SC28-1304-1
File No. S370-39

Program Product

TSO Extensions CLISTS: Implementation and Reference

Second Edition (December, 1985)

This is a major revision of, and obsoletes, SC28-1304-0. See the Summary of Amendments following the Contents for a summary of the changes made to this manual. Technical changes or additions to the text and illustrations are indicated by a vertical line to the left of the change.

This edition applies to TSO Extensions (TSO/E) Release 2, Program Number 5665-285, and all subsequent releases until otherwise indicated in new editions or Technical Newsletters. Changes are made periodically to the information herein: before using this publication in connection with the operation of IBM systems, consult the latest *IBM System/370 Bibliography*, GC20-0001, for the editions that are applicable and current.

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM program product in this publication is not intended to state or imply that only IBM's program product may be used. Any functionally equivalent program may be used instead.

Publications are not stocked at the address given below. Requests for IBM publications should be made to your IBM representative or to the IBM branch office serving your locality.

A form for readers' comments is provided at the back of this publication. If the form has been removed, comments may be addressed to IBM Corporation, Information Development, Department D58, Building 921, PO Box 390, Poughkeepsie, New York 12602. IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Preface

Audience

This publication is intended for programmers who design and code CLISTS for installation-dependent applications. Although the applications will vary from installation to installation, some possible ones are:

- Extensions to TSO
- Production programs
- Interfaces between an end user and TSO
- Interfaces between an end user and existing applications

To exploit the full capabilities of the CLIST language, a programmer should have a prerequisite knowledge of TSO and ISPF.

Organization

"Chapter 1: Introduction" describes the types of functions CLISTs perform.

"Chapter 2: Developing and Executing CLISTs" describes how to set up and invoke CLISTs.

"Chapter 3: Programming Tools" describes how to use CLIST statements, functions, and variables and how they interrelate.

"Chapter 4: Implementation" provides examples of CLISTs that perform a broad range of application tasks. Each example is preceded by a description of the concepts the example illustrates. Generally, the more complex examples more fully develop the concepts introduced in the simple examples.

"Chapter 5: Reference" contains complete syntax descriptions of all of the CLIST statements and two TSO commands - EXEC and END. This chapter also includes a figure describing error codes.

Related Publications

The following publications are referenced in, or related to, this publication:

- *IBM System/370 Reference Summary*, GX20-1850
- *TSO Extensions Command Language Reference*, SC28-1307
- *TSO Extensions User's Guide*, SC28-1333
- *TSO Guide to Writing a Terminal Monitor Program or Command Processor*, SC28-1136.
- *Interactive System Productivity Facility Dialog Management Services*, SC34-2137
- *Interactive System Productivity Facility/Program Development Facility for MVS: Program Reference*, SC34-2139
- *MVS/370 JCL User's Guide*, GC28-1349; *MVS/370 JCL Reference*, GC28-1350.
- *MVS/Extended Architecture JCL User's Guide*, GC28-1351; *MVS/Extended Architecture JCL Reference*, GC28-1352.
- *MVS/370 Data Management Services*, GC28-4058
- *MVS/Extended Architecture Data Administration Guide*, GC26-4013

Referenced Products

All occurrences of ISPF refer to Interactive System Productivity Facility, Program Number 5665-319.

All occurrences of ISPF/PDF refer to Interactive System Productivity Facility/Program Development Facility, Program Number 5665-317.

All occurrences of RACF refer to Resource Access Control Facility, Program Number 5740-XXH.

Contents

Figures

Summary of Amendments

Summary of Amendments
for SC28-1304-1
for TSO Extensions Release 2

Changes have been made throughout this publication to support TSO Extensions (TSO/E) service updates.

Chapter 1. Introduction

The CLIST language provides functions that enable you and others to do work more efficiently. You can organize related activities so that users can simply invoke a CLIST to perform a given task or group of tasks. CLISTs can handle any number of activities, from issuing multiple TSO commands to coordinating testing sessions.

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

The CLIST language includes an extensive set of arithmetic and logical operators, as well as string-handling functions.

CLIST statements let you structure your programs, perform I/O, define and modify symbolic variables, and handle errors and attention interrupts. In addition, you can use CLIST statements in conjunction with TSO commands and JCL statements to write applications.

Based on input supplied by the invoker, your CLISTs can invoke programs in either the foreground or the background.

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

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

CLISTs That Perform Routine Tasks

A user may perform certain tasks on a regular basis. Checking on the status of data sets, allocating data sets for particular programs, and printing files are examples of such tasks.

You can write CLISTs that significantly reduce the amount of time that a user has 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; thereby, increasing end-user productivity. Such a CLIST may consist entirely of TSO commands or of a combination of TSO commands and CLIST statements.

If the tasks require specific input, you may define variables on a PROC statement, use WRITE and READ statements, or allow TSO commands to prompt for the input.

CLISTs That are Self-contained Applications

The CLIST language includes the basic tools required to write applications. Any CLIST can invoke another CLIST, which is referred to as a nested CLIST. Therefore, you can structure complex applications using CLISTs. The GLOBAL statement lets you define common data among CLISTs and the PROC statement allows you to pass parameters to a CLIST.

CLISTs can issue ISPF commands, such as ISPEXEC, to display full-screen panels. Conversely, ISPF panels can invoke CLISTs, based on input provided by the user. When the user changes a value on a panel, the change to the variable on the associated panel definition also applies to the value of the variable in the CLIST that displayed the panel.

CLISTs That Manage Applications Written in Other Languages

You may have access to applications that are written in other programming languages and perform useful services. However, the interfaces required to invoke these applications may not be easily mastered by a casual user. Rather than write new applications, you can write CLISTs that act as intermediaries between a user and such applications.

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

Chapter 2. Creating, Editing, and Executing CLISTs

This chapter describes how to create, edit, and execute CLISTs. The descriptions include:

- the steps you take to create and edit CLIST data sets
- the attributes of CLIST data sets
- the different methods you can use to execute a CLIST
- how to allocate and concatenate CLIST data sets

Data Sets and CLIST Libraries

CLISTs are programs that reside in either sequential or partitioned data sets (PDSs). A sequential CLIST data set consists of only one CLIST, while a PDS may 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. (A CLIST library may also consist of a concatenation of individual CLIST libraries.)

CLISTs are stored and cataloged like other TSO data sets. Your installation may allocate a PDS to be used as a production CLIST library. You may create your own CLIST library by making your CLISTs members of a PDS. If you store CLISTs in a PDS, you can allocate that data set to the file SYSPROC and execute the CLISTs implicitly as described under 'Executing CLISTs'.

Creating and Editing CLIST Data Sets

Prior to coding a CLIST, you should create a CLIST data set. As you code the CLIST, you can edit it interactively. Once the CLIST is completed, you can execute it for testing purposes, make any necessary changes or corrections to it directly on the screen, and re-execute it.

There are two ways to create and edit CLIST data sets:

1. Using options 2 (EDIT) and 3 (UTILITIES) of ISPF/PDF.

   - Allocate a CLIST data set using the allocate panel in ISPF (usually option 3.2 on the primary menu).

   - Create your CLIST in the full-screen environment using the ISPF/PDF editor (usually option 2).

- Modify the CLIST by making corrections directly to the data on the screen.

For a complete description of creating and editing data sets under ISPF/PDF, refer to *Interactive System Productivity Facility/Program Development Facility for MVS: Program Reference.*

2. Using the TSO EDIT command and its subcommands. (This method includes option 6 of ISPF/PDF.)

   - Include the CLIST keyword on the EDIT command to inform TSO that you are creating a CLIST data set.
   - Enter and save your CLIST statements, TSO commands, and TSO subcommands.
   - Use subcommands of EDIT to modify the CLIST.

   For a complete description of creating and editing data sets under TSO, refer to *TSO Extensions Command Language Reference.*

CLIST Data Set Attributes

If no explicit assignment is made, the system assigns the following default attributes:

Record format	Variable-blocked
Logical record length	255
Blocksize	3120

When creating a CLIST data set, you have the option of overriding the default attributes. However, you should allocate your CLIST data set using the same attributes as your installation's production CLIST library.

Note: CLIST record formats must be fixed- or variable-blocked only.

Considerations for Copying CLIST Data Sets

When copying an existing CLIST data set into another data set under ISPF/PDF, be aware of the record formats of the data sets. Variable-blocked data sets may contain line numbers in columns 1-8 that do not normally appear when you are editing 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 manually.

When copying fixed-blocked data sets into variable-blocked data sets, the system copies the sequence numbers from columns 72-80 into the variable-blocked data set. This data must also be removed manually. See the *ISPF Program Reference*, SC34-2139, for information on how to delete all the line numbers from a variable block data set.

Executing CLISTs

To execute a CLIST, use the EXEC command. When in TSO EDIT mode, use the EXEC subcommand. CLISTs executed under EDIT can execute only EDIT subcommands and CLIST statements. However, in a CLIST you can terminate EDIT mode using the END subcommand to allow the CLISTS to execute TSO commands.

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

The EXEC command (or subcommand) has two forms:

- Explicit form -- Enter "exec" or "ex" followed by the one of the following:

  1. the name of the data set and the member name, with the member name enclosed in parentheses (the CLIST is a member of a PDS whose library name is CLISTLIB and type is CLIST)

     For example, if a CLIST, LISTPGM, is a member of a PDS named PREFIX.CLISTLIB.CLIST, enter:

     ```
     {exec}     clistlib(listpgm)
     {ex  }
     ```

  2. the name of the CLIST enclosed in parentheses (the CLIST is a member of a PDS whose type is CLIST

     For example, if a CLIST, LISTPGM, is a member of a PDS named PREFIX.CLIST, enter:

     ```
     {exec}     (listpgm)
     {ex  }
     ```

  3. the name of the CLIST (the CLIST is in a sequential data set whose type is CLIST)

     For example, if the CLIST is in a sequential data set named PREFIX.LISTPGM.CLIST, enter:

     ```
     {exec}     listpgm
     {ex  }
     ```

  4. the fully qualified name of a data set enclosed in single quotes

     For example, if the CLIST is in a data set named PREFIX.LISTPGM, enter:

     ```
     {exec}     'prefix.listpgm'
     {ex  }
     ```

- Implicit form -- Enter only the name of the CLIST, optionally preceded by a percent sign (%). The implicit form works only when executing CLISTs

that are members of a PDS allocated to the file SYSPROC. The two implicit forms are as follows:

1. Enter only the member name, for example:

```
listpgm
```

   When you use this form, TSO searches several libraries before it searches the SYSPROC file to ensure that the name you entered is not a TSO command.

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

```
%listpgm
```

   When you use this form, called the extended implicit form, TSO searches only the SYSPROC file for the name, thus reducing the amount of search time.

For a complete syntactical definition of the EXEC command, refer to Chapter 5.

Methods of Allocating Data Sets to SYSPROC

There are three methods of allocating partitioned CLIST data sets to SYSPROC:

1. Allocate them by a CLIST that you execute when you log on.
2. Allocate the data sets at the terminal using the ALLOCATE command.
3. Have your logon procedure modified to allocate them.

If you use method 1 or 2, include the REUSE operand on the ALLOCATE command. The REUSE operand enables you to use an already allocated filename without having to free it.

Concatenating Data Sets

To create a CLIST library that spans several data sets, use the ALLOCATE command to concatenate the data sets and allocate them to the file name specified on the FILE or DDNAME keyword. The concatenated data sets are treated as one data set for the duration of your TSO session.

If you want to implicitly execute the CLISTs in the concatenation, allocate the data sets to the file SYSPROC. When you allocate data sets to SYSPROC, include in the concatenation all data sets that were previously allocated to SYSPROC if you want to implicitly invoke any CLISTs in those data sets. To find out which data sets are allocated to SYSPROC, issue the LISTALC command with the STATUS keyword.

To concatenate data sets using the ALLOCATE command, enter their names in the order in which you want TSO to concatenate them. The concatenation order establishes the order in which TSO searches the data sets to find specified CLISTs.

The block sizes of data sets may affect their concatenation order. If block sizes vary, TSO requires that you specify the data set with the largest block size first. If, for example, you want to concatenate your private library to the installation's library so that your library is the first in the concatenation order, make sure that your library's block size is at least as large as that of the installation's library. To determine the block size of any library, display its data set attributes with the LISTDS command.

Do not mix variable-blocked with fixed-block data sets in a concatenation.

For example, assume your userid is STEVE and you want to concatenate the CLIST libraries whose fully-qualified data set names are as follows:

```
MARK.CLISTLIB.CLIST
STEVE.CMPRCLIB.CLIST
MASTER.PROFLIB.CLIST
ISR.V1R1M0.ISRCLIB
```

After you determine that the blocksizes are in descending order, the largest first, the following ALLOCATE command concatenates the data sets in the order listed and allocates them to the file SYSPROC so that CLISTs from any of them may be invoked implicitly:

```
allocate file(sysproc) da('mark.clistlib.clist'-
  'steve.cmprclib.clist' 'master.proflib.clist' -
  'isr.v1r1m0.isrclib') shr reu
```

The hyphen at the end of the first line indicates that the command continues on the next line. The REU keyword enables you to allocate SYSPROC without having to free and reallocate it. The disposition of SHR allows more than one person to use the allocated data sets concurrently.

Chapter 3. Programming Tools

This chapter describes the programming tools provided by the CLIST language and how to use them in conjunction with TSO commands and JCL statements.

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 correctly in both the command and subcommand environment. They fall into the categories shown in Figure 3-1.

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

Figure 3-1. CLIST Statement Categories

Subsequent topics in this chapter describe all the statements in detail.

Syntax Rules

This section provides the syntax rules for CLIST statements relative to those for TSO commands.

Delimiters

Most CLIST statements have operands. Operands are variables or data that provide information to be used in processing the statement. As is the case with TSO commands, 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 equivalent. 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 would have to insert a blank before the plus sign for correct execution.

Note: If you use a continuation character in a line containing comments, the continuation character must appear *after* the comments, as in the following example:

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

Capitalization

All CLIST statement names must be capitalized. If lowercase letters are used when uppercase are required, the CLIST abnormally terminates. Capitalization of CLIST variable names and built-in function names is optional. Capitalization of TSO commands and subcommands in a CLIST is also optional.

Formatting

The CLIST language does not provide any formatting aids. For example, the interpreter does not align DO statements with their corresponding END statements; you must align them yourself if formatting is desired. You may use blank lines as a formatting aid. Note that a blank line after a continuation character ends continuation, unless the blank line is also continued.

Comments

You may include a comment:

- On a line by itself
- Before, in the middle of, or after a CLIST statement or TSO command that is not continued on the following line

You define a comment by placing the descriptive text between facing slash-asterisk combinations (comment delimiters) as follows:

```
/* This statement opens the data set containing the input records */
```

As shown in the following example, when a comment appears at the end of a *logical* line, the closing comment delimiter is unnecessary.

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

Note that comments may be in both upper-case and lowercase letters. *No symbolic substitution is performed in comments.*

TSO Commands and JCL Statements

You can include TSO commands and subcommands, and JCL statements in a CLIST to provide functions that add capabilities and flexibility to that CLIST.

TSO Commands

You may include TSO commands and subcommands (and user-written commands and subcommands) in a CLIST at any point where the specific functions (for example, allocate, free, etc.) are required. For certain applications, a CLIST may consist entirely of commands and subcommands. You can also substitute CLIST variables as operands in commands and subcommands, or as commands themselves, to add flexibility to a CLIST.

JCL Statements

From a CLIST, you may want to submit a jobstream for execution. In the CLIST, you can include the required JCL statements (EXEC, DD, etc.). 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.
2. A statement containing a single ampersand (&) or a double ampersand (&&) - use the &NRSTR built-in function or the &SYSSCAN control variable.
3. JCL comments - use the &STR built-in function.

Examples of using the built-in functions and the control variable are provided later in this chapter.

Operators and Expressions

Operators cause a CLIST to perform evaluations on data; the data may be numeric or character, or may be a variable or a built-in function. Operations fall into three categories: arithmetic, comparative, and logical, as shown in Figure 3-2.

- Arithmetic operators perform integer arithmetic on numeric operands. The operators connect integers, variables, or built-in functions to form expressions, such as *4-2*.

- Comparative operators perform comparisons between two expressions, to form comparative 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 Figure 3-2, if more than one accepted value exists for an operator, the values are separated by commas.

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

Figure 3-2. Arithmetic, Comparative, and Logical Operators

Order of Evaluations

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

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

You may 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 may 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 in expressions can range from -2,147,483,648 (-2^{31}) 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.

Symbolic Variables

A symbolic variable is any alphameric character string, usually preceded by an ampersand, for which you can substitute different values at different times. The variable name may be a maximum of 252 alphameric characters (not including the ampersand); variables used as keywords have a maximum length of 31 characters. The first character must be an alphabetic or national character, or the underscore. (However, any parameter specified on a PROC statement must only begin with an alphabetic character.) Valid alphameric characters include letters, digits, national characters (#,$,@), and the character underscore (_).

Note: The system recognizes the following hexadecimal representations of the U.S. national characters: @ as X'7C'; $ as X'5B'; and # as X'7B'. In countries other than the U.S., the U.S. national characters represented on terminal keyboards might generate a different hexadecimal representation and cause an error. For example, in some countries the $ character may generate a X'4A'.

You can include variables on a subset of the CLIST statements, on TSO commands and subcommands, and on JCL statements. When a CLIST is executed, it scans each line and replaces the symbolic variables with their actual values. This process is called symbolic substitution.

Valid Values of Variables

The maximum length of a value that you can assign to a variable is 32,768 characters.

Defining Variables and Assigning Values

You can use a number of CLIST statements to define and assign values to variables. A short description of each statement follows:

- Use the PROC statement to define variables and assign values to them.
- Use the GLOBAL statement to define variables whose values are accessed by more than one CLIST.
- Use the SET, READ, and READDVAL statements to define variables, assign values to them, and modify them.
- Use the OPENFILE statement to define a variable used to send records to, and retrieve records from, a data set.

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

The PROC Statement

The PROC statement enables both the CLIST and its invoker to assign values to variables by CLIST invocation, prompting, and default parameter values. The PROC statement may contain positional parameters, keyword parameters, and keyword parameters with associated values. When used, the PROC statement must precede all other executable statements and commands. However, comments or blank lines may precede the PROC statement.

The GLOBAL Statement

The GLOBAL statement enables a CLIST to reference the value of a variable if the variable was defined on a GLOBAL statement in both:

1. the CLIST itself
2. the highest-level CLIST of those that directly or indirectly invoked this CLIST

For more information on using the GLOBAL statement, refer to "Nested CLISTs" later in this chapter.

The SET Statement

You use the SET statement to assign a specific value to a symbolic variable. 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 increment or decrement 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
```

Note that an ampersand is required when the variable appears in the expression on the right side of the equal sign, but is optional when the variable appears on the left-hand side of the equal sign.

You can also use control variables and built-in functions on SET statements. Refer to "Control Variables" and "Built-in Functions" for additional information.

The READ Statement

The READ statement creates or modifies variables. You can assign terminal input directly to variables and do so to multiple variables using one statement.

The CLIST successively assigns each input string supplied after a READ statement to the variables included on the READ statement.

If you code a READ statement with no operands, the input is stored in the &SYSDVAL control variable. You then can use a READDVAL statement to assign the contents of &SYSDVAL to a variable or set of variables.

The READDVAL Statement

Like the READ statement, the READDVAL statement enables you to define and assign values to multiple variables using one statement.

The CLIST successively assigns each input string contained in the control variable &SYSDVAL to the variables included on the READDVAL statement. (For a description of &SYSDVAL, refer to "Control Variables" later in this chapter.)

I/O Statements

You use the OPENFILE, GETFILE, PUTFILE, and CLOSFILE statements to perform I/O to a physical sequential data set or to a member of a PDS.

Performing I/O to data sets involves steps that involve variables. For input, define a variable that receives the retrieved record. For output, assign to a variable the value (record) to be sent to the data set. In either case, the variable created must have the same name as the file name of the data set on which I/O is being performed. For examples, refer to "Performing I/O."

The CLIST scans I/O variables only once to obtain the variable's value.

Nesting Symbolic Variables

The value substituted for a symbolic variable can be another (nested) symbolic variable. If a CLIST encounters nested symbolic variables in a line, it normally scans the line 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
```

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

Concatenating Symbolic Variables

You may concatenate a symbolic variable to another symbolic variable to form a third symbolic variable.

Assume a CLIST invokes programs that are contained in data sets whose names are sequentially numbered. By concatenating &PROGRAM and &N you can use the DO-WHILE-END conditional loop structure to invoke PROGRAM1 through PROGRAM10 as follows:

```
SET N=0
SET PROGRAM = PROGRAM
DO WHILE &N¬>10
 SET N = &N+1
 call mylib(&PROGRAM&N)
END
```

By changing the value of &N in a loop, a CLIST could invoke the following set of programs without having to modify the CALL command.

```
PROGRAM1
PROGRAM2
 .
 .
PROGRAM10
```

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

```
&PROGRAM.1
```

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

```
1&PROGRAM
```

Double Ampersands and Symbolic Variables

In some situations, you may want to assign the name of another variable to a variable, or, modify the name of a variable. When a CLIST encounters a double ampersand, it removes the first one, producing a variable name. For example, to set a variable called &VARIABLE to the variable string &LINE, code:

```
SET VARIABLE = &&LINE
```

When you use double ampersands, you are telling a CLIST *not* to perform symbolic substitution on a variable.

Suppose you want to set a variable to the name of another variable and then modify the name of the other variable. For example, if you have to set

&VARIABLE to different variables such as &LINE1, &LINE2, and so on, during processing, you could code many SET statements, or code the following sequence:

```
SET NUMBER=0
SET VARIABLE=&&LINE&NUMBER /* Initialize &VARIABLE to &LINE0 */
DO WHILE &NUMBER¬>8  /* Perform processing 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
```

If you nest variables whose values contain double ampersands, only the variable that was set to the value contains the name of the 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
```

Character Set Supported in Symbolic Variables

Using CLIST I/O statements can cause characters other than those you can enter at a terminal to become part of the value of a symbolic variable. Certain hexadecimal codes are used by the system in internal processing and should not appear in data processed by CLIST I/O statements. CLISTs support all codes from x'40' through x'FF', with the understanding that lowercase characters are translated to uppercase, unless otherwise specified by NOCAPS or ASIS, and lowercase numbers (x'B0'-x'B9') are translated to standard numbers (x'F0'-'F9'). In addition, the following control characters are supported:

x'05' HT (horizontal tab)
x'14' RES (restore)
x'16' BS (backscore)
x'17' IL (Idle)
x'24' BYP (bypass)
x'25' LF (line feed)

All other codes between x'00' and x'3F' are reserved for internal processing; the use of I/O statements to process data sets containing these codes is not supported. For example, I/O statements cannot be used to process OBJ or LOAD type data sets.

Refer to *IBM System/370 Reference Summary*, GX20-1850, for the characters associated with the internal hexadecimal codes.

Control Variables

Control variables are variables which are assigned special meaning in a CLIST. Generally, they provide information about the environment during CLIST execution. When a CLIST encounters a control variable, it replaces the variable with either the information it is intended to provide or the information you have assigned to a modifiable control variable. You can assign values only to a subset of the control variables.

You code a control variable as you would a symbolic variable, for example:

```
alloc da('&SYSPREF..jcl.cntl') shr
where &SYSPREF is a control variable
```

Figure 3-3 lists the control variables in the categories in which they are described in this section.

Category	Variable	Modifiable
Current date and time	&SYSDATE &SYSJDATE &SYSSDATE &SYSTIME &SYSSTIME	No No No No No
Terminal-related	&SYSLTERM &SYSWTERM	No No
User-related	&SYSUID &SYSPREF &SYSPROC	No No No
System-related	&SYSCPU &SYSSRV &SYSISPF &SYSRACF	No No No No
CLIST-related	&SYSSCAN &SYSENV &SYSICMD &SYSPCMD &SYSSCMD &SYSNEST	Yes No No No No No
Input-related	&SYSDVAL &SYSDLM	Yes No
Output-related	&SYSOUTTRAP &SYSOUTLINE	Yes Yes
Return codes	&LASTCC &MAXCC	Yes Yes

Figure 3-3. Control Variable Categories

Figure 3-4 gives brief descriptions of the control variables you can modify.

Modifiable Variable	Contents
&LASTCC	Contains the return code from the last operation (TSO 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.
&SYSDVAL	(1) Contains the input line supplied by the user when he returned control to the CLIST after a TERMIN 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=.
&SYSOUTLINE	Contains the number of lines of command output saved in CLIST variables; points to the variables containing the output.
&SYSOUTTRAP	Contains the maximum number of lines of TSO command output to be saved.
&SYSSCAN	Contains the maximum number of times a CLIST may rescan a line to evaluate variables. The default is 16 times. The maximum value is +2,147,483,647. The minimum is 0.

Figure 3-4. Modifiable Control Variables

Figure 3-5 gives brief descriptions of those control variables you cannot modify.

Non-modifiable Variable	Contents
&SYSCPU	Contains the number of CPU seconds used during the session in the form: *seconds.hundredths-of-seconds*.
&SYSDATE	Contains the current date in the form: *month/day/year*.
&SYSDLM	Contains the character string the user entered to return control to the CLIST after a TERMIN statement.
&SYSENV	Indicates whether the CLIST is executing in the foreground or background environment.
&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*.
&SYSLTERM	Contains the number of lines available on the screen.
&SYSNEST	Indicates whether the currently executing CLIST was invoked by another CLIST.
&SYSPCMD	Contains the name (or abbreviation of the name) of the most recently executed TSO command in this CLIST.
&SYSPREF	Contains the prefix that TSO uses to fully qualify data set names.
&SYSPROC	Contains the name of the logon procedure used when the TSO user logged on.
&SYSRACF	Indicates whether the Resource Access Control Facility (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*.
&SYSSRV	Contains the number of system resource manager (SRM) service units used during the session.
&SYSSTIME	Contains the time of day in the form: *hours:minutes*.
&SYSTIME	Contains the time of day in the form: *hours:minutes:seconds*.
&SYSUID	Contains the user ID under which the current session is logged.
&SYSWTERM	Contains the width of the screen.

Figure 3-5. Non-modifiable Control Variables

Control Variables Related to the Current Date and Time

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

&SYSDATE, &SYSSDATE, and &SYSJDATE

Three variables provide the current date.

&SYSDATE provides the date in the standard form: *month/day/year.* If executed on June 27, 1983, the following statement displays the message 'Today is 06/27/83':

```
WRITE Today is &SYSDATE
```

&SYSSDATE provides the date in a sortable form: *year/month/day.* If executed on June 27, 1983, the following statement displays the message 'Today is 83/06/27':

```
WRITE Today is &SYSSDATE
```

&SYSJDATE provides the date in the Julian form: *year.days.* If executed on June 27, 1983, the following statement displays the message 'Today is 83.178':

```
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 not necessary on the WRITE statement.

```
IF &STR(&SYSDATE) = &STR(06/27/83) THEN +
 WRITE On &SYSDATE, 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
```

Control Variables that Describe Terminal Characteristics

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

&SYSLTERM and &SYSWTERM

&SYSLTERM provides the number of lines available on the 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)/
```

Control Variables Related to the User

Three control variables provide information related to the user.

&SYSUID

&SYSUID provides the user ID under which the current TSO session is logged on. You can use this variable 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('&SYSUID..records.data') shr reuse
```

Two periods are required between &SYSUID 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('userid.records.data') shr reuse
```

You may also use &SYSUID in messages and wherever logic depends on, or references, the user ID.

&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. For example, suppose a CLIST allocates many data sets that have the same identification qualifier, D1984F. To avoid having to code full qualifications for every data set name, you can ensure that TSO uses the desired prefix using &SYSPREF as follows:

```
SET SAVEPREF=&SYSPREF
IF &STR(&SYSPREF) ¬= D1984F1 THEN +
 profile prefix(D1984F1)
ELSE /* null ELSE */
(Allocations)
PROFILE PREFIX(&SAVEPREF)
```

Once the PROFILE command in the previous example is issued, all non-fully qualified data set names have D1984F1 as their identification qualifier. If you attempt to allocate non-fully qualified data sets that have identification qualifiers other than D1984F1, TSO may allocate the wrong data sets or the allocations may fail. &SYSPREF enables you to keep track of the prefix to avoid these problems.

&SYSPROC

&SYSPROC provides the name of the logon procedure used when the user logged on to the current TSO 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
```

Control Variables Related to the System

Four control variables provide information related to the system environment under which the CLIST is executing.

&SYSCPU and &SYSSRV

&SYSCPU provides the number of central processing unit (CPU) seconds used during the session in the form: *seconds.hundredths-of-seconds.* &SYSSRV provides the number of system resource manager (SRM) service units used during the session. These variables are can be used for:

- Measuring the performance of applications
- 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 CPU seconds and SRM service units used by the program PAYROLL.

&SYSISPF and &SYSRACF

&SYSISPF indicates whether or not 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

&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

Control Variables Related to the CLIST

Six control variables provide information related to 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 give &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 may 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 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 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 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 or not 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.

Control Variables Related to Input

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

&SYSDLM

&SYSDLM contains a number that identifies the position (first, second, third, and so on) of the TERMIN 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), 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

At any given time, &SYSDVAL contains one of the following:

- A null value
- The input the user entered, in addition to the character string or null line, when returning control to the CLIST after a TERMIN statement
- The user's response after a READ statement without operands
- The value assigned to &SYSDVAL with 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 character string or null line after a TERMIN 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.

Control Variables Related to TSO Command Output

Two control variables are related to TSO command output -- &SYSOUTTRAP and &SYSOUTLINE. These variables save output from TSO commands and allow a CLIST or application to display the output. You can use assignment statements to modify the values of &SYSOUTTRAP and &SYSOUTLINE.

&SYSOUTTRAP

Use &SYSOUTTRAP to specify the maximum number of lines of TSO command output to be saved for each command. If you want to save all the output from a TSO command, set &SYSOUTTRAP to a number greater than or equal to the number of output lines that the command produces.

&SYSOUTLINE

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

The CLIST uses the variable &SYSOUTLINE to record the actual number of output lines saved. This number is limited by the one in &SYSOUTTRAP; the CLIST saves no more output lines than &SYSOUTTRAP specifies.

The CLIST saves the actual command output in the variables &SYSOUTLINEnn, where 'nn' represents the positional number of the line being saved. 'nn' can be any 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.

Note: Whenever a CLIST executes a new TSO command, it resets &SYSOUTLINE to zero. However, if a CLIST invokes a non-CLIST program containing TSO commands, the program does not reset &SYSOUTLINE to zero for each TSO command. Therefore, to save command output lines in a non-CLIST program, use an assignment statement to reset &SYSOUTLINE to zero for each TSO command. See *TSO Guide to Writing a Terminal Monitor Program or Command Processor* for information on assigning a value to CLIST variables in a non-CLIST environment.

Be aware of the following considerations when using &SYSOUTTRAP and &SYSOUTLINE:

- If you try to display a line of output in &SYSOUTLINEnn where 'nn' is greater than the value of &SYSOUTTRAP, the &SYSOUTLINEnn variable contains unreliable data.
- If you try to display a &SYSOUTLINEnn variable that contains no command output, the CLIST returns a null line.
- If a TSO command produces fewer output lines than a previous command, the remaining &SYSOUTLINEnn variables retain output from the previous

command. To avoid this possibility, you can reset &SYSOUTTRAP between commands.

- Because CLISTs use the TSO EXEC command to invoke nested CLISTs, &SYSOUTLINE saves the WRITE statements of nested CLISTs as TSO command output.

- &SYSOUTLINE does not save command output sent to the terminal by a TPUT macro, such as normal or error prompting messages.

For an example of using &SYSOUTTRAP and &SYSOUTLINE when saving command output, refer to "Allocating Data Sets to SYSPROC" in Chapter 4.

Control Variables Related to Return and Reason Codes

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

&LASTCC

&LASTCC contains the return code from the last TSO 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.

&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
```

&MAXCC

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

You can use &MAXCC in conjunction 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 +
.
.
.
```

Variables Related to the Use 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, respectively, the ABEND code, ABEND reason code, and command return code produced by the most previous command invoked by the TSOEXEC command. You can use these variables in situations similar to those in which you would use &LASTCC and &MAXCC. For example, to determine if the TRANSMIT command terminated abnormally, you could code:

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

Built-in Functions

Built-in functions enable you to perform certain functions on variables, expressions, and character strings. A CLIST evaluates the variable or expression first, if necessary, and then performs the requested function. The built-in function is then replaced by the result of the evaluation.

A built-in function name (identifier) is followed by an argument, enclosed in parentheses, upon which a particular predetermined function is performed. The name identifies the function. Blanks are not allowed between a built-in function identifier and its argument.

The built-in functions are described in individual topics in this section. A short description of each of them follows:

Built-in Function	Function
&DATATYPE(expression)	Indicates whether the evaluation of *expression* is a character string or a numeric value.
&EVAL(expression)	Performs an arithmetic evaluation of *expression*.
&LENGTH(expression)	Evaluates *expression* if necessary and indicates the number of characters in the result.
&NRSTR(string)	Preserves the integrity of a character string.
&STR(string)	Defines data to be used as a character string.
&SUBSTR(exp[:exp],string)	Uses part of a character string.
&SYSCAPS(string)	Translates the string to uppercase characters.
&SYSDSN(dsname[(member)])	Indicates whether the specified data set exists.
&SYSLC(string)	Translates the string to lowercase characters.

Determining the Data Type of an Expression - &DATATYPE

Use the &DATATYPE built-in function to determine whether the evaluation of a given expression is a numeric value or a character string. After evaluating the expression, a CLIST replaces this built-in function with either the string CHAR or the string NUM. The strings indicate the following:

- CHAR -- The evaluated expression contains at least one non-numeric character.
- NUM -- The evaluated expression is entirely numeric.

The following examples show the evaluations of various expressions:

- &DATATYPE(ALPHABET) - CHAR
- &DATATYPE(1234)- NUM
- &DATATYPE(SYS1.PROCLIB) - CHAR
- &DATATYPE(3*2/4) - NUM
- &DATATYPE(12.34) - CHAR

For example, the following clause evaluates as true:

```
IF &DATATYPE(12.34)=CHAR 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 - &LENGTH

Use the &LENGTH built-in function to determine the number of characters 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 characters in the result of the addition, 11.

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

&LENGTH may 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.

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 zeroes 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))
```

Preserving the Integrity of a Character String - &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

Also, you can use &NRSTR when performing I/O to data sets that contain JCL statements that include the name of a temporary data set (for example, &&A) or a symbolic parameter (for example, &LIBRARY). Using &NRSTR prevents a CLIST from:

- changing the name of a temporary data set (&&A) to a symbolic parameter (&A)
- performing erroneous symbolic substitution on a symbolic parameter (&LIBRARY)

In either case, the use of &NRSTR prevents the subsequent execution of the JCL statement from causing a JCL error.

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 reading or writing records that contain JCL statements.

Temporary Data Set Names: If a JCL statement contains a temporary data set name (for example, &&A), enclose the statement in a &NRSTR built-in function to prevent a CLIST from removing the first ampersand. The following CLIST writes a JCL statement containing a temporary data set name to a data set.

```
 .
 .
alloc f(to) da(data1.ctl) shr
OPENFILE TO OUTPUT
SET TO = &NRSTR(//DD3 DD DSN=&&A(ADD),UNIT=3350,+
           DISP=(OLD,KEEP),VOL=SER=MYOWN2)
PUTFILE TO
CLOSFILE TO
 .
 .
```

When reading the same statement from the data set and writing it to a different data set, you can code the following CLIST to ensure the integrity of the temporary data set name.

```
 .
 .
alloc f(from) da(data1.ctl) shr
alloc f(to) da(data2.ctl) shr
OPENFILE FROM INPUT
OPENFILE TO OUTPUT
 .
 .
GETFILE FROM
SET TO=&NRSTR(&FROM)  /* Do not change the record
PUTFILE TO  /* Write it as is
CLOSFILE FROM
CLOSFILE TO
 .
 .
```

Symbolic Parameters: If a JCL statement contains a symbolic parameter (for example, &LIBRARY), you can use &NRSTR to prevent a CLIST from performing erroneous symbolic substitution. Assume a record contains the following JCL statement:

```
//DO2 DD DSN=&LIBRARY,DISP=(OLD,KEEP),UNIT=3400,VOL=SER=MYOWN1
```

You can use the same CLIST code shown in the previous example under 'Temporary Data Set Names' to read the statement from one data set and write it to another data set without having the CLIST perform symbolic substitution on the symbolic parameter &LIBRARY. (The single substitution substitutes the JCL statement for the variable &FROM.)

However, if you wanted to write the statement directly to a data set from a CLIST or do some processing on the statement within a CLIST, use the &SYSSCAN control variable in place of the &NRSTR built-in function to prevent symbolic substitution.

Defining Character Data - &STR

Use the &STR built-in function to define character data. The data may be any expression or statement. Nested variables are also permitted in &STR built-in functions.

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 could not make the desired assignment because a CLIST would 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. In order to preserve the character string, code the following:

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

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 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 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 would 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)
//&SYSUID.1 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 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 would not be necessary because the program's statements would appear in the data set containing MATRIX. Thus, they would be associated with the program, not the CLIST.

Defining a Substring - &SUBSTR

Use the &SUBSTR built-in function to request that a CLIST recognize only part 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 may 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 last name. To add a blank between the initials and the last 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 last name

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

If you want to substring only one position, the colon and end-expression may be omitted. For example, if you are interested only in the first letter of the last 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 substringed data represents a part number.

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

Translating Character Strings to Uppercase Characters - &SYSCAPS

Use &SYSCAPS to translate character input strings to uppercase characters. A CLIST does not modify numbers, national characters, or special characters included in the data string. You may use variables containing the character strings in &SYSCAPS built-in functions.

You can use &SYSCAPS in conjunction with &SYSLC to control the capitalization of text in a CLIST. For an example, refer to "Controlling Uppercase and Lowercase in READ Statement Input" later in this chapter.

Determining Whether a Data Set Exists - &SYSDSN

Use the &SYSDSN built-in function to determine whether or not a specified data set exists or a specified data set and member exist. &SYSDSN can return one of the following values:

- OK -- data set exists or data set and member exist
- 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, xxxx
- MISSING DATA SET NAME

For example, you can use the &SYSDSN built-in function in conjunction with conditional logic 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 quotes when they appear in &SYSDSN built-in functions. You may use variables containing data set names in &SYSDSN built-in functions.

Translating Character Strings to Lowercase Characters - &SYSLC

Use &SYSLC to translate character strings to lowercase characters. A CLIST does not modify numbers, national characters, or special characters included in the string. You may 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.

Prompting for Input

A CLIST can prompt for input in a number of different ways:

- using positional parameters and keywords with associated values on a PROC statement
- using WRITE and WRITENR statements
- using TSO commands

PROC Statement

When you include positional parameters on a PROC statement, the invoker must supply a value for each of them. If the invoker does not specify a value at invocation, the CLIST prompts until the invoker specifies one.

When you include keywords that require associated values on a PROC statement and the invoker specifies only the keyword at invocation, the CLIST prompts until the invoker specifies the value.

For details on the use of the PROC statement, refer to "Parameter Definitions the PROC Statement" in this chapter.

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. For details on the use of the WRITE and WRITENR statements, refer to "Communicating with the Terminal User" in this chapter.

TSO Commands

Some TSO commands, for example LISTDS, require more information than just the name of the command and prompt when that information is not supplied. However, TSO commands included in a CLIST can prompt for input only when the CLIST allows prompting, which is controlled by PROFILE and EXEC, TSO commands, and by CONTROL, a CLIST *statement.*

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.

	Prompting by TSO commands allowed in CLIST	
Specifications	**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

Notes:

- PROFILE PROMPT is the default specification and applies to a TSO session not just to a particular CLIST. The explicit specification of PROFILE PROMPT is unnecessary unless needed to override a prior PROFILE NOPROMPT command.
- The PROFILE command may be executed either outside of, or within, a CLIST.

- EXEC NOPROMPT is the default specification and applies only to the CLIST that it invokes.
- The CONTROL statement applies only to the CLIST in which it appears.
- If a CONTROL statement does not appear in a CLIST, CONTROL NOPROMPT is implied.

The DATA PROMPT-ENDDATA Sequence

Use the DATA PROMPT-ENDDATA sequence in a CLIST to designate responses to prompts by TSO commands, subcommands, or READ statements.

To use the DATA PROMPT-ENDDATA sequence, code:

```
DATA PROMPT
 .
 .  /* Responses */
ENDDATA
```

If the sequence is not immediately preceded by a TSO command or subcommand that prompts, or by a READ statement, an error occurs (error code 968). You can ignore the error condition if a command or subcommand that *could* prompt, does not prompt.

The responses in the DATA PROMPT-ENDDATA sequence must appear exactly as if entered by the user. Each DATA PROMPT-ENDDATA sequence can respond only to prompts issued by the immediately preceding command, subcommand, or READ statement. 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.

Some TSO 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 prompts for control statements. If you include such a LINK command in a CLIST that is run in the background, place the control statements within a DATA PROMPT-ENDDATA sequence. The following CLIST link edits the member *X*, which resides in the file *DD1*:

```
SET NULL =
CONTROL PROMPT LIST
link (*) /* Prompt terminal for control statements */ +
 load('d32kds1.load') pr(*) ncal xref list let
DATA PROMPT
include dd1(x)
entry x
name x
ENDDATA
&NULL
```

There are additional considerations for using the DATA PROMPT-ENDDATA sequence:

- The CLIST must allow prompting, except in the case of the READ statement
- The CLIST performs symbolic substitution before using the responses to satisfy the prompt. (You may include variables in the responses.)

Parameter Definitions - - the PROC Statement

The PROC statement enables both the CLIST and its invoker to assign values to variables by CLIST invocation, prompting, and default parameter values. The PROC statement may contain positional parameters, keyword parameters, and keyword parameters with associated values. When used, the PROC statement must precede all other executable statements and commands. However, comments or blank lines may precede the PROC statement.

When you specify positional parameters on the PROC statement, the CLIST needs to know the number of them. To inform the CLIST, specify the number as the first operand on the PROC statement. If you do not specify any positional parameters, just keyword parameters, specify the number 0.

Positional Parameters: Use positional parameters when the CLIST requires input from the invoker that may vary from one invocation to the next. For each positional parameter name, the CLIST defines a variable of the same name and initializes it with the value entered for the parameter. As a result, positional parameter names have the same syntax rules as symbolic variables.

Positional parameter values may be up to 252 alphameric characters in length. The invoker enters only a value for the parameter because the CLIST identifies it by its position, not its name. The CLIST assigns the values to the positional parameters sequentially. As a result, the invoker must know the order of the parameters on the PROC statement so that the CLIST assigns the values entered to the correct variables. If the invoker does not supply values for all positional parameters, the CLIST prompts for the data.

Assume a CLIST called BUDGET resides in a data set called PROC.CLIST and that it references an account number to perform its processing. If account numbers could be different for different invocations, code the following:

```
PROC 1 ACCT
```

The "1" indicates that the PROC statement contains one positional parameter, ACCT.

If BUDGET is invoked explicitly, the invoker must enclose the account number parameter in single quotes as follows:

```
ex proc(budget) 'd5880p'
```

If the CLIST is invoked implicitly, the invoker does not enclose the positional parameter in single quotes, for example,

```
%budget d5880p
```

or

```
budget d5880p
```

If the invoker enters only the name of the CLIST on either an explicit or implicit invocation, the CLIST prompts for an account number. Regardless of how the CLIST was invoked, the missing information may then be entered without single quotes.

Keyword Parameters: Use keyword parameters when the input is either optional or capable of having a default value. Keywords do not have the same syntax rules as symbolic variables. Keyword parameters must be from one to 31 characters in length, and they may or may not have associated values.

For each keyword parameter on a PROC statement, a CLIST defines a variable of the same name. If the keyword does not have an associated value and the invoker supplies the keyword name, the CLIST initializes the variable to the keyword name. If the invoker does not supply a keyword name, the CLIST initializes the variable to a null value.

If the keyword has an associated value, the CLIST initializes the variable to the value supplied by the invoker or to a default value.

Using Keyword Parameters: Use a keyword parameter to enable the invoker to indicate if the CLIST should perform a certain action.

The invoker may include the *name* of the keyword or omit it. On the invocation, the CLIST does not recognize an abbreviation of the name.

To enable the invoker to indicate whether the results of BUDGET should be printed, you could code the following:

```
PROC 1 ACCT PRINT
```

The *1* indicates that there is only one positional parameter. ACCT is the positional parameter; PRINT is a keyword parameter. The CLIST may perform a check on the variable &PRINT to see if the invoker wants the results printed. You can code this check using an IF-THEN-ELSE sequence.

```
IF &PRINT= PRINT THEN (Print results)
```

Another use for keywords involves options on CLIST statements. Suppose you want the invoker of BUDGET to be able to indicate whether to display TSO commands and CLIST statements at the terminal. You could code the following:

```
PROC 1 ACCT PRINT DEBUG
IF &DEBUG=DEBUG THEN +
CONTROL LIST CONLIST SYMLIST
```

If the invoker includes DEBUG, the CLIST displays TSO commands and CLIST statements. If the invoker does not include DEBUG (&DEBUG has a null value), the CLIST does not display TSO commands and CLIST statements.

Using Keywords with Associated Values: Use a keyword with an associated value if the input, for example an account number, is optional. The length restriction for subparameter values is the same as that for symbolic variable values.

You could code the following:

```
PROC 0 ACCT()
```

If the invoker of the CLIST wants to supply an account number, the invoker enters the word *ACCT* along with the number in parentheses. If the invoker enters only the word *ACCT*, the CLIST prompts for the account number. If the invoker omits the word *ACCT*, the variable has a null value.

Another case occurs when there is a standard account number that applies to most, but not all, invokers of the CLIST. You can supply a default value that the CLIST uses if the invoker does not supply a value. For example:

```
PROC 0 ACCT(D5880P)
```

Regardless of the default value, if the invoker enters *ACCT(D90)* on the invocation of the CLIST, the value of &ACCT becomes D90.

Special Considerations for Values Containing Single Quotes: If the invoker of a CLIST wants to pass values containing single quotes (or apostrophes), the invoker must adhere to certain guidelines that are shown in Chapter 5 under the EXEC command.

Communicating with the Terminal User

The WRITE, WRITENR, READ, READDVAL, and TERMIN statements provide a means of communication between a CLIST and the terminal user:

- The WRITE and WRITENR statements issue messages to the user, usually stating why the user received control, and prompt for input.
- The READ statement reads user input either into variables included on the READ statement or into the &SYSDVAL control variable when no variables are included on the READ statement.
- The READDVAL statement accesses the contents of the &SYSDVAL control variable.
- The TERMIN statement causes a CLIST to pass control to the user. TERMIN suspends CLIST execution until the user indicates that execution should resume. The statement also defines character strings, any of which the user may enter to return control to the CLIST. TERMIN statements commonly follow WRITE or WRITENR statements.

Writing Messages to the Terminal User - WRITE and WRITENR

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

You can use either statement to send messages. You may find WRITENR preferable when the message prompts the user for input. 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 ınvalid.
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. Since 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.
```

Since CONTROL CAPS is specified, the message is translated to all capital letters before being displayed.

Reading Input from the Terminal - READ and READDVAL

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 or from the DATA-PROMPT-ENDDATA sequence. 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 may 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 could 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 READ 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 on 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 would all 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 quotes). For

example, assume that the CLIST sends a message to the user requesting four successive 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 quotes 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 quotes (according to TSO 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 quotes preserved, by using the READ statement with the &SYSDVAL control variable.

The following CLIST uses a READ statement and &SYSDVAL to preserve single quotes around a data set name. It also checks for the quotes to see if the user entered a fully qualified data set name and, if not, adds the quotes 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 data set is not fully qualified,
  SET &DSN = '&SYSPREF..&DSN'               /* add prefix and quotes */
 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 not
 WRITE acceptable.
 READ ALPHA
 SET SPACEVENTS = &ALPHA
END
```

Controlling Uppercase and Lowercase for READ Statement Input

To control uppercase and lowercase for READ statement input, use of the CAPS/ASIS/NOCAPS operand on the CONTROL statement, as well as the &SYSLC and &SYSCAPS built-in functions. The CAPS/ASIS/NOCAPS operand indicates whether the CLIST should translate *all* READ statement input to uppercase characters. (The CLIST does not modify numbers, national characters, or special 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
```

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
```

&SYSLC and &SYSCAPS enable you to tailor individual strings as well as substrings of input strings.

For example, a CLIST that prompts for first, middle, and last names, and saves the data for inclusion in a report, may want to guarantee that the name is properly capitalized before saving it in a variable. The following section of code shows a way to do so:

```
CONTROL ASIS /* Do not translate all 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 - TERMIN"
- Information that the CLIST explicitly placed into &SYSDVAL with an assignment statement

&SYSDVAL is updated only when a CLIST:

- Executes a READ statement without operands
- Executes a TERMIN statement
- Explicitly modifies &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 the symbolic variables listed on the following statement:

```
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 their original values.

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 not enough 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 - TERMIN

The TERMIN statement transfers control to the terminal and establishes a means for the user to return control to the CLIST.

Note: If a CLIST containing a TERMIN statement is executed under ISPF, the TERMIN statement terminates the CLIST. As a result, users should not invoke such a CLIST from ISPF unless termination at that point is acceptable.

The TERMIN 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 statement normally does not function alone. WRITE statements preceding the TERMIN 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 enables the user to enter commands or subcommands, and invoke programs before responding to the WRITE statement prompts.

As soon as the CLIST executes the TERMIN statement, the user receives control. The user might or might not receive a mode message after the TERMIN statement executes. 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 may think the CLIST has terminated. You may want to avoid any confusion by telling them otherwise in the WRITE statement that precedes the TERMIN statement.)

Returning Control After a TERMIN Statement

Code the TERMIN 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 contains 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 statement. Multiple strings enable the user to indicate desired actions to the CLIST.

You may allow a null line as one of the valid strings but it must be the first string on the TERMIN 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 one of the following:

- null line (pressing the ENTER key)
- PROCESS
- TERMINATE

You can issue a TERMIN 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 command processors use null lines as function delimiters (for example, to switch between input and edit modes under EDIT).

Entering Input After a TERMIN 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 upper case, 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 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
```

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 . To request that they be displayed:

```
CONTROL MSG
```

To suppress the display of informational messages, code:

```
CONTROL NOMSG
```

The MSG/NOMSG option has no effect on error messages, they are always displayed.

Structuring CLISTs

You may structure a CLIST using the following CLIST functions:

- DO-groups
- DO-WHILE-END sequences
- IF-THEN-ELSE sequences
- GOTO statements and labels for branching

DO-Groups

DO-groups enable you to organize the instructions in a CLIST into groups to be executed conditionally. The DO-END sequence appears only in an IF-THEN-ELSE sequence and is described in "Making Decisions - the IF-THEN-ELSE Sequence." The DO-WHILE-END sequence is a DO-group that executes repeatedly while a condition is true.

Creating Loops - The DO-WHILE-END Sequence

The DO-WHILE-END sequence is the loop structure in the CLIST language. A loop is a programming structure that repeats instructions as long as a condition is true. When the condition becomes false, the loop ends and execution continues at the instruction after the loop.

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 may code multiple conditions, in which case the comparative expressions (and/or variables) must be joined by logical operators.

The *action* can be one or more instructions. The CLIST executes the instructions within the sequence repeatedly *while* the condition included 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 (usually 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 GT 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 causes a test of the value of this counter each time the CLIST begins to execute the DO-WHILE-END sequence. As long as 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 bypasses the action, and continues processing at the instruction following the END statement.

Nesting DO-WHILE-END Sequences

You can nest DO-WHILE-END sequences within other DO-WHILE-END sequences. The condition governing the execution of the action of the nested sequence may or may not be the same as that governing the execution of the sequence in which it is nested. The following example includes a nested DO-WHILE-END sequence whose conditional action is the same as that of the sequence in which it is nested.

```
SET &COUNTER = 5 /* Initialize counter
/* Perform the action while counter is greater than 0 */
DO WHILE &COUNTER GT 0
 .
 .  (Set of instructions)
 DO WHILE &COUNTER GT 3
  .
  . (Subset of instructions)
  .
 SET COUNTER = &COUNTER - 1 /* Decrease counter by 1 */
 END
 .
 .
 SET COUNTER = &COUNTER - 1 /* Decrease counter by 1 */
END
```

Making Decisions - 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 (and/or variables) must be joined by logical operators.

The *action* can be one or more instructions. If the condition(s) is true, the CLIST executes the instructions in the THEN action. If the condition(s) 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. (If an action involves only one statement or command, it is not necessary to use the DO-group.)

```
IF condition THEN +
 DO
  action
 END
ELSE action
```

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 invoke the CLIST      */
/* 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
 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 could 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 statement in one of two ways: omit the ELSE statement entirely or just code ELSE without operands (an action). The following IF-THEN-ELSE sequence performs the desired processing:

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

You can also code the following:

```
IF &PRINT=YES THEN
 DO
  WRITE We are printing the data set as you requested.
  %printds
 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 statement that effectively branches to the end of the ELSE statement, avoiding the code that prints the data set.

The following IF-THEN-ELSE sequence performs the desired processing:

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

Distinguishing END Statements from END Commands or Subcommands

When you include END commands or subcommands in the action of a DO-group, a CLIST allows you to distinguish the END commands or subcommands from the END statement.

The CONTROL Statement

One way to distinguish between an END statement and an END command or subcommand is by coding a CONTROL statement with the END operand. The value you code for the END operand may then be substituted for the END statement in the DO-group. Once you have established this value in a given CLIST, use it to end all DO-groups unless another CONTROL END statement 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
  .
  . (Set of instructions)
  .
  test 'datapak(newpgm)'
    .
    . (TEST subcommands)
    .
  end
  .
  . (more instructions)
  .
 SET COUNTER = &COUNTER - 1 /* Decrease counter by 1 */
ENDO
```

The DATA-ENDDATA Sequence

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

```
SET COUNTER = 10
DO WHILE &COUNTER GT 0
  .
  . (Set of instructions)
  .
  DATA
    test 'datapak(newpgm)'
    .
      (TEST subcommands)
    .
    end
  ENDDATA
  .
  . (more instructions)
  .
 SET COUNTER = &COUNTER - 1 /* Decrease counter by 1 */
END
```

Only TSO commands and subcommands can appear within the DATA – ENDDATA sequence. If a CLIST statement is included, TSO attempts to execute it as a TSO command, causing an error.

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. You cannot use a GOTO statement to branch to another CLIST. Examples of using GOTO statements are:

```
    IF &A = 555 THEN GOTO A1
    ELSE GOTO A2
A1:  processing
     .
     .
A2:  processing
     .
     .
```

```
    SET TARGET = B1
    IF &X = 666 THEN GOTO &TARGET
    ELSE +
      DO
       SET TARGET = B2
        .
        .
       IF LASTCC = 0 THEN +
       SET TARGET = B1
       ELSE GOTO &TARGET
      END
B1:  processing
B2:  processing
     .
     .
```

Nested CLISTs

A CLIST may invoke another CLIST, which in turn may invoke another, and so forth. CLISTs that are invoked by other CLISTs are called nested CLISTs. A nested CLIST 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 levels of CLISTs in the same way that you can design complex programs with main routines and subroutines. 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 3-6, PROC1 is the outer-level CLIST. It invokes PROC2 and PROC3, which are nested within it. PROC2 invokes PROC4, and PROC4 invokes PROC5. PROC4 is nested within PROC2, and PROC5 within PROC4.

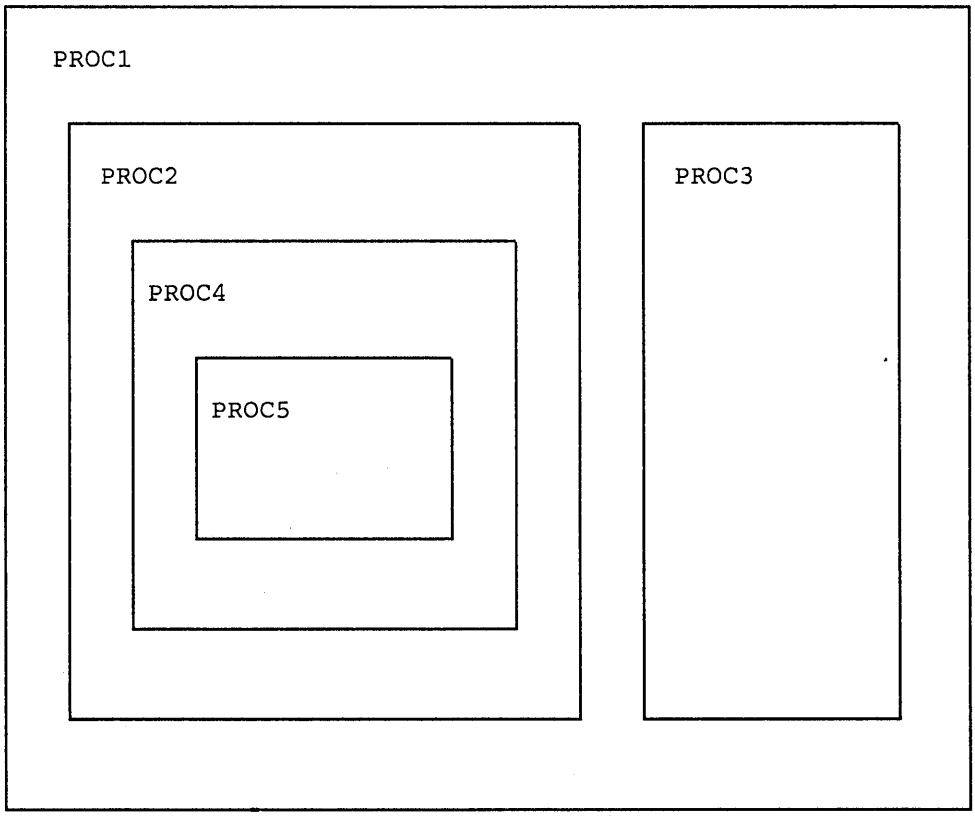


Figure 3-6. Nested CLISTS

If, for a given invocation, PROC2 invokes PROC4 and PROC5, it must do so before returning control to PROC1. PROC1 cannot invoke PROC3 until PROC2 finishes 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.

Any special options established by a nested CLIST are in effect only when that nested CLIST is executing. In particular, CONTROL statement options and ATTN exits are no longer in effect when a nested CLIST returns control to the CLIST that called it.

Nested CLISTs in the subcommand environment may execute only subcommands and CLIST statements. They cannot execute TSO commands. A CLIST at any given level determines the execution environment (command or subcommand) for the CLISTs nested at all levels beneath it.

Global Variables

Global variables are variables defined on a GLOBAL statement. They are designed to 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 3-6, 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 amongst 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 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. You cannot set a return code. To use the END command, code:

```
end
```

Using the EXIT Statement

To cause a nested CLIST to return control to the CLIST that invoked it (one level upward), 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 encountered during execution.

To pass a return code when you exit, code:

```
EXIT CODE(expression)
```

The expression must be an integer or a symbolic variable whose value, after substitution, is an integer. The CLIST stores the value of the expression into the control variable &LASTCC.

If the proper control options are in effect, you can also have control passed back to a CLIST that is protected from having an error or attention interrupt terminate its execution. 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 on writing error and attention routines, refer to "ATTN and ERROR ROUTINES."

If no CLIST in the nested chain is protected from being terminated after an error or an attention interrupt, coding QUIT causes TSO to receive control. When this situation occurs, the user sees a READY message indicating that TSO has returned control to the terminal.

Performing I/O

CLISTs can perform I/O to a physical sequential data set or a member of a PDS. CLISTs can also perform I/O to the directory of a PDS if the record format is not variable (V or VB). Four CLIST statements are available for opening, accessing, and closing data sets:

- OPENFILE opens a previously allocated data set (file) for input, output, or updating. You may have allocated the file using the TSO 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 you perform I/O, include an error routine that can handle end-of-file conditions and errors that may occur; for example, during pre-allocation or allocation.

When performing I/O to a physical sequential data set that has a block size of 80 and a record format of U, TSO truncates the last character in the record. (VTAM/TCAM uses the last byte for an attribute character notation.)

Opening a File

To open a file, use an OPENFILE statement that includes the file name of the data set. Code either the name included on the FILE keyword of the ALLOCATE command that allocates the data set; or if you used step allocation, the ddname of the JCL statement that allocated the data set. If you use the ALLOCATE command, include the FILE keyword or you cannot open the data set for I/O. On the allocation, create the file name; it is an arbitrary value.

For example, you can code the following:

```
.
.
alloc 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
.
.
alloc file(&FILEID) da('d58tan1.checks.data') shr
OPENFILE &FILEID
.
.
```

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.

When coding the file name on the CLOSFILE, statement, code the same file name as that specified on the associated OPENFILE statement. For example, if you opened a file by coding:

```
OPENFILE &FILEID
```

close that file by coding:

```
CLOSFILE &FILEID
```

For examples of CLOSFILE, refer to the examples in the following two sections.

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.

Assume a data set called D58TAN1.CHECKS.DATA has a variable-blocked record format and contains the following records:

```
200BLACKBUY
449REFY
450YARRUM
```

To read the records into three variables, you could code the following:

```
 .
 . (Error routine)
 .
alloc file(paycheks) da('inst.emp.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 */
```

Writing a Record to a File

To write a record to an open file, use a PUTFILE statement. After issuing the OPENFILE statement but before issuing PUTFILE, create a variable of the same name as the file name and place into it the record you are writing to the data set.

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, 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.CURRENTSAL.DATA, you could code the following:

```
alloc file(salaries) da('d58tan1.currentsal.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, 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)
 .
alloc 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 */
```

Note: On a GETFILE or PUTFILE statement, code the actual **filename**; DO NOT code the file name as a symbolic variable. You can, however, use a

symbolic variable on the OPENFILE and CLOSEFILE statements, as in the following example:

```
 .
 .
 .
SET FILEID = PAYCHEKS
 .
 .
 .
alloc f(&FILEID) da('d58tan1.checks.data') shr
OPENFILE &FILEID
 .
 .
 .
GETFILE PAYCHEKS
 .
 .
 .
PUTFILE PAYCHEKS
CLOSFILE &FILEID
```

Special Considerations for Performing I/O on Records Containing JCL Statements

If a CLIST reads or writes records containing JCL statements, that CLIST could make incorrect modifications to the statements by symbolic substitution. To prevent the incorrect modifications, you can use the &STR built-in function or the &SYSSCAN control variable. Refer to "Preserving the Integrity of a Character String - &NRSTR" and &SYSSCAN for details and example.

End-of-File Processing

Whenever you perform I/O, you should 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. For a complete description of how to write an error routine, refer to "Error Routines" later in this chapter.

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
    free f(paycheks)
    WRITE No record to update because end-of-file was reached.
    SET EOF=ON
    RETURN
   END
 END
alloc 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
```

ATTN and ERROR ROUTINES

Two types of actions 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 within the CLIST to handle attention interrupts and errors. They are ATTN and ERROR respectively. The ATTN statement is described in "Attention Routines." The ERROR statement is described in "Error Routines."

An attention interrupt occurs when the user presses the key (usually PA1 or ATTN) on the terminal keyboard and suspends execution of a program. The user may enter an attention interrupt for any number of reasons, such as to terminate an infinite loop or simply to end the CLIST.

An error can occur for any number of reasons, such as a numeric value that exceeds 2^{31}-1, an end-of-file condition, a non-zero return code from a TSO command, etc.

Attention Routines

Use the ATTN statement to create an attention routine that defines an action to be taken when the user enters an attention interrupt.

To create an attention routine, code:

```
ATTN action
```

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 command, TSO subcommand, or null line may be executed. If the one TSO command executed is an invocation of an attention handling CLIST, you may execute as many TSO commands or subcommands as you wish in the attention handling CLIST.

You should inform the user at the beginning of the attention routine that TSO is processing the attention interrupt. Otherwise, the user may enter another attention interrupt. For a description of how TSO processes multiple attention interrupts, refer to *TSO Guide to Writing a Terminal Monitor Program or a Command Processor.*

Cancelling 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 previously established attention routine. Do not use ATTN OFF within an attention routine.

You may also code attention routines that override previous ones. You may initialize new attention routines as many times as you wish. Each attention routine overrides all previous ones.

Unless the action terminates the CLIST, it must execute a RETURN statement. 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.

Protecting the Input Stack for Attention Routines

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 obtains its input (TSO commands, CLIST statements).

For nested CLISTs, the input stack holds the contents of the CLISTs in the order in which they are nested.

If you write an attention routine that does anything other than 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 that must appear before the

attention routine. The MAIN operand indicates that the CLIST is the main CLIST in the invoker's TSO environment and automatically prevents TSO from flushing the input stack in the event of an attention interrupt.

If you use global variables in both a main CLIST and one that is invoked by the ATTN action, code CONTROL MAIN in the main CLIST so that the global variables can be maintained.

Attention routine processing depends on whether or not CONTROL MAIN has been coded, as well as whether the routine executes a null line. If CONTROL MAIN is not in effect, the CLIST terminates and the user sees the READY message, indicating that control has returned to the terminal. If CONTROL MAIN is in effect, and a command is issued, the CLIST ultimately continues at the statement or command following the one that was executing when the user entered the attention interrupt. If CONTROL MAIN is in effect, and a null line executes in the attention routine, the CLIST continues at the point where the attention occurred, which may not be the next command or statement.

The ALLOCATE CLIST shown in Figure 3-7 contains an attention routine that prompts the user to indicate whether he wants to terminate the CLIST. If the user responds affirmatively, the CLIST determines whether any data sets have been allocated, and, if so, invokes a CLIST called HOUSKPNG to free allocated data sets. HOUSKPNG determines which data sets to free by referencing the CLEANUP global variable. CLEANUP contains the number one, two, or three. The CLIST containing the attention routine frees the data sets in the inverse order of that in which it allocates them. The HOUSKPNG CLIST is shown in Figure 3-8.

THE ALLOCATE CLIST

```
/*****************************************************************/
/* 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)
CONTROL PROMPT
GLOBAL DS1 DS2 CLEANUP
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 NOFLUSH MAIN  /* return control to the next CLIST    */
                             /* instruction after HOUSKPNG finishes */
   IF &FOOTPRINT = YES THEN +
     %houskpng
   ELSE
     DO
       SET &NULL =
       &NULL                 /*issue null line */
     END
  STOP
alloc f(input) da(&ds1..text) shr reu
SET FOOTPRNT = 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 FOOTPRNT =   /* Set FOOTPRNT back to null */
```

Figure 3-7. A CLIST Containing an Attention Routine - The ALLOCATE CLIST

THE HOUSKPNG 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 GLOBAL VARIABLE         */
/* CLEANUP, THE CLIST FREES FROM ONE TO THREE OF THE DATA SETS   */
/* ALLOCATED IN ALLOCATE.                                        */
/*****************************************************************/
CONTROL END(ENDO)
ATTN +
 EXIT QUIT
GLOBAL DS1 DS2 CLEANUP
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 3-8. An Attention Handling CLIST - The HOUSKPNG CLIST

Note that the ATTN action in Figure 3-7 itself issues only one TSO command: *%houskpng* or the null line. However, when HOUSKPNG is invoked, one to three commands are issued.

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 nonzero return code. (Figure 5-1 lists the CLIST error codes.) The action is 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 commands and subcommands, subject to the mode in which the CLIST is executing when the error occurs. An error routine action is not limited to issuing only one TSO command or subcommand, as is an attention routine.

Unless the action terminates the CLIST, it must execute a RETURN statement. The RETURN statement returns control to the CLIST statement, TSO command,

or TSO 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. You may initialize new error routines as many times as you want. Each error routine overrides all previous ones.

Cancelling Error Routines

To cancel the most previously established error routine in a CLIST, code either:

```
ERROR OFF
```

or

```
ERROR
```

following the error routine to be cancelled. For ERROR OFF, the CLIST continues execution without any error processing.

For ERROR, the CLIST continues execution without any error processing but displays the statement, command, or subcommand that caused the error, together with explanatory error messages. After displaying the information, the CLIST attempts to continue execution with the next sequential statement, command, or subcommand.

Protecting the Input Stack for Error Routines

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 obtains its input (TSO 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 with the NOFLUSH or MAIN operand. The CONTROL statement must appear before any error routine. preferably at the beginning of the CLIST.

Using Error Routines

The COPYDATA CLIST, shown in Figure 3-9, 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.

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
  IF &RCODE=400 THEN GOTO EOF
  /* If error occurred before allocation, set exit code to 4
  IF &FOOTPRINT=0 THEN SET ECODE=4
  /* If allocation of file OUTDS failed, free file INDATA and set
  /* exit code to 8
  IF &FOOTPRINT=1 THEN +
   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
  IF &FOOTPRINT=2 THEN +
   DO
    free f(indata) da(text.data)
    free f(outds)
    SET ECODE=12
   ENDO
  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 1=1 /* Use infinite loop to reach EOF
 GETFILE INDATA /* Read input record
 SET OUTDS=&INDATA /* Set output record to value of input record
 PUTFILE OUTDS /* Write output record to output data set
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
 .
 .
```

Figure 3-9. The COPYDATA CLIST

Chapter 4. Implementation

This chapter contains examples of CLISTs that illustrate the implementation of the CLIST tools discussed in Chapter 3. The examples assume that the CLISTs reside in a PDS allocated to SYSPROC.

Figure 4-1 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.

CLIST	Function
LISTER	Including TSO commands
DELETE	Simplifying routine tasks
CALC	Creating arithmetic expressions from user supplied input
CALCFTND	Performing front-end prompting
SCRIPTDS	Initializing and invoking system services
SCRIPTN	Invoking CLISTs to perform subtasks
SUBMITD	Including JCL; performing front-end prompting
SUBMITFQ	Performing substringing; adding flexibility
RUNPRICE	Allowing foreground or background submittal of jobs
TESTDYN	Providing invoker with options and performing initialization based on options specified
COMPRESS	Simplifying routine, system-related tasks
CASH	Simplifying invoker's interface to complex applications
PHONE	Performing I/O; reading records into &SYSDVAL
SPROC	Using &SYSOUTTRAP and &SYSOUTLINE variables to manage command output
PROFILE	Using ISPF dialog management services in CLISTs to create full-screen applications

Figure 4-1. CLIST Examples and Their Functions

Including TSO Commands - The LISTER CLIST

You can organize related activities so that users can simply invoke a CLIST to perform a given task or group of tasks. The simplest example is a CLIST that groups TSO commands together.

The LISTER CLIST consists of two TSO commands. (See Figure 4-2.) 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 userid. TSO 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.

THE LISTER CLIST

```
listc
lista st
```

Figure 4-2. The LISTER CLIST

Simplifying Routine Tasks - The DELETE 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 could confuse users who simply want to delete some of their data sets. For those users, you could write a CLIST that simplifies the process. The DELETE CLIST shown in Figure 4-3 is an example of such a CLIST. It prompts the invoker for a data set name or a completion indicator.

THE DELETE CLIST

```
/*********************************************************************/
/*   THIS CLIST (DELETE) 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 quotes.
 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 4-3. The DELETE CLIST

Creating Arithmetic Expressions from User-Supplied Input - The CALC CLIST

The CALC CLIST, shown in Figure 4-4, 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, invalid input causes the &EVAL built-in function to fail and generate an error code.

THE CALC CLIST

```
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 4-4. The CALC CLIST

Using Front-End Prompting - The CALCFTND CLIST

Front-end prompting verifies the data before the CLIST uses it in other statements. For example, the CALC CLIST 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 Figure 4-5, 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 INVALID, CONTINUE PROMPTING THE USER TO ENTER  */
/* AN ACCEPTABLE VALUE.                                         */
/*************************************************************/
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 INVALID, 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 INVALID, 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)
```

Figure 4-5. The CALCFTND CLIST

Initializing and Invoking System Services - The SCRIPTDS CLIST

A user can invoke the SCRIPTDS CLIST to run the SCRIPT program against an input data set and have the output printed.

As shown in Figure 4-6, SCRIPTDS contains a positional parameter, &DSN. The invoker supplies a unique name for this parameter. The CLIST includes the &DSN variable in the member name of the input data set parameter 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: MEMOS.TEXT(&DSN).                          */
/*********************************************************************/
script   '&SYSPREF..memos.text(&DSN)' +
 message(delay id trace) device(3800n6) twopass +
 profile('script.r2.text(ssprof)') +
 lib('script.r2.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.    */
/*********************************************************************/

CONTROL NOMSG
free f(sysprint,sysut1,sysut2,sysin)
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)
```

Figure 4-6. The SCRIPDS CLIST

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.

SCRIPTN, shown in Figure 4-7, 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 immediately invokes a CLIST called SCRIPTD, which sets up the environment required to script the input data set and then issues the SCRIPT command (See Figure 4-8). When finished with these tasks, SCRIPTD automatically 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 Figure 4-9). 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 EXECUTE THE SCRIPT COMMAND.  */
/*********************************************************************/
%scriptd
/*********************************************************************/
/*   INVOKE THE OUTPUT CLIST TO PRINT 2 COPIES OF THE SCRIPTED       */
/*   DATA SET ON THE 3800 PRINTER.                                   */
/*********************************************************************/
%output
 END
 ELSE +
   WRITE The name entered must be less than 9 characters long and +
         the first character must not be numeric.
```

Figure 4-7. The SCRIPTN CLIST

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
/*********************************************************************/
/* 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.r2.text(ssprof)') +
 lib('script.r2.maclib') +
 sysvar(c 1 d yes) +
 bind(8 8) chars(gt12 gb12) file('&SYSPREF..&DSNAM..list') continue
```

Figure 4-8. The SCRIPTD CLIST

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
free f(sysprint,sysut1,sysut2,sysin)
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)
```

Figure 4-9. The OUTPUT CLIST

Including JCL Statements - The SUBMITDS CLIST

You can include job control language (JCL) statements in CLISTs. The SUBMITDS CLIST, shown in Figure 4-10, makes use of the SUBMIT * command, which indicates 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.

Since an account number may contain leading zeros that are ignored by the &LENGTH built-in function, use the &STR built-in function in the evaluation of the length of &ACCT.

The design of SUBMITDS 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.

THE SUBMITDS CLIST

```
PROC 2 DSN ACCT CLASS(C)
/*********************************************************************/
/* IF &ACCT IS INVALID, 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($$)
//&SYSUID.1 JOB   &ACCT,&SYSUID,CLASS=&CLASS,notify=&sysuid
/&SLSHASK  THIS STEP COPIES THE INPUT DATASET TO SYSOUT=A
//COPY      EXEC  PGM=COPYDS
//SYSUT1    DD    DSN=&SYSUID..&DSN,DISP=SHR
//SYSUT2    DD    SYSOUT=A
$$
```

Figure 4-10. The SUBMITDS CLIST

Performing Substringing on Input Strings - The SUBMITFQ CLIST

It is possible to use the &SUBSTR built-in variable to modify input supplied by the invoker. The SUBMITFQ CLIST, shown Figure 4-11, determines whether the data set name supplied by the invoker is a fully qualified name or not. SUBMITFQ makes the determination 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 quotes. (Unlike on the ALLOCATE command, fully qualified data set names are not enclosed in single 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 referenced on the SYSUT1 JCL statement.

THE SUBMITFQ CLIST

```
PROC 2 DSN ACCT CLASS(C)
/*********************************************************************/
/* IF &ACCT IS INVALID, 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 QUOTES.  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 /* required for the jcl comment statement */
SUBMIT *   END($$)
//&SYSUID.1 JOB   &ACCT,&SYSUID,CLASS=&CLASS
/&SLSHASK   THIS STEP COPIES THE INPUT DATASET TO SYSOUT=A
//COPY      EXEC  PGM=COPYDS
//SYSUT1    DD    DSN=&DSN,DISP=SHR
//SYSUT2    DD    SYSOUT=A
$$
```

Figure 4-11. 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 implement this capability to enable users who are not familiar 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 4-12, illustrates these advantages.

RUNPRICE either executes a COBOL program called APRICE in the foreground or submits a job that executes it in the background. The CLIST determines which type of invocation to perform based on whether or not the invoker includes the BATCH keyword on the invocation of RUNPRICE.

THE RUNPRICE CLIST

```
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 EXECUTE 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 4-12. The RUNPRICE CLIST

Including Options - The TESTDYN CLIST

You can code options in CLISTs that add flexibility to an application. The TESTDYN CLIST, shown in Figure 4-13, 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 or not the invoker wants the system messages that PARMTEST produces sent to a data set rather than to the terminal. TESTDYN includes a keyword parameter on its PROC statement called SYSPRINT and assigns it a default value of *, which sends system messages to the terminal. The invoker can override that default value and have them 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 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.
```

Figure 4-13. The TESTDYN CLIST

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 Figure 4-14, performs all of the functions required to compress a data set.

COMPRESS could 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 OR NOT 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 EXECUTE, 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 */
```

Figure 4-14. The COMPRESS CLIST

Simplifying Interfaces to Applications - The CASH CLIST

You may have access to applications written in other programming languages that perform useful services. However, the interfaces required to invoke these programs may not be easily mastered by the casual users of the system. 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 Figure 4-15, 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.

THE CASH CLIST

```
/* 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 4-15. 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 could contain a name and a phone 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 4-16, takes advantage of this technique. PHONE receives a last 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 last name, followed by a blank and a phone 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 phone 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 provides for loop termination (SET DONE=YES).

THE PHONE CLIST

```
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 */
/* PHONE 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, SET UP AN ERROR ROUTINE TO HANDLE END-OF-FILE,       */
/* AND OPEN THE 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 PHONE 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 4-16. The PHONE CLIST

Allocating Data Sets to SYSPROC - The SPROC CLIST

The purpose of SPROC is: first, find all data sets currently allocated to SYSPROC and concatenate them; then add the invoker's data set to the beginning of the concatenation and allocate the concatenation to SYSPROC.

The CLIST, shown in Figure 4-17, uses &SYSOUTTRAP to intercept the output from the LISTALC STATUS command and saves the command output in &SYSOUTLINEnn 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 the file begins in the third position, whereas a data set name begins in the first position of the output line. The steps in the process are:

1. Loop through &SYSOUTLINEnn 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, set a variable to the name of the previous data set in the list and enclose it in single quotes.

3. Begin with the &SYSOUTLINEnn 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 you have 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, you can determine the contents of the line.

   If the line contains a disposition, decrease &SYSOUTLINEnn by one to get the data set name and add it to the variable (&CONCAT) representing the data sets in the new concatenation. Repeat this procedure until another file name is encountered or until all command output has been searched. Once all data sets have been added to the concatenation list, issue 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 it 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 quotes 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, IN ORDER 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).                     */
/*********************************************************************/
```

Figure 4-17 (Part 1 of 2). The SPROC CLIST

```
lista st
SET SPROC = &STR(SYSPROC)
SET FOUND = NO
SET I=1
DO WHILE &STR(&FOUND) = NO AND &SYSOUTLINE     <= &I
 SET DSN = &&SYSOUTLINE&I
 IF &LENGTH(&STR(&DSN))   >=9  THEN +
   IF &STR(&SUBSTR(3:9,&DSN)) = &SPROC THEN +
    DO
     SET FOUND = YES
     SET I = &I-1
     SET DSN = &&SYSOUTLINE&I
     SET CONCAT = '&DSN'
    END
   ELSE SET I = &I+1
 ELSE SET I = &I+1
END
/*********************************************************************/
/*  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
  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
 END
/*********************************************************************/
/*  ONCE 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
```

Figure 4-17 (Part 2 of 2). The SPROC CLIST

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. The PROFILE CLIST is an example of a CLIST that displays entry panels on which the user can modify information.

PROFILE receives control after the user enters a choice on a primary selection menu. PROFILE allows the user to perform any of the following functions to modify his profiles:

- Set terminal characteristics
- Set LOG/LIST parameters
- Set PF keys (1-12)
- Set PF keys (13-24)

The PROFILE CLIST receives control from another CLIST that displays a higher-level panel. The higher-level panel prompts the user to indicate which function he wants to perform (QCMD); and if the function is setting PF keys, which PF keys he wants to view (QKEYS). Then, the CLIST invokes PROFILE, passing along 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 and/or 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, he can switch to the other one by pressing the enter key. The value of &QPFKSW is initially 0. PROFILE modifies its value to 1 if the user switches to another PF key panel. PROFILE also 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-WHILE-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 automatically stored in the associated variables on the panel definitions.

Figure 4-18 contains the purpose of, and figures containing, the PROFILE CLIST and its supporting four panel definitions.

CLIST/Panel	Purpose	Figure
PROFILE	Manage user profile panels	4-19
XYZABC10	Terminal characteristics panel	4-20
XYZABC20	LOG/LIST parameters panel	4-21
XYZABC30	PF keys 1-12 panel	4-22
XYZABC40	PF keys 13-24 panel	4-23

Figure 4-18. Purpose of, and Figures Containing, PROFILE CLIST and Supporting Panels

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.                                            */
/*********************************************************************/

IF &QCMD = 1 THEN +
 DO WHILE (&FINISH = NO)
  ISPEXEC DISPLAY PANEL(XYZABC10)
  IF &LASTCC = 8 THEN +
   SET FINISH = YES
 ENDO

/*********************************************************************/
/*  IF THE USER WANTS TO UPDATE LOG/LIST PARAMETERS, DISPLAY          */
/*  THE ASSOCIATED PANEL.                                            */
/*********************************************************************/

IF &QCMD = 2 THEN +
  DO WHILE (&FINISH = NO)
   ISPEXEC DISPLAY PANEL(XYZABC20)
   IF &LASTCC = 8 THEN +
    SET FINISH = YES
  ENDO

/*********************************************************************/
/*  IF THE USER WANTS TO UPDATE PF KEYS, DETERMINE WHICH GROUP HE     */
/*  WANTS TO UPDATE: 1-12 or 13-24.  DISPLAY THE ASSOCIATED PANEL.    */
/*********************************************************************/
```

Figure 4-19 (Part 1 of 2). The PROFILE CLIST

```
IF &QCMD =3 THEN +
 DO
  SET QPFKSW = 0
  DO WHILE (&FINISH = NO)
   IF &QKEYS ¬= 24 THEN +
    DO
     ISPEXEC DISPLAY PANEL(XYZABC30)
     IF &LASTCC = 8 THEN +
      SET FINISH = YES
     ELSE +
      DO
       SET QPFKSW = 1
       SET QKEYS = 24
      ENDO
    ENDO
   ELSE +
    DO
     ISPEXEC DISPLAY PANEL(XYZABC40)
     IF &LASTCC = 8 THEN +
      SET FINISH = YES
     ELSE +
      IF &QPFKSW = 1 THEN +
       SET QKEYS = 12
    ENDO
  ENDO
 ENDO
/*
/*   EXIT ROUTINE
/*
FINAL: +
SET FCODE = 0
EXIT CODE(&FCODE)
```

Figure 4-19 (Part 2 of 2). The PROFILE CLIST

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 FOR THE USER
%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
  .HELP = ICQAE120           /* Insert name of tutorial panel    */
  .ZVARS = '(ZTERM ZKEYS ZPADC ZSF ZDEL)'
  &ZSF  = TRANS (&ZFMT D,DATA S,STD M,MAX *,' ')
)PROC
 IF (&ZCMD ¬= ' ') .MSG = ISPZ001      /* INVALID 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
```

Figure 4-20. The Terminal Characteristics Panel Definition (XYZABC10)

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 FOR THE USER
%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
  .HELP = ICQAE135           /* Insert name of tutorial panel    */
  .ZVARS = '(ZLOGFDSP,ZLSTFDSP,ZLOGCLA,ZLSTCLA,ZLOGPID,ZLSTPID, +
             ZLOGLIN,ZLSTLIN,ZLOG1PG,ZLST1PG,ZLOG2PG,ZLST2PG)'
```

Figure 4-21 (Part 1 of 2). The LOG/LIST Characteristics Panel Definition (XYZABC20)

```
)PROC
  IF (&ZCMD ¬= ' ') .MSG = ISPZ001      /* INVALID 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
```

Figure 4-21 (Part 2 of 2). The LOG/LIST Characteristics Panel Definition (XYZABC20)

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 FOR THE USER
%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
  .HELP = ICQAE180           /* Insert name of tutorial panel     */
 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
```

Figure 4-22 (Part 1 of 2). The PF Keys 1-12 Panel Definition (XYZABC30)

```
)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
```

Figure 4-22 (Part 2 of 2). The PF Keys 1-12 Panel Definition (XYZABC30)

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 FOR THE USER
%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
  .HELP = ICQAE165           /* Insert name of tutorial panel    */
 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
```

Figure 4-23 (Part 1 of 2). The PF Keys 13-24 Panel Definition (XYZABC40)

```
)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
```

Figure 4-23 (Part 2 of 2). The PF Keys 13-24 Panel Definition (XYZABC40)

Chapter 5. Reference

This chapter describes the syntax of the CLIST statements and the two TSO commands - EXEC and END - that are closely associated with CLIST processing. In addition, it lists the error codes returned by CLIST statements.

Coding the Statements and Commands

The notation used to define the statement and command syntax and format in this publication is described in the following paragraphs.

1. The set of symbols listed below is used to define the syntax, but never use them in a statement or command.

hyphen	-
underscore	_
braces	{ }
brackets	[ ]
ellipsis	...

   The special uses of these symbols are explained in the following paragraphs.

2. Use uppercase letters, numbers, and the set of symbols listed below in a statement or command exactly as shown in the syntax.

apostrophe or single quote	'
asterisk	*
comma	,
equal sign	=
parentheses	( )
period	.
ampersand	&
percent	%
colon	:

3. Lowercase letters, and symbols appearing in the syntax represent variables for which you substitute specific information in the statement or command.

   Example: If *name* appears in the syntax, substitute a specific value (for example, ALPHA) for the variable when you enter the statement or command.

4. Hyphens join lower-case words and symbols to form a single variable.

   Example: If *member-name* appears in the syntax, substitute a specific value (for example, BETA) for the variable in the statement or command.

5. A stack groups related items, such as alternatives.

   Example: The representation

   A
   B
   C

   indicates select A or B or C. Select one item and only one item; and specify it explicitly in the statement or command.

6. An underscore indicates a default option. If you select an underscored alternative, you need not specify it when you enter the statement or command.

   Example: The representation

   A
   B
   C

   indicates select A or B or C; however, if you select B, you need not specify it in the statement or command because it is the default option.

7. Braces group related items, such as alternatives.

   Example: The representation

   ALPHA=({ A / B / C } ,D)

   indicates choose only one of the items enclosed within the braces. If you select A, specify ALPHA=(A,D) in the statement or command.

8. Braces group related items, such as alternatives.

   Example: The representation

   ALPHA=({ A / B / C } ,D)

   indicates choose only one of the items enclosed within the braces. If you select A, specify either ALPHA=(A,D) or ALPHA=(,D). If you select A, you need not specify it in the statement or command because it is the default option.

9. Brackets also group related items; however, everything within the brackets is optional and may be omitted.

   Example: The representation

   ```
   ALPHA=(┌A┐
          │B│,D)
          └C┘
   ```

   indicates choose only one of the items enclosed within the brackets or omit all of the items within the brackets. If you select only D, specify ALPHA=(,D) in the statement or command.

10. An ellipsis indicates that the preceding item or group of items can be repeated more than once in succession.

    Example:

    ```
    ALPHA[,BETA]...
    ```

    indicates that ALPHA can appear alone or can be followed by ,BETA any number of times in succession in the statement or command.

11. Alphameric characters: unless otherwise indicated, an alphameric character is one of the following:

    - alphabetic: A-Z
    - numeric: 0-9
    - national: $ # @

12. CLIST statements and TSO commands may be prefixed with a label. 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 set up a routine that TSO executes when the user causes an attention interrupt. The attention interrupt is designed to halt execution of a CLIST so that the user can terminate or alter its processing.

```
[label:]          ATTN  { OFF    }
                        { action }
```

label
a name the CLIST can reference in a GOTO statement to branch to this ATTN statement. *label* is one-to-eight alphameric 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 command, commonly an EXEC command that invokes an attention processing CLIST, or a null (blank) line. An attention processing CLIST can execute multiple TSO commands, while the action can execute only one.

2. A DO-group constituting an attention exit routine. This routine must specify either one TSO command, an EXIT statement, or a null on the line preceding the RETURN statement. It may contain CLIST statements.

If a null line is executed, TSO 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 command is executed, control is given to the command.

Once a TSO command. an EXIT statement, or a null line is executed, TSO ignores all other CLIST statements and commands in the action.

CLOSFILE Statement

Use the CLOSFILE statement to close a file (data set) that has been previously opened by an OPENFILE statement. Only one file can be closed with each CLOSFILE statement.

```
[label:]        CLOSFILE  {file-name                }
                          {&symbolic-variable-name  }
```

label
a name the CLIST can reference in a GOTO statement to branch to this CLOSFILE statement. *label* is one-to-eight alphameric 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.

symbolic-variable-name
the symbolic variable to which you assigned file-name.

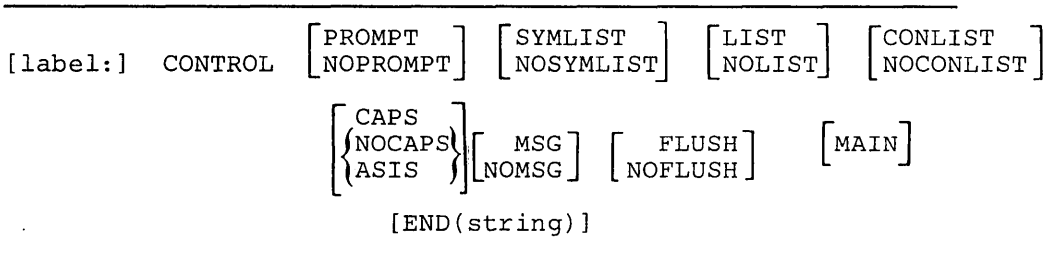
CONTROL Statement

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.

CLISTs that do not issue CONTROL statements 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-eight alphameric characters, beginning with an alphabetic character.

PROMPT
TSO commands in the CLIST may prompt the terminal for input. (The PROMPT operand on the PROFILE command must also be in effect.)

NOPROMPT
TSO commands in the CLIST may not prompt the terminal for input.

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
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
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
character strings are translated to uppercase letters before being processed.

NOCAPS or ASIS
character strings are not translated to uppercase before being processed.

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
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 on an execution error.

NOFLUSH
the system cannot flush the input stack below the CLIST with NOFLUSH specified.

MAIN
this is the main CLIST in your TSO 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 or not 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-group. *string* is 1-4 alphameric 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 on to TSO for execution.

Do not include CLIST statements in a DATA-ENDDATA sequence because TSO attempts to execute them as commands or subcommands.

Symbolic substitution is performed before execution of the group.

```
[label:]                DATA
                          .
                          .
                          .
                          .
                        ENDDATA
```

label

a name the CLIST can reference in a GOTO statement to branch to this DATA-ENDDATA sequence. *label* is one-to-eight alphameric characters, beginning with an alphabetic character.

DATA PROMPT-ENDDATA Sequence

Use the DATA PROMPT-ENDDATA sequence to designate responses to prompts by TSO commands or subcommands, or READ statements. An error condition (error code 968) occurs unless the sequence is immediately preceded by a command or subcommand, or by READ statement, issuing a prompt.

```
DATA PROMPT
.
.
.
ENDDATA
```

Note: There are certain rules to remember when using the DATA PROMPT-ENDDATA sequence. They are:

- The CLIST must allow prompting.
- Symbolic substitution is performed before a reply is sent.

DO-WHILE-END Sequence

Use the DO-WHILE-END sequence to group commands, subcommands, and statements. This sequence can include decision-making using the DO-WHILE statement. The DO statement indicates the beginning of a DO-group. The END statement concludes the DO-group.

You use DO-groups:

- With IF-THEN-ELSE sequences
- With WHILE statements
- In attention and error routines

A DO-group includes the actions you want executed when the logical expression for the THEN, ELSE, or WHILE statement is true. The WHILE DO-group executes repeatedly until the logical expression is false.

The string specified on the END operand of the CONTROL statement can be used instead of the END statement.

```
[label:]        DO  [WHILE logical-expression]
                  .
                  .
                  .
                  .
[label:]        END
```

label
a name the CLIST can reference in a GOTO statement to branch to this DO-WHILE-END sequence. *label* is one-to-eight alphameric characters, beginning with an alphabetic character.

logical-expression
a group of comparative expressions grouped by logical operators. The minimal entry for a logical expression is a comparative expression.

END Command

You may use the END command to end a CLIST. When an END command is encountered in a CLIST, and the CONTROL MAIN option is not in effect, execution of the CLIST is terminated. (If the CONTROL MAIN option is in effect, use the EXIT statement to terminate the execution of the CLIST.

```
END
```

ERROR Statement

Use the ERROR statement to set up an environment that checks for nonzero return codes from commands, subcommands, and CLIST statements in the currently executing CLIST. When an error code is detected, processing automatically continues at the ERROR routine active for the code that registered the error. If an ERROR routine is not active for the code, 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. The CLIST then attempts to continue with the next sequential statement if possible.

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:]        ERROR  [OFF   ]
                       [action]
```

label
a name the CLIST can reference in a GOTO statement to branch to this ERROR statement. *label* is one-to-eight alphameric characters, beginning with an alphabetic character.

OFF
any action previously set up by an ERROR statement is nullified.

action
any executable statement, commonly a DO-group constituting a routine. The action may execute TSO commands, subcommands, and CLIST statements. .

Note: Coding ERROR OFF within the DO-group routine itself prevents the routine from returning control to the CLIST.

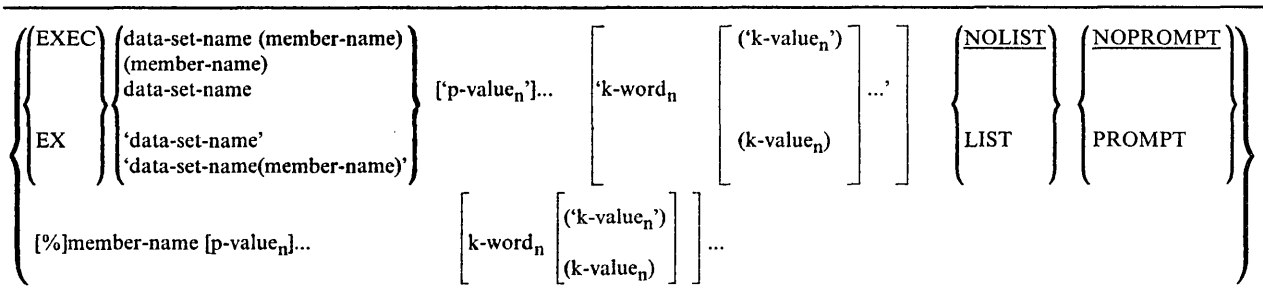
EXEC Command

Use the EXEC command to execute a CLIST. You can specify the EXEC command or the EXEC subcommand of EDIT in three ways:

- **The explicit form:** Enter EXEC or EX followed by the name of the data set that contains the CLIST.
- **The implicit form:** Enter only the member-name (a member of a CLIST library). A CLIST library is a PDS allocated to the SYSPROC file. TSO determines if the specified name is a command before searching SYSPROC for the name.
- **The extended implicit form:** Enter a percent sign, followed by the member-name. TSO searches only the SYSPROC file for the specified name.

Some of the commands in a CLIST may have symbolic variables for operands. When you specify the EXEC command, you may supply actual values for the CLIST to use in place of the symbolic variables.

The EXEC subcommand of EDIT performs the same basic functions as the EXEC command. However, a CLIST which is executed with the EXEC subcommand of EDIT can only execute CLIST statements and EDIT subcommands.



data-set-name
the unqualified name of a PDS whose type is CLIST. (data-set-name is the library name.)

member-name
the name of the CLIST

member-name
a member of a PDS whose type is CLIST. (member-name is the name of the CLIST.)

data-set-name
the unqualified name of a sequential data set whose type is CLIST. (data-set-name is the name of the CLIST.)

'data-set-name'
the fully qualified name of a sequential data set.

'data-set-name(member-name)'
the fully qualified name of a PDS. (member-name is the name of the CLIST)

A data set may contain line numbers according to the following format:

- Variable blocked - First eight characters in each record. If the data in columns 1-8 is not numeric, the CLIST treats it as data.
- Fixed blocked - Last eight characters in each record

Variable blocked records are recommended, although fixed blocked can be used.

member-name
the name of a CLIST. The CLIST is a member of a PDS that is either *not* concatenated to SYSPROC; or concatenated to SYSPROC, but the user did not use the extended implicit form.

%member-name
member-name is the name of a CLIST. The CLIST is a member of a PDS that *is* concatenated to SYSPROC.

p-value
the actual value a user specifies for each positional parameter on the PROC statement. Lower-case values are changed to upper case.

The user must specify a *p-value* for each positional parameter in the same sequence as each appears on the PROC statement (for example, p-value_1 p-value_2 ...).

If a user does not specify a *p-value* for a positional parameter, the CLIST prompts for the value. Nested CLISTs prompt for the value only if PROMPT appears on the CONTROL statement of the first CLIST.

k-word
the actual keyword a user specifies. *k-word* must be the exact name of a keyword parameter on the PROC statement.

The specification of *k-word* must follow all *p-value* specifications; but *k-words* may be specified in any order.

k-value
a value associated with k-word

'k-value'
k-value is a quoted string

Specification on the PROC statement: keyword()

- If the user specifies *k-word* without a *k-value*, the CLIST prompts for the value.

- If the user does not specify *k-word*, the associated keyword has a null value.

Specification on the PROC statement: keyword(default-value)

- If the user specifies *k-word* without a *k-value* or does not specify *k-word*, the CLIST uses the *default-value*.

- If the user specifies *k-word* with a *k-value*, the CLIST uses *k-value*.

Considerations for specifying parameters that:

- contain single quotes (apostrophes) - specify two apostrophes for each apostrophe within the string. For example, to pass the string: It's 2 o'clock specify: It''s 2 o''clock

- are quoted strings

  - implicit invocation

    - p-value - specify the exact string. For example, to pass the fully qualified data set name 'USER33.MASTER.BACKUP' specify: 'user33.master.backup'

    - k-word('k-value') - to pass the same fully qualified data set name as shown in the previous example as a k-value, specify: dsn('''user33.master.backup''')

  - explicit invocation

    - p-value - specify two quotes for each enclosing quote. For example, to pass the fully qualified data set name 'USER33.MASTER.BACKUP' specify: '''user33.master.backup'''

      The outermost set of quotes is required as part of the syntax.

    - k-word('k-value') - to pass the same fully qualified data set name as shown in the previous example as a k-value, specify: 'dsn(''''''user33.master.backup'''''')'

      The number of enclosing quotes must be doubled because the entire specification is itself a quoted string.

NOLIST
do not display commands and subcommands at the terminal.

LIST
display commands and subcommands at the terminal as they are executed.

PROMPT
allow prompting to the terminal during the execution of a CLIST. The PROMPT keyword implies LIST, unless NOLIST has been explicitly specified.

NOPROMPT
do not allow prompting during the execution of a CLIST.

EXIT Statement

Use the EXIT statement to cause control to be returned 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 the value in control variable &LASTCC.

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 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 the CONTROL statement). If no such CLIST is found, control passes to TSO, which flushes all CLISTs from the input stack and passes control to the terminal.

```
[label:]          EXIT   [CODE(expression)] [QUIT]
```

label
a name the CLIST can reference in a GOTO statement to branch to this EXIT statement. *label* is one-to-eight alphameric characters, beginning with an alphabetic character.

CODE(expression)
a user-defined return code, a decimal integer or a simple expression. When CODE is not specified, the system uses the contents of &LASTCC.

QUIT
control is passed up the nested hierarchy until either a CLIST is found with the MAIN or NOFLUSH option active or TSO receives control.

GETFILE Statement

Use the GETFILE statement to read a record from a 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.

```
[label:]          GETFILE    file-name
```

label
a name the CLIST can reference in a GOTO statement to branch to this GETFILE statement. *label* is one-to-eight alphameric 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 that will reference the values uses the GLOBAL statement to define them to global variables. Lower-level CLISTs must include a GLOBAL statement if they intend to refer to the global variables defined by the highest-level CLIST. The number of global variables defined in the highest-level CLIST is the maximum number that can be referenced by any lower-level CLIST.

The global variables are positional, and the order is set 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 only needs to define two global variables.

The GLOBAL statement must precede any statement that uses or defines its variables.

```
[label:]        GLOBAL variable1[variablen] ...
```

---

label

a name the CLIST can reference in a GOTO statement to branch to this GLOBAL statement. *label* is one-to-eight alphameric characters, beginning with an alphabetic character.

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:]          GOTO  {target   }
                        {&variable}
```

label
a name the CLIST can reference in a **GOTO** statement to branch to this GOTO statement. *label* is one-to-eight alphameric characters, beginning with an alphabetic character.

target
a label on a statement or command

variable
a symbolic variable that contains a valid label

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:]        IF logical-expression THEN [action]
                [ELSE [action]]
```

label
a name the CLIST can reference in a GOTO statement to branch to this IF-THEN-ELSE sequence. *label* is one-to-eight alphameric characters, beginning with an alphabetic character.

logical-expression
a group of comparative expressions grouped by logical operators. The minimal entry for a logical expression is a comparative expression.

action
an executable command, subcommand, or CLIST statements. (Enclose an action consisting of more than one statement in a DO-group.) 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.

OPENFILE Statement

Use the OPENFILE statement to open a 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.

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 miscellaneous abnormal terminations.

Specify NOFLUSH for a CLIST that uses file I/O. (See the CONTROL statement.)

If a system action causes TSO 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:]    OPENFILE {file-name              } {INPUT  }
                     {&symbolic-variable-name} {OUTPUT }
                                               {UPDATE }
```

label
a name the CLIST can reference in a GOTO statement to branch to this OPENFILE statement. *label* is one-to-eight alphameric characters, beginning with an alphabetic character.

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
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 to the CLIST using the EXEC command. PROC is optional. If you use it, it must be the first executable statement in the CLIST.

If the name of a positional parameter on the PROC statement is the same as the name of a GLOBAL variable, an error occurs. You cannot predefine a GLOBAL variable.

```
PROC      positional-specification [positional-parameter_n]...
            [keyword-parameter_n[([default-value_n])]]...
```

positional-specification
the number of required positional parameters to be passed. Enter 1-5 decimal digits. Enter 0 if none.

positional-parameter
a positional parameter passed to the CLIST.

A positional parameter name may be 1-252 alphameric characters in length, beginning with an alphabetic character.

keyword-parameter
a keyword parameter passed to the CLIST.

A keyword parameter name may be 1-31 alphameric characters in length, beginning with an alphabetic character.

default-value
the value assigned to the corresponding variable in the CLIST if the user does not specify a value on the associated keyword on the EXEC command.

omitted value (empty parentheses)
the user may supply a value on the associated keyword on the EXEC command

All parameters have an initial value at the time the CLIST 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 is active. Values passed in lower case are translated to upper case by the the EXEC command.

PUTFILE Statement

Use the PUTFILE statement to write a record to an open file. Each execution of PUTFILE transfers one record. Unless the user wants the same record sent more than once, the file name variable must be initialized to a different record using an assignment statement before the next PUTFILE statement is issued.

```
[label:]          PUTFILE   file-name
```

label
a name the CLIST can reference in a GOTO statement to branch to this PUTFILE statement. *label* is one-to-eight alphameric 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 created on the READ statement or elsewhere in the CLIST. The READ statement is usually preceded by a WRITE statement that requests the expected input from the terminal.

```
[label:]        READ[variable1 [variablen]...]
```

---

label

a name the CLIST can reference in a GOTO statement to branch to this READ statement. label is one-to-eight alphameric characters, beginning with an alphabetic character.

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.

omitted operand

store the input in the &SYSDVAL control variable

READDVAL Statement

Use the READDVAL statement to cause the current contents of the &SYSDVAL control variable to be assigned to a specified symbolic variable.

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:]          READDVAL  variable1 [variablen]...
```

---

label
provides a name the CLIST can reference in a GOTO statement to branch to this READDVAL statement. *label* is one-to-eight alphameric characters, beginning with an alphabetic character.

variable
any valid variable name. A variable need not have been previously defined.

RETURN Statement

The RETURN statement returns 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.

RETURN is valid only when issued from an activated error routine or an activated attention routine in this CLIST. If neither of these conditions exists, RETURN is treated as a no-operation.

```
[label:]    RETURN
```

---

label

a name the CLIST can reference in a GOTO statement to branch to this RETURN statement. *label* is one-to-eight alphameric characters, beginning with an alphabetic character.

SET Statement

Use the SET statement to assign a value to a symbolic variable, control variable, or built-in function.

```
                  {[&]symbolic-variable-name}     {=  }
[label:]    SET   {&control-variable-name    }     {EQ } value
```

label
a name the CLIST can reference in a GOTO statement to branch to this SET statement. *label* is one-to-eight alphameric characters, beginning with an alphabetic character.

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 (Refer to Figure 3-9 for those control variables that you can modify.)

EQ or =
the operator 'equal'

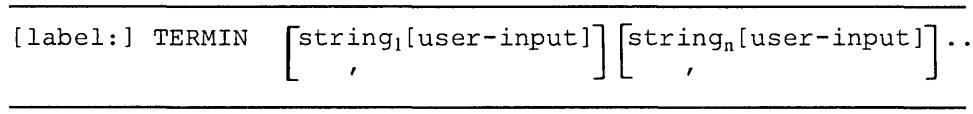
value
any valid numeric value or character string

TERMIN Statement

Use the TERMIN statement to pass control from the CLIST to the terminal user. You can also use TERMIN to define the character strings, including a null line, that a user enters to return control to the CLIST. TERMIN is usually preceded by a WRITE statement that requests the expected response from the terminal user.

Do not use TERMIN if the CLIST may be executed under ISPF or in the background.

Control returns to the CLIST at the statement after TERMIN. 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-eight alphameric characters, beginning with an alphabetic character.

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 $string_2$, etc.).

user-input
additional input entered by the terminal user. The input is stored in the &SYSDVAL control variable.

,
the terminal user can enter a null line (press the ENTER key) to return control to the CLIST.

no operands 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:]        {WRITE   } text
                {WRITENR}
```

label
a name the CLIST can reference in a GOTO statement to branch to this WRITE/WRITENR statement. *label* is one-to-eight alphameric characters, beginning with an alphabetic character.

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.

Error Codes

Figure 5-1 lists the error codes returned by CLIST statements. Except as otherwise noted, the codes are in decimal format.

Error Code	Explanation
16	Not enough virtual storage
300	User tried to update a control variable that can only be updated by the system.
304	Invalid keyword on EXIT statement.
308	CODE keyword specified, but no code given on EXIT statement.
312	Internal GLOBAL processing error.
316	TERMIN delimiter greater than 256 characters.
324	GETLINE error.
328	More than 64 delimiters on TERMIN.
332	Invalid file name syntax.
336	File already open.
340	Invalid OPEN type syntax.
344	Undefined OPEN type.
348	File specified did not open. (For example, the file name was not allocated.)
352	GETFILE - file name 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.
400	GETFILE - end of file. TSO treats this condition as an error that can be handled by an ERROR action.
404	PUTFILE - tried to write to a file open for INPUT.
408	GETFILE - tried to read from a file open for OUTPUT.
8xx	Evaluation routine error codes.
	800 Data found where operator was expected.
	804 Operator found where data was expected.
808	A comparison operator was used in a SET statement.
812	(Reserved).
816	Operator found at the end of a statement.
820	Operators out of order; data may resemble operators.
824	More than one exclusive operator found.
828	More than one exclusive comparison operator found.
832	The result of an arithmetic calculation is outside the valid range extending from -2,147,483,647 to +2,147,483,647.

Figure 5-1 (Part 1 of 2). CLIST Statement Error Codes

Error Code	Explanation
836	(Reserved).
840	Not enough operands.
844	No valid operators.
848	Attempt to load character from numeric value.
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 too large.
900	Single ampersand found.
904	(Reserved).
908	Error occurred in an error action range that received control because of another error.
912	Substring range invalid.
916	Non-numeric value in substring range.
920	Substring range value too small (zero or negative).
924	Invalid substring syntax.
932	Substring 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	Invalid symbolic variable.
944	A label was used as a symbolic variable.
948	Invalid label syntax on a GOTO statement.
952	GOTO label was not defined.
956	GOTO statement has no label.
960	&SYSSCAN was set to an invalid value.
964	&LASTCC was set to an invalid 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.
999	Internal CLIST error.
Sxxx	A system ABEND code, printed in hexadecimal.
Uxxx	A user ABEND code, printed in hexadecimal.

Figure 5-1 (Part 2 of 2). CLIST Statement Error Codes

Index

TSO Extensions CLISTS: Implementation and Reference

SC28-1304-1

S370-39

Printed in U.S.A.

TSO Extensions
CLISTS: Implementation
and Reference

SC28-1304-1

READER'S
COMMENT
FORM

This manual is part of a library that serves as a reference source for systems analysts, programmers, and operators of IBM systems. You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Note: ***Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.***

Possible topics for comment are:

Clarity Accuracy Completeness Organization Coding Retrieval Legibility

If you wish a reply, give your name, company, mailing address, and date:

_____

_____

_____

_____

Please use pressure sensitive or other gummed tape to seal this form.

Cut or Fold Along Line

What is your occupation? _____

How do you use this publication? _____

Number of latest Newsletter associated with this publication: _____

Thank you for your cooperation. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments or you may mail directly to the address in the Edition Notice on the back of the title page.)

TSO Extensions CLISTS: Implementation and Reference

SC28-1304-1 S370-39

Reader's Comment Form

Cut or Fold Along Line

Fold and tape Please Do Not Staple Fold and tape

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, N.Y.

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
Department D58, Building 921-2
PO Box 390
Poughkeepsie, New York 12602

Fold and tape Please Do Not Staple Fold and tape

Printed in U.S.A.

IBM®

TSO Extensions
CLISTS: Implementation
and Reference

SC28-1304-1

READER'S
COMMENT
FORM

This manual is part of a library that serves as a reference source for systems analysts, programmers, and operators of IBM systems. You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Note: ***Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.***

Possible topics for comment are:

Clarity Accuracy Completeness Organization Coding Retrieval Legibility

If you wish a reply, give your name, company, mailing address, and date:

_____

_____

_____

_____

What is your occupation? _____

How do you use this publication? _____

Number of latest Newsletter associated with this publication: _____

Thank you for your cooperation. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments or you may mail directly to the address in the Edition Notice on the back of the title page.)

Please use pressure sensitive or other gummed tape to seal this form.

Cut or Fold Along Line

TSO Extensions CLISTS: Implementation and Reference

SC28-1304-1 S370-39

Reader's Comment Form

Cut or Fold Along Line

Fold and tape Please Do Not Staple Fold and tape

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, N.Y.

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
Department D58, Building 921-2
PO Box 390
Poughkeepsie, New York 12602

Fold and tape Please Do Not Staple Fold and tape

Printed in U.S.A.

IBM®

SC28-1304-01