GC28-0645-1
File No. S370-39

Systems

OS/VS2 TSO Terminal User's Guide

VS2 Release 2

IBM

Second Edition (February, 1974)

This is a major revision of, and obsoletes, GC28-0645-0. See the Summary of Amendments following the contents. Changes or additions to the text and illustrations are indicated by a vertical line to the left of the change.

This edition applies to release 2 of OS/VS2 and to all subsequent releases until otherwise indicated in new editions or Technical Newsletters. Changes are continually made to the information herein; before using this publication in connection with the operation of IBM systems, consult the latest **IBM System/360 and System/370 Bibliography**, GA22-6822, for the editions that are applicable and current.

Requests for copies of 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, Programming Systems Publications, Department D58, Building 706-2, PO Box 390, Poughkeepsie, N.Y. 12602. Comments become the property of IBM.

Preface

This publication explains how to use the TSO Command Language. The TSO commands can be used to perform the following functions:

- Start and end a terminal session.
- Enter and manipulate data.
- Execute programs at the terminal.
- Test a program.
- Write and use command procedures.

This publication tells you how commands are used to perform these functions. For details on how to code each command, refer to the publication *OS/VS2 TSO Command Language Reference,* GC28-0646.

This publication is based on the following:

- Program products are not discussed in this manual.
- All examples in this manual show the user's input in lowercase letters and the system output in uppercase letters.
- All examples in this manual assume that you are using an IBM 2741 Communication Terminal, and that you must press the RETURN key to enter data. For information on your type of terminal refer to the publication *OS/MVT and OS/VS2 TSO Terminals,* GC28-6762.

Publications referenced in this manual include:

OS/MVT and OS/VS2 TSO Terminals, GC28-6762
OS/VS2 TSO Command Language Reference, GC28-0646
OS/VS Access Method Services, GC26-3836
OS/VS2 JCL, GC28-0692
OS/VS Data Management Services Guide, GC26-3783
OS/VS Linkage Editor and Loader, GC26-3803
OS/VS2 Data Areas, SYB8-0606
IBM System/370 Principles of Operation, GA22-7000

Contents

Figures

Summary of Amendments for GC28-0645-1 OS/VS2 Release 2

This publication has been redesigned into six distinct sections:

Section I: Basic Information for Using TSO
Section II: Starting and Ending a Terminal Session
Section III: Entering and Manipulating Data
Section IV: Executing Programs at a Terminal
Section V: Testing a Program at a Terminal
Section VI: Command Procedures

In addition to routine technical and editorial changes to both text and examples, the individual sections have been updated as follows:

Section I: Basic Information for Using TSO

This section has been redesigned and updated to provide the necessary information to use TSO at a terminal. Specifically, this section describes the use of:

- Terminals
- TSO commands
- System-provided aids
- Data set naming conventions

Section II: Starting and Ending a Terminal Session

This section has been updated to include additional and enhanced functions of LOGON, PROFILE, and TIME commands.

Section III: Entering and Manipulating Data

This section has been updated to provide information concerning the functions available under EDIT with the ALLOCATE, SEND, and SUBMIT subcommands of EDIT.

Section IV: Executing Programs at a Terminal

This section has been expanded to include a discussion of data set allocation, assigning attributes to data sets, and freeing allocated data sets.

Section V: Testing a Program at a Terminal

This section has been expanded to include information previously included only in **TSO Guide to Writing a Terminal Monitor Program or a Command Processor**, GC28-0648.

Section VI: Command Procedures

This section has been expanded and redesigned to provide a broader overiew of the nature and function of command procedures as well as the specific information required to write and use them.

Introduction

TSO is a time sharing system that lets you use the facilities of a computer at a terminal. A terminal is a typewriter-like device connected through telephone or other communication lines to the computer. A terminal can be at any distance from the computer -- in the same room or in another city. Because the system processes instructions much faster than you can enter them through the terminal, it can process input from many terminals at the same time it is processing work entered in the conventional manner in the computer room. However, due to the speed of the system, you will be able to work almost as though you had exclusive use of the system.

You can tell the system what work you want done by typing in one or more of the commands that form the TSO command language. The command language can be used to:

- Enter, store, modify, and retrieve data at the terminal.
- Develop programs written in Assembler, FORTRAN, COBOL, PL/I, or other languages.
- Execute programs.

Your installation determines which of the facilities of the system you can use by determining which commands are available to you.

When you enter a command in the system, the system performs the work requested by that command and sends messages back to your terminal. Messages tell you the status of your program and whether the system is ready to accept another command.

If you fail to include some necessary information with the command, the system sends you a message prompting you for the necessary information. You may then respond by typing in the information requested.

Whenever you are not sure which command to use or how to use a particular command, you can type HELP. HELP is a command that provides you with information about all other TSO commands.

This manual explains how to perform various functions using the command language. The manual is divided into the following sections:

1. Basic information for using TSO
2. Starting and ending a terminal session
3. Entering and manipulating data
4. Executing programs at a terminal
5. Testing a program at a terminal
6. Command procedures

The first three items must be known by all system users. Items 4 - 6 describe specific functions that you may wish to perform.

This manual tells you how commands are used to perform the functions mentioned above. For details on how to enter each command, refer to the manual *TSO Command Language Reference.*

Section I: Basic Information For Using TSO

Before using TSO you should know how to use:

- A terminal
- TSO commands
- System provided aids
- Data set naming conventions

Using a Terminal

A terminal session is designed to be relatively simple: a terminal user identifies himself to the system and then issues commands to request work from the system. As the session progresses, the user has a variety of aids available at the terminal which he can use if he encounters any difficulties.

Entering Information at a Terminal

All TSO terminals have a typewriter-like keyboard. The features of each keyboard vary from terminal to terminal; for example, one terminal may not have a backspace key, while another may not allow for lowercase letters. The features of each terminal as they apply to TSO are described in the publication, *TSO Terminals.* The examples in this book are addressed to a user of an IBM 2741 Communication Terminal.

Standard Terminal Conventions

Certain conventions apply to the use of all TSO terminals. They are:

- Any lowercase letters you type are interpreted by the system as uppercase letters. For example, if you type in:

  ```
  abcDe8-fg
  ```

  the system interprets it as:

  ```
  ABCDE8-FG
  ```

  The only exceptions are certain text-handling applications which allow you to type in text with both uppercase and lowercase letters. Text handling is discussed in the section "Entering and Manipulating Data."

- All messages or other output sent to you by the system comes out in uppercase letters. The only exception is the output from the special text-handling applications mentioned previously which comes out both in uppercase and lowercase.

Character and Line Deletion

To correct typing mistakes you can request that the character you just typed be deleted or that all the preceding characters in the line be deleted. You can define character-deletion and line-deletion control characters, or you can use the default characters in the system. For example, if the control characters are the quotation mark (") for deleting the preceding character, and the percent sign (%) for deleting the current line, and you type the following message:

```
first ent%Sect"onft""d ENR"try
```

it is received by the system as:

```
SECOND ENTRY
```

Note that you can use the character-deletion character repetitively (to delete more than one of the preceding characters in the line).

The blank space produced when you hit the space bar is also considered to be a character, and you can delete it using the character-deletion or line-deletion characters. For example, if you type the following line:

```
a b%cd  "E "f
```

it is received by the system as:

```
CD EF
```

Normally, you will use the default characters in the system, (usually the backspace and the attention key). However, you can use the PROFILE command to establish your own character-deletion and line-deletion characters. The PROFILE command is described in the section, "Starting and Ending a Terminal Session." The ability to change the character-deletion and line-deletion characters is useful when you use more than one type of terminal. For example, any time you have to use a terminal that does not have backspace and attention keys, you can use the PROFILE command to select two other suitable characters as the character-deletion and line-deletion characters.

Line by Line Data Entry

After you type a line and make any necessary corrections, you can enter that line as follows:

- Press the RETURN key on an IBM 2741 Communication Terminal.
- Press the RETURN key on an IBM 1052 Printer-Keyboard (If the 1052 does not have the automatic EOB feature, hold down the ALTN coding key and press the EOB(s) key.)[1]
- Hold the CTRL key and press the XOFF key on a Teletype[2] terminal.

Notes:

- All examples in this manual assume that you are using an IBM 2741 Communication Terminal, and that you must press the RETURN key to enter a line.
- If you want to enter a line of blanks, press the key used to enter a line (RETURN key on the 2741) after entering at least one blank.

You cannot use the character-deletion and line-deletion characters to make corrections to the line after you enter it. If the line you entered was a command, you must use the attention interruption (described later in this section) to cancel the line. If the line you entered was data, you can change it by using the EDIT command (described in the section, "Entering and Manipulating Data").

Using TSO Commands

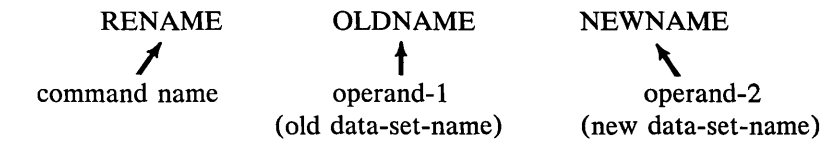
A command consists of a command name followed, usually, by one or more operands. A command name is typically a familiar English word, that describes the funtion of the command. For instance, the RENAME command changes the name of a data set. Operands

---

[1] For information about the terminal you are using, refer to *TSO Terminals.*

[2] Trademark of the Teletype Corporation.

provide the specific information required for the command to perform the requested operation. For instance, operands for the RENAME command identify the data set to be renamed and specify the new name:



Two types of operands are used with the commands: *positional* and *keyword.*

Positional Operands

Positional operands follow the command name in a prescribed sequence. In the command descriptions within this manual, the positional operands are shown in lowercase characters. A typical positional operand is:

```
data-set-name
```

You must replace "data-set-name" with an actual data set name when you enter the command.

When you want to enter a positional operand that is a list of several names or values, the list must be enclosed within parentheses. The names or values must not include unmatched right parentheses.

Keyword Operands

Keywords are specific names or symbols that have a particular meaning to the system. You can include keywords in any order following the positional operands. In the command descriptions within this book, keywords are shown in upper case characters. A typical keyword is:

```
TEXT
```

In some cases you may specify values with a keyword. The value is entered within parentheses following the keyword. The way a typical keyword with a value appears in this book is:

```
LINESIZE( integer )
```

Continuing this example, you would select the number of characters that you want to appear in a line and substitute that number for the "integer" when you enter the operand:

```
LINESIZE( 80 )
```

Note: If conflicting keywords are entered, the last keyword entered overrides the previous ones.

Abbreviating Keyword Operands

You must enter keywords spelled exactly as they are shown or you may use an acceptable abbreviation. You may abbreviate any keyword by entering only the significant characters; that is, you must type as much of the keyword as is necessary to distinguish it from the other keywords of the command or subcommand. For instance, the LISTBC command has four keywords:

```
MAIL     NOTICES
NOMAIL   NONOTICES
```

The abbreviations are:

```
M   for MAIL (also MA and MAI)
NOM for NOMAIL (also NOMA and NOMAI)
NOT for NOTICES (also NOTI, NOTIC, and NOTICE)
NON for NONOTICES (also NONO, NONOT, NONOTI, NONOTIC,
    and NONOTICE)
```

Delimiters

When you type a command, you should separate the command name from the first operand by one or more blanks. You should separate operands by one or more blanks or a comma. Do not use a semicolon as a delimiter because the characters entered after a semicolon are treated as comments. Using a blank or a comma as a delimiter, you can type the LISTBC command like this:

```
LISTBC NOMAIL NONOTICES
```

or like this:

```
LISTBC NOMAIL,NONOTICES
```

or like this:

```
LISTBC NOMAIL    NOTICES
```

A list of items may be enclosed in parentheses and separated by blanks or commas, for example:

```
LISTDS (MYDSA MYDSB,MYDSC)
```

Enter a blank by pressing the space bar at the bottom of your terminal keyboard. You can also use the TAB key to enter one or more blanks.

Subcommands

The work done by some of the commands is divided into individual operations. Each operation is defined and requested by a subcommand. To request one of the individual operations, you must first enter the command. You can then enter a subcommand to specify the particular operation that you want performed. You can continue entering subcommands until you enter the END subcommand.

The commands that have subcommands are ACCOUNT, EDIT, OPERATOR, OUTPUT and TEST.

Syntax Notation Conventions

The notation used to define the 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 format but you should never type them in the actual statement. The special uses of these symbols are explained in paragraphs 5-9.

   hyphen -

   underscore _

   braces {}

   brackets []

   ellipsis ...

2. You should type uppercase letters, numbers, and the set of symbols listed below in an actual command exactly as shown in the statement definition.

   apostrophe '

   asterisk *

   comma ,

   equal sign =

   parentheses ()

   period .

3. Lower-case letters, and symbols appearing in command definition represent variables for which you should substitute specific information in the actual command.

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

4. Stacked items represent alternatives. You should select only one such alternative.

   Example: The representation

   A
   B
   C

   indicates that either A or B or C is to be selected.

5. Hyphens join lowercase words and symbols to form a single variable.

   Example: If member-name appears in a command definition, you should substitute a specific value (for example, BETA) for the variable in the actual command.

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

   Example: The representation

   A
   B
   C

   indicates that you are to select either A or B or C; however, if you select B, you need not specify it, because it is the default option.

7. Braces group related items, such as alternatives.

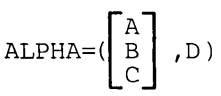
   Example: The representation

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

   indicates that you must choose one of the items enclosed within the braces. If you select A, the result is ALPHA=(A,D).

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

Example: The representation



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

9. 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.

When to Enter a Command or Subcommand

The system lets you know when it is ready to accept a new command by sending you the message:

```
READY
```

The system remains able to receive commands until you enter one of the commands that have subcommands. The system then accepts only that command's subcommands until you request a READY message by entering the END subcommand.

Using System-Provided Aids

Several aids are available for your use at the terminal:

- The attention interruption stops processing so that you can enter a command.
- The HELP command provides information about the commands.
- The conversation messages guide you at the terminal.

The Attention Interruption

The attention interruption allows you to interrupt processing at any time so that you can enter a command or subcommand. For instance, if you are executing a program and the program gets in a loop, you can use the attention interruption to halt execution. As another example, when you are having the data listed at your terminal and the data that you need has been listed, you may use the attention interruption to stop the listing operation instead of waiting until the entire data set has been listed.

If, after causing an attention interruption, you want to continue with the operation that you interrupted, you can do so by pressing the return key before typing anything else; however, input data that was being typed or output data that was being printed at the time of the attention interruption may be lost. You can also request an attention interruption while at the command level, enter the TIME command, and then resume with the interrupted operation by pressing the return key.

Note: One output record from the interrupted programs may be printed at the terminal after you enter your next command. This is normal for some programs.

If your terminal has an interruption facility, you can request an attention interruption by pressing the appropriate key (the ATTN key on IBM 2741 Communication Terminals). Whether or not your terminal has a key for attention interruptions, you can use the TERMINAL command to specify particular operating conditions that the system is to interpret as a request for an attention interruption. More specifically, you can specify a sequence of characters that the system is to interpret as a request for an attention interruption. In addition, you can request the system to pause after a certain number of seconds of processing time has elapsed or after a certain number of lines of output have been displayed at your terminal. When the system pauses, you can enter the sequence of characters that you define as a request for an attention interruption.

Note: If you are using the attention key as a line-delete indicator, pressing the attention key (after entering characters in a line, and before pressing the carriage return,) will cause the line you entered to be ignored by the system. Another depression of the attention key is required to cause an interruption.

These are three types of responses to an attention interruption entered by a terminal user:

System Response	Explanation
I	Ignored (no more attention exits available).
D	Input line has been deleted. List of messages is made available.
"attention message"	One of the message types is made available.

Messages

There are four types of messages:

- Mode messages.
- Prompting messages.
- Informational messages.
- Broadcast messages.

Mode Messages

A mode message tells you when the system is ready to accept a new command or subcommand. When the system is ready to accept a new command it prints:

```
READY
```

When you enter a command that has subcommands and the system is ready to accept that command's subcommands, it prints the name of the command, for example:

```
EDIT
```

You can then enter the subcommands you want to use. The TEST message also appears after each TEST subcommand has been processed. If the system has to print any output or other messages, as a result of the previous command or TEST subcommand, it does so before printing the mode message. (The use of mode messages in the EDIT command is discussed in the section "Entering and Manipulating Data.")

Sometimes you can save a little time by entering two or more commands in succession without waiting for the intervening READY message. The system then prints the READY messages in succession after the commands. If you enter the following commands without waiting for the intervening mode messages, your listing will be:

```
READY
attrib...
allocate...
test...
READY
READY
READY
```

There is a drawback to entering commands withput waiting for the intervening mode messages. If you make a mistake in one of the commands, the system sends you messages telling you of your mistake, and then it cancels the remaining commands you have entered. After you correct the error, you have to reenter the other commands.

Unless you are sure that there are no mistakes in your input, you should wait for a READY message before entering a new command.

Note: Some terminals "lock" the keyboard after you enter a command, and therefore you cannot enter commands without waiting for the intervening READY message. Terminals which do not lock the keyboard may occasionally do so, for example when all buffers allocated to the terminal are used.

Prompting Messages

A prompting messages tells you that required information is missing or that information you supplied was incorrectly specified. A prompting message asks you to supply or correct that information. For example, partitioned-data-set-name is a required operand of the CALL command; if you enter the CALL command without that operand the system will prompt you for the data-set-name and your listing will look as follows:

```
READY
call
ENTER DATA SET NAME -
```

You should respond by entering the requested operand, in this case the data set name, and by pressing the RETURN key to enter it. For example if the data set name is ALPHA.DATA you would complete the prompting message as follows:

```
ENTER DATA SET NAME-
alpha.data
```

If you wish, you will receive prompting messages when appropriate. However, the PROFILE command can be used to suppress prompting.

If an informational message ends with a plus sign (+) you can request an additional message by entering a question mark (?) after READY. Informational messages have only one second level message, while prompting messages may have more than one.

To request an additional level of message:

1. Type a question mark (?) in the first position of the line.
2. Press the RETURN key.

```
level 1 ENTER DATA SET NAME+
level 2 ENTER THE NAME OF A PARTITIONED DATA SET AND MEMBER THAT
        CONTAINS THE PROGRAM TO BE EXECUTED.
```

If you enter a question mark, and there are no messages to provide further detail, you receive the following message:

```
NO INFORMATION AVAILABLE
```

You can stop prompting sequence by entering the requested information or by requesting an attention interruption.

Informational Messages

An informational message tells you about the status of the system and your terminal session. For example, an informational message can tell you how much time you have used. Informational messages do not require a response.

If an informational message ends with a plus sign (+) you can request an additional message by entering a question mark (?) after READY. Informational messages have only one second level message, while prompting messages may have more than one.

Broadcast Messages

Broadcast messages are messages of general interest to users of the system. Both the system operator and any user of the system can send broadcast messages. The system operator can send messages to all users of the system or to individual users. For example, he may send the following message to all users:

```
DO NOT USE TERMINALS #4, 5 AND 6 ON 6/30. THEY ARE RESERVED FOR
DEPARTMENT 791.
```

You, or any other user, can send messages to other users or to the system operator. For example, you may send, or receive, the following message:

```
DEPARTMENT NO.4672 WILL BE CHANGED TO 4675 STARTING 8/25
```

A message sent by another user will show his user identification so you will know who sent you the message.

The HELP Command

The HELP command can be used by a terminal user to receive all the information necessary to use any TSO command. The information requested will be printed out at the user's terminal.

Explanations of Commands

To receive a list of all the TSO commands in the SYS1.HELP data set along with a description of each, enter the HELP command as follows:

```
help
```

Information about installation written commands may be placed in the SYS1.HELP data set. You can also get all the information available on a specific command in SYS1.HELP by entering the specific commandname as an operand on the HELP command, as follows:

```
help call
```

If you want to know just the function of a particular command, DELETE, for instance, enter the HELP command as follows:

```
READY
help delete function
```

If you want to know just the syntax of a particular command, TEST, for instance, enter the HELP command as follows:

```
READY
help test syntax
```

If you want to know both the function and the operands of a particular command, EXEC, for instance, enter the HELP command as follows:

```
READY
help exec function operands
```

Syntax Interpretation of HELP Information

The syntax notation used to present HELP information at your terminal is different from the syntax notation used in this publication. Since the HELP information resides in the SYS1.HELP data set, it is restricted to characters that can be represented at your terminal. If you want to use the HELP command, you should become familiar with the syntax interpretation by entering the HELP command as follows:

```
READY
help help
FUNCTION -
    THE HELP COMMAND PROVIDES FUNCTION, SYNTAX, AND OPERAND INFORMATION
    ON COMMANDS.

SYNTAX -
    HELP  'COMMAND NAME' FUNCTION SYNTAX
          OPERANDS('KEYWORD LIST') ALL

    REQUIRED-       NONE
    DEFAULTS-       ALL IF FUNCTION, SYNTAX, OR OPERANDS NOT SPECIFIED.
    ALIAS   -       H
    NOTE    -       IF HELP IS ENTERED WITHOUT ANY OPERANDS A LIST OF
                    AVAILABLE COMMANDS WITH A SHORT DESCRIPTION OF EACH
                    WILL BE DISPLAYED.
    NOTE    -       'KEYWORD LIST' IS OPTIONAL WHEN OPERANDS IS USED.
```

Syntax Interpretation -

1. User supplied values are apostrophes. Two sets of apostrophes means the value should be supplied within a set of apostrophes.
2. Words without apostrophes are to be entered as shown.
3. Commas, periods, parentheses, and asterisks are to be entered as shown.
4. Exclusive choices are indicated by slash (/).
5. Mutually exclusive formats are separated by 'or'.

Explanations of Subcommands

You can also receive a list of all the subcommands of a command having subcommands. For example, to get a list of the subcommands of EDIT, you must first get the system to issue the edit mode message. The following simulated listing shows how to successfully enter EDIT command that specifies an existing data set, so as to receive the EDIT message:

```
.....
READY
edit cmdlang old asm
EDIT
help
```

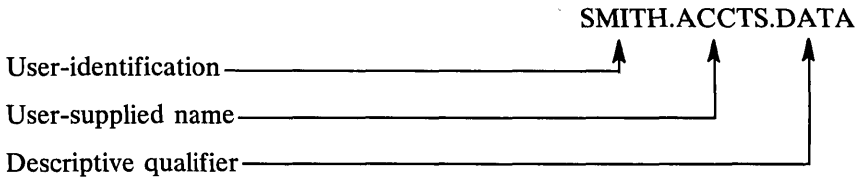
Help entered without any operands will produce a list of subcommands of EDIT.

Using Data Set Naming Conventions

The name you give a data set should follow certain conventions. A TSO data set name normally has three fields:

- Identification qualifier (used to make the data set name unique).
- User-supplied name (optional for a partitioned data set).
- Descriptive qualifier, which has meaning to the TSO commands.

The fields must be separated by periods. Each field consists of 1-8 alphameric characters and begins with an alphabetic or national ($, @, and #) character. The total length of the name, including periods, must not exceed 44 characters. For example, a typical data set name is:



When you create a data set you need specify only the user-supplied name. The system supplies values for the other two fields. The identification qualifier is either the user identification you specified with the LOGON command, or a qualifier you assign to yourself by using the PROFILE command. The user-supplied name can be a simple name or several simple names separated by periods. The descriptive qualifier is one of those listed in Figure 1. Sometimes, the system infers the descriptive qualifier from the data set type operand entered with the EDIT command. If you do not specify a data set type the EDIT command prompts you for it. (You should be aware of the distinction between EDIT command data set type and a descriptive qualifier.) You specify the descriptive qualifier as part of a data set name, for example:

```
PARTS.DATA
```

Exceptions to Data Set Naming Conventions

You may specify a fully qualified name (a name with all three qualifiers) by enclosing it in apostrophes. For example,

```
'JONES.PROG1.ASM'
```

This is required when you have to use a data set with an identification qualifier other than your own user identification. This procedure will also reduce response time since fewer system functions must be performed.

Descriptive Qualifier	Data Set Contents
ASM	Assembler (F) input
BASIC	ITF:BASIC Statements
CLIST	TSO commands and subcommands
CNTL	*JCL and SYSIN for SUBMIT command
COBOL	American National Standard COBOL statements
DATA	Uppercase text
FORT	FORTRAN IV (E, G, G1, or H) statements and free- or fixed-format Code and Go FORTRAN statements
IPLI	ITF:PL/I statements
LINKLIST	Output listing from linkage editor
LIST	Listings
LOAD	Load module
LOADLIST	Output listing from loader
OBJ	Object module
OUTLIST	*Output listing from OUTPUT command
PLI	PL/I (F), PL/I Checkout, or PL/I Optimizing compiler statements
STEX	STATIC external data from ITF:PLI
TESTLIST	Output listing from TEST command
TEXT	Uppercase and lowercase text
VSBASIC	VSBASIC statements

*Refer to Appendix A in the publication: TSO Command Language Reference

Figure 1. Descriptive Qualifiers

Any name that does not conform to the naming conventions must be enclosed in apostrophes. For example, if you have a data set named RECORDS, with no identification or descriptive qualifiers, enter:

```
'records'
```

The system will not append the identification and descriptive qualifiers to data set names that are enclosed in apostrophes.

You can refer to an existing data set by its user-supplied name and descriptive qualifier. For example, if your data set is named:

```
SMITH.PART1.DATA
```

You may want to specify the data set name as:

```
part1.data
```

or specify the data set type if you are using the EDIT command. For example:

```
edit part1 old data
```

If you specify:	The input data set name is:	The output data set name will be:
EDIT PARTS ASM	UID.PARTS.ASM	UID.PARTS.ASM
LINK PARTS or		
LINK (PARTS)	UID.PARTS.OBJ	UID.PARTS.LOAD(TEMPNAME)
CALL PARTS	UID.PARTS.LOAD(TEMPNAME)	—
EDIT PARTS(JAN) ASM	UID.PARTS.ASM(JAN)	UID.PARTS.ASM(JAN)
LINK PARTS(JAN) or		
LINK (PARTS(JAN))	UID.PARTS.OBJ(JAN)	UID.PARTS.LOAD(JAN)
CALL PARTS(JAN)	UID.PARTS.LOAD(JAN)	—
EDIT (PARTS) ASM	UID.ASM(PARTS)	UID.ASM(PARTS)
LINK ((PARTS))	UID.OBJ(PARTS)	UID.LOAD(PARTS)
CALL (PARTS)	UID.LOAD(PARTS)	—

Figure 2. Default Names Supplied by the System

Note: In these examples, UID stands for your user identification, or an identifier assigned by the PROFILE command. TEMPNAME is the membername supplied by the system.

Note: Member names must be enclosed in parentheses to distinguish them from data set names.

	DESCRIPTIVE QUALIFIERS		
Command	**Input**	**Output**	**Listing**
ASM	ASM	OBJ	LIST
CALC	STEX	STEX	—
CALL	LOAD	—	—
COBOL	COBOL	OBJ	LIST
CONVERT	IPLI	PLI	—
	FORT	FORT	—
EXEC	CLIST	—	—
FORMAT	TEXT	—	LIST
FORT	FORT	OBJ	LIST
LINK	OBJ	LOAD	LINKLIST
	LOAD	—	—
LOADGO	OBJ	—	LOADLIST
	LOAD	—	—
OUTPUT	—	—	OUTLIST
RUN	ASM	—	—
	FORT	—	—
	BASIC	—	—
	COBOL	—	—
	IPLI	—	—
SUBMIT	CNTL	—	—
TEST	OBJ	—	TESTLIST
	LOAD	—	—

Figure 3. Descriptive Qualifiers Supplied by Default

Most of these commands require the listed descriptive qualifier if data set names are unqualified. Other commands such as LISTDS do not require any descriptive qualifier (if the name is unique).

Specifying Data Set Passwords

When referencing password protected data sets, you may specify the password as part of the data set name (you will be prompted for it otherwise). The password is separated from the data set name by a slash (/) and optionally, by one or more standard delimiters (tab, blank, or comma).

Partitioned Data Sets

You can also create and edit partitioned data sets. A partitioned data set consists of one or more data sets called members. Each member can be created and edited separately and each has a name. A member name is enclosed in parentheses and appended to the right of the fully qualified data set name. For example, the fully qualified name of member MEM1 of the SMITH.PART1.DATA data set is:

```
SMITH.PART1.DATA(MEM1)
```

You need only use the user-supplied name and member name to refer to the member. The system appends the identification and descriptive qualifiers and moves the member name to the end to form the fully qualified name. For example, to refer to member MEM1 you can specify:

```
part1(mem1)
```

or you might specify

```
part1.data(mem1)
```

In the second example, the system will append only the identification qualifier.

The following example uses the EDIT command to create member ONE of a partitioned data set named JONES.T42.DATA. The second EDIT command, creates member TWO of JONES.T42.DATA. Note that the NEW operand must be specified in both cases. The third EDIT command, specifies that changes are to be made to member ONE (the OLD operand is the default).

```
READY
edit t42.data(one) new data
INPUT
.
.
.
READY
edit t42.data(two) new data
INPUT
.
.
.
READY
edit t42.data(one) data
EDIT
.
.
.
```

Data Set Types for the EDIT Command

After you specify the data set name and the NEW or OLD operand, you should specify the data set type. The data set type is an operand that describes the contents of the data set. The type operand is one of the sources from which the system can obtain the descriptive qualifier. (If the descriptive qualifier is a valid data set type, you may specify the descriptive qualifier as part of the data set name, rather than giving data set type: specify EDIT myds.DATA instead of EDIT myds DATA.) The valid types are:

```
ASM
BASIC
CLIST
CNTL
COBOL
DATA
FORTE
FORTG
FORTGI
FORTH
GOFORT
IPLI
PLI
PLIF
TEXT
VSBASIC
```

Note: Any user data set types, specified at system generation time, are also valid data set types.

If the system cannot find the data set type from other sources, you are prompted for it.

Section II: Starting and Ending a Terminal Session

This section describes the commands you can use to:

- Identify yourself to the system.
- Define operational characteristics of your session.
- Receive and send broadcast messages.
- Display session time used.
- End your terminal session.

Identifying Yourself to the System

The first thing you must do to start a terminal session is to turn on the power according to instructions provided by your installation. In many cases, you will find an instruction sheet such as the one shown in Figure 4 attached to the terminal. In the example shown in Figure 4, instructions 1 through 8 must be followed to turn on the power and to establish and maintain connection with the system.

After you turn on the power you must use the LOGON command to identify yourself to the system. You supply, as operands of LOGON, the user attributes assigned to you by your installation. Your user attributes will consist of, at the minimum, a userid. The others listed below are optional and will be prompted for if required:

- User identification (required) -- The name or code by which you are known to the system.
- Password (required if your installation assigns you one) -- A further identification used for additional security protection.
- Account number (optional) -- The account to which your terminal session is charged.
- Procedure name (optional) -- The name of a series of statements that defines your job to the system.
- Performance group (optional) -- The performance group you wish to use during the session.

TERMINAL #7

(Available 9:00 a.m. - 3:00 p.m.
For additional time call A. Jones ext 1234)

1. Turn ON/OFF switch to ON.

2. Make sure the COM/LCL switch is set to COM.

3. Remove handset from telephone (data set).

4. Press TALK button on telephone.

5. Dial ext._____ or_____.

6. Wait for a high pitched tone. When you hear this tone you are in contact with the computer. (If you get a busy signal or no answer, hang up and repeat from step 3 trying another extension).

7. Push the DATA button on the telephone. If DATA button light goes off at any point during session, repeat from step 3.

8. Replace handset on the cradle.

9. Enter LOGON command:

   logon___/____ acct(___) proc(___) size(___) [notices / nonotices] [mail / nomail] [PERFORM(value)] [RECONNECT]

   userid password account no. procedure nnnn

10. The default TERMINAL command for an IBM 2741 Terminal is:

    terminal nolines noseconds noinput break notimeout linesize(120)

    If you want to change any of the defaults, use this form of the TERMINAL command:

    terminal lines( ) seconds( ) input( ) linesize( )

11. The default PROFILE command for an IBM 2741 Terminal is:

    profile char(bs) line(attn) prompt intercom nopause nomsgid nomode prefix(userid) nowtpmsg

    If you want to change your user profile (any of the above defaults), use this form of the PROFILE command:

    profile char( ) line( ) prompt nointercom pause msgid

    nochar noline

    The following operands are recommended for this terminal:
    char(bs) and line(attn)

Note: Please turn ON/OFF switch to OFF after you enter LOFOFF.

Figure 4. Sample Instruction Sheet for an IBM 2741 Terminal

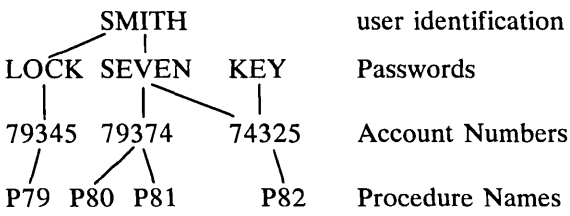
Your user attributes are recorded in the system together with the attributes of all other terminal users. When you log on, the system compares the attributes you specify in the LOGON command to the attributes recorded in your user profile, to determine if you are an authorized user of the system.

User Attributes

You can have a simple set of attributes, such as the following:

SMITH	User identification
LOCK	Password
79345	Account Number
P79	Procedure name

or a more complex set, such as



The latter set has three passwords (LOCK,SEVEN, and KEY) associated with your user identification. If you use the password LOCK, you can have your processing charged only to account 79345 and you can use only procedure P79. If you use the password SEVEN, you can have your processing charged to either account 79374 or 74325. If you choose account 79374, you can use either procedure P80 or P81. If you choose account 74325, you can use only procedure P82. Another way of using procedure P82 is to choose password KEY. KEY only has account 74325 and procedure P82 associated with it.

Logging On

The LOGON command is a simple means of telling the system your user identification, password, account number, procedure name, performance group, and whether you want the reconnect option. For example, if you want to use procedure P81, you must enter:

```
logon smith/seven acct (79374) proc(p81)
```

Whenever there is only one account number or procedure name associated with the user identification and password the system selects it by default. For example, account 79345 and procedure P79 are the only account and procedure associated with password LOCK. Therefore, when you log on you need only enter:

```
logon smith/lock
```

instead of:

```
logon smith/lock acct(79345) proc(p79)
```

Note: Some terminals have a feature which inhibits the printing of passwords on the console listing.

If you choose password SEVEN, you must specify which account number you want. If you select account 74325, you do not have to specify the procedure because there is only one procedure associated with the account.

```
logon smith/seven acct(74325)
```

If you select account 79374, you must also select a procedure name because there are two procedures associated with the account. For example,

```
logon smith/seven acct(79374) proc(p80)
```

If you choose password KEY, you do not have to specify to account number and procedure name because there are only one account number and one procedure name associated with KEY.

Note: In some instances your installation may require a modification in the way that you enter the LOGON command; for example, you may have to precede LOGON with a quotation mark. Your installation's management is responsible for advising you of such a change.

Defining Operational Characteristics

Operational characteristics can be divided into terminal characteristics and a user profile. Terminal characteristics identify:

- How you can request an attention interruption.
- Whether the keyboard is to lock up if you do not enter anything for a while.
- The length of the line that can be displayed or printed at your terminal.

Some of the characteristics a user profile identifies:

- What your character-deletion and line-deletion characters are.
- Whether you want to receive prompting messages.
- Whether you will accept messages from other terminals.

Refer to the PROFILE and TERMINAL commands in *TSO Command Language Reference* for additional information about defining terminal and user profile characteristics.

Terminal Characteristics

Your installation establishes default terminal characteristics for all the TSO terminals. If you want to change any of those characteristics for the duration of your session you can use the TERMINAL command. After your session is over, the defaults selected by the installation will again be valid for that terminal. For example, assume that the default for the number of lines of continuous output that are printed before you receive an automatic interruption is 50. You can use the TERMINAL command to request that 100 lines be printed before you receive an interruption. When you log on for your next session at that terminal, 50 lines will again be the default, provided there has been a logoff prior to the logon. The terminal characteristics remain the same for a re-logon terminal session and assume the default values with a logoff.

Your User Profile

The system has a user profile for you (see the default PROFILE command in Figure 4). When you log on that profile will be in effect. If you want to change any item in your profile, you can do so with the PROFILE command. Any change you make becomes a permanent part of your profile. That is, the next time you log on that change will be in effect. For example, assume that the line-deletion character in your profile is a percent (%) sign. You can use the PROFILE command to change it to a number (#) sign, throughout the current session. When

you log on for your next session your line deletion character will be the number sign. If you want to change it back to the original percent sign you must again use the PROFILE command.

Receiving and Sending Broadcast Messages

There are two types of broadcast messages you can receive: notices and mail. Notices are messages sent by the system operator to all users. Mail consists of messages sent by the operator or another user directly to you. You can send mail to other users and to the system operator.

Receiving Broadcast Messages

You can use three commands to control which broadcast messages you receive: LOGON, PROFILE, and LISTBC.

When you log on, broadcast messages sent to all users (notices) and those intended only for you (mail) are displayed at your terminal. You can use the following operands of the LOGON command to prevent printing either type of message at your terminal:

- NONOTICES suppresses printing of broadcast messages intended for all terminal users.
- NOMAIL suppresses printing of broadcast messages intended specifically for you.

For example, if you enter:

```
logon smith acct(72411) nomail
```

You will not receive mail but you will receive all notices that are available at the time.

NONOTICES and NOMAIL suppress those broadcast messages outstanding at the time you log on. You will automatically receive any broadcast messages issued after you log on. You cannot stop the operator from sending you notices, but you can specify that you do not want to receive any mail by using the NOINTERCOM operand of the PROFILE command. For example, if you enter the following commands:

```
READY
profile nointercom
```

you request that all broadcast messages (notices and mail) available at logon be displayed, but that all mail sent to you after logon be suppressed throughout your session. (Note that NOINTERCOM can be a default of your user profile, and therefore you may not have to specify it with the PROFILE command.)

At any time during your session you can use the LISTBC command to request that either all available notices for users, or all your mail (or both) be displayed. If you enter:

```
listbc
```

you will get all broadcast messages (notices and mail).

If you enter:

```
listbc nomail
```

you will get only notices.

If you enter:

```
listbc nonotices
```

you will get only your mail.

The notices you get are both the notices available at the time you logged on and those issued throughout your session. This enables you to see what notices were available at logon time, if you specified NONOTICES in your LOGON command. (The system operator can delete notices at any time. Consequently you will get only those notices he has not deleted.)

Mail messages sent directly to you are automatically deleted by the system after you receive them. Therefore the mail you get when you use the LISTBC command are those messages available at logon time, if you specified NOMAIL in your LOGON command, and those suppressed as a result of the NOINTERCOM operand of the PROFILE command. After you use the LISTBC command to see your mail, the NOINTERCOM operand will again be in effect.

If there are no messages available when you use the LISTBC command you will receive the following message:

```
NO BROADCAST MESSAGES
```

If you want to cancel the effect of the NOINTERCOM operand, enter:

```
profile intercom
```

You will receive any mail issued after you enter this command. To obtain your mail messages issued before you entered INTERCOM, use the LISTBC command.

Sending Messages

You can use the SEND command to send mail messages to another terminal user or to a system operator. The SEND command can be used at any time after you log on.

You can send a mail message to another user only if you know his user identification. For example, the command:

```
send 'do not use procedure 245 until notified'
user(jones,smith)
```

will send the message enclosed in quotes to the two users whose identifications are JONES and SMITH.

When you send a message to another user, he will receive it immediately if he is logged on and is accepting messages. If he is not logged on or is not accepting messages, you are notified and your message is deleted. For example, assume that SMITH is not logged on, JONES is not accepting messages, and CLARK is both logged on and accepting messages. When you send the following message:

```
send 'this is a message' user(smith,jones,clark)
```

SMITH and JONES do not receive the message, you are notified, and the message is deleted. CLARK receives the message.

You can request the system to save your message until the user you sent it to logs on or decides to accept messages, by using the LOGON operand of the SEND command. For example, if you enter:

```
send 'this is a message' user(smith,jones,clark) logon
```

SMITH will receive your message when he logs on, JONES will receive it when he uses the LISTBC command, and CLARK will receive it immediately.

You can send a message to only one operator at a time. With the SEND command, you can identify an operator by a number. For example,

```
send 'important message' operator(7)
```

If there is only one operator at your installation, you can omit the operand. For example,

```
send 'important message'
```

If there are several operators and you omit the operand, your message is sent to the mail operator.

Displaying Session Time Used

Use the TIME command to obtain the following information:

- Cumulative CPU time (from logon)
- Cumulative session time (from logon)
- Service units used
- Local time of day
- Today's date

You must first cause an attention interruption prior to entering the TIME command while a program is executing. The TIME command has no effect upon the executing program.

Testing Long-Running Programs

If a TSO command has been executing longer than expected, you can interrupt it to check its CPU and execution time. Then, depending on your analysis of the times returned, you can either resume processing from the point of interruption, or, you can cancel the processing of that command. The following example shows how a LOADGO command was interrupted; a TIME command was entered successfully; and a carriage return was entered to resume the processing of the LOADGO command.

```
READY
loadgo pehtest
>   (an attention interruption was entered here)
READY
time
(Your time information is printed here)

READY
(a carriage return was entered here)
VALID TYPES FOR DATA SET PEHTEST ARE LOAD AND OBJ
ENTER TYPE-
obj
READY   (indicates that LOADGO has completed successfully)
```

Note: If the user had decided to cancel the processing of LOADGO, he would only have had to issue another command after the third READY to cancel LOADGO.

Ending Your Terminal Session

You can end your terminal session in two ways:

- By entering the LOGOFF command to end the session.
- By entering the LOGON command to start a new session.

The LOGOFF command:

- Logically disconnects your terminal from the system. If LOGOFF HOLD is specified, the terminal remains physically connected and you can enter a new LOGON command; however, terminal characteristics established by a TERMINAL command during the previous session are no longer in effect.

A typical logoff follows:

```
READY
logoff
D58PEH LOGGED OFF TSO AT 15:24:47 on MAY 22, 1973+
```

The LOGON command:

Terminates your current session and starts a new one at the same time.

A typical logon follows:

```
READY
logon d58peh/d58paswd 10781525
D58PEH LOGGED OFF TSO AT 10:14:06 ON MAY 23, 1973+
D58PEH LOGON IN PROGRESS AT 10:14:40 ON MAY 23, 1973
READY
```

Note: In the case of a re-logon as shown above, the terminal characteristics of the old session are carried over into the new session.

Section III: Entering and Manipulating Data

The processing of data is an important part of almost all system applications. Therefore, you should learn how to enter data into the system and how to modify, store, and retrieve data after it has been entered. For example, a data set may contain:

- Text used for information storage and retrieval.
- A source program.
- Data used as input to a program.

When you create a data set you must give it a name. The system uses the name to identify the data set whenever you want to modify or retrieve it.

Using the Edit Command

The EDIT command, which is used to enter and manipulate data sets, operates in either of two modes: input mode or edit mode. When you use the EDIT command to enter data into a data set, you are using the input mode. When you use the EDIT command to enter subcommands to manipulate the data in a data set you are using the edit mode.

Entering Data in Input Mode

In input mode, you can type a line of data and then enter it into the data set by pressing the RETURN key. You can continue entering lines of data as long as EDIT is operating in input mode. If you enter a command or subcommand while in input mode the system adds it to the data set as input data. The command or subcommand is taken as data and is not executed.

You can have the system assign a line number to each line as it is entered. Line numbers make edit mode operations much easier, since you can refer to each line by its own number. When you are working with a line-numbered data set, you can request the system to print out the new line number at the start of each new input line. If the data set does not have line numbers, you can request that a prompting character be displayed at the terminal before each line is entered.

Entering Subcommands in Edit Mode

After you finish entering data in the data set, you can switch to edit mode by entering a null line. (Press the RETURN key to enter a null line.)

The system lets you know you are in edit mode by printing the following message:

```
EDIT
```

In edit mode you can enter subcommands to point to particular lines of the data set, to modify or renumber lines, to add and delete lines, or to control editing of input.

When EDIT is operating in edit mode, it uses an indicator called the currentline pointer to keep track of the next line of data to be processed. The operations you indicate with the subcommands are performed starting at the line indicated by the pointer. For example, the DELETE subcommand deletes the line indicated by the pointer. After a subcommand is executed the system repositions the pointer in accordance with the subcommand you are using.

You may want to reposition the pointer before a subcommand is executed. You can do so by using one of two methods: line number editing or context editing. Line number editing can be used only if your data set has line numbers. You can specify a line number as an operand of a subcommand and the system will move the pointer to that line before it executes the subcommand. Context editing can be used for data sets with or without line numbers. A set of

subcommands UP, DOWN, TOP, BOTTOM, and FIND allows you to move the pointer up or down a specified number of lines, or to find a line with a particular series of characters in it and move the pointer to it. After the pointer is positioned you can enter the subcommand that performs the functions required. The subcommand may contain an asterisk (*) instead of a line number to specify the line indicated by the pointer, or it may default to the current line.

Switching Modes

After you finish editing the data, you can switch to input mode in two ways:

- Entering the INPUT or INSERT subcommand.
- Entering a null line. (Press the RETURN key to enter a null line.)

The system lets you know you have selected input mode by printing the following message:

```
INPUT
```

You can terminate the EDIT command at any time by switching to edit mode (if not already in edit mode) and entering the END subcommand. The system then prints a READY message, and you can enter any command you choose.

Note: If you want to enter a blank line in your data set, you must enter a blank by pressing the space bar, and then press the RETURN key. You can then enter other lines after the blank line. If you fail to enter a blank and press only the RETURN key, you enter a null line which causes EDIT to switch modes from INPUT mode to EDIT mode.

Functions of Edit Subcommands

The remainder of this chapter describes how you can use the subcommands of EDIT to:

- Identify whether a data set is new or old.
- Create a data set.
- Place data into columns.
- Find and position the current line pointer.
- Updata a data set.
- List the contents of a data set.
- Store a new or updated data set.
- Allocate a data set.
- Submit a data set for batch execution.
- Send a message.
- End the EDIT functions.

Functions of Other Commands

The following functions described in this chapter are performed with commands other than EDIT:

- Rename a data set.
- Delete a data set.
- Allocate a data set.
- Free an allocated data set.
- List information about your data sets.

Note: A data set may be allocated by using the ALLOCATE command or the ALLOCATE subcommand of EDIT.

Identifying Data Sets

The EDIT command is used to specify the name of a data set and whether you want to create it or edit it. If you indicate that you are going to create a new data set, the system enters input mode. If you indicate that you are going to edit an existing data set, the system enters edit mode after you enter the EDIT command. For example, the NEW operand in the following EDIT command specifies that you are going to create a new data set named ACCTS.DATA. After you enter the command the system enters input mode.

```
READY
edit accts.data new
INPUT
```

In the following example, the OLD operand of the EDIT command specifies that you want to edit an existing data set named PARTS.TEXT. After you enter the command, the system enters edit mode.

```
READY
edit parts.text old
EDIT
```

As you can see, the NEW operand specifies that you are going to create a data set, and the OLD operand specifies that the data set already exists.

Creating a Data Set

You create a data set when EDIT is in input mode. You request input mode when you enter one of the following:

- The NEW operand in the EDIT command.
- The INPUT subcommand while in edit mode.
- The INSERT subcommand with no operands, while in edit mode.
- A null line if the system is in edit mode.

After you enter the EDIT command with the NEW operand the system sends you the following message:

```
INPUT
```

After this message, the system prints the first line number of your data set, unless you specified NONUM in the EDIT command. The first line number printed is 00010. Type the first line of input to the right of the line number and press the RETURN key to enter it. The system then prints the second line number, which is 00020, and you may then enter your second line of input, and so on.

Caution: A hyphen at the end of an input line indicates logical continuation of the line. In input mode, logical continuation is meaningful only if you are using the syntax checking facility. Whether or not you are syntax checking, however, the input processor will delete the hyphen from the end of the line except in a few special instances. The rules governing handling of a hyphen at the end of a line in input mode are detailed in *TSO Command Language Reference.*

When you reach the end of the data you want to enter, press the RETURN key without entering anything (a null line) and the system switches to edit mode. The following example illustrates the points just discussed:

```
READY
edit accts new data
INPUT
00010     #23942        5        @2.75        acme inc
00020     #32135       21        @3.90        bbb corp
00030     #32174       12        @1.80        alpha inds
00040     #49213       35        @7.95        xyz dist
00050     #52221       50        @2.35        beta mfg
00060   (null line)
EDIT
```

In the example, the line numbers have the standard increment of 10. If you prefer a different increment, you can use the INPUT subcommand to specify another increment. To do this you must first request a switch to edit mode by entering a null line after you receive the INPUT message. Then enter the INPUT subcommand specifying the number of the first line and the size of the increment. After entering the INPUT subcommand the system switches to input mode and prompts you with the first line number. For example, to start with line 5 and use increments of 5, you could use the following sequence:

```
READY
edit accts new data
INPUT
00010   (null line)
EDIT
input 5 5
INPUT
00005     #23942        5        @2.75        acme inc
00010     #32135       21        @3.90        bbb corp
00015     #32174       12        @1.80        alpha inds
00020     #49213       35        @7.95        xyz dist
00025     #52221       50        @2.35        beta mfg
00030   (null line)
EDIT
```

You can create the same data set in edit mode. However, you must enter the line numbers you wish to use.

```
READY
edit accts new data
INPUT
00010   (null line)
EDIT
5         #23942        5        @2.75        acme inc
10        #32135       21        @3.90        bbb corp
15        #32174       12        @1.80        alpha inds
20        #49213       35        @7.95        xyz dist
25        #52221       50        @2.35        beta mfg
```

Note: Requesting an increment larger than 1 makes it easier to insert lines in your data set later on.

Placing Data into Columns

You can use the TAB key of your terminal to align data in columns, just as you would with an ordinary typewriter. However, this mechanical tab setting is not recognized by the system, which interprets each striking of the TAB key as a space. For example, if you enter the following three lines and align them with the TAB key, they appear at the terminal as follows:

```
39427     abcde         49211     72669      ab4
22        fghijkl       441       123456     72de
987654    mnop          2         31          xyz
```

but they are received by the system as follows:

```
39427 ABCDE 49211 72669 AB4
22 FGHIJKL 441 123456 72DE
987654 MNOP 2 31 XYZ
```

If you want the system to place your data into columns, you must establish logical tab settings with the TABSET subcommand of the EDIT command or else use the defaults provided by the system. If you have established logical tab settings for your data set, the system will arrange each item in its proper column whenever you press the TAB key. The mechanical tab settings in your terminal need not correspond to the logical tab settings. For example, assume that the logical tab settings for the data set are columns 10,20, and 30, while the mechanical tab settings in the terminal are columns 5, 10, and 15. When you type in the following seven lines using the TAB key:

```
abc      def ghi          jkl
mno      pqr stu          vwx
yz0      123 456          789
                                        column 15
                                        column 10
                                        column  5
                                        column  1
```

they are arranged by the system as follows:

```
ABC        DEF          GHI        JKL
MNO        PQR          STU        VWX
YZ0        123          456        789
                                              column 30
                                              column 20
                                              column 10
                                              column  1
```

You may find it convenient to make the mechanical tab settings coincide with the logical tab settings. Details for doing this are given in the section describing the TABSET subcommand, under the EDIT command, in the *TSO Command Language Reference* manual.

If you do not use the TABSET subcommand, the default tab settings used by the system vary with the data set type. The defaults are shown in Figure 5.

Descriptive Qualifier	Default Tab Setting Columns
ASM	10,16,31,72
BASIC	10,20,30,40,50,60
CLIST	10,20,30,40,50,60
CNTL	10,20,30,40,50,60
COBOL	8,12,72
DATA	10,20,30,40,50,60
FORT	7,72
IPLI	5,10,15,20,25,30,35,40,45,50
PLI	5,10,15,20,25,30,35,40,45,50
TEXT	5,10,15,20,30,40
VSBASIC	10,15,20,25,30,35,40,45,50,55
User Defined Qualifier	10,20,30,40,50,60

Figure 5. Default Tab Settings

If you want to change the default settings or other settings you previously established, or nullify all tabs, you must use the TABSET subcommand. If you want to change the default settings, you will probably do so before you create the data set. That means you must request edit mode after you enter the EDIT command, then enter the TABSET subcommand and return to the input mode to create the data set. For example, if you want to create a TEXT data set with the logical tabs at columns 10,25, and 35, you can use the following sequence:

```
READY
exit series new text
INPUT
00010 (null line)
EDIT
tabset  (10 25 35)
        (null line)

INPUT
00010
```

If you prefer, you can define tab settings by entering a line containing t's in positions corresponding to desired tab settings. For example, to establish tab settings in columns 10, 25, and 35 you can use the TABSET subcommand as follows:

```
tabset image
123456789tbbbbbbbbbbbbbtaaaaaaaaat
```

You must fill the spaces between the t's with blanks or characters other than t. Do not use the TAB key when entering the IMAGE line, nor the backspace except as a character-delete character.

If you want to nullify the existing tab settings for the data set, enter the TABSET subcommand as follows:

```
tabset off
```

The maximum number of logical tab settings that can be defined is ten.

Finding and Positioning the Current Line Pointer

Unless you plan to use line numbers for all your edit operations, you should know how to find and reposition the current line pointer. These operations are described in the following paragraphs.

Finding the Current Line Pointer

The location of the current line pointer is determined by the last subcommand you entered. If you are editing an old data set, the current line pointer is positioned at the first line of the data set upon initial entry into edit mode. Figure 6 shows the location of the pointer at the end of each subcommand. If you do not remember this information, you can use the LIST subcommand with the * operand to find the line at which the pointer is positioned. For example:

```
list *
THIS IS THE LINE AT WHICH THE CURRENT LINE POINTER IS
POSITIONED
```

Edit Subcommands	Value of the Pointer at Completion of Subcommand
ALLOCATE	No change
BOTTOM	Last line (or line zero for empty data sets)
CHANGE	Last line changed
DELETE	Line preceding deleted line, if any, else zero
DOWN	The line n down from where you were at the start of the subcommand, or the bottom of the data set. (n is the value of the 'count' parameter.)
END	No change
FIND	Found line, if any, else no change
HELP	No change
INPUT	Last line entered
INSERT	Last line entered
Insert/Replace/Delete	Inserted or replaced line, or line preceding the deleted line, if any, or else zero.
LIST	Last line listed
PROFILE	No change
RENUM	Same relative record
RUN	No change
SAVE	No change
SCAN	Last line referred to, if any
SEND	No change
SUBMIT	No change
TABSET	No change
TOP	Zero value
UP	The line n lines up from where you were at the start of the subcommand, or the top of the data set. (n is the value of the 'count' parameter.)
VERIFY	No change

Figure 6. Values of the Current Line Pointer Referred to by an Asterisk (*)

You can also have the system display the line at which the pointer is positioned every time the pointer changes as a result one of the EDIT subcommands. To do this use the VERIFY subcommand as follows:

```
verify
```

The VERIFY subcommand is in effect until you enter it again with the OFF operand:

```
verify off
```

Positioning the Current Line Pointer

You can use the UP, DOWN, TOP, BOTTOM and FIND subcommands to move the current line pointer.

The UP subcommand moves the pointer a specified number of lines up, relative to the beginning of your data set. For example, to move the pointer so that it refers to a line located five lines before the location currently referred to, enter:

```
up 5
```

The DOWN subcommand moves the pointer a specified number of lines down, relative to the end of your data set. For example, to move the pointer so that it refers to a line located 12 lines after the location currently referred to, enter:

```
down 12
```

The TOP subcommand moves the pointer to the position preceding the first line of your data set. (For line numbered data sets, the pointer is set to zero. If line number zero exists, then line number zero becomes the current line.) TOP is often used in combination with the DOWN subcommand. For example, if you want the pointer to refer to the third line of your data set, use the following sequence:

```
top
down 3
```

The BOTTOM subcommand moves the pointer to the last line of the data set.

The FIND subcommand moves the pointer to a line that contains a specified sequence of characters. For example, to move the pointer to the line that contains PLACED BEFORE ENTRY enter:

```
find xplaced before entry
```

The "x" inserted before "placed" is a special delimiter that marks the beginning of the sequence of characters the system has to search for. The special delimiter can be any character other than a number, apostrophe, semicolon, blank, tab, comma, parenthesis, asterisk, or one of the characters in the sequence you want to find. The special delimiter must be placed next to the first character of the sequence you want to find. Any blanks inserted between the special delimiter and the first character are considered to be part of the sequence of characters.

An alternate method for specifying the sequence of characters for FIND is quoted-string notation. With this method, the specified sequence must start and end with an apostrophe. If an apostrophe is one of the characters in the specified sequence, you must enter two apostrophes for the single apostrophe in the specified sequence. For example, to find the character sequence:

```
single 'quote'
```

using quoted-string notation, enter:

```
FIND 'single ''quote'''
```

If you prefer, you can have the system search for the sequence of characters starting at the same column of each line. For example, if you want to search for PLACED BEFORE ENTRY in column seven of each line, enter:

```
find xplaced before entry x7
     or
find 'placed before entry '7
```

Notice that the same special delimiter or apostrophe used at the beginning of the sequence of characters must also precede the column number.

The FIND subcommand starts looking for the sequence of characters beginning with the line at which the pointer is located. Therefore, unless you are sure the characters are in a line following the one indicated by the pointer, you should use the TOP subcommand to move the pointer to the beginning of the data set. For example:

```
top
find xplaced before entry
```

The following is a data set used to illustrate the examples of positioning the current line pointer. Although this data set has line numbers, they are not used in the examples.

```
00010       TEMPERATURE DATA FOR 7/29/70
00020       HIGHEST, 90 AT 12:30 P.M.
00030       LOWEST, 73 AT 5:40 A.M.
00040       MEAN, 83
00050       NORMAL ON THIS DATE, 77
00060       DEPARTURE FROM NORMAL, +6
00070       HIGHEST TEMPERATURE THIS DATE, 99 IN 1949
00080       LOWEST TEMPERATURE THIS DATE, 59 IN 1914
00090       TEMPERATURE HUMIDITY INDEX, 81
```

Assume that you do not know the present location of the current line pointer, and would like to move it to the fifth line (00050). Enter:

```
top
down 5
```

To move the pointer from the fifth line (00050) to the third line (00030), enter:

```
up 2
```

To move the pointer to the line that contains FROM NORMAL enter:

```
find xfrom normal
```

To move the pointer to the last line (00090), enter:

```
bottom
```

Updating a Data Set

The subcommands of the EDIT command allow you to update a data set. That is, they allow you to:

- Delete data from a data set.
- Insert data in a data set.
- Replace data in a data set.
- Renumber lines of a data set.

These functions are described in the following paragraphs.

Deleting Data From a Data Set

If you want to delete only one line of data you do not need a subcommand. Indicate only the line number or an asterisk. For example, if you want to delete line 30, enter:

```
30
```

If you want to delete the line indicated by the current line pointer, enter:

```
*
```

You can also use the DELETE subcommand to perform the same function. For example,

```
delete 30
    or
delete *
```

DELETE also allows you to delete more than one consecutive line. To do so you can specify the line numbers of the first and last lines to be deleted, or the number of lines to be deleted starting with the line indicated with the current line pointer. For example, if you want to delete all the lines between, and including lines 15 and 75, enter:

```
delete 15 75
```

If you want to delete 12 lines starting with the line indicated by the current line pointer, enter:

```
delete * 12
```

If you want to delete all the lines in your data set, use the TOP and DELETE subcommands in combination, specifying for DELETE a number of lines greater than the number of lines in your data set.

```
top
delete * 99999999
```

After the system deletes the lines you requested, the current line pointer is positioned at the line before the first deleted line.

Inserting Data in a Data Set

To insert only one line of data in a line-numbered data set, you do not need a subcommand; indicate only the line number. The line number referred to should not exist. (That is, it should fall between two existing line numbers in the data set.) For example, if you want to insert "RECORDED DAILY IN CENTRAL" as line 22, enter:

```
22 recorded daily in central
```

The characters you want to enter must be separated from the line number or the asterisk by a single blank or a comma. Any additional blanks or commas are considered to be part of the input data. You may optionally use the tab key to separate characters from the line number or asterisk. In this case all blanks, including the first, resulting from the tab will be part of your input data. The number of blanks resulting from the tab is determined by the logical tab setting. The logical tab setting results from the TABSET subcommand or the default tab setting.

To insert one line of data after the current line, use the INSERT subcommand with the insert-data operand. For example:

```
list *
TAKE ME OUT
insert to the ballgame
```

The rules for separating inserted data from the subcommand name are the same as for separating data from line numbers.

To insert more than one line, use the INSERT or INPUT subcommands. INPUT or INSERT can be used for data sets with or without line numbers.

The INSERT subcommands inserts one or more lines of data following the location pointed to by the current line pointer.

For example, assume that you have the following data set:

```
A. CARSON   DEPT A72
T. DANIELS  DEPT 792
C. DICKENS  DEPT 981
R. EMERSON  DEPT 245
E. FARRELL  DEPT B32
C. LEVI     DEPT 229
D. MADISON  DEPT D49
```

To insert three lines after the entry for E. FARRELL and before the entry for C. LEVI, you must first position the current line pointer at the fifth line. Your listing would look like this:

```
EDIT
top
down 5
insert
INPUT
e. glotz dept 741
p. henry dept 333
h. hill  dept R92
       (null line)
EDIT
```

You must enter a null line to indicate the end of your input.

The INPUT subcommand is used in a manner similar to the INSERT subcommand if your data set does not have line numbers. Use an asterisk in the INPUT subcommand to indicate that the lines of input that follow are to be inserted in the location following the current line pointer. For example, assume that you have the following data set:

```
A. CARSON   DEPT A72
T. DANIELS  DEPT 795
C. DICKENS  DEPT 981
R. EMERSON  DEPT 245
E. FARRELL  DEPT B32
C. LEVI     DEPT 229
D. MADISON  DEPT D49
```

To insert three lines after the line for E. FARRELL and before the line for C. LEVI, your listing would look like the following:

```
EDIT
top
down 5
input *
INPUT
e. glotz dept 741
p. henry dept 333
h. hill  dept R92
       (null line)
EDIT
```

Note: that after you enter the INSERT or the INPUT subcommand, EDIT switches to input mode.

If your data set has line numbers, you can use the INPUT or INSERT subcommand to insert one or more lines of data between two existing lines of the data set. You can also indicate a smaller increment for the new line numbers so that they fit between the line numbers of the existing lines. For example, assume you have the following data set:

```
00010     1932      $1.50
00020     2579      $1.39
00030     4798      $1.75
00040     5344      $2.49
```

To insert three lines between lines 20 and 30, to have the first line numbered 22, and to increment this number by two in the following lines, your listing would look as follows:

```
EDIT
input 22 2
INPUT
00022     2795      $0.79
00024     3241      $2.81
00026     4152      $1.79
00028     (null line)
EDIT
```

The updated data set would look like this:

```
00010     1932      $1.50
00020     2579      $1.39
00022     2795      $0.79
00024     3241      $2.81
00026     4152      $1.79
00030     4798      $1.75
00040     5344      $2.49
```

Another way to insert three lines between lines 20 and 30 is to use the INSERT subcommand, as follows:

```
EDIT
top
down 2
insert
INPUT
00021     2795      $0.79
00022     3241      $2.81
00023     4152      $1.79
00024     (null line)
EDIT
```

Note: that INSERT automatically increments the line numbers by one.

The updated data set would look like this:

```
00010     1932      $1.50
00020     2579      $1.39
00021     2795      $0.79
00022     3241      $2.81
00023     4152      $1.79
00030     4798      $1.75
00040     5344      $2.49
```

If you do not change the increment, and there is no room for the new lines, you receive an error message. If you wish, you can renumber the lines of your data set. This procedure is explained in the paragraph entitled "Renumbering Lines of Data."

To enter lines at the end of the data set, enter the INPUT subcommand without operands. If the data set has line numbers you will be prompted with the line number. For example:

```
EDIT
input
INPUT
00050     6211      $3.95
00060     7199      $0.85
00070     (null line)
EDIT
```

Replacing Data in a Data Set

You can replace an entire line, or a sequence of characters in a line or in a range of lines.

If you are only replacing one line of data, you do not need a subcommand. Indicate only the line number or an asterisk. For example, if you want to replace the contents of line 70 with "SEVERAL REPORTS WERE MADE", enter:

```
70 several reports were made
```

If you want to replace the contents of the line indicated by the current line pointer, enter:

```
* several reports were made
```

The characters you want to enter must be separated from the line number or the asterisk by a single blank or a comma. Any additional blanks or commas are considered to be part of the input data. You may optionally use the tab key to separate characters from the line number or asterisk. In this case all blanks, including the first, resulting from the tab will be part of your input data. The number of blanks resulting from the tab is determined by the logical tab setting. The logical tab setting results from the TABSET subcommand or the default tab setting.

You can also replace lines of data when you use the INPUT subcommand. If you use the R operand, the lines starting with the line indicated by the line number or the asterisk are replaced by the lines you enter. For example, assume that you have the following data set:

```
COMPLETION SCHEDULE
STAGE 1   7/19
STAGE 2   8/15
STAGE 3   9/29
```

To replace the third and fourth lines, you must first position the current line pointer at the third line.

```
EDIT
top
down 2
input * r
INPUT
stage 2   8/21
stage 3   9/15
       (null line)
EDIT
```

Your updated data set would look like this:

```
COMPLETION SCHEDULE
STAGE 1   7/19
STAGE 2   8/21
STAGE 3   9/15
```

In the following example, assume that the data set has line numbers:

```
00010   COMPLETION SCHEDULE
00020   STAGE 1     7/19
00030   STAGE 2     8/15
00040   STAGE 3     9/29
```

To replace lines 30 and 40, your listing should look as follows:

```
EDIT
input 30 r
INPUT
00030   stage 2     8/21
00040   stage 3     9/15
00050   (null line)
EDIT
```

Your updated data set will look as follows:

```
00010   COMPLETION SCHEDULE
00020   STAGE 1     7/19
00030   STAGE 2     8/21
00040   STAGE 3     9/15
```

If the data set has line numbers, you can replace a line and insert additional lines. For example, assume the same data set:

```
00010   COMPLETION SCHEDULE
00020   STAGE 1     7/19
00030   STAGE 2     8/15
00040   STAGE 3     9/29
```

To replace line 30 and insert two lines with a line increment of 2, your listing should look as follows:

```
EDIT
input 30 2 r
INPUT
00030    stage 2      part 1   8/15
00032    stage 2      part 2   8/21
00034    stage 2      part 3   9/15
00036    (null line)
EDIT
```

Your updated data set will look as follows:

```
00010    COMPLETION SCHEDULE
00020    STAGE 1      7/19
00030    STAGE 2      PART 1   8/15
00032    STAGE 2      PART 2   8/21
00034    STAGE 2      PART 3   9/15
00040    STAGE 3      9/29
```

To replace more than one line with a greater number of lines, you can also use the DELETE subcommand to delete those lines and then use either INPUT or INSERT to insert the replacement lines. Use this procedure when the data set does not have line numbers.

Use the CHANGE subcommand to change only part of a line or lines. For example, to change the characters "DAILY INVENTORY" to "WEEKLY REPORT" in line 12 of your data set, enter:

```
change 12 /daily inventory/weekly report/
```

The "/" placed before the characters to be changed and before the replacement characters is a special delimiter that marks the beginning of those sequences of characters. The special delimiter can be any character other than a number, blank, tab, comma, semicolon, apostrophe, parenthesis, or asterisk. Make sure the character you select as a special delimiter does not appear in the sequence of characters you specify. If you leave blanks between the last character to be replaced and the special delimiter for the replacement characters, the blanks are considered part of the characters to be replaced. The special delimiter need not appear at the end of the replacement characters unless other parameters are to follow.

Instead of using a line number you can use an asterisk. For example if the change is to be made to the line indicated by the current line pointer, enter:

```
change * xdaily inventoryxweekly reportx
```

You can have the system search for a sequence of characters in a range of lines rather than in one line. You can indicate the range of lines by giving the numbers for the first and last lines of the range, or by indicating the current line pointer and the number of lines you want to have searched. For example, if the characters "DAILY INVENTORY" appear somewhere between lines 15 and 19, enter:

```
change 15 19 !daily inventory!weekly report!
```

If the characters appear within the 10 lines starting with the one indicated by the current line pointer, enter:

```
change * 10 ?daily inventory?weekly report?
```

You can change the sequence of characters every time it appears within the range of lines. To do this specify the ALL operand after the replacement sequence. The special delimiter must be used to terminate the replacement string before typing "all." For example,

```
change 15 19 !daily inventory!weekly report! all
or
change * 10 !daily inventory!weekly report! all
```

If you wish, you can have the system locate a sequence of characters in a line and print that line up to those characters. You can then type new characters to complete the line and enter the new line when you press the RETURN key. For example, assume that you want to change the characters "TUESDAY" in the following line:

```
00015 PARTS DELIVERIES ARE MADE ON TUESDAY
```

Your listing will look as follows:

```
change 15 /tuesday
00015 PARTS DELIVERIES ARE MADE ON
```

If the characters you want to change are in the line indicated by the current line pointer, your listing would look like this:

```
change * /tuesday
00015 PARTS DELIVERIES ARE MADE ON
```

You can also request that the system print out a specified number of characters of a given line. Then you can enter the characters you want to replace the remaining characters in the line. For example, you can request that the first 26 characters of the line "PARTS DELIVERIES ARE MADE ON TUESDAY" be printed:

```
change 15 26
00015 PARTS DELIVERIES ARE MADE
```

You can have the system print the first several characters of a range of lines. This is particularly useful when you want to change a column in a table. For example, assume that you have the following data set:

```
00010   ENROLLMENT DATES
00012   P. JONES     MAY 15  JUNE 12
00014   A. SMITH     MAY 31  JULY 19
00016   J. DOE       JUNE 7  JULY 17
00018   B. GREEN     JUNE 9  AUGUST 3
```

If you want to change the data in the last column, which begins in position 17, enter:

```
change  10 18 17
00010   ENROLLMENT DATES
00012   P. JONES     MAY 15
00014   A. SMITH     MAY 31
00016   J. DOE       JUNE 7
00018   B. GREEN     JUNE 9
```

If you want to change the data in the last column and the current line pointer is at line 10, enter:

```
change  * 5 17
00010   ENROLLMENT DATES
00012   P. JONES     MAY 15
00014   A. SMITH     MAY 31
00016   J. DOE       JUNE 7
00018   B. GREEN     JUNE 9
```

You can insert a sequence of characters at the beginning of the line. For example, if line 15 of your data set is as follows:

```
00015 EMPLOYEE ABSENTEEISM
```

enter:

```
change 15 //weekly report of /
```

to obtain:

```
00015 WEEKLY REPORT OF EMPLOYEE ABSENTEEISM
```

You can also delete a sequence of characters using the CHANGE subcommand. For example, to delete WEEKLY from line 15 above, enter:

```
change 15 /weekly//
```

to obtain:

```
00015 REPORT OF EMPLOYEE ABSENTEEISM
```

Quoted String Notation

In these examples of the CHANGE subcommand of EDIT, special-delimiter notation has been used to specify character sequences. You may, however, use an alternate form of notation, the quoted-string notation. General rules for quoted-string notation are:

- Begin and end each sequence with an apostrophe. (The system will ignore the apostrophes in its operations on your character sequence.)
- Separate character sequences with a blank.
- Specify two apostrophes in place of one whenever you wish to include an apostrophe within a character sequence.

For example, to replace WEEKLY with DAILY in the current line, you can use the special-delimiter notation:

```
change * /weekly/daily/
```

or the quoted-string notation:

```
change * 'weekly' 'daily'
```

To delete DAILY from the current line, you can use:

```
change * 'daily' "
```

instead of:

```
change * /daily//
```

To insert WEEKLY at the beginning of line 15, you can use:

```
change 15 " 'weekly'
or
change 15 //weekly/
```

To replace characters after TUESDAY'S in line 30 of your data set, you can use the special-delimiter notation:

```
00030 THIS IS TUESDAY'S CHILD
change 30 /tuesday's/
00030 THIS IS monday's child
```

or the quoted-string notation:

```
00030 THIS IS TUESDAY'S CHILD
change 30 'tuesday''s'
00030 THIS IS monday's child
```

Renumbering Lines of Data

You can use the RENUM subcommand of EDIT to assign line numbers to a data set without line numbers, or to renumber the lines of a data set with line numbers. If you enter:

```
renum
```

the system assigns new line numbers to all the lines of the data set. The first line will be assigned the number 10 and subsequent lines will be incremented by 10.

You can assign a number to the first line of the data set. For example, if you want the first line to have number 5, enter the following:

```
renum 5
```

The remaining line numbers will be 15,25,35, etc.

You can specify an increment other than 10 in addition to the number of the first line. For example if you want the first line to be number one, and the remaining line numbers to increase by 3, enter:

```
renum 1 3
```

If your data set already has line numbers, you can specify that renumbering is to start at a given line. You must also specify the new number for this line (which must be equal to or greater than the old line number) at line 23, and the new line number is to be 25 and the increment is to be 5, enter:

```
renum 25 5 23
```

The preceding example shows renumbering of all lines following a given line. You may want to limit the renumbering to a range of lines. You must specify the new line number (greater than the line prior to the old line number), the increment to be used, the old line number (first line to be renumbered), and the end line number (last line to be renumbered). For example, if you want to renumber lines 25 through 50, assigning line number 40 to the first renumbered line and using an increment of 2, enter:

```
renum 40 2 25 50
```

If you use the RENUM subcommand to renumber your data set, the renumber increment that you specify is used when you enter the INPUT subcommand the next time during the edit session. For example, if the following sequence occurred:

```
list
00010    LINE 1 OF DATA
00020    LINE 2 OF DATA
00030    LINE 3 OF DATA
END OF DATA
renum 3
input
INPUT
00012    line 4 of data
00015    line 5 of data
00018    (null line)
EDIT
```

Your data set would look like this:

```
00003    LINE 1 OF DATA
00006    LINE 2 OF DATA
00009    LINE 3 OF DATA
00012    LINE 4 OF DATA
00015    LINE 5 OF DATA
```

If you want to override the existing line number increment use the increment operand on the INPUT subcommand.

Listing the Contents of a Data Set

The LIST subcommand of EDIT allows you to display the contents of a data set at your terminal. To list the entire contents of the data set, enter:

```
list
```

To list a group of lines, enter the number of the first and last lines of the group. For example, to list lines 20 through 110 of the data set, enter:

```
list 20 110
```

If your data set does not have line numbers, you can use the current line pointer and the number of lines to be listed. For example, to list the 20 lines that begin with the line indicated by the pointer enter:

```
list * 20
```

To list only one line, indicate the line number or the current line pointer. For example, if you wish to list line 22, enter:

```
list 22
```

If you want to list the line pointed at by the current line pointer, enter:

```
list *
```

You can use the SNUM operand of LIST to suppress listing the line numbers of a line-numbered data set. (If your data set does not have line numbers, this operand has no effect.) For example, any of the following commands produces a listing of the lines indicated without their line numbers:

```
list snum
list 20 110 snum
list * 20 snum
list 22 snum
list * snum
```

The LIST subcommand uses a standard listing format. If you list a non-line-numbered data set, or a line-numbered data set using the SNUM operand (to suppress line numbers), the lines displayed will consist of only the data portion of the records. For example, to list a non-line-numbered data set:

```
list
LINE 1 OF DATA
LINE 2 OF DATA
LINE 3 OF DATA
END OF DATA
```

If you list a line-numbered data set, the system will suppress up to three leading zeros in each line number, and separate the line number from the data with a blank. The line number prints to the left of the data. For example data with an 8-digit line number would print:

```
list
00010   LINE 1 OF DATA
00020   LINE 2 OF DATA
00030   LINE 3 OF DATA
END OF DATA
```

If you are editing a line-numbered COBOL data set, with a six-character sequence (line number) field, either one or three leading zeros will be deleted depending on the command. For the INPUT command, one leading zero is suppressed; for the LIST command three leading zeros are suppressed, as follows:

```
edit a new cobol
INPUT
00010   identification division
00020   program-id.  calc.
00030   environment division
00040   (null line)
EDIT
list
010 IDENTIFICATION DIVISION
020 PROGRAM-ID.  CALC.
030 ENVIRONMENT DIVISION
END OF DATA
```

Storing a New Data Set

The data set you create or change is retained by the system only until you finish using the EDIT command and its subcommands. That is, as soon as you notify the system that you want to use another command and you get a READY message, your newly created data set, or your

new set of changes, is discarded. If you want the system to make your new data set a permanent data set, or you want the system to incorporate your changes into the existing data set, you must use the SAVE subcommand of the EDIT command.

For example, in the following sequence you create a data set named RECORDS and ask the system to store it as a permanent data set:

```
READY
edit records new data
INPUT
00010   project 21      7/10-8/25    a. jones
00020   project 23      7/10-9/12    p. smith
00030   project 39      8/1-9/15     r. brown
00040   (null line)
EDIT
save
EDIT
end
READY
```

In the following sequence you add a line to the RECORDS data set and ask the system to make it part of the data set:

```
READY
edit records old data
EDIT
40  project 428/15-9/21      s. green
save
EDIT
end
READY
```

Creating an Updated Copy of a Data Set

In some cases you may want to preserve the existing data set intact and have the system make the changes to a data set that is a copy of the original data set. To do this you must enter a new data set name for the copy when you enter the SAVE subcommand. For example, if you want to keep the RECORDS data set intact, and you want your changes to be made to a copy of RECORDS named PROJS, use the following sequence:

```
READY
edit records old data
EDIT
40  project 428/15-9/21      s. green
save projs
EDIT
end
READY
```

Now you have two data sets. The one named RECORDS looks like this:

```
00010   PROJECT 21  7/10-8/25   A. JONES
00020   PROJECT 23  7/10-9/12   P. SMITH
00030   PROJECT 39  8/1-9/15    R. BROWN
```

The data set named PROJS looks as follows:

```
00010   PROJECT 21  7/10-8/25   A. JONES
00020   PROJECT 23  7/10-9/12   P. SMITH
00030   PROJECT 39  8/1-9/15    R. BROWN
00040   PROJECT 42  8/15-9/21   S. GREEN
```

Saving Updates To A Data Set

You can use the SAVE subcommand whenever you are using the EDIT command. For example, you can create a data set and save it. Then you can start making changes to the data set and once you are satisfied with those changes you can save them to make them part of the data set. For example, in the following sequence you create a data set, save it, replace line 30, insert three lines after line 50, list the data set, delete line 56, renumber the data set, and save it.

```
READY
edit phones new text
INPUT
00010           telephone listing - sales dept
00020           j. adams        1291
00030           c. allan        2431
00040           a. bailey       3255
00050           b. crane        4072
00060           e. foster       1384
00070           f. graham       2291
00080           d. murphy       9217
00090   (null line)
EDIT
save
EDIT
30              c. alden        2241
input   52 2
INPUT
00052           l. davis        4119
00054           j. egan         6835
00056           e. foster       1384
00058   (null line)
EDIT
list

00010           TELEPHONE LISTING - SALES DEPT
00020           J. ADAMS        1291
00030           C. ALDEN        2241
00040           A. BAILEY       3255
00050           B. CRANE        4072
00052           L. DAVIS        4119
00054           J. EGAN         6835
00056           E. FOSTER       1384
00060           E. FOSTER       1384
00070           F. GRAHAM       2291
00080           D. MURPHY       9217
delete  56
renum
save
EDIT
end
READY
```

Ending the Edit Functions

Use the END subcommand to terminate the operation of the EDIT command. If you have made changes to your data set and have not entered the SAVE subcommand, the system will ask you if you want to save the modified data set. If so you can enter the SAVE subcommand. If you do not want to save the changes, reenter the END subcommand.

After you enter the END subcommand you receive the READY message. You can then enter another command.

Renaming a Data Set

The RENAME command allows you to:

- Change the name of a nonVSAM data set. (The Access Method Services ALTER command changes the name of a VSAM data set or a nonVSAM data set in a VSAM catalog.) For additional information about ALTER, refer to *OS/VS Access Method Services.*
- Change the name of a member of a partitioned data set.
- Assign an alias to a member of a partitioned data set.
- Rename common qualifiers.

Renaming a Data Set

If your LOGON user identification is SMITH and you have a data set named SMITH.RECPT.DATA and you want to change it to SMITH.ACCT.DATA, you can do so with any of the following RENAME commands:

```
rename  'smith.recpt.data' 'smith.acct.data'
rename  recpt.data acct.data
rename  recpt acct
```

Notice that the fully qualified name must be enclosed in apostrophes.

The simple user-supplied name can be used if you have only one data set under that name. However, if you have two data sets under the same user-supplied name, SMITH.RECPT.DATA and SMITH.RECPT.TEXT, you must specify either RECPT.DATA or 'SMITH.RECPT.DATA' in the RENAME command. If you do not specify the descriptive qualifier, the system will prompt you for it.

The following examples show how you can use RENAME to change either the identification qualifier or the descriptive qualifier.

```
rename  'smith.acct.data' 'jones.acct.data'
rename  acct.data acct.text
```

The following examples show how you can change more than one qualifier at a time.

```
rename  'smith.acct.data' 'jones.recpt.text'
rename  acct.data recpt.text
```

Renaming a Member of a Partitioned Data Set

When changing the name of a member of a partitioned data set, you must specify the existing data set name and member name along with the new member name. For example, to change the name of a member of SMITH.AB79. DATA from INPUT to ENTRY, you can do so with any of the following commands:

```
rename  'smith.ab79.data(input)' (entry)
rename  ab79.data(input) (entry)
rename  ab79(input) (entry)
```

Assigning an Alias to a Member

Use the ALIAS operand to indicate that the new member name is an alias and not a replacement. For example to assign the alias DAILY to member INPUT of SMITH.AB79.DATA, use any of the following:

```
rename  'smith.ab79.data(input)' (daily) alias
rename  ab79.data(input) (daily) alias
rename  ab79(input) (daily) alias
```

After entering this command the member can be referred to as either SMITH.AB79.DATA(INPUT) or SMITH.AB79.DATA(DAILY).

Renaming Common Qualifiers

Sometimes you may have two or more data set names that are identical in all but one of their qualifiers. For example, you may have these data sets:

```
JONES.ALPHA.DATA
JONES.BETA.DATA
```

or

```
JONES.ALPHA.DATA
JONES.ALPHA.ASM
```

or

```
JONES.ALPHA.DATA
SMITH.ALPHA.DATA
```

You can use the RENAME command to replace one or both of their common qualifiers. For example, you may want to change the group:

```
JONES.ALPHA.DATA
JONES.BETA.DATA
```

to

```
JONES.ALPHA.TEXT
JONES.BETA.TEXT
```

or to

```
SMITH.ALPHA.DATA
SMITH.BETA.DATA
```

or to

```
SMITH.ALPHA.TEXT
SMITH.BETA.TEXT
```

In order to make the change, replace the dissimilar qualifier with an asterisk. For example,

```
jones.*.data
```

stands for "all data sets whose identification qualifier is JONES and whose descriptive qualifier is DATA". If your logon identifier is Jones, you can then enter the RENAME command as follows:

```
rename *.data *.text
```

to change the group

```
JONES.ALPHA.DATA
JONES.BETA.DATA
```

to

```
JONES.ALPHA.TEXT
JONES.BETA.TEXT
```

Enter the command

```
rename  'jones.*.data' 'smith.*.data'
```

to change the group

```
JONES.ALPHA.DATA
JONES.BETA.DATA
```

to

```
SMITH.ALPHA.DATA
SMITH.BETA.DATA
```

Enter the command

```
rename  'jones.*.data' 'smith.*.text'
```

to change the group

```
JONES.ALPHA.DATA
JONES.BETA.DATA
```

to

```
SMITH.ALPHA.DATA
SMITH.BETA.DATA
```

Listing Information About Your Data Sets

Use the LISTALC, LISTCAT, and LISTDS commands to list the names of your data sets and obtain further information about them.

LISTALC list the data sets presently allocated to you and tells how many more data sets you can dynamically allocate using the ALLOCATE command. Other information can be obtained about these data sets depending on the parameters you specify.

LISTCAT list the names of all cataloged data sets that have your user identification. Cataloged data sets are those whose names are entered in the system catalog. The system catalog is a list the system keeps of the names and locations of cataloged data sets.

LISTDS gives you information on specific data sets which are currently cataloged or allocated, or both. The information you receive, which is described in detail in the publication, *JCL Reference,* includes:

- The serial number of the volume on which the data set resides.
- The record format, logical record length, and blocksize of the data set.
- The data set organization.
- Directory information for a member of a partitioned data set.

For more information on the LISTALC, LISTCAT, and LISTDS commands refer to *TSO Command Language Reference.* LISTCAT is also discussed in *OS/VS Access Method Services.*

Protecting Your Data Sets

The PROTECT command protects only nonVSAM data sets; an error message will be issued if you attempt to protect a VSAM data set. To protect VSAM data sets, use the Access Method Services ALTER and DEFINE commands. These commands are discussed in *OS/VS Access Method Services*.

Deleting a Data Set

Use the Access Method Services DELETE command to delete one or more data sets or one or more members of a partitioned data set. DELETE is discussed in *OS/VS Access Method Services* and *TSO Command Language Reference*.

Section IV: Executing Programs at a Terminal

You can use the TSO commands to compile, link edit, and execute (or compile and load) your source program at the terminal. TSO also allows you to use other programs, such as utilities, at the terminal. That is, instead of taking your job to the computing room to run it, you can use the TSO commands to enter it through your terminal. These commands reduce your job turnaround time because you get immediate results at the terminal. Since TSO commands are designed to operate on cataloged data sets, data sets created in the background for use with TSO in the foreground should be cataloged.

You can also use the terminal to submit your job for processing at the computer in the conventional manner. That is, you submit your job through the terminal even if you do not want to get immediate results at the terminal. The results are sent to you from the computer room after your job is executed or you may obtain them at the terminal at a later time. Jobs submitted in this manner are called batch jobs.

Most compilers or assemblers that can be used under OS/VS2 can be used from your TSO terminal. They can be used to obtain results at the terminal, or for background jobs. In addition to these programs, your installation may have one or more of the special TSO programs for your use at the terminal. They are:

- Interactive Terminal Facility (ITF):PL/I -- A problem-solving language processor.
- Interactive Terminal Facility (ITF):BASIC -- A problem-solving language processor.
- Code and Go FORTRAN -- A FORTRAN compiler designed for a very fast compile-execute sequence at the terminal.
- FORTRAN IV (G1) -- A version of the FORTRAN IV (G) compiler modified for the terminal environment.
- TSO FORTRAN Prompter -- An initialization routine to prompt you for options and invoke the FORTRAN IV (G1) Processor.
- FORTRAN Interactive Debug -- A tool for dynamic debugging of FORTRAN programs (used in confunction with Code and Go FORTRAN or FORTRAN G1).
- FORTRAN IV Library (Mod I) -- Execution-time routines for use with either Code-and-Go FORTRAN or FORTRAN IV (G1).
- Full American National Standard COBOL Version 3 or Version 4 -Versions of the American National Standard COBOL compilers with extensions for the terminal environment.
- TSO COBOL Prompter -- An initialization routine to prompt you for options and invoke the full American National Standard COBOL Version 3 or 4 Processor.
- COBOL Interactive Debug -- A tool for dynamic debugging of COBOL programs (used in conjunction with ANS COBOL Version 4).
- TSO Assembler Prompter -- An initialization routine to prompt you for options and invoke the Assembler.
- PL/I Optimizing compiler and PL/I Checkout compiler -- Both compilers include the PL/I Prompter, which is an initialization routine that prompts you for options and invokes the compiler.

If your installation has the OS PL/I Optimizing compiler or the PL/I Checkout compiler, you can compile and execute PL/I programs under TSO. The compilers are program products and each includes the PL/I Prompter, which is an initialization routine that checks compiler options, allocates data sets required by the compiler, and then invokes it.

If your installation has one or more of the TSO program product PL/I compilers, it will provide you with documentation that explains how to use them. This section explains how to use the programs normally available under OS/VS2. The following paragraphs describe how you can:

- Allocate a data set
- Assign data set attributes
- Free an allocated data set
- Create a program
- Compile your program
- Link edit a compiled program
- Execute a program
- Load a program

It is assumed that you are familiar with a programming language. The options and data set requirements of the compilers, linkage editor, and loader are summarized in the programmer's guide for the compiler you are using.

Allocating a Data Set

There are three basic times when you should allocate data sets with the ALLOCATE command or the ALLOCATE subcommand of EDIT:

- To allocate data sets required by the program or compiler you intend to invoke.
- To allocate a data set for which special characteristics have been defined with the ATTRIB command.
- To allocate data sets required by the linkage editor or loader when you use the CALL command.

You should identify the data set requirements for any program that you intend to invoke. In some cases, compilers have prompters that allocate the required data sets for you. The documentation for a program or compiler specifies data set requirements. A data set with unique characteristics assigned by the ATTRIB command may be allocated with the USING operand of the ALLOCATE command.

This section is intended for those users who are going to compile, link edit, or execute (or load) a program. Knowledge of a programming language (such as Assembler, COBOL, FORTRAN or PL/I) and of the Job Control Language (JCL) statements required to compile, link edit, and execute the program is useful for understanding this section.

The compiler, linkage editor, loader, and your own program require data sets in order to operate. In an operating system without TSO these data sets are defined with data definition (DD) JCL statements. In TSO, these data sets are defined through the EDIT and ALLOCATE commands. You can use the EDIT command to define and create input data sets. You can use the ALLOCATE command to define output and work data sets and libraries, and to allocate the data sets you created with the EDIT command. This section discusses the ALLOCATE command and the ALLOCATE subcommand of EDIT.

Note: Compilers that have prompters associated with them will allocate data sets for you. Your installation can tell you if these program product facilities are available to you. The data sets for the linkage editor and loader are allocated for you by the LINK and LOADGO commands, respectively. You need only allocate them if you invoke the linkage editor or the loader with the CALL command.

The number of data sets you need is determined by the program (compiler, linkage editor, loader, or your own program) you are going to use. (The publications associated with the IBM-supplied programs list the data set requirements.) The number of data sets you can allocate depends on the number of data sets assigned to you in your LOGON procedure. The LOGON procedure defines a series of data sets. Some of these data sets are fully defined and correspond to data sets that you always need in your processing. The remaining data sets are left undefined; they are defined when you define a data set with an ALLOCATE or EDIT command.

When you define a data set with the ALLOCATE command or subcommand, it remains allocated until you use the FREE command to free it. You may allocate a data set to the terminal by using an asterisk (*) with the data set name.

When you create a data set with the EDIT command, the system uses one of the undefined data sets in the LOGON procedure to define the data set. When you save the data set and end the EDIT command, the system saves the data set, enters its name in the system catalog, and frees the definition in the LOGON procedure for further use. When you again use the EDIT command to make changes to the saved data set, the system finds the data set through the system catalog and uses another of the available definitions to define the data set. When you end the EDIT command, the system frees the data set definition. If you want the data set to remain allocated in your LOGON procedure, you must use the ALLOCATE command or subcommand.

You can list the data sets allocated to you with the LISTALC command (described in "Listing the Names of Your Data Sets"). The system lets you know, as part of the LISTALC listing, how many DD statements are available for allocation. For example, if there are five available data sets you get the following message:

```
5 DATA SETS CAN BE ALLOCATED DYNAMICALLY
```

You can allocate as many data sets as there are available definitions. If you need more data sets you can free a previously allocated data set with the FREE command. After you free a data set, you can use the available definition to allocate another data set with the ALLOCATE command.

If you have to allocate the same data sets every time you log on, you can have your installation allocate them in the form of fully defined data sets in the LOGON procedure or you can build a command procedure containing your ALLOCATE statements and execute that procedure as soon as you are logged on. In either case you do not have to type the same ALLOCATE commands every time you log on.

The example in Figure 7 illustrates the use of the ALLOCATE command for allocating the data sets required for an execution of the Assembler F compiler. The assembler requires eight data sets for this compilation. They are:

SYSLIB	The macro library (usually SYS1.MACLIB).
SYSUT1	Work data set.
SYSUT2	Work data set.
SYSUT3	Work data set.
SYSPRINT	Output listing data set. Your terminal is allocated for this purpose.
SYSPUNCH	Data set for a punched deck of an object module. It is to be produced on the standard message output class. (To change this output class to a punch output class, see "Freeing an Allocated Data Set".)
SYSGO	Data set for the object module.
SYSIN	Input source statements to the assembler. It is entered with the EDIT command and defined to the assembler with the ALLOCATE command.

```
.
.
.
READY
edit input.asm new
INPUT
.
.source statements
.
.
EDIT
save
EDIT
end
READY
allocate dataset('sys1.maclib') file(syslib) shr
READY
allocate file(sysut1) new block(400) space(400,50)
READY
allocate file(sysut2) new block(400) space(400,50)
READY
allocate file(sysut3) new block(400) space(400,50)
READY
allocate dataset(*) file(sysprint)
READY
allocate file(syspunch) sysout
READY
allocate dataset(prog.obj) file(sysgo) new block(80) space(200,50)
READY
allocate dataset(input.asm) file(sysin) old
READY
.
.
.
```

Figure 7. Allocating Data Sets for the Assembler

Assigning Attributes to a Data Set

TSO data set characteristics are called attributes. Generally, you do not have to be concerned with attributes because TSO assigns them automatically. In some instances, however, you may want to allocate a data set with attributes different from those assigned automatically. The ATTRIB command provides a way for you to do this.

The ATTRIB command is used to assign DCB and other parameters, such as retention and expiration dates, to a data set, dynamically. This command function allows you to run existing programs that are dependent upon JCL for specifying certain DCB parameters.

Basically, you use the ATTRIB command to build a list of the attributes that you want to assign to a data set. Then you use the ALLOCATE command, specifying the name of the attribute list as the value for the USING (attr-list-name) operand. The attributes in the list are assigned to the data set when it is allocated.

You can refer to the attribute list any number of times during the remainder of your terminal session. When you finish using an attribute list, use the FREE command to delete it from the system.

The operands of the ATTRIB command (as discussed in *TSO Command Language Reference*) correspond to data control block (DCB) and other parameters discussed in the following publications:

- *JCL Reference*
- *Data Management Services Guide*

Note: Not all DCB parameters can be specified via ATTRIB.

You should understand the purpose of DCB parameters as presented in these publications before using the ATTRIB command.

The example in Figure 8 illustrates the use of the ATTRIB command. In this example, the attributes are the logical record length, the block size, and the expiration date.

```
attr dcbparms lrecl(24) blksize(96) expdt(72111)
READY
alloc da('attr.show') using(dcbparms) new bl(80) sp(1,1) vol(231400)
READY
free attrlist(dcbparms)
```

Figure 8. Assigning Attributes to a Data Set

Freeing an Allocated Data Set

Use the FREE command to release any data sets allocated to you. You can also use this command to change the output class of a SYSOUT data set, or to release attribute lists created by the ATTRIB command.

To free a data set specify its data set name or its file name (ddname). If your terminal has been allocated as a data set, you must free it through its file name. You can use the LISTALC command to obtain the file names and data set names of the data sets allocated to you.

The following example frees the data sets allocated in Figure 7. The output class of the SYSPUNCH and SYSPRINT data sets is changed to B.

```
free dataset('sys1.maclib',prog.obj,input.asm) file(sysut1,
sysut2,sysut3,sysprint,syspunch) sysout(b)
```

Creating a Program

Before your source program is compiled you must introduce it into the system. You do so with the EDIT command, as described in the section, "Entering and Manipulating Data."

When you enter the EDIT command you must specify the type operand or give a descriptive qualifier to the data set name. The type (or descriptive qualifier) tells the system which programming language you are using. If you are writing a program and JCL statements to be submitted as a background job, use CNTL as the type or descriptive qualifier.

The EDIT command allows you to specify certain options for your source program. You can use the SCAN operand to request syntax checking when the data set type is GOFORT, FORTE, FORTG, FORTGI, FORTH, BASIC, PLIF, PLI, or IPLI. You can use the LINE operand to specify the length of the input line for PL/I source programs. The length of the input line for the Assembler, FORTRAN, and COBOL is 80 characters.

After you create your source program you must use the SAVE subcommand to save the data set before you end the EDIT command. Your source program is now ready for compilation.

The example in Figure 9 shows the creation of an assembler source program.

```
READY
edit    prog1    new      asm
INPUT
.
.
.       source program
.
.
EDIT
save
EDIT
end
READY
```

Figure 9. Creating an Assembler Source Program

Compiling a Program

If you are using a TSO program product compiler and prompter, you can ignore this section. The prompter allocates data sets and calls the compiler for you.

You can use the CALL command to invoke the compiler that will compile your source program. Before you use the CALL command to invoke the compiler you must use ALLOCATE commands to allocate all the data sets required for compilation. The data sets required by your compiler are described in that program product's user's guide publication.

You must give the data set name of your compiler in the CALL command. The data set names are shown in Figure 10. (In the example, the compilers are stored in LINKLIB.)

Compiler	Data Set Name
Assembler	'SYS1.LINKLIB(IEUASM)'
American National Standard COBOL	'SYS1.(IKFCBL00)'
FORTRAN E	'SYS1.LINKLIB(IEJFAA0)'
FORTRAN G	'SYS1.LINKLIB(IEYFORT)'
FORTRAN H	'SYS1.LINKLIB(IEKAA00)'
PL/I F	'SYS1.LINKLIB(IEMAA)'

Figure 10. Data Set Names of the Compilers

Notice that the data set name is a fully qualified name and must be enclosed in apostrophes. For example, if you want to use the FORTRAN H compiler, enter:

```
READY
call 'sys1.linklib(iekaa00)'
```

In addition to the compiler's data set name, you can enter the compiler options you desire in the CALL command. These options are those specified with the PARM parameter of the EXEC statement in JCL. For example, if you want to use the MAP, NOID, and OPT=2 options of the FORTRAN H compiler, enter:

```
READY
call 'sys1.linklib(iekaa00)' 'map noid opt=2'
```

Any messages and other output produced by the compiler will appear in your listing after the CALL command. Once the compiler completes its processing you receive the READY message. You can then free any allocated data sets you no longer need.

Figure 11 shows the commands required to create a COBOL source program, allocate the eight data sets required for compilation, call the COBOL compiler, and free all allocated data sets except the one that contains the compiled program (object module). It is assumed you are using your user identification as part of all data set names except SYS1.COBLIB.

```
READY
edit    prog2   new     cobol
INPUT
.
.
.       source program
.
.
EDIT
save
EDIT
end
READY
allocate dataset('sys1.coblib') file(syslib) shr
READY
allocate file(sysut1) new block(460) space(700,100)
READY
allocate file(sysut2) new block(460) space(700,100)
READY
allocate file(sysut3) new block(460) space(700,100)
READY
allocate file(sysut4) new block(460) space(700,100)
READY
allocate dataset(*) file(sysprint)
READY
allocate dataset(prog2.obj) file(syslin) new block(80) space(500,100)
READY
allocate data set(prog2.cobol) file(sysin) old
READY
call 'sys1.linklib(ikfcbl00)' 'map load nodeck flagw'
.
.
.
.   COBOL listings and messages
.
.
.
READY
free file(syslib,sysut1,sysut2,sysut3,sysut4,sysprint,sysin)
READY
```

Figure 11. COBOL Compilation

Link Editing a Compiled Program

The LINK command makes available to you the services of the linkage editor. The linkage editor processes the compiled program (object module) and readies it for execution. The processed object module becomes a load module. Optionally, the linkage editor can process more than one object module and/or load module and transform them into a single load module.

In your LINK command you must first list the name or names of the object modules you want to link edit. (If you omit the descriptive qualifier the system assumes OBJ.) After the names of the object modules you should use the LOAD operand to indicate the name of a member of a partitioned data set where you want the load module placed. (If you omit the user-supplied name of the load module data set the system assumes it has the same user-supplied name as the object module. If you omit the descriptive qualifier the system assumes LOAD. If you omit the member name the system assumes TEMPNAME.) For example, if you want to link edit the load module in the JONES.PROG2.OBJ data set and place the resultant load module in member TEMPNAME of the JONES.PROG2.LOAD data set, enter:

```
link prog2
```

If you want to link edit the load module in the JONES.PROG2.OBJ data set and place the resultant load module in member ONE of the JONES.MODS.LOAD data set, enter:

```
READY
link prog2 load(mods(one))
```

The following example shows how to link edit the two object modules in the SMITH.PGM1.OBJ and SMITH.PGM2.OBJ data sets. The resultant load module is placed in member TEMPNAME of the SMITH.LM.LOAD data set.

```
READY
link (pgm1,pgm2) load(lm)
```

You can control the link editing process with linkage editor control statements. These control statements can be in a previously created data set, or can be introduced through the terminal. You must give the name of the data set, or an asterisk (indicating that you will introduce the control statements through the terminal) in the list of input data sets. The following example shows how to link edit the object module in the CARTER.TRAJ.OBJ data set. There are control statements in the CARTER.CNTL.DATA data set. The load module is placed in member TEMPNAME of CARTER.TRAJ.LOAD.

```
READY
link (traj,cntl.data)
```

Using the same example, if you want to introduce the control statements through your terminal, enter:

```
READY
link (traj,*)
```

The system will prompt you for the control statements at the appropriate time. You must follow your last control statement with a null line.

You can also have the linkage editor search a subroutine library to resolve external references. (External references are references to other modules.) The subroutine library is usually one of the language libraries and it is specified with one of the following operands:

Operand	Subroutine Library
COBLIB	SYS1.COBLIB
FORTLIB	SYS1.FORTLIB
PLILIB	SYS1.PL1LIB

In addition to, or instead of a language library, you can use the LIB operand to specify the name of one or more user libraries. The libraries are searched in the order you specify.

The following example shows how to link edit the object module in JAMES.PRG.OBJ. The load module is placed in JAMES.PRG.LOAD(TEMPNAME). The libraries SYS1.PL1LIB, and DEPT39.LIB.SUBRT2 are to be searched to resolve external references.

```
READY
link prg plilib lib('dept39.lib.subrt2')
```

The LINK command also lets you specify the linkage editor options. These options are those specified with the PARM parameter of the EXEC statement when you are running the linkage editor directly under the operating system rather than through TSO. For example, if you want to use the LET and XCAL options when the object module in AGNES.RET.OBJ is link edited and placed in AGNES.TBD.LOAD(MOD), enter:

```
READY
link ret load(tbd(mod)) let xcal
```

Linkage editor listings (specified with the MAP, XREF, and LIST options) are directed to a data set or to your terminal. You indicate your choice with the PRINT operand. The following example shows that the object module in BILL.PRGM.OBJ is to be link edited and placed in BILL.PRGM.LOAD(TEMPNAME). The listing produced by the MAP option is to be placed in the BILL.LIST.LINKLIST data set.

```
READY
link prgm map print(list)
```

Note that if you omit the descriptive qualifier from the print data set name, the system assumes LINKLIST. If you omit the user-supplied name, the systems it has the same user-supplied name as the object module. For example if the listing is to be placed in the BILL.PRGM.LINKLIST data set, enter:

```
READY
link prgm map print
```

Using the same example, if you want the listing to appear on your terminal, enter an asterisk in the PRINT operand.

```
READY
link prgm map print(*)
```

Error messages are listed at the terminal as well as on the print data set when you specify a data set name instead of an asterisk. If you want the error messages to appear only on the print data set, enter the NOTERM operand. For example,

```
READY
link prgm map print noterm
```

Executing a Program

You can use the CALL command to execute your program after it has been link edited. You can also use CALL to execute any other program in the load module form, such as utilities and compilers.

Before you use CALL to execute your program you can use the EDIT and ALLOCATE commands to define your data sets. Use EDIT to create your input data sets, and ALLOCATE to allocate your input, work, and output data sets.

You must specify the data set name and member name of the member that contains your program in load module form. If you want to execute a program that resides in DEPTB.PROGS.DAILY(NUM3), enter:

```
READY
call 'deptb.progs.daily(num3)'
```

If you omit the descriptive qualifier and member name, the system assumes LOAD and TEMPNAME, respectively. For example, if your LOGON identifier is "JONES" and if your program resides in JONES.LIB.LOAD(MEM2), enter:

```
READY
call lib(mem2)
```

If your program resides in JONES.LIB.LOAD(TEMPNAME), enter:

```
READY
call lib
```

You can pass parameters to your program if you wrote it in assembler or PL/I(F). These are the parameters you would specify with the PARM parameter of the EXEC statement in JCL. For example, if you want to pass the parameters OPTION1 and OPTION5 to a program that resides in JONES.ASMPG.LOAD(MEM3), enter:

```
READY
call asmpg(mem3) 'option1 option5'
```

Figure 12 shows how the COBOL program created and compiled in Figure 11 can be link edited and executed. In Figure 11, the compiled program (object module) was placed in PROG2.OBJ. After link editing, the load module is placed in PROG2.LOAD(TEMPNAME). Your program requires three data sets for execution. The input data set, INPUT.DATA, is created with the EDIT command. ALLOCATE commands are used to allocate the input data set, a work data set, and an output data set. CALL is used to execute your program. The PROG2.COBOL, PROG3.OBJ, PROG2.LOAD, and INPUT.DATA data set are deleted. (The other data sets, allocated in Figure 11, are automatically deleted because they were not given a data set name when allocated.) It is assumed you are using your user identification as part of the data set names.

If your program has an error termination, you can use the facilities of the TEST command to debug your program.

```
READY
link prog2 print(*) map
.
.
.
.
.       linkage editor messages and listings
.
.
.
READY
edit input.data new
INPUT
.
.
.
.
.
        input data
.
.
.
.
.
EDIT
save
EDIT
end
READY
allocate dataset(input.data) file(input) old
READY
allocate file(work) new block(100) space(300,10)
READY
allocate dataset(*) file(print)
READY
call prog2
.
.
.
.
.       output from your program
.
.
.
READY
delete (prog2.* input.data)
READY
```

Figure 12. Link Editing and Executing a Program

Loading a Program

The LOADGO command makes available to you the services of the loader. The loader combines the basic functions of the linkage editor and program fetch. That is, the loader link edits and executes your program. Therefore, the LOADGO command combines the basic functions of the LINK and CALL commands. No load module is produced. For complete information on the loader, refer to the publication, *Linkage Editor and Loader.*

The loader can process and execute a compiled program (object module) or a link edited program (load module). Optionally, it can combine object modules and/or load modules and execute them. (If you want to load and execute a single load module, the CALL command is more efficient.)

Before you use the LOADGO command you can use the EDIT and ALLOCATE commands to create and allocate any data sets required to execute your program.

In your LOADGO command you must list the name or names of the object and load modules you want to load. For example, if you want to load the object module in JONES.PROG3.OBJ, enter:

```
READY
loadgo prog3
```

If you want to load the object modules in JONES.PROG3.OBJ, JONES.COB.OBJ and the load module in JONES.COB.LOAD(TWO), enter:

```
READY
loadgo (prog3 cob.obj cob.load(two))
```

You can also pass parameters to your program if you wrote it in assembler or PL/I(F). These are the parameters you would specify with the PARM parameter of the EXEC statement in JCL. For example, if you want to pass the parameters OPTION1 and OPTION5 to a compiled program that resides in JONES.ASMPG.OBJ, enter:

```
READY
loadgo asmpg 'option1 option5'
```

You can have the loader search a subroutine library to resolve external references. The subroutine library is usually one of the language libraries. If so, it is specified with one of the following operands:

Operand	Subroutine Library
COBLIB	SYS1.COBLIB
FORTLIB	SYS1.FORTLIB
PLILIB	SYS1.PL1LIB

In addition to, or instead of, a language library you use the LIB operand to specify the name of one or more user libraries. The libraries are searched in the order you specify.

The following example shows how to load the object module in JONES.PRG.OBJ. The libraries SYS1.PL1LIB, and DEPT39.LIB.SUBRT2 are to be searched to resolve external references.

```
READY
loadgo prg plilib lib('dept39.lib.subrt2')
```

The LOADGO command also lets you specify the loader options. These options are those specified with the PARM parameter of the EXEC statement in JCL. For example, if you want to use the LET and EP(MAIN) options when the object module in JONES.CIR.OBJ is loaded, enter:

```
READY
loadgo cir let ep(main)
```

Loader listings (specified with the MAP option) are directed to a data set or to your terminal. You indicate your choice with the PRINT operand. The following example shows that the object module in JONES.PRGM.OBJ is to be loaded. The listing produced by the MAP option is to be placed in the JONES.LISTING.LOADLIST data set.

```
READY
loadgo prgm map print(listing)
```

Note: that if you omit the descriptive qualifier from the print data set name, the system assumes LOADLIST. If you omit the user-supplied name, the system assumes it has the same user-supplied name as the object module. For example, if the listing is to be placed in the JONES.PRGM.LOADLIST data set, enter:

```
READY
loadgo prgm map print
```

Using the same example, if you want the listing to appear on your terminal, enter an asterisk in the PRINT operand.

```
READY
loadgo prgm map print(*)
```

Error messages are listed on the terminal as well as on the print data set when you specify a data set name instead of an asterisk. If you want the error messages to appear only on the print data set, enter the NOTERM operand. For example,

```
READY
loadgo prgm map print noterm
```

Figure 13 shows how the COBOL program created and compiled in Figure 11 can be loaded. The loading operation shown in Figure 13 is the equivalent of the link editing and execution shown in Figure 12. The same data sets required for execution of your program in Figure 12 are required in this example. The object module resides in PROG2.OBJ. A load module is not produced by the loader, therefore, only PROG2.COBOL, PROG2.OBJ, and INPUT.DATA are deleted at the end. It is assumed you are using your user identification as part of the data set names.

```
READY
edit input.data new
INPUT
.
.
.
.
.
.       input data
.
.
.
EDIT
save
EDIT
end
READY
allocate dataset(input.data) file(input) old
READY
allocate file(work) new block(100) space(300,10)
READY
allocate dataset(*) file(print)
READY
loadgo prog2 map print(*)
.
.
.
.
.       loader listings and output from your program
.
.
.
READY
delete   (prog2.* input.data)
READY
```

Figure 13. Loading a Program

Section V: Testing a Program at a Terminal

The operating system provides you with facilities to test your program from the terminal. They are the test facilities, if any, provided by your compiler, and the TSO TEST command. The compiler test facilities are described in the publications associated with the compiler. A brief description of the TEST command follows.

The TEST command allows you to "debug" your program. That is, it helps you to test a program for proper execution and to find programming errors. To use TEST effectively, you should be familiar with the assembler language. If you are using another language, for example COBOL, you can still use the TEST command to obtain listings and other information to give to your installation's system programmer who can help you debug your program. (You can use the full facilities of the TEST command to debug your program if you can correlate the statements in your source program listing to the resultant assembler language statements in the object listing.)

Refer to *TSO Command Language Reference* for a complete description of the facilities of the TEST command. If you prefer, you can elect not to test your program. Simply enter any command you wish after receiving the abnormal termination and READY messages.

If you are not an assembler language programmer, your system programmer will probably provide you with a test procedure. The most common situation he may provide for occurs when your program is executing and you receive a message that the program has abnormally terminated. If you enter a carriage return after the error message and "READY", a dump will be taken. Your other choices are to enter any command, or to enter the word 'TEST' with no operands. He may tell you to enter the TEST command and then the LOAD subcommand with the name of a program that will test your program. For example, if the name of the program that will test yours is DPTEST, use the following sequence.

```
MYPROG ENDED DUE TO ERROR +
?
SYSTEM ABEND CODE 0C1
READY
test
TEST
load (dptest)
```

If the system programmer does not give you the name of a testing program, he may instruct you to use the TEST command and a set of its subcommands that produce listings of your program and other pertinent information. For example, he could ask you to perform procedures similar to the following.:

Example 1:

```
MYPROG ENDED DUE TO ERROR +
READY
test
TEST
listpsw
SYSTEM MASK KEY AMWP INTRPT CODE ILC CC PROG MASK INSTR ADDR
  11111111   D  0101    0061      11 00    0000     067AB8
TEST
where 67ab8.
67AB8.  LOCATED AT +38 IN (load-module name.csectname) UNDER TCB
 LOCATED AT 660D0.
TEST
list 67ab8.-32n length(32)
```

First, you begin testing by entering the TEST command. You can now use the subcommands of TEST to "debug" your program.

Enter the LISTPSW subcommand to determine the address of the instruction that failed in your program. The last five characters of the PSW that is listed can then be entered with the WHERE subcommand and the system will then provide the location and the program name in which the abend occurred. When LIST is entered in the preceeding manner, the thirty-two bytes of instructions prior to the abend will be displayed.

At this time all the registers may be listed in the following manner to aid you in solving the problem.

```
list 0r:15r
```

If you wish to trace the execution of your program you may enter the following:

Example 2:

```
at +0:+200 (go)
at +32
at +8c
at +10a
go +0
```

In this case breakpoints will be set at every instruction in your program between relative addresses 0 and 200 (inclusive), stopping at the first invalid opcode encountered. Breakpoints set at relative address 32, 8C, and 10A supplement the previous settings. The last GO causes the program to resume execution from the beginning (assuming the first address contains a valid instruction). Before execution of the instruction at any of the breakpoint locations a message is printed at the terminal. If the location is other than 32, 8C, or 10A, execution continues because of the GO subcommand in the subcommand list of the first AT. Before 32, 8C or 10A are executed, the associated AT subcommand causes control to return to the terminal so that you can enter any TEST subcommands before continuing execution.

Example 3:

To supply new values for a range of registers, you can enter:

```
0r=(x'0',x'0',x'0')
```

The values specified would be assigned starting with register 0, register 1, etc. until all values in the list have been assigned.

Example 4:

If you want to display storage at a known relative address you may enter:

```
list +34
+34 47F0C220
```

If you want not only to display storage, but also to find out the absolute address associated with the relative address, you can enter:

```
list +34+0
A0680.  47F0C220
```

Example 5:

To list an area of storage greater than 256 bytes, you must use the MULTIPLE keyword of the LIST subcommand. For example, to find a module name that is a DC within the instructions of a module, enter:

```
list a0680 c 1(256) m(4)
```

(List the storage beginning at location A0680, translate into printable characters, for length 4 x 256)

When You Would Use Test

There are two basic situations in which you might want to use the TEST command:

- You want to TEST a program currently active in the system.
- You want to TEST a program not currently being executed.

You may want to TEST a currently executing program either because it has begun to abnormally terminate, or because you want to check through the current environment to see that the program is executing properly.

If a program has begun to abnormally terminate, you receive a diagnostic message from the Terminal Monitor Program and then a READY message. The TMP is in effect asking, "Do you want to terminate your program or test it?" If you respond with anything but TEST, your program is abnormally terminated by the abend routine. If, however, you issue the TEST command (no program name should be supplied), the TEST command processor is given control, and you can use the TEST subcommands to debug the defective program.

If you just want to look at the current environment of an executing program that is not terminating abnormally, enter an attention. The currently active program is not detached and the TMP responds to your interruption by issuing its usual READY message. Issue the TEST command (no program name) and the currently active program remains in storage under the control of the TEST command processor. You can then use the TEST subcommands to investigate the current storage situation.

Note: that in the case of both the ABEND and the attention interruption, you do not enter a program name following the TEST command. If you enter the TEST command followed by the name of the currently active program, you lose the current in-storage copy of the program and TEST loads a new copy.

The second use of the TEST command processor, testing a program not currently being executed, requires that you enter a program name along with the TEST command. When the Terminal Monitor Program issues a READY message to request a command, enter the command, TEST program name. (There are other optional operands of the TEST command but they are not necessary for this example.) The TEST command processor is given control and it loads a copy of the named program. The program can be newly written TMP, CP, or applications program.

Programs to be tested in this manner must be linkage edited members of partitioned data sets, or object modules in sequential or partitioned data sets, loadable by the Loader program.

While the program is under the control of TEST, you can step through the program, investigate or alter the environment at any time, change instructions or register contents, force entry into various subroutines, and perform other debugging operations online and immediately.

It is this second use of TEST command processor, especially the debugging of newly written code, that this section discusses.

This section is not intended to be a complete discussion of the TEST command processor. For additional discussion of the TEST command and its operands, see *TSO Command Language Reference*. The TEST subcommands are listed in Figure 14.

Subcommand Name	Function
= (Assignment)	Assigns values to one or more locations.
AT	Establishes breakpoints at specified locations.
CALL	Initiates execution of a program at a specified address.
COPY	Moves data fields or addresses.
DELETE	Deletes a load module.
CROP	Removes symbolic addresses from the symbol table.
END	Terminates all functions of the TEST command.
EQUATE	Adds symbolic address to the symbol table.
FREEMAIN	Frees a specified number of bytes of real storage.
GETMAIN	Acquires a specified number of bytes of real storage for use by the program being processed.
GO	Restarts a program at the point of interruption or at a specified address.
LIST	Displays the contents of specified areas of real storage or the contents of specified registers.
LISTDCB	Lists the contents of a Data Control Block (DCB). You must specify the address of the DCB.
LISTDEB	Lists the contents of a Data Extent Block (DEB). You must specify the address of the DEB.
LISTMAP	Displays a storage map of any real storage assigned to a program.
LISTPSW	Displays the Program Status Word (PSW). You may specify the address of any PSW.
LISTTCB	Lists the contents of the Task Control Block (TCB). You may specify the address of any TCB.
LOAD	Loads a program into real storage for execution.
OFF	Removes breakpoints.
QUALIFY	Establishes the starting or base location for symbolic or relative addresses; resolves external symbols within load modules.
RUN	Voids all breakpoints so that a program can execute to termination.
WHERE	Displays the absolute address of a symbol or entrypoint, and its relative location within the CSECT.

Figure 14. The TEST Subcommands

Addressing Restrictions

The TEST command processor can resolve internal and external symbolic addresses only if these addresses are available and can be obtained by TEST. Within certain limitations, symbolic addresses are available for both object modules (processed by the loader) and load modules (fetched by contents supervision). To ensure availability of symbols, use the EQUATE subcommand of TEST to define the symbols you intend to use.

External symbols, such as CSECT names, can be available for both object modules and load modules. Object modules require that the OS Loader had enough real storage to build in-storage CESD entries. LOAD modules must have been processed by the linkage editor with the TEST parameter specified, or must have been fetched to main storage by the TEST command or its LOAD subcommand.

Internal symbols are available only for load modules. You can refer to most internal symbols in load modules if you specified the TEST parameter during both assembly and link editing. Certain internal symbols, however, are not available. These include the names on EQU, DSECT, LTORG, and ORG assembler statements, and the symbolic names contained in system routines that operate in zero protection key.

Symbolic addresses normally cannot be obtained for modules fetched from data sets which have been concatenated to SYS1.LINKLIB by use of a link library list in a member of SYS1.PARMLIB. If, however, these modules are brought into real storage by the TEST command processor (with the LOAD subcommand, or as an operand on the TEST command), then the symbolic addresses within these modules are available to TEST.

If the necessary conditions for symbol processing are not met, you can use absolute, relative, or register addressing, but you cannot refer to symbols, unless you have previously defined them with the EQUATE subcommand of TEST.

Executing a Program Under The Control of TEST

Any program, if it is a linkage edited member of a partitioned data set or an object module in a sequential or partitioned data set, can be executed under the control of the TEST command processor.

Issue the command TEST followed by the program name and those operands of the TEST command that either define the program or are necessary to its operation. These operands may consists of parameters necessary to the operation of the program under test, the keyword LOAD or OBJECT depending upon whether the program is a load or an object module, and the keyword CP or NOCP depending upon whether the program to be tested is a command processor or not.

Any parameters that you specify in the TEST command are passed to the named program as a standard operating system parameter list; that is, when the program under test receives control, register one contains a pointer to a list of addresses that point to the parameters.

If the program to be tested is a command processor, include the keyword CP (the default is NOCP). The test routine creates a Command Processor Parameter List, and places its address into register 1 before loading the program.

Establishing and Removing Breakpoints Within a Program

Use the AT subcommand to establish breakpoints within the program being tested. Then issue the GO subcommand to begin execution of the program. To begin executing a newly loaded program, merely enter the subcommand GO - no address is required. When the breakpoints are encountered, as the program is being executed, processing is temporarily halted, and the message AT address, is written to the terminal. You can then examine the executing program, its registers, and data areas to see that it has been executing properly.

There are two methods of accomplishing this.

- You can specify a list of subcommands when you issue the AT subcommand. When a breakpoint is encountered, the TEST command processor issues each of the specified subcommands as if it had been entered from the terminal at that time. The subcommands execute and display the results of their execution at the terminal. If you specify GO as the last subcommand, control is automatically returned to the program under TEST at the point of interruption. If you do not specify GO as the last subcommand in the list, control is returned to you, at the terminal, after the last subcommand is executed. If you determine from the information displayed by the subcommands, that your program has executed properly up to that breakpoint, issue the GO subcommand. Your program

resumes execution at the point of interruption and continues execution until another breakpoint, or the end of the program, is reached.

- If you do not specify a list of subcommands when you issue the AT subcommand, the TEST command processors returns control to you at the terminal each time a breakpoint is encountered. You can then check on your program's execution by entering the TEST subcommands directly from the terminal.

Issue the OFF subcommand with no address operand to remove all breakpoints previously established. Issue the OFF subcommand followed by an address, a list of addresses, or a range of addresses to remove a single breakpoint, several breakpoints, or all breakpoints occurring within the range of addresses.

Displaying Selected Areas of Storage

Use the various LIST subcommands to display the contents of a specified area of real storage, registers, or various control blocks at your terminal, or to write this information to a data set. There are six variations of the LIST subcommand; they are:

LIST
LISTMAP
LISTTCB
LISTDEB
LISTDCB
LISTPSW

LIST: Use the LIST subcommand to display areas of storage or the contents of registers. The address required as an operand of the LIST subcommand can be one address, a list of addresses, or a range of addresses. The address may be specified as a symbolic address if a symbol table exists and contains the requested symbolic address. If no symbolic table exists (the program was not linkage edited or did not have a symbol table), you can use the EQUATE subcommand to create a symbolic address for any location within the program, or you can specify the address as a relative address, an absolute address, or as a register containing an address.

If you use the LIST subcommand to list information found at an address specified by a symbol contained in a symbol table, the information is displayed in the character type and the length specified in the symbol table. You can, however, override the attributes contained in the symbol table by including attribute operands on the LIST subcommand.

Use the LIST subcommand at any point during the execution of your program (use AT or an ATTENTION to stop the execution of the program), to determine whether data areas and registers contain proper data. If the data displayed is not what it should be, use the TEST subcommands to determine why the data is not as expected, or to modify the data in real storage and continue execution of the program.

LISTMAP: Use the LISTMAP subcommand to display at your terminal a map of all real storage assigned to the program under test. Some of the information displayed after issuance of the LISTMAP subcommand is:

- Region size.
- Task Control Block address.
- Program name, length, and location in real storage.
- Active Request Blocks, RB types, and the names of the programs associated with each of the RBs.

LISTTCB: Use the LISTTCB subcommand to display the entire Task Control Block of the program under test, or any fields of that TCB. The information displayed is formatted, and each field is identified according to the field names contained in the publication *OS/VS2 System Data Areas.*

If you want to display the TCB for the program under test, enter the subcommand LISTTCB with no address. If you want to display another TCB on the TCB queue, you must include the address of the TCB as an operand of the LISTTCB subcommand.

LISTDEB: Use the LISTDEB subcommand to display the Basic section and any direct access sections of any valid Data Extent Block (DEB), or any fields of that DEB. The information displayed is formatted according to the field names of the Data Extent Block as contained in the publication *OS/VS2 System Data Areas*.

The LISTDEB subcommand requires the address of a DEB as an operand.

LISTDCB: Use the LISTDCB subcommand to display the contents of a Data Control Block (DCB). The information displayed is formatted, and each field is identified according to the field names contained in the publication *OS/VS2 System Data Areas.*

The LISTDCB subcommand requires the address of a DCB as an operand. If you have created the DCB within the program under test, use the address of the DCB macro instruction used to create the DCB. You can also obtain the address of the DCB from the DEBDCBAD field of the DEB displayed with the LISTDEB subcommand.

LISTPSW: Use the LISTPSW subcommand to display the current Program Status Word or any of the PSWs at your terminal. If you issue the subcommand LISTPSW with no address following the subcommand, the current PSW is displayed at your terminal. If you want to display any of the other PSWs at your terminal, supply the address of the PSW you want to see as an operand of the LISTPSW subcommand. A list of the permanent real storage locations of all PSWs can be found in the *IBM System/370 Principles of Operation* publication.

The PSW is displayed formatted by field, i.e., system mask, key, AMWP, interruption code, ILD, CC, program mask, and instruction address.

Changing Instructions, Data Areas, or Register Contents

Once you have listed those areas of real storage that help you determine just what has occurred in your program, you can use the assignment function of the TEST command to make corrections within the real storage copy of the code, or to change the contents of data areas or registers.

Simply enter the address at which you want the new data entered, a code indicating the data type, and the new data you want entered at that address. The address must conform to the address restrictions already discussed. The new data must be contained within single quotes. The data type codes can be found in the publication *TSO Command Language Reference.*

One problem that can arise during a debugging session occurs when you want to replace a section of the program under test but the replacement code is longer than the section to be replaced. If you merely type in the beginning address of the section to be replaced, followed by a portion of code longer than the segment to be replaced, you will overlay some functional code. You can solve this problem with the GETMAIN subcommand of TEST.

Issue the TEST subcommand GETMAIN to obtain a work area in which to build your replacement segment of code. The GETMAIN subcommand writes out the address of the beginning of the real storage area it obtained for you. Use the Assignment of Values function of the TEST command to place a branch to the new area at the address in your module that begins the code you want to replace. Use the Assignment of Values function to build your

code segment in the newly written code, place a branch back to the point within your module at which you want processing to resume. You can then use the GO subcommand to restart your program at some point before the branch. Your program will execute through the branch instruction and into the new instructions and branch back into your original code. Later, you can use the LIST subcommand to display the newly written code in a form useful to you, enter it into your program with the TSO EDIT command, and reassemble your now executable module.

Forcing Execution of Program Subroutines

Certain paths through some programs are difficult to test because the combination of events leading to that path is difficult to produce.

One example of this problem is processing after return codes. Your module might respond differently according to the codes returned to it by some other module or some other, not yet written, section of code. You can use the AT subcommand to insert a breakpoint in your program at the point where it passes control to the not yet existing code; the assignment function of TEST to set register 15 to the desired return code; and the GO subcommand to begin execution of your program at the point where control would have been returned. Using this sequence of TEST subcommands, you can test your module's response to each possible return code.

Using TEST After a Program Abend

If a program running under TSO begins to ABEND, a diagnostic message containing the ABEND code is written to the terminal, ABEND processing is halted, and control is returned to either the TMP or TEST. If the program was running under the control of the TEST command processor, control is returned to TEST and you can immediately begin to use the TEST subcommands to determine the cause of the error. If the program was not running under TEST, control is returned to the Terminal Monitor Program. You can then enter the command TEST (no program name should be entered), to place the abnormally terminating program under control of the TEST command processor.

Use the ABEND code to determine the type of interruption that occurred. Issue the WHERE subcommand to determine where the interruption occurred.

The WHERE subcommand is especially helpful. If you enter the WHERE subcommand, the current instruction address is displayed at the terminal. If you then enter WHERE followed by that instruction address, WHERE responds by printing out the program name, the CSECT name, the offset of the current instruction address within the CSECT, and the address of the abnormally terminating task's TCB.

The instruction address, and the information returned by the WHERE subcommand pinpoint the point of error.

Use the LIST subcommand to display the instructions leading up to the error condition, and to display data areas and registers used in those instructions. This information should be sufficient to determine the cause of the error.

Determining Data Set Information

If you want to investigate the condition of any of your data sets, perform the following operations:

1. Use the LISTTCB subcommand to display the TCB for the terminating task.
2. Use the contents of the TCBDEB field as an operand of the LISTDEB subcommand to gain access to the Data Extent Block queue.
3. Use the contents of the DEBDCBAD field in each of the DEBs in the DEB queue, or the addresses of any DCB macro instructions coded within your program, as an operand of the LISTDCB macro instruction, to list the Data Control Blocks.

These control blocks contain the addresses of other control blocks useful in the debugging process.

Section VI: Command Procedures

When a function is to be performed frequently and the commands necessary would require considerable entry time at a terminal, the function may be performed by using a command procedure. A command procedure is a data set containing the appropriate TSO commands for performing a specific function. The command procedure data set is created with the EDIT command by specifying a CLIST data set type and then coding the commands you need. You use the EXEC command to invoke the command procedure.

Although the function remains the same, the specific values for the operands of the commands may vary each time the command procedure is invoked. To allow you the flexibility of varying the input to a command procedure, symbolic values may be coded within the procedure itself. Then, when you invoke the command procedure with the EXEC command, you can supply specific values (as operands of EXEC) that replace the symbolic values.

The execution of a command procedure may be controlled by using three control statements: PROC, WHEN, and END. The first statement in a command procedure is the PROC statement, which indicates those values within the command procedure that are to be symbolic. The WHEN statement analyzes return codes from a program or compiler invoked by the command procedure and returns control to a specified command within the procedure if a return code condition is met. The END statement is used to terminate the execution of a command procedure.

Once a command procedure has been written, you can document it by specifying what function it performs and what information it needs.

The major topics discussed in this section are the following:

- Creating Command Procedures
- Establishing Symbolic Values
- Writing a Conditional Command Procedure
- Calling Command Procedures
- Allocating a Terminal
- Nested Procedures

For additional information about the syntax and function of command procedure statements or any TSO commands used in a command procedure refer to *TSO Command Language Reference.*

Creating Command Procedures

You use the EDIT command to create a command procedure. The CLIST operand designates the data set being created as a command list (CLIST) data set - one that contains a command procedure.

Command Procedure Statements

A command procedure can also contain command procedure statements that control the execution of the procedure. The command procedure statements are:

PROC defines the symbolic values in a procedure. (See "Writing a Command Procedure with Symbolic Values.")

WHEN initiates or terminates a procedure (or a command within a procedure) according to certain conditions. (See "Writing a Conditional Command Procedure.")

END is used in the WHEN statement to end a procedure.

Writing a Simple Command Procedure

The following sample EDIT session shows how to create a CLIST data set named "listpgm" that loads and passes control to a program named "weekly", after allocating three data sets required by "weekly."

```
edit listpgm clist new

INPUT

alloc dataset(input) file(indata) old
alloc dataset(output) block(100) space(300,10)
alloc dataset(list) file(print)
call weekly
(null line)

EDIT

end

NOTHING SAVED
ENTER SAVE OR END-

save

EDIT    The data set "listpgm", containing the command
        procedure created above, is retained as a
        permanent data set.
```

Note: The fully qualified name (see naming conventions) assigned to this data set will be:

```
userid.LISTPGM.CLIST
```

Writing a Command List (CLIST) into a Partitioned Data Set

The following sample EDIT session shows how to create the same command procedure as in the previous example, but this session will place the procedure into a member of a partitioned data set, "list" member of "clistlib."

```
edit clistlib(list) clist new

INPUT

alloc dataset(input) file(indata) old
alloc dataset(output) block(100) space(300,10)
alloc dataset(list) file(print)
call weekly

save

EDIT    The system allocates a partitioned
        data set name "clistlib".  When SAVE
        is entered, the system creates the
        member "list" and stores this command
        list into it.
```

Defining a Private Command Procedure Library

A terminal user can define his own partitioned data set to contain command procedures. The sample session in the preceding paragraph shows how to define a partitioned data set named "clistlib." A member is created and given the name of the command procedure it is to contain.

Subsequent command procedures can be kept in additional members of CLISTLIB.

Once created and cataloged, this personal command procedure library (CLISTLIB data set) would have to be allocated by a potential user. The following ALLOCATE command shows how to allocate CLISTLIB as a command procedure:

```
ALLOC DATASET(CLISTLIB) FILE(SYSPROC)
```

A Compiler Command Procedure

Figure 15 shows a command procedure that invokes the PL/I (F) compiler. This procedure would be created with the EDIT command as a command list (CLIST) data set under an appropriate member name such as PLIG.

```
1 PROC 1 NAME
2 ALLOCATE DATASET(&NAME..PLI)FILE(SYSIN)
3 ALLOCATE DATA SET(&NAME..LIST) FILE(SYSPRINT) BLOCK(125)
4 SPACE(300,100)
5 ALLOCATE DATASET(&NAME..OBJ)FILE(SYSLIN) BLOCK(80) SPACE(250,100)
6 ALLOCATE FILE(SYSUT1) BLOCK(1024) SPACE(60,60)
7 ALLOCATE FILE(SYSUT3) BLOCK(80) SPACE(250,100)
8 CALL 'SYS1.LINKLIB(IEMAA)' 'LIST,ATR,XREF,STMT,MACRO'
9 FREE FILE(SYSUT1,SYSUT3,SYSIN,SYSPRINT)
```

Figure 15. A Command Procedure to Invoke the PL/I(F) Compiler

Record 1 is a PROC statement, defining a single positional parameter to be supplied by the user when the procedure is invoked, in this case, the name of his program. Whatever value the user specifies when calling the procedure will be filled into the following commands wherever '&NAME' appears.

Records 2 through 7 allocate the data sets required by the PL/I compiler. Record 2 allocates the input data set containing the source program. Although this data set is probably already allocated, since the user has most likely just created it with EDIT, this ALLOCATE command will reallocate it with the DDNAME 'SYSIN.' This data set is always OLD; NOBLOCK or SPACE values have to be supplied. The data set name will be formed from the program name supplied by the EXEC command, followed by the characters '.PLI' Two periods are necessary in the model command, since the first one indicates the following characters are to be concatenated to the supplied value. Records 3 through 5 similarly allocate and assign standard names to the data sets to hold the program listing and the object program. Since these are new data sets, the BLOCK and SPACE values must be supplied. Records 6 and 7 allocate the two utility, or temporary work, data sets the compiler needs. No data set name is specified, so a system-generated name will be assigned to them, and the data sets will automatically be deleted by a FREE command. All the other data sets will be kept and cataloged. To use the same procedure again for the same program, the user should enter DELETE commands for SYSIN and SYSPRINT.

Record 8 invokes the PL/I compiler by its load module name, and passes to it the list of options to control execution. When the compiler completes processing, the FREE command in record 9 releases all the data sets except the object module.

Figure 16 shows how the procedure might be used from the terminal. The user enters record 1, is the EXEC command invoking the procedure contained in data set "PLIF". The LIST keyword on the command specifies that each command is to be printed out at the terminal as it is executed. The system responds with records 2-9. Note that the name supplied with the EXEC command has been filled in as part of the data set name field in the ALLOCATE commands. The system continues to list commands through line 8, then notifies the user it is again ready to accept commands from the terminal with the READY message in line 9. The

user enters the LOADGO command to bring his compiled object program into storage for execution.

```
1   exec plif 'exp' list
2   ALLOCATE DATA SET(EXP.PLI) FILE(SYSIN)
3   ALLOCATE DATA SET(EXP.LIST) FILE(SYSPRINT) BLOCK(125)
    SPACE(300,100)
4   ALLOCATE DATA SET(EXP.OBJ) FILE(SYSLIN) BLOCK(80)
    SPACE(250,100)
5   ALLOCATE FILE(SYSUT1) BLOCK(1024) SPACE(60,60)
6   ALLOCATE FILE(SYSUT3) BLOCK(80) SPACE(250,100)
7   CALL 'SYS1.LINKLIB(IEMAA)' 'LIST,ATR,XREF,STMT,MACRO'
8   FREE FILE(SYSUT1|SYSUT3,SYSIN,SYSPRINT)
9   READY

10  allocate dataset(*) file(sysin)
11  READY

12  allocate dataset(*) file(sysout)
13  READY

14  loadgo exp.obj plilib
```

Figure 16. Use of a Command Procedure

If the procedure is a member of the command procedure library, the user can use the EXEC command implicitly, as shown in Figure 17. When the system does not find 'PLIF' defined in the command library, it looks for the command procedure in the command procedure library. The individual commands are not displayed at the terminal. When the procedure completes, the READY message is displayed, and the user can load his program for execution.

```
plif exp
READY
```

Figure 17. Implicit Use of EXEC Command

Establishing Symbolic Values

A Sample Proc Statement

The following PROC statement indicates that there are three positional symbolic values(&INPUT, &OUTPUT, and &LIST) and one optional symbolic value (&LINES) in the command procedure that this PROC statement applies to:

```
PROC 3 INPUT OUTPUT LIST LINES( )
```

The number '3' identifies the number of positional operands (symbolic values) to be substituted for by the potential user. If none of the symbolic values being defined are positional, a "0" must be entered following "PROC". Following the required number operand, the positional symbolic values appear (minus their ampersands). Following the positional symbolic values (if any), the keyword symbolic values appear (minus their ampersands). The command procedure defined by the PROC statement above appears in the following paragraph.

Writing a Command Procedure With Symbolic Values

A symbolic value is a symbol that is replaced by an actual value each time a command procedure executes. Without symbolic values a command procedure can perform only one fixed function. But with them, each user can tailor the command procedure to his specific needs by substituting actual values for the symbolic ones.

The PROC statement is the means by which you can define certain values in a command procedure referred to by symbolic values. The potential user actually substitutes values by

including them as operands on the EXEC command when requesting the services of your command procedure. For the syntax of the EXEC command or the PROC statement, refer to the publication: *TSO Command Language Reference.*

Symbolic Values

Symbolic values in a command procedure (identified by ampersands) are defined (without ampersands) by the operands on the PROC statement. There are two types of symbolic values used in command procedures:

Positional a required operand; it must be substituted for (in the order in which it appears) in the EXEC command entered by a potential user. It cannot exceed 252 characters.

Keyword an optional operand; it may be substituted for (in any order following the positional operands) in the EXEC command entered by a potential user. It cannot exceed 31 characters. A keyword may have a default value.

A Sample Command Procedure with Symbolic Values

The following example uses the preceding sample PROC statement to establish symbolic values in the command procedure example that appears under the earlier heading, "Writing a Simple Command Procedure":

```
PROC 3 INPUT OUTPUT LIST LINES ( )
ALLOC  DATASET( &INPUT. ) FILE( INDATA ) OLD
ALLOC  DATASET( &OUTPUT. ) BLOCK( 100 ) SPACE( 300,10 )
ALLOC  DATASET( &LIST. ) FILE( PRINT )
CALL  WEEKLY '&LINES.'
```

Note:

- When the symbolic value must be followed by a special character (i.e., a right parenthesis, apostrophe, or a period), you must end the symbolic value with a period. (See each of the symbolic values in the example above).
- The optional (keyword) symbolic value "&LINES" has been added to the basic example to show how a parameter string that is to be passed to the program named "WEEKLY" can also be designated as a symbolic value.

Assigning Defaults for Optional Symbolic Values

Another function of the PROC statement is assigning default values to optional symbolic values in a command procedure. You can assign a default value by enclosing it in parentheses that immediately follow the symbolic value in the PROC statement. Then, if the user of your command procedure fails to code an actual value for the symbolic value in question, the system will substitute your default value for it.

A PROC Statement That Assigns a Default

In the basic command procedure example being used in this section, you might want to assign the number "35" as a default value for the optional symbolic value "&LINES". The following example illustrates how this is accomplished:

```
proc 3 input output list lines( 35 )
```

Note: You cannot assign default values to positional symbolic values.

Documenting a Command Procedure with Symbolic Values

A command procedure containing symbolic values can be made available to all the programmers at an installation. The efficient use of your command procedure by the

programmers is dependent on how well it is documented. Figure 18 illustrates this documentation.

```
Command Procedure Name: CLISTLIB(LIST)

Purpose: Scan Master Input File & Print a Report
Symbolic Values:

INPUT       OUTPUT      LIST        LINES(35)

INPUT    - Required.    Replace with name of input data set.
OUTPUT   - Required.    Replace with name of output data set.
LIST     - Required.    Replace with name of output data set.
LINES(35)  Optional.    Code the parameter string value you
                        want passed to the load module named
                        "WEEKLY" when it receives control.
                        If you do not code this value, the default
                        value "35" is passed to "WEEKLY".
Command Procedure:

    PROC 3 INPUT OUTPUT LIST LINES(35)
    ALLOC DATASET(&INPUT.) FILE(INDATA) OLD
    ALLOC DATASET(&OUTPUT. BLOCK9100) SPACE(300,10)
    ALLOC DATASET(&LIST.)FILE(PRINT)
    CALL WEEKLY '&LINES.'
```

Figure 18. Documentation of a Command Procedure with Symbolic Values

Examples of Symbolic Substitution

- Positional parameters (These must be specified on the EXEC command).

  Here is the procedure statement for a CLIST data set called PR1:

  ```
  PROC 3 PARM1 PARM2 PARM3
  ```

  If the user enters at the terminal

  ```
  exec pr1 10 20 input
  ```

  TSO makes the following substitutions within the command procedure:

  ```
  10      replaces    &PARM1
  20      replaces    &PARM2
  INPUT   replaces    &PARM3
  ```

- Keyword parameters (these need not be specified on the EXEC command).

  Here is the procedure statement for a CLIST data set called PR2:

  ```
  PROC 0 KEY1 KEY2(1) KEY3(10)
  ```

  (The zero indicates there are no positional parameters)

Figure 19 describes the results of substitution within the command procedure:

EXEC COMMAND ENTERED AT TERMINAL	VALUES SUBSTITUTED FOR &KEY1	&KEY2	&KEY3
exec pr2	null string	null string	10
exec pr2 'key1 key2 key3'	KEY1	null string	null string
exec pr2 'key1(8) key2(input) key3(5)'	(not valid)	INPUT	5

Figure 19. Substitution Using Keyword Parameters

Writing a Conditional Command Procedure

The WHEN statement can be used in any command procedure that contains a CALL or a LOADGO command. It allows you to insert processing checkpoints at strategic points in the procedure. If a program should experience an error condition, any commands in the command procedure that have not been executed would be executed, regardless of the error condition. To prevent this potentially wasteful processing within a command procedure, the WHEN statement has been included as a command procedure function.

Using the WHEN Statement

The WHEN statement is the means by which you can test the return codes of programs that have been invoked with a CALL or a LOADGO command from within a command procedure. If the return you test for is found, you can either end the procedure, or else execute another command or procedure at that point. If the return code you test for is not found, the remainder of your procedure will execute normally.

SYSRC Operand of WHEN

You use the SYSRC operand to specify the return code, or range of return codes, that you want to test for (from programs invoked by an immediately preceding CALL or LOADGO command, only).

Executing an Alternate Procedure

Again refering to the sample command procedure "LISTPGM", assume that you want to execute a backup procedure that resides in the data set named "COSCO.CHECKOUT.CLIST", if the program "WEEKLY" produces a return code of 8. The following WHEN statement would be inserted:

```
ALLOC DATASET(INPUT) FILE(INDATA) OLD
ALLOC DATASET(OUTPUT) BLOCK(100) SPACE(300,10)
ALLOC DATASET(LIST) FILE(PRINT)
CALL WEEKLY
WHEN SYSRC(= 8) EXEC CHECKOUT
```

Note:

- When CHECKOUT completes processing, the terminal returns to command mode or to the command procedure that invoked LISTPGM.
- The complete syntax of the WHEN statement appears in the publication: *TSO Command Language Reference.*

Executing an Alternate Command

Again refering to the sample command procedure "LISTPGM", assume that you want to execute a LIST command if the program "WEEKLY" produces a return code greater than 8. The following WHEN statement would be inserted:

```
.....
CALL WEEKLY
WHEN SYSRC(GT 8) LIST COSCO.DEBUG
.....
```

Ending a Command Procedure Strategically

Assume that you want to end a command procedure prematurely if a given CALL command produces a return code less than 8. The following WHEN statement would be inserted:

```
.....
CALL 'SYS1.LINKLIB(IEQCB100)' 'NODECK'
WHEN SYSRC(LT 8) END
.....
```

Testing Conditions for Termination

The programs invoked with a CALL or LOADGO command can issue a return code (a number to indicate its relative 'success'). The return codes of IBM-supplied programs are listed in the publications associated with the program. Only those user programs written in the assembler language or PL/I can issue return codes. (For description of how to issue return codes, see *Assembler F Programmer's Guide* and *PL/I (F) Programmer's Guide.)* User return codes are usually standardized in each installation.

You can insert a WHEN statement after any CALL or LOADGO command or a processor (such as a compiler or link editor) in the command procedure to test its return code. If the test you request is true, you have the option of ending the command procedure or of executing another procedure or another command. If the test you request is not true, the command procedure will continue its course. The test is specified with the SYSRC operand of the WHEN statement. For example, assume that you want to end a procedure named proc4 if a given CALL command produces a return code of 8. Enter the following WHEN statement after the command you want to test:

```
.
.
call 'sys1.linklib(ieqcb100)' 'nodeck'
when sysrc(eq 8) end
.
.
.
```

If instead of ending proc4 when the test is true, you want to execute another procedure that resides in the JONES.PROC5.CLIST data set, enter:

```
.
.
.
when sysrc(eq 8) exec proc5
.
.
.
```

If the test is true, proc5 will replace the procedure that requested its execution. When proc5 is done, no other commands in proc4 will be executed. Now the system will be ready for a command from the terminal or will return to the command procedure that invoked proc4. If instead of executing a procedure, you want to enter a LIST command, enter:

```
.
.
.
when sysrc(eq 8) list pgm.list snum
.
.
.
```

Ending the Command Procedure

You may write an END statement after the last line of the command procedure. When the system encounters an END statement in a command procedure it sends a READY message to the terminal so you can enter another command.

Calling Command Procedures

You use the EXEC command to call a CLIST data set, or a member of the Command Procedure Library, in order to execute the command procedure that resides therein.

Calling a Command Procedure in a CLIST Data Set

The following sample EXEC command shows how to call the command procedure "LISTPGM" that appears in a previous paragraph entitled, "Writing a Simple Command Procedure."

```
EXEC LISTPGM.CLIST
or
EXEC LISTPGM
```

Note:

- CLIST, the descriptive qualifier of the command procedure name, is the default value. Thus, it is sufficient to code the command procedure name without the descriptive qualifier if the command procedure resides in the CLIST data set.
- Command procedures do not have to be stored in CLIST data sets.

Calling a Command Procedure in a Command Procedure Library

An installation's command procedures are stored in a partitioned data set known as the Command Procedure Library.

Implicit Form of the Exec Command

To call a command procedure that is stored in a member of a command procedure library data set, you may use the implicit form of EXEC. If this is the case, you do not code EXEC on the command. The following example of the implicit form of the EXEC command shows how to call the command procedure that is stored in the member named "COMPILE" of your installation's command procedure library:

```
compile
```

Note: The system searches the command library, before it searches the command procedure library, for the name that you enter. Therefore, a member in your command procedure library should not have the same name as a command in your command library.

Calling a Command Procedure in any Other Data Set

Normally, a command procedure is stored in either a CLIST data set or in a member of a command procedure library. However, if a command procedure happens to be stored in any other type data set, it can only be called by entering the fully qualified data set name, enclosed in apostrophes, with the EXEC command. The following sample EXEC command shows how to call a command procedure that is stored in a data set named "userid.COMPROC.CP":

```
EXEC 'userid.COMPROC.CP'
```

Calling a Command Procedure With Symbolic Values

When you decide to use one of your installation's command procedures, first decide which of the symbolic values you must replace with your own values. Then decide which of the optional symbolic values (if any) you want to use. Don't forget to consider the optional symbolic values that have defaults. Then enter your own values as operands on the EXEC command that you use to call the command procedure. Your values must follow the name of the data set, or of the data set member, that contains the procedure. The required values must be entered in the same order in which they appear on the PROC statement that defines the procedure you are calling. Optional values must be entered following the required values, in any order. The following example shows how to call the command procedure named "CLISTLIB(LIST)", that is documented in Figure 14.

Known: The name of the master tape data set containing last week's inventory . . . . . PH.OLDDATA.DATA

The name of the master type data set to contain this week's inventory PH.NEWDATA.DATA

The name of the output data set that will go to the terminal user . . . . PH.OUTDATA.DATA

The parameter value to be passed to the program named "WEEKLY" 10

Enter: exec clistlib list olddata newdata outdata lines(10)

Allocating a Terminal

In TSO programming, you can use the ALLOCATE command to allocate your terminal as a data set, for either input or output. The ALLOCATE command is discussed in the publication: *TSO Command Language Reference.* You can also allocate your terminal from within a command procedure.

How to List Output at Your Terminal

An installation can include an optional symbolic value in command procedures to allow their programmers to have their own listings printed out at their terminals if the output is generated by a command procedure.

Figure 20 shows the sample documentation of a command procedure named "LISTUPDT(NEWPART)". This command procedure contains an optional symbolic value, "OUTPUT(*)", that can allocate the user's terminal as an output data set.

```
Command procedure Name: LISTUPDT (NEWPART)
Purpose: Update Inventory List
Symbolic Values:

    WEEKIN      WEEKOUT         NEW     OUTPUT(*)

WEEKIN      - Required.     Replace with name of input data set.
WEEKOUT     - Required.     Replace with name of output data set.
OUTPUT(*)   - Optional.     Routes reports generated by this procedure
                            to your terminal. If you want to route
                            the report to a data set, replace
                            the * with the data set name.

Command Procedure:

PROC 2 WEEKIN WEEKOUT OUTPUT(*)
ALLOC DATASET(&WEEKIN.) FILE(INPUT) OLD
ALLOC DATASET(&WEEKOUT.) FILE(OUTPUT) NEW
ALLOC DATASET(&OUTPUT.) FILE(REPORT)
CALL INVUPDT
```

Figure 20. Allocating a Terminal in a Command Procedure

Nested Procedures

A command procedure can be made into a compile-load-go sequence -- the equivalent of the RUN command -- by using the procedure nesting and conditional execution capabilities. For instance, in Figure 21, note that the user enters two ALLOCATE commands, defining terminal input and output for execution time, and a LOADGO command to invoke his program. Like the commands used to invoke the compiler, these would normally be used every time the user wants to invoke his program, and therefore can be reasonably placed in a command procedure. This second procedure can be called from the compiler-invoking procedure, making it a compile-load-go procedure.

The procedure to load and execute the user program might be defined under a suitable name such as LDGO. The FREE command in record 2 is the same as the one in the PLIF procedure. It needs to be repeated here since it will not be executed in that procedure, as explained below. Records 3 and 4 allocate the terminal for SYSIN or SYSPRINT I/O statements in the user program, and statement 5 is the LOADGO command causing the program to be brought into storage and given control.

```
1   PROC 1 NAME1
2   FREE FILE(SYSUT1,SYSUT3,SYSIN,SYSPRINT)
3   ALLOCATE DATASET(*) FILE(SYSIN)
4   ALLOCATE DATASET(*) FILE(SYSPRINT)
5   LOADGO &NAM1..OBJ PLILIB
6   END
```

Figure 21. A Command Procedure to Invoke a User Program

It would be possible to call this procedure from the PLIF procedure by inserting a record containing:

```
exec ldgo '&name'
```

However, it would be preferable to call it only when the return code from the compiler indicates successful execution is likely, that is, no serious errors were detected during compilation. To test the compiler return code, the user inserts a WHEN statement:

```
when sysrc(le 4) exec ldgo '&name'
```

The WHEN statement immediately follows the CALL command invoking the compiler. If the compiler return code is less than or equal to four ('LE 4'), indicating that no errors or only minor errors were detected, the EXEC command is executed, and the procedure ends. If the return code is greater than four, the EXEC command will be ignored, the FREE command is executed, and the procedure ends. The terminal returns to command mode, and the user will probably use the LIST command to display the compiler listing, determine the errors in the source program, correct them with the EDIT command, and reinvoke the procedure for another compilation. Figure 22 shows the modified PLIF command procedure. A DELETE command has been added for the object module, since it is not executable. Figure 23 shows a use of the procedure for a successful compilation. The LIST operand is specified to display each command as it is executed.

```
PROC 1,NAME
ALLOCATE DATASET( &NAME..PLI ) FILE(SYSIN)
ALLOCATE DATASET( &NAME..LIST) FILE(SYSPRINT) BLOCK( 125 ) SPACE( 300,100 )
ALLOCATE DATASET( &NAME..OBJ ) FILE(SYSLIN) BLOCK( 80 ) SPACE( 250,100 )
ALLOCATE FILE(SYSUT1 ) BLOCK( 1024 ) SPACE( 60,60 )
ALLOCATE FILE(SYSUT3 ) BLOCK( 80 ) SPACE( 250,100 )
CALL 'SYS1.LINKLIB( IEMAA )' 'LIST,ATR,XREF,STMT,MACRO'
WHEN SYSRC( LE 4 ) EXEC LDGO '&NAME.'LIST
FREE FILE( SYSUT1,SYSUT3 )
DELETE &NAME..OBJ
END
```

Figure 22. A Command Procedure for a Compile-Load-GO Sequence

```
exec plif 'derv' list
ALLOCATE DATASET( DERV.PLI ) FILE( SYSIN )
ALLOCATE DATASET( DERV.LIST ) FILE( SYSPRINT ) BLOCK( 80 ) SPACE( 300,100 )
ALLOCATE DATASET( DERV.OBJ ) FILE( SYSLIN ) BLOCK( 80 ) SPACE( 250,100 )
ALLOCATE FILE( SYSUT1 ) BLOCK( 1024 ) SPACE( 60,60 )
ALLOCATE FILE( SYSUT3 ) BLOCK( 80 ) SPACE( 250,100 )
CALL 'SYS1.LINKLIB( IEMAA )' 'LIST,ATR,XREF,STMT,MACRO'
WHEN SYSRC( LE 4 ) EXEC LDGO 'DERV' LIST
FREE FILE( SYSUT1,SYSUT3,SYSIN,SYSPRINT )
ALLOCATE DATASET( * ) FILE( SYSIN )
ALLOCATE DATASET( * ) FILE( SYSPRINT )
LOADGO DERV.OBJ PLILIB
```

Figure 23. Using a Compile-Load-Go Command Procedure

Index

Where more than one page reference is given, the major reference is first.

Indexes to systems reference library manuals are consolidated in the **OS/VS2 Master Index**, GC28-0663. For additional information about any subject in this index, refer to other publications listed for the same subject in the **Master Index.**

GC28-0645-1

OS/VS2 TSO Terminal User's Guide (S370-39) Printed in U.S.A. GC28-0645-1

IBM®

International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, New York 10604
(U.S.A. only)

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
(International)

OS/VS2 TSO Terminal User's Guide

GC28-0645-1

READER'S COMMENT FORM

Your views about this publication may help improve its usefulness; this form will be sent to the author's department for appropriate action. Using this form to request system assistance or additional publications will delay response, however. *For more direct handling of such requests, please contact your IBM representative or the IBM Branch Office serving your locality.*

Possible topics for comment are:

Clarity Accuracy Completeness Organization Index Figures Examples Legibility

Cut or Fold Along Line

What is your occupation? _____

Number of latest Technical Newsletter (if any) concerning this publication: _____

Please indicate in the space below if you wish a reply.

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.

Cut or Fold Along Line

Your comments, please . . .

This manual is part of a library that serves as a reference source for system analysts, programmers, and operators of IBM systems. Your comments on the other side of this form will be carefully reviewed by the persons responsible for writing and publishing this material. All comments and suggestions become the property of IBM.

Fold Fold

First Class
Permit 81
Poughkeepsie
New York

Business Reply Mail
No postage stamp necessary if mailed in the U.S.A.

Postage will be paid by:

International Business Machines Corporation
Department D58, Building 706-2
PO Box 390
Poughkeepsie, New York 12602

Fold Fold

IBM

International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, New York 10604
(U.S.A. only)

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
(International)

OS/VS2 TSO Terminal User's Guide (S370-39) Printed in U.S.A. GC28-0645-1