF keys). Each guide includes the default settings for the PF keys.

- *z/OS TSO/E Primer*
- *z/OS TSO/E User's Guide*
- *z/OS V2R2 ISPF User's Guide Vol I*

Dotted decimal syntax diagrams

Syntax diagrams are provided in dotted decimal format for users who access IBM Knowledge Center with a screen reader. In dotted decimal format, each syntax element is written on a separate line. If two or more syntax elements are always present together (or always absent together), they can appear on the same line because they are considered a single compound syntax element.

Each line starts with a dotted decimal number; for example, 3 or 3.1 or 3.1.1. To hear these numbers correctly, make sure that the screen reader is set to read out

punctuation. All the syntax elements that have the same dotted decimal number (for example, all the syntax elements that have the number 3.1) are mutually exclusive alternatives. If you hear the lines 3.1 USERID and 3.1 SYSTEMID, your syntax can include either USERID or SYSTEMID, but not both.

The dotted decimal numbering level denotes the level of nesting. For example, if a syntax element with dotted decimal number 3 is followed by a series of syntax elements with dotted decimal number 3.1, all the syntax elements numbered 3.1 are subordinate to the syntax element numbered 3.

Certain words and symbols are used next to the dotted decimal numbers to add information about the syntax elements. Occasionally, these words and symbols might occur at the beginning of the element itself. For ease of identification, if the word or symbol is a part of the syntax element, it is preceded by the backslash (\) character. The * symbol is placed next to a dotted decimal number to indicate that the syntax element repeats. For example, syntax element *FILE with dotted decimal number 3 is given the format 3 * FILE. Format 3* FILE indicates that syntax element FILE repeats. Format 3* * FILE indicates that syntax element * FILE repeats.

Characters such as commas, which are used to separate a string of syntax elements, are shown in the syntax just before the items they separate. These characters can appear on the same line as each item, or on a separate line with the same dotted decimal number as the relevant items. The line can also show another symbol to provide information about the syntax elements. For example, the lines 5.1*, 5.1 LASTRUN, and 5.1 DELETE mean that if you use more than one of the LASTRUN and DELETE syntax elements, the elements must be separated by a comma. If no separator is given, assume that you use a blank to separate each syntax element.

If a syntax element is preceded by the % symbol, it indicates a reference that is defined elsewhere. The string that follows the % symbol is the name of a syntax fragment rather than a literal. For example, the line 2.1 %OP1 means that you must refer to separate syntax fragment OP1.

The following symbols are used next to the dotted decimal numbers.

? indicates an optional syntax element
The question mark (?) symbol indicates an optional syntax element. A dotted decimal number followed by the question mark symbol (?) indicates that all the syntax elements with a corresponding dotted decimal number, and any subordinate syntax elements, are optional. If there is only one syntax element with a dotted decimal number, the ? symbol is displayed on the same line as the syntax element, (for example 5? NOTIFY). If there is more than one syntax element with a dotted decimal number, the ? symbol is displayed on a line by itself, followed by the syntax elements that are optional. For example, if you hear the lines 5 ?, 5 NOTIFY, and 5 UPDATE, you know that the syntax elements NOTIFY and UPDATE are optional. That is, you can choose one or none of them. The ? symbol is equivalent to a bypass line in a railroad diagram.

! indicates a default syntax element
The exclamation mark (!) symbol indicates a default syntax element. A dotted decimal number followed by the ! symbol and a syntax element indicate that the syntax element is the default option for all syntax elements that share the same dotted decimal number. Only one of the syntax elements that share the dotted decimal number can specify the ! symbol. For example, if you hear the lines 2? FILE, 2.1! (KEEP), and 2.1 (DELETE), you know that (KEEP) is the

default option for the FILE keyword. In the example, if you include the FILE keyword, but do not specify an option, the default option KEEP is applied. A default option also applies to the next higher dotted decimal number. In this example, if the FILE keyword is omitted, the default FILE(KEEP) is used. However, if you hear the lines 2? FILE, 2.1, 2.1.1! (KEEP), and 2.1.1 (DELETE), the default option KEEP applies only to the next higher dotted decimal number, 2.1 (which does not have an associated keyword), and does not apply to 2? FILE. Nothing is used if the keyword FILE is omitted.

*** indicates an optional syntax element that is repeatable**

The asterisk or glyph (*) symbol indicates a syntax element that can be repeated zero or more times. A dotted decimal number followed by the * symbol indicates that this syntax element can be used zero or more times; that is, it is optional and can be repeated. For example, if you hear the line 5.1* data area, you know that you can include one data area, more than one data area, or no data area. If you hear the lines 3* , 3 HOST, 3 STATE, you know that you can include HOST, STATE, both together, or nothing.

Notes:

1. If a dotted decimal number has an asterisk (*) next to it and there is only one item with that dotted decimal number, you can repeat that same item more than once.
2. If a dotted decimal number has an asterisk next to it and several items have that dotted decimal number, you can use more than one item from the list, but you cannot use the items more than once each. In the previous example, you can write HOST STATE, but you cannot write HOST HOST.
3. The * symbol is equivalent to a loopback line in a railroad syntax diagram.

+ indicates a syntax element that must be included

The plus (+) symbol indicates a syntax element that must be included at least once. A dotted decimal number followed by the + symbol indicates that the syntax element must be included one or more times. That is, it must be included at least once and can be repeated. For example, if you hear the line 6.1+ data area, you must include at least one data area. If you hear the lines 2+, 2 HOST, and 2 STATE, you know that you must include HOST, STATE, or both. Similar to the * symbol, the + symbol can repeat a particular item if it is the only item with that dotted decimal number. The + symbol, like the * symbol, is equivalent to a loopback line in a railroad syntax diagram.

Notices

This information was developed for products and services offered in the U.S.A. or elsewhere.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

Site Counsel
IBM Corporation
2455 South Road
Poughkeepsie, NY 12601-5400
USA

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

COPYRIGHT LICENSE:

This information might contain sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Policy for unsupported hardware

Various z/OS elements, such as DFSMS, HCD, JES2, JES3, and MVS, contain code that supports specific hardware servers or devices. In some cases, this device-related element support remains in the product even after the hardware devices pass their announced End of Service date. z/OS may continue to service element code; however, it will not provide service related to unsupported hardware devices. Software problems related to these devices will not be accepted

for service, and current service activity will cease if a problem is determined to be associated with out-of-support devices. In such cases, fixes will not be issued.

Minimum supported hardware

The minimum supported hardware for z/OS releases identified in z/OS announcements can subsequently change when service for particular servers or devices is withdrawn. Likewise, the levels of other software products supported on a particular release of z/OS are subject to the service support lifecycle of those products. Therefore, z/OS and its product publications (for example, panels, samples, messages, and product documentation) can include references to hardware and software that is no longer supported.

- For information about software support lifecycle, see: IBM Lifecycle Support for z/OS (http://www.ibm.com/software/support/systemsz/lifecycle/)
- For information about currently-supported IBM hardware, contact your IBM representative.

Programming interface information

This document describes intended Programming Interfaces that allow the customer to write programs to obtain the services of z/OS TSO/E.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available at Copyright and Trademark information (http://www.ibm.com/legal/copytrade.shtml).

Index

Special characters

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

Y

Z

Product Number: 5650-ZOS

Printed in USA

SA32-0978-01