Publication Number
SC34-4018-0

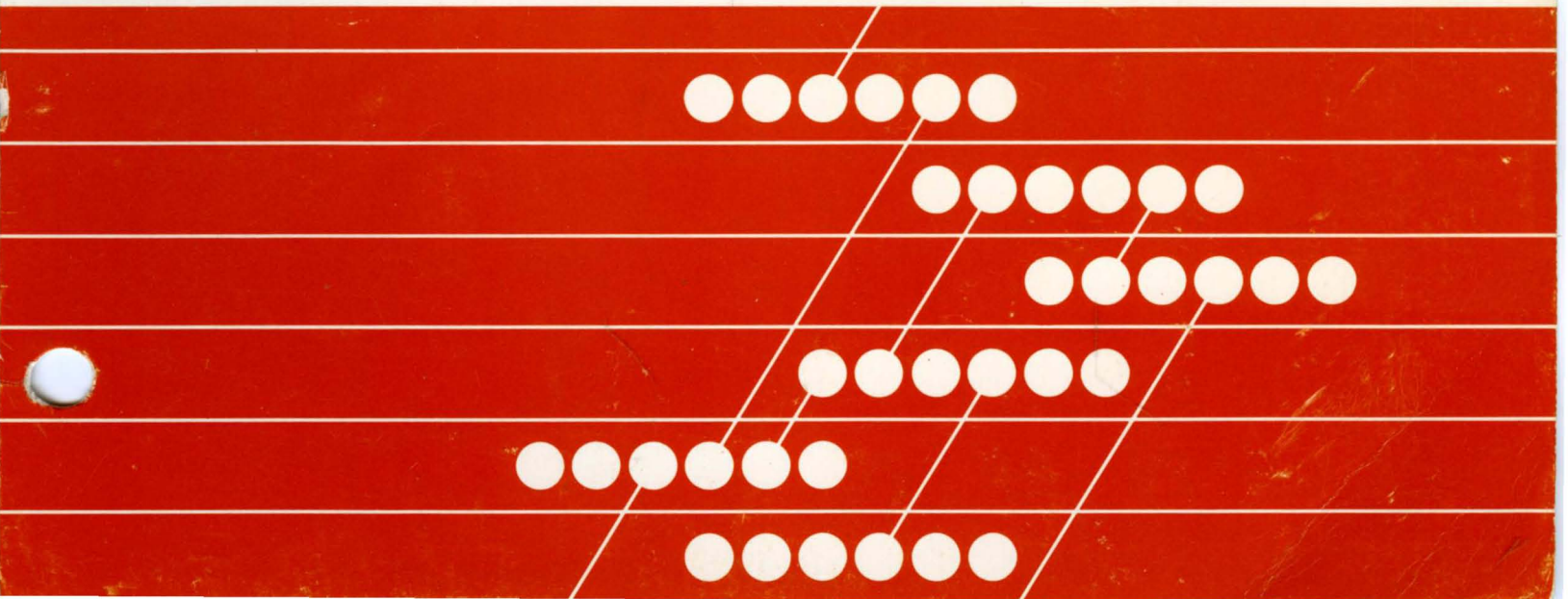
Interactive System Productivity Facility/ Program Development Facility

Version 2

Edit Macros

MVS/Extended Architecture

IBM



Interactive System Productivity Facility/ Program Development Facility

Version 2

Edit Macros

MVS/Extended Architecture

Publication Number
SC34-4018-0

File Number
S370/4300-39

Program Number
5665-317

First Edition (December 1984)

This is a new manual and supports ISPF for MVS/XA and MVS/370. You should use this manual if you have ISPF with an MVS/XA environment or if you are using APL2 with ISPF in either an MVS/370 or an MVS/XA environment. This manual is a revision of, but does not obsolete, SC34-2134-0 for ISPF/PDF for MVS. Changes or additions to the text and illustrations are indicated by a vertical line to the left of the change or addition.

This edition applies to version 2, release 1, modification 2 of the Interactive System Productivity Facility (ISPF) Program Product, (Program Number 5665-319) and to the ISPF/Program Development Facility (ISPF/PDF or PDF) Program Product (Program Number 5665-317) for use with OS/VS2 MVS Release 3.8 or MVS/SP Release 1.1.1 and to all subsequent releases until otherwise indicated by new editions or technical newsletters.

Changes are made periodically to the information herein; before using this publication in connection with the operation of IBM systems, consult the latest IBM System/370 and 4300 Processors Bibliography, GC20-0001, for the editions that are applicable and current.

It is possible that this material may contain reference to, or information about, IBM products (machines and programs), programming, or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such IBM products, programming, or services in your country.

Publications are not stocked at the address given below. 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 reader's comments is provided at the back of this publication. If the form has been removed, comments may be addressed to IBM Corporation, Dept. T46, P. O. Box 60000, Cary, North Carolina, 27511. IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

PREFACE

The Interactive System Productivity Facility (ISPF) and the ISPF/Program Development Facility (ISPF/PDF or PDF) are related IBM program products. Together, they are designed to improve user productivity in the development of applications, and contain special functions for the development, test, and use of interactive applications, called dialogs. Specifically:

- ISPF is a dialog manager for interactive applications. It provides control and services to permit execution of dialogs.

- PDF is a facility that aids in the development of dialogs and other types of applications. It makes use of display terminals and an interactive environment to assist with many of a programmer's tasks.

This manual describes the edit macro facility, the statements that are available, and how to code them. It assumes that you are an application or systems programmer, engaged in program development, and are familiar with coding CLISTs in the MVS environment and with the ISPF/PDF editor.

Use this publication with the following publications:

OS/VS2 TSO Command Language Reference (GC28-0646) - Provides reference information about the CLIST statements coded in PDF edit macros.
OS/VS2 TSO Terminal User's Guide (GC28-0647) - Provides usage information about the CLIST statements coded in PDF edit macros.

The following publications provide further information about ISPF and PDF:

ISPF/PDF for MVS/XA Program Reference (SC34-4024) - Provides information on the use of PDF under MVS.

ISPF for MVS/XA Dialog Management Services (SC34-4021) - Provides a detailed description of the dialog management services and related information required to develop an interactive application that runs under ISPF.

ISPF for MVS/XA Dialog Management Services Examples (SC34-4022) - Provides a set of examples of the use of dialog management services.

ISPF and ISPF/PDF for MVS/XA Installation and Customization (SC34-4019) - Provides information needed to install ISPF and ISPF/PDF and to custom tailor these products for a particular installation.

ISPF/PDF for MVS/XA Services (SC34-4023) - Provides a detailed description of the dialog management services required to develop an interactive application that runs under ISPF with PDF.

ISPF/PDF for MVS/XA Library Management (SC34-4025) - Provides a description of the library management facilities available with ISPF/PDF.

In this document, the following notation conventions are used to describe macro formats:

- Uppercase commands and their operands should be entered as shown, but not necessarily in uppercase. Operands shown in lowercase are variable; you substitute your own value for them.

- Operands shown in brackets [ ] are optional, with a choice indicated by the OR symbol (|) or by stacking the operands. You may choose one or none; the defaults are underscored.

- Operands shown without brackets are required. If several operands are separated by the OR symbol (|) or are stacked and shown in braces { }, you must select one of the choices.

- Command name truncations are shown stacked under the full command name without braces.

CONTENTS

FIGURES

CHAPTER 1. INTRODUCTION

The ISPF/Program Development Facility (ISPF/PDF, or PDF) provides you with the ability to write and execute macros for use with the PDF editor.

Edit macros allow you to:

- **Extend** edit with macro commands.
- **Override** existing PDF edit commands.
- **Specify** an initial macro to be executed automatically when edit is invoked. This initial macro receives control after the data has been read, but before the data has been displayed.
- **Access**:
  - The data currently being edited (line by line).
  - The current cursor position and the location of the screen "window".
  - Edit modes and other related information, including the mask line, tabs line, and boundaries.
- **Access external data** from a library or data set using library access services.
- **Invoke** any ISPF or PDF dialog service.

WHY USE EDIT MACROS?

As a user of PDF edit, you know that there are a large number of edit commands that can be used in editing a program or document. When you are at your terminal, you can use edit commands such as FIND, RESET, SAVE, CHANGE, and LOCATE. Lines can be inserted or deleted, and you can add or change data by overtyping data that is already displayed on the screen.

For simple editing, you may be able to enter a command that does exactly what you want to do. If you want to find the word 'COFFEE', for example, you can enter the command ==> FIND WORD 'COFFEE'.

For more complex tasks, there may not be a command that does what you want to do. You may have to execute a series of commands to accomplish the task. Suppose that you want to delete every line that begins with a '-' in column 1, except for the first such line.

One approach would be to visually scan the file and manually delete the lines. This approach would work well if there were only a few lines to be deleted from a small file.

A second approach would be to perform a series of edit commands to accomplish the task. For example, to delete the lines, you could enter:

```
===> RESET EXCLUDED
===> EXCLUDE ALL '-' 1
===> FIND FIRST  '-' 1
===> DELETE ALL EXCLUDED
```

This second approach would be a better approach if there were several hundred lines to be deleted. The RESET command would reset any excluded lines in the file. The EXCLUDE command would exclude all of the lines with a hyphen (-) in column 1. The FIND command would show (unexclude) the first such line. And the DELETE command would then delete the lines that remained excluded.

CLEANUP Macro

With PDF edit macros, there is a third approach, one that would be appropriate if the task has to be done many times. The third approach would be to write a macro named CLEANUP that includes the four edit commands. Once the macro was written, you could perform the line deletion simply by entering CLEANUP as an edit command. Edit would then execute the four commands, one after another, performing the task in a single interaction.

Thus, by writing a macro named CLEANUP, you would have in effect created a new edit command. In this manual, we will normally refer to the built-in commands such as FIND and END as 'commands' and edit macro commands such as CLEANUP as 'macros'. Later on, we will see how an edit macro can be written to replace the END command. (Refer to "Defining Macros" on page 32.) When discussing special cases like this where confusion might result, the terms 'edit macro command' and 'edit built-in command' will be used.

To an end user, there is little difference between an edit macro and an edit built-in command. Edit macros are executed exactly the same way that edit commands are executed. You simply type the macro name in the command field on the edit data display, along with any operands that may be required, and then press ENTER.

This manual assumes that you are familiar with the CLIST language. Macros can be written in either CLIST language or compiled or assembled languages. A discussion of coding and invoking program macros will be found in Chapter 6, "Writing Program Macros" on page 55.

Let's take a look at the implementation of the CLEANUP macro in the CLIST language. The macro (Figure 1) consists of six lines that are stored as a member of a CLIST partitioned data set. The name of the

member is CLEANUP, and the PDS must be part of your SYSPROC concatenation.

```
/*                                                                  */
/*   CLEANUP MACRO - DELETE LINES WITH A '-' IN COLUMN 1            */
/*                                                                  */
ISREDIT MACRO
  ISREDIT RESET EXCLUDED                  /* Ensure no lines are excluded  */
  ISREDIT EXCLUDE ALL '-' 1               /* Exclude lines with '-' in col1*/
  ISREDIT FIND FIRST  '-' 1               /* Show the first such line      */
  ISREDIT DELETE ALL EXCLUDED             /* Delete all lines left excluded*/
EXIT CODE(0)
```

Figure 1. CLEANUP Macro

Don't worry if you don't understand everything about this macro and the macros that follow in this chapter. They are included only as illustrations.

The lines that begin with ISREDIT are edit commands, and are documented in this manual. The first such command identifies this CLIST as being an edit macro. Every edit macro must include an ISREDIT MACRO statement. The last statement, EXIT, is a CLIST statement. While the EXIT statement is not required by the CLIST processor, we will include it to clearly identify the ending of the macro. Indentation is used to show the structure of the macro. In this manual, complete valid macros will always be shown beginning with an ISREDIT MACRO command and ending with an EXIT statement.

Note: Do not code a CLIST PROC statement at the beginning of your edit macro.

TESTGEN Macro

In the CLEANUP macro, four commands were executed, one after another, just as though you had executed the four commands separately. You may find this approach useful for simple tasks; for more complex tasks, however, you will need to repeat some steps and execute other steps only under certain conditions.

For example, suppose that you wanted to generate some test data. The following macro (Figure 2) changes the first TEST-# to TEST-1, the second TEST-# to TEST-2, up through nine occurrences of TEST-#.

```
/*                                                                    */
/*  TESTGEN MACRO - GENERATE TEST DATA                                */
/*                                                                    */
ISREDIT MACRO
  SET &COUNT = 1                          /* Initialize loop counter  */
  DO WHILE &COUNT <= 9                    /* Loop up to 9 times       */
    ISREDIT FIND 'TEST-#'                 /* Search for 'TEST-#'      */
    SET &RETCODE = &LASTCC                /* Save the FIND return code*/
    IF &RETCODE = 0 THEN                  /* If string was found,     */ -
      DO                                  /*                          */
        ISREDIT CHANGE '#' '&COUNT'       /*   Change # to a digit and*/
        SET &COUNT = &COUNT + 1           /*   increment loop counter */
      END                                 /*                          */
    ELSE                                  /* If string is not found,  */ -
      SET &COUNT = 10                     /*   Set counter to exit loop*/
  END                                     /*                          */
EXIT CODE(0)
```

Figure 2. TESTGEN Macro

This macro has limited use, but it illustrates how logic can be included in an edit macro. The CLIST DO WHILE statement loops up to nine times, finding and then changing data. The IF and ELSE statements cause different statements to be executed, depending on whether the 'TEST-#' string was found or not.

STRCOUNT Macro

A general-purpose macro is often required. Such a macro might be passed a parameter to indicate data which is to be processed. It might need to retrieve information known to the editor. And it could return information to the end user by setting up a message to be displayed by the editor.

The following macro (Figure 3) is written to be passed a string of characters. After searching the file, it displays a message that indicates how many times the string was found. This example is included to show that a macro can get information from the end user, that it can get information from the editor, and that it can display a message to the end user.

```
/*                                                                  */
/* STRCOUNT - COUNT THE NUMBER OF OCCURRENCES OF A STRING           */
/*                                                                  */
ISREDIT MACRO (PARMSTR)
  ISREDIT SEEK ALL &PARMSTR
  ISREDIT (COUNT) = SEEK_COUNTS
  SET &COUNT = &COUNT
  SET &ZEDSMSG = &STR("&PARMSTR" FOUND &COUNT TIMES)
  SET &ZEDLMSG = &STR(THE STRING "&PARMSTR" WAS FOUND +
                      &COUNT TIMES IN THE FILE.)
  ISPEXEC SETMSG MSG(ISRZ000)
EXIT CODE(0)
```

Figure 3. STRCOUNT Macro

The ISREDIT MACRO command includes the name of the variable &PARMSTR into which edit stores the parameter. The ISREDIT SEEK command is used for find all occurrences of the string. It is identical to the FIND command, except that it does not alter the exclude status of a line, and therefore is recommended for use in macros. The line following the SEEK command assigns the SEEK_COUNTS to the variable &COUNT. The SET &COUNT = &COUNT statement strips leading zeros from the variable &COUNT, to prepare it for display. The SET &ZEDSMSG = ... statement sets the "edit short message variable", and the SET &ZEDLMSG = ... statement sets the "edit long message variable". These two variables make up the short and long messages for ISRZ000, which is referenced in the dialog manager service SETMSG. Finally, the macro exits, at which point the short message is displayed by the editor. If the user enters HELP, the long message is then displayed.

If you enter the macro exactly as it is shown, and then enter the SAVE command, you can test the macro by executing it while you are still editing it. If you enter:

```
  COMMAND ===> STRCOUNT ISREDIT
```

the short and long messages which would be displayed would be:

```
                                                  "ISREDIT" FOUND 3 TIMES
COMMAND ===>
THE STRING "ISREDIT" WAS FOUND 3 TIMES IN THE FILE.
```

EDIT MACRO ELEMENTS

An edit macro is a CLIST. As such, it is made up of CLIST statements, each of which falls into one of the following categories.

- CLIST command procedure statements
- Edit macro statements
- ISPF and PDF dialog services requests
- TSO commands and subcommands

All statements are initially processed by the TSO command processor, which scans them, and performs symbolic substitution. It is important to distinguish among these different kinds of statements because:

- They are processed by different components of the system.
- They have different syntax rules and error handling.
- Their descriptions will be found in different documents.

CLIST Command Procedure Statements

Command procedure statements provide for the handling of CLIST variables, and for control flow within a CLIST. When a command procedure statement is encountered during the execution of a CLIST, it is processed by the TSO command processor. Some of the command procedure statements which are commonly seen in PDF edit macros are:

- SET statement
- IF-THEN-ELSE statement
- DO-WHILE-END sequence
- EXIT statement

For a complete list and description of the command procedure statements see OS/VS2 TSO Command Language Reference.

Edit Macro Statements

Any statement that begins with ISREDIT is assumed to be an edit macro statement. When such a statement is encountered during the execution of a CLIST, the PDF editor is given control. The editor then processes the statement, performing any requested functions. Edit macro statements are the only statements that are processed by the PDF editor. Examples of edit macro statements are:

```
ISREDIT FIND "TEST475"
ISREDIT BOUNDS = 1,60
ISREDIT (WIDTH) = LRECL
ISREDIT PROCESS
```

A description of each macro statement can be found in Chapter 7.

ISPF and ISPF/PDF Dialog Service Requests

Any statement which begins with ISPEXEC is assumed to be an ISPF or PDF dialog service request. When such a statement is encountered during the execution of a CLIST, the appropriate ISPF or PDF service is executed. Some examples of service requests which might be found in a PDF edit macro are:

```
ISPEXEC SETMSG ...
ISPEXEC VPUT ...
ISPEXEC DISPLAY ...
ISPEXEC EDIT ...
ISPEXEC LMINIT ...
```

ISPF service requests are described in ISPF for MVS/XA Dialog Management Services. PDF service requests are described in ISPF/PDF for MVS/XA Services.

TSO Commands and Subcommands

Any statement that is not recognized as a command procedure statement and does not begin with ISPEXEC or ISREDIT is assumed to be a TSO command or subcommand. TSO commands can be either CLISTs or programs. When a TSO command is encountered, it is executed. Examples of TSO commands are:

```
ALLOCATE ...
FREE ...
DELETE ...
RENAME ...
```

TSO subcommands are statements which are processed by a TSO command. Examples are TSO EDIT subcommands which are processed by the TSO EDIT command, and TSO TEST subcommands which are processed by the TSO TEST command.

For a complete list and description of the TSO commands and subcommands see TSO Command Language Reference.

CHAPTER 2. BASIC EDIT MACRO CONCEPTS

In this chapter, we will examine some of the basic concepts used in writing edit macros. Some of the topics will be explained more fully in Chapter 4, "Advanced Edit Macro Concepts" on page 23.

VARIABLES

An edit macro is simply a CLIST which may contain edit macro statements. If you understand how variables are handled in a CLIST, you also understand their use in edit macros. Since variables are important in any CLIST, let's review a few important points.

CLISTs can contain three types of variables, all of which begin with an ampersand (&). They are:

- Symbolic variables
- Control variables
- Built-in functions

Symbolic variables can be set by using a CLIST SET statement. For example, if you code

```
SET &COUNT = 3
SET &LABEL = TEST14
```

the first statement sets the variable &COUNT to a value of 3 and the second sets the variable &LABEL to the string of characters 'TEST14'. (The variable on the left side of a CLIST SET assignment statement can be coded without the ampersand (&) but including it improves readability.)

You can perform arithmetic functions with a SET statement and numeric variables. Examples are:

```
SET &COUNT = &COUNT + 1
SET &RIGHT = &LEFT + &MARGIN + &WIDTH - 1
```

Control variables are predefined variables which are normally set by the command processor. Some of them can be overridden by setting them with a SET statement. A list of the CLIST control variables and their descriptions can be found in the OS/VS2 TSO Command Language Reference. Examples of control variables are:

```
&LASTCC
&SYSDATE
&SYSUID
```

Built-in functions are also predefined by the command processor. A list of the CLIST built-in functions and their descriptions can be found in the TSO Command Language Reference. Examples of built-in functions are:

```
&LENGTH(expression)
&STR(string)
```

VARIABLE SUBSTITUTION

Every statement in a CLIST is scanned by the TSO command processor. When a variable is found, it is replaced by its value. Then the statement is evaluated and is processed by the command processor, by the PDF editor, by ISPF or PDF dialog services, or is executed as a command.

The CONTROL LIST statement can be added to a macro to turn on list mode and cause the TSO command processor to write out each statement after substitution has been performed. CONTROL NOLIST can be used to turn off list mode.

Some examples of the use of variables and substitution:

Example 1

Before substitution:

```
SET &A = ABC
SET &B = DEF
IF &A = &B THEN ...
```

After substitution:

```
SET &A = ABC
SET &B = DEF
IF ABC = DEF THEN ...
```

Example 2

Before substitution:

```
ISREDIT CHANGE xx:xx:xx &SYSTIME
```

After substitution:

```
ISREDIT CHANGE xx:xx:xx 14:50:07
```

Example 3

Before substitution:

```
SET &COL = 12
ISREDIT CHANGE XXXXXX TEST01  1  &EVAL(&COL + 4)
```

After substitution:

```
SET &COL = 12
ISREDIT CHANGE XXXXXX TEST01  1  16
```

EDIT PRIMARY COMMANDS

Any command that you can execute from the edit command line can be included in a macro. Simply prefix the command with ISREDIT. For example:

From the command line:

```
COMMAND ===> LOCATE 10
```

From an edit macro:

```
ISREDIT LOCATE 10
```

From the command line:

```
COMMAND ===> CHANGE ALL ' ' '-' 1 10
```

From an edit macro:

```
ISREDIT CHANGE ALL ' ' '-' 1 10
```

EDIT ASSIGNMENT STATEMENTS

A large number of edit macro statements can be grouped together under the heading of edit assignment statements.

An edit assignment statement has the format

```
ISREDIT  xxx = yyy
```

and is used as the primary means of communication between an edit macro and the editor.

Let's take an example. The editor knows whether or not caps mode is currently set ON or OFF. The macro would like to know the current caps mode so that it can take different logic paths. The problem is how to pass the setting of caps mode from the editor to the CLIST. The answer is by using the following edit assignment statement:

```
ISREDIT (CAPMODE) = CAPS
```

When the editor sees this statement, it takes the current CAPS mode and puts it into the variable &CAPMODE.

It may not be obvious why the variable &CAPMODE is not preceded by an ampersand (&) in the statement, and why it is enclosed in parentheses.

The PDF editor needs to know the name of the variable in order to assign a value to it. If the name were preceded by an ampersand (&), the TSO command processor would have replaced the name of the variable with its value, and the editor would never get to see the name.

The parentheses simply indicate to the PDF editor that the enclosed name is the name of a symbolic variable. Although the TSO command processor allows long variable names, only names with eight or fewer characters are allowed by PDF edit.

Let's take another example. The macro wants to tell the editor to set caps mode. Because any primary command can be executed, the following edit macro statement could be used:

```
ISREDIT CAPS ON
```

In addition, either of the two following assignment statements could be used:

```
ISREDIT CAPS = ON
ISREDIT CAPS = (CAPMODE)
```

When the editor sees these statements, it takes the value from the right side of the equals sign (=), and assigns it to caps mode. The value can be coded directly or it can be in a variable, such as &CAPMODE.

Of course you could also code either

```
ISREDIT CAPS &CAPMODE
ISREDIT CAPS = &CAPMODE
```

in which case the TSO command processor would replace the variable &CAPMODE with its value before the PDF editor processed the statement, making them equivalent to the statements in the previous example.

All of the above three examples are valid. When coding a macro, you should use the format which seems to make the most sense, and which makes the macro as readable as possible.

Some information can best be passed back and forth between the editor and the macro in pairs. The following examples show assignment statements that pass two values:

```
ISREDIT (LEFTBND,RIGHTBND) = BOUNDS
ISREDIT BOUNDS = (LB,RB)
```

In the first statement, the current left and right boundaries are stored into the variables &LEFTBND and &RIGHTBND. In the second statement, the values from the variables &LB and &RB are used to change the current boundaries.

The descriptions of the edit commands in Chapter 7 will indicate which edit macro commands are used with one variable, and which are used with two variables.

Many macros need to examine or change data in the file that is being edited. Edit assignment statements are used to communicate data from the file between the editor and the macro.

```
ISREDIT (LINEDATA) = LINE 1
```

When the editor sees this statement, it takes line 1 from the file and puts it into the variable &LINEDATA.

To replace the first line in the file, using the data from the variable &LINEDATA, use:

```
ISREDIT LINE 1 = (LINEDATA)
```

To add a new line after line 1 in the file using the variable &NEWDATA, use:

```
ISREDIT LINE_AFTER 1 = (NEWDATA)
```

PERFORMING LINE COMMAND FUNCTIONS

There is no way to execute PDF edit line commands directly from an edit macro. For example, the TS (text split) line command cannot be executed from an edit macro.

However, most of the functions provided by line commands can be performed by an edit macro. Functions such as move, copy, or repeat can be performed within a macro by using edit assignment statements or by executing edit primary commands. To move a line you can assign the line to a CLIST variable, delete the original line using the DELETE command, and assign the variable to a new line in the file.

Functions provided by the I (insert) line command, the shift line commands, and the text line commands are provided by new commands that can be executed only from within a macro. The following list identifies these commands, the corresponding line commands, and the functions performed.

Edit Macro Statement	Corresponding Line Command	Function
INSERT	I	To insert temporary lines
SHIFT (	(	To shift columns left
SHIFT)	)	To shift columns right
SHIFT <	<	To shift data left
SHIFT >	>	To shift data right
TENTER	TE	To enter text entry mode
TFLOW	TF	To perform text flow
TSPLIT	TS	To perform text split

For example:

```
ISREDIT TFLOW 1
```

causes the paragraph starting on line 1 to be flowed, using the same rules as the TF (text flow) line command.

PASSING PARAMETERS

You may have a need for the user to supply information to the macro, in the form of parameters. These parameters, to be accepted by the PDF editor, must be identified on the ISREDIT MACRO statement, enclosed in parentheses. If your macro FIXIT needs one piece of information, a filename, for instance, you can code:

```
ISREDIT MACRO (FILNAM)
```

When the end user enters

```
====> FIXIT ABCD
```

the value ABCD is assigned to the variable &FILNAM, to be used by the macro.

The TRYIT macro (Figure 4) is useful in trying out edit commands.

```
/*                                                                    */
/*  TRYIT - SIMPLE MACRO FOR TRYING OUT EDIT MACRO STATEMENTS         */
/*                                                                    */
ISREDIT MACRO (COMMAND)
  IF &STR() = &STR(&COMMAND) THEN          /* If no command specified, */ -
    WRITE MISSING COMMAND PARAMETER        /*   indicate problem       */
  ELSE                                     /* Else parameter exists;   */ -
    DO                                     /*   invoke edit command    */
      ISREDIT &COMMAND                     /*   and save return code   */
      SET &RETCODE = &LASTCC               /*   for WRITE message      */
      WRITE &COMMAND RETURN CODE IS &RETCODE
    END
EXIT CODE(&RETCODE)
```

Figure 4. TRYIT Macro

If TRYIT is invoked as "TRYIT RESET", the variable &COMMAND is set to RESET; if it is invoked as "TRYIT FIND A", the variable &COMMAND is set to FIND A.

EDIT MACRO MESSAGES

To display a message from your macro to the user, you can use the SETMSG dialog service. There are two system variables, &ZEDSMSG and &ZEDLMSG, that you can use to set up the short and long messages you wish to issue. For example, if your macro needs to put out a short message saying "INVALID PARAMETER, " and a long message saying "THE ABC PARAMETER MUST BE A 4-DIGIT NUMBER", you can code:

```
SET &ZEDSMSG = &STR(INVALID PARAMETER)
SET &ZEDLMSG = &STR(THE ABC PARAMETER MUST BE A 4-DIGIT NUMBER)
```

You can use these variables in either of two predefined edit macro messages, ISRZ000 (for informational messages) and ISRZ001 (for error messages). ISRZ001 causes the audible alarm to be sounded when it issues the message. To issue the messages defined above, just code:

```
ISPEXEC SETMSG MSG(ISRZ001)
```

RETURN CODES

A macro invoked from the command line may issue the following return codes. These codes affect the command line and cursor position on the next display of edit data:

- Code 0 indicates normal completion of the macro. The cursor position is left as set by the macro. The command line is blanked.
- Code 1 indicates normal completion of the macro. The cursor is placed on the command line. The command line is blanked. Use this code to make it easy to enter another macro or edit command on the command line.
- Codes 4 and 8 are reserved. They are functionally equivalent to return code 0.
- Codes 12 and above are error codes. The cursor is placed on the command line and the macro command is left displayed. Use this code with the dialog manager SETMSG service to prompt the user for an incorrect or omitted parameter.

Every edit macro command sets &LASTCC with a return code. The codes range from 0 to 20.

- Code 0 indicates normal completion of the command.
- Codes 4 and 8 are information codes. They indicate a special condition which is not necessarily an error. These codes can be tested or ignored, depending on the requirements of the macro.
- Codes 12 and above are error codes. Normally an error code will cause the macro to be abnormally terminated and an error panel to be displayed. The error panel will indicate the type of error, and

list the statement which caused the error condition. (A description of how a macro can capture and handle errors is included in Chapter 4.)

Each command description in Chapter 7 includes a list of return codes that are possible for the command. Since &LASTCC is set for every statement, you must either test it in the statement immediately following the command that sets it, or you must save its value in another variable, using a command such as:

```
SET &RETCODE = &LASTCC
```

The variable &RETCODE can then be tested anywhere in the macro until it is changed.

In many cases, the only two possible codes are 0 and 20. The CAPS command is an example of such a command. Any valid form of the CAPS command returns a code of 0.

All of the following forms of the CAPS command have some type of syntax error, and return a code of 20:

```
ISREDIT CAPS XYZ             - Invalid value for CAPS
ISREDIT CAPS ()              - Invalid value for CAPS
ISREDIT CAPS (CAPMODE)       - Not an assignment statement
ISREDIT CAPS ON ON           - Too many parameters coded
ISREDIT CAPS = ()            - No variable name specified
ISREDIT CAPS = (-=+*)        - Invalid variable name
ISREDIT CAPS = (CAPSMODE1)   - Variable name too long
ISREDIT CAPS = ON ON         - Too many parameters coded
ISREDIT CPAS = ON            - Keyword not recognized
ISREDIT () = CAPS            - No variable name specified
ISREDIT (-=+*) = CAPS        - Invalid variable name
ISREDIT (CAPSMODE1) = CAPS   - Variable name too long
ISREDIT CAPS = (VAR1,VAR2)   - Too many parameters coded
ISREDIT (CAPMODE) = CPAS     - Keyword not recognized
```

An example of a command returning a code of 4 is the following CHANGE command:

```
ISREDIT CHANGE "TEST" "DONE"
```

This command sets &LASTCC to 4 if "TEST" is not found. This is not considered an error, since it is unlikely to represent an error in the macro.

An example of a command with a return code of 12 or greater is the CURSOR assignment statement, which sets the cursor position. The following statement:

```
ISREDIT CURSOR = 80 1
```

sets the cursor to line 80, column 1, and sets &LASTCC to 0 unless there is no line 80. In this case, &LASTCC is set to 12, and the macro is

abnormally terminated. If such an error occurs, it indicates a probable error in the macro.

LABELS

A macro frequently wants to look at the data that was found by a FIND or SEEK command. Both commands position the cursor to the search argument. The following statement assigns the data from the line on which the cursor is positioned to the variable &CSRDATA:

```
ISREDIT (CSRDATA) = LINE .ZCSR
```

.ZCSR is a label that names the line at the cursor position. It is a label set by the editor. .ZFIRST and .ZLAST are labels associated with the first and last data lines. You can retrieve the number of data lines with the following statement:

```
ISREDIT (NBRLINES) = LINENUM .ZLAST
```

.ZCSR will be moved by the editor to a new line when one of the following commands moves the cursor: FIND, CHANGE, SEEK, EXCLUDE, TSPLIT or CURSOR. .ZFIRST and .ZLAST also may move when data is added and deleted.

A macro can also label lines that it will refer to frequently. Labels that a macro assigns stay with one line until the macro moves or deletes the label.

A label is an alphabetic character string that must begin with a period (.) followed by from one to eight alphabetic characters, the first of which must not be Z. No special characters or numeric characters are allowed.

For example:

```
ISREDIT  LINE .NEXT = (DATAVAR)
ISREDIT  LINE_AFTER .XYZ = (DATAVAR)
```

The first example stores new data into the line that currently has the label .NEXT. The second example creates a new line after the line whose label is .XYZ, and stores data into the new line.

A macro may assign a label to a line with the LABEL assignment statement. For example:

```
SET &LNUM = 10
ISREDIT LABEL &LNUM = .HERE
```

This assigns the label .HERE to the line whose relative line number is contained in variable &LNUM, line 10 in this case. The .HERE label is set to allow the macro to keep track of a line whose relative line number may change. When the macro completes execution, the .HERE label is deleted.

A macro may obtain the current relative line number of a labeled line with the LINENUM assignment statement. For example:

```
ISREDIT  (LNUM1) = LINENUM .ZLAST
ISREDIT  (LNUM2) = LINENUM .ABC
```

The first example stores the relative line number of the last data line into variable &LNUM1. The second example stores the relative line number of the line with label .ABC into variable &LNUM2.

To delete a label, set the label to blanks:

```
ISREDIT LABEL lptr = ' '
```

CHAPTER 3. TESTING EDIT MACROS

At this point, you should be able to write and execute an edit macro that uses CLIST logic and invokes simple edit commands. However, even an experienced edit macro writer occasionally includes a bug that causes a macro to terminate abnormally, or writes a macro that does not work as expected. When this occurs, it is necessary to debug your macro, just as you would debug any other type of program which you were implementing.

If you have not coded a CLIST before, there are some additional considerations you should be aware of when you test and debug your edit macro. Refer to Appendix A for these CLIST considerations.

ERROR HANDLING

Because edit macros consist of different kinds of statements, there are different kinds of errors that can occur, and different ways that the errors are handled. Later on we will discuss how you can code macros that capture and handle error conditions. In debugging macros that do not handle errors, you run into the following types of errors:

- Errors may be detected during variable substitution, or in the processing of command procedure statements, by the TSO command processor. They normally result in the screen being cleared, and the statement in error being displayed, along with an error message. For example, if the symbolic variable &DATA was referenced in an edit FIND command before it was set, the error messages might be:

  ```
  ISREDIT FIND &DATA
  THIS STATEMENT HAS AN UNDEFINED SYMBOLIC VARIABLE
  ***
  ```

  When ENTER is pressed, the macro is abnormally terminated, and you return to edit where the message "xxxxx MACRO ERROR" is displayed.

- Edit command errors are detected by the editor. They result in an PDF edit macro error panel being displayed. The command in error is displayed, along with an error message that identifies the error.

  - Using PDF as a normal user, with ISPF test mode off, you are prompted to press ENTER to terminate the macro. The macro is abnormally terminated, and you return to edit.
  - Using PDF in test mode, with ISPF test mode on, you can override the abnormal termination and attempt to continue by entering YES on the PDF edit macro error panel.

- Dialog service errors are detected by ISPF. They result in an ISPF dialog error panel being displayed. A message identifying the error is displayed, along the with the statement which is in error.

  - Using PDF as a normal user, with ISPF test mode off, you are prompted to press ENTER to terminate the dialog. The edit session is abnormally terminated, and the primary option menu is displayed.
  - Using PDF in test mode, with ISPF test mode on, you can override the abnormal termination and attempt to continue by entering YES on the ISPF dialog error panel.

Note: If you enter ISPF with TEST as a parameter, ISPF test mode is set on. Once you are executing ISPF, you can cause ISPF test mode to be set on by going to Option 7 (Dialog Test) from the primary option panel. Once test mode is set on, it remains on until you end your ISPF session.

USING CLIST WRITE STATEMENTS

The CLIST WRITE statement can be a valuable tool in tracking down edit macro problems. On the WRITE statement you can identify the position of the statement within the macro, as well as display the value of variables. For example, if you are having trouble debugging the TESTGEN macro (Figure 2), you might include some WRITE statements, as shown in Figure 5.

```
/*                                                                  */
/*  TESTGEN MACRO - GENERATE TEST DATA                              */
/*                                                                  */
ISREDIT MACRO
  SET &COUNT = 1                          /* Initialize loop counter     */
  DO WHILE &COUNT <= 9                    /* Loop up to 9 times          */
    ISREDIT FIND 'TEST-#'                 /* Search for 'TEST-#'         */
    SET &RETCODE = &LASTCC                /* Save the FIND return code   */
WRITE RESULT OF FIND, RC = &RETCODE
    IF &RETCODE = 0 THEN                  /* If string was found,        */ -
      DO                                  /*                             */
        ISREDIT CHANGE '#' '&COUNT'       /*   Change # to a digit and   */
        SET &COUNT = &COUNT + 1           /*   increment loop counter    */
WRITE COUNT IS NOW UP TO &COUNT
      END                                 /*                             */
    ELSE                                  /* If string is not found,     */ -
      SET &COUNT = 10                     /*   Set counter to exit loop  */
  END                                     /*                             */
EXIT CODE(0)
```

Figure 5. TESTGEN Macro with WRITE Statements

In this example, the WRITE statements are not indented, which makes them stand out. You may want to use this approach in your debugging and testing to make the statements easier to delete once the macro is complete.

USING CLIST CONTROL STATEMENTS

You can use the CLIST CONTROL statement with the LIST, SYMLIST, or CONLIST operands to display statements from a macro as it is being interpreted and executed. See the TSO Command Language Reference for more details.

- LIST displays commands and subcommands (including ISREDIT statements), after substitution, but before execution. This allows you to see an ISREDIT statement in the form that the editor sees the statement.

- CONLIST displays a CLIST statement (IF, DO, SET, etc) after substitution, but before execution. You may be able to tell why an IF statement didn't work properly by using CONLIST.

- SYMLIST displays lines before symbolic substitution. It is less useful than LIST and CONLIST, but in some cases may be required to help debug a macro.

The NOLIST, NOSYMLIST, and NOCONLIST operands can be used to terminate the display of statements.

CHAPTER 4. ADVANCED EDIT MACRO CONCEPTS

This chapter contains information that you will need to know when you begin to code more advanced edit macros.

EDIT PRIMARY COMMANDS

As was noted in Chapter 2, any command that you can execute from the edit command line can be included in a macro by prefixing it with ISREDIT. There are however, some minor differences between executing a command from the command line, and executing the same command from a macro.

One difference is that if you execute the command from the command line, an information or error message will frequently be displayed, while if you execute the command from a macro, a return code will be set.

A further difference may be experienced if a series of commands are executed. Since the display is not formatted for each command that is executed from a macro, the lines being displayed may be different.

An example would be a series of FIND commands, some of which position the cursor within the displayed screen, and others which causes scrolling to take place. The series of FIND commands executed from the command line might result in line 000050 being the first line displayed, with the cursor in the middle of the screen, while the same commands executed from a macro might result in line 000060 being the first displayed line, with the cursor on the second line of the screen. The ISREDIT (varname) = DISPLAY_LINES assignment statement forces the display screen to be formatted and may be used within a macro if necessary to simulate a series of end user commands.

Some of the commands have additional operands that are permitted in a macro but are not permitted from the command line. For example:

```
ISREDIT COPY 1 10 MYDATA AFTER .ZLAST
```

which copies lines 1 through 10 from the member MYDATA after the last line of the current file, is valid from a macro, but is not valid from the command line.

EDIT ASSIGNMENT STATEMENTS

Edit assignment statements are used for communication between a macro and the editor. Edit assignment statements differ from CLIST assignment statements in the following ways:

- Certain keyphrases may appear on the left or right side of an equal sign. A keyphrase is either a single keyword, or a keyword followed by a line number or label.

- Variable names that are to be passed to the editor are enclosed in parentheses, with no leading ampersand. In some cases, two variable names may appear within the parentheses.

- Arithmetic expressions are not allowed in an edit assignment statement, but in certain cases a plus sign (+) may be used to indicate partial overlay of a line.

In general, edit assignment statements are used to set or retrieve the contents of a line of data, the cursor position, the boundaries, the contents of the mask and tabs lines, or the settings of edit modes. Each of these is represented by a keyphrase.

For example, the keyphrase "LINE lptr" represents the contents of the data line pointed to by a line pointer (lptr), which may be either a relative line number or a label. The statements:

```
ISREDIT  (OLD) = LINE 5
ISREDIT  LINE 5 = (NEW)
```

save the contents of the fifth line in a variable named &OLD, and then replace the contents of the same line from a variable named &NEW.

Frequently, the line pointer (lptr) operand is specified as a variable. For example:

→ see page 111

```
ISREDIT  (OLD) = LINE &LINENUM
ISREDIT  LINE &LINENUM = (NEW)
```

where the variable &LINENUM contains a relative line number.

Note: The line pointer variable (&LINENUM, in this example) is not coded in parentheses because it is to be replaced with its current value by the CLIST processor before the command is passed to the editor.

Certain keyphrases have two associated values. For example, the first value associated with the CURSOR keyphrase is the relative line number on which the cursor is currently located, and the second value is the relative column position of the cursor. The statements:

```
ISREDIT  (ROW,COL) = CURSOR
ISREDIT  CURSOR = 1,40
```

save the current position of the cursor in variables ROW and COL, and then set the cursor to line 1, column 40.

Either of the variables or values in the pair can be omitted. For example, the statements:

```
ISREDIT  (ROW) = CURSOR
ISREDIT  CURSOR = 1
```

save and then set the cursor line number without changing the column position, whereas the statements:

```
ISREDIT  (,COL) = CURSOR
ISREDIT  CURSOR = ,40
```

save and then set the cursor column position without changing the line number.

The "value" in an edit assignment statement may be one of the following:

- A literal (character string) that may be:

  A simple string
  : Any series of characters not enclosed within apostrophes ('), quotation marks ("), parentheses, or less-than (<) and greater-than signs (>), and not containing any embedded blanks or commas.

  A delimited string
  : Any string starting and ending with an apostrophe (') but not containing imbedded apostrophes, or starting and ending with a quotation mark (") but not containing imbedded quotation marks. The delimiting apostrophes or quotation marks are not considered part of the data.

- A variable name enclosed in parentheses:

  ```
  (varname)
  ```

  The entire contents of the variable are considered part of the data, including any quotes, apostrophes, blanks, commas, or other special characters.

Statements with Two-Valued Keyphrases

An edit assignment statement that contains a two-valued keyphrase has one of the following formats:

```
ISREDIT  (varname,varname) = keyphrase
ISREDIT  keyphrase = value-pair
```

where "value-pair" is one of the following:

- Two literals that may be separated by a comma or blank. Examples:

```
ISREDIT  CURSOR = 1,40
ISREDIT  CURSOR = 1 40
```

Note: Apostrophes or quotes may **not** be used when specifying two numeric values. Both of the following, for example, are **invalid:**

```
ISREDIT  CURSOR = '1','40'
ISREDIT  CURSOR = '1,40'
```

- Two variable names enclosed in parentheses and separated by a comma or blank:

  (varname,varname) or (varname varname)

  where each variable contains a single value.

In any edit assignment statement that contains a two-valued keyphrase, either of the variables or values in a pair can be omitted. The general format then becomes:

```
ISREDIT  (varname) = keyphrase
ISREDIT  keyphrase = single-value

ISREDIT  (,varname) = keyphrase
ISREDIT  keyphrase = ,single-value
```

Note: Even though blanks may be used instead of commas to separate paired variables or values, a leading comma is required whenever the first variable or value has been omitted.

Line Data Statements

In addition, there are certain statements that allow additional flexibility. These statements, which add or replace lines of data, are LINE, LINE_AFTER, LINE_BEFORE, MASKLINE, and TABSLINE. They may have any of the following formats:

```
ISREDIT  keyphrase = keyphrase
ISREDIT  keyphrase = value
ISREDIT  keyphrase = keyphrase + value
ISREDIT  keyphrase = value + value
```

When two values, or a keyphrase and a value, are separated by a plus sign (+), nonblank characters in the value on the right overlay corresponding characters in the value on the left. For example:

```
ISREDIT  LINE .ZCSR = LINE + '//'
ISREDIT  MASKLINE = MASKLINE + <40 '/*' 70 '*/'>
```

The first example causes two slashes to replace the first two column positions of the line containing the cursor. The remainder of the line is unchanged. The second example uses a template to cause columns 40-41

of the current mask line to be replaced with "/*" and columns 70-71 to be replaced with "*/". A template of the form:

```
<col-1 literal-1 col-2 literal-2 ... >
```

may be coded with col indicating a starting column position and literal indicating the data to start at that column. The entire template is delimited with less-than (<) and greater-than (>) signs. A template may be coded using variable names, enclosed in parentheses, for "col" or "literal" (or both). All of the following forms are valid:

```
<(colvar-1) (datavar-1) (colvar-2) (datavar-2) ... >
<(colvar-1,datavar-1)   (colvar-2,datavar-2)   ... >
<(colvar-1)  literal-1   col-2     (datavar-2) ... >
```

HANDLING ERRORS

As indicated in "Return Codes" in Chapter 2, every edit macro statement causes &LASTCC to be set to a return code. Codes of 12 or greater are considered errors, and the default is to terminate macros which issue return code 12 or greater.

If you want to handle all errors that might occur in your macro you should code

```
ISPEXEC CONTROL ERRORS RETURN
```

indicating that, in case of errors, control is to return to the macro. If you do not want to handle errors, you should code

```
ISPEXEC CONTROL ERRORS CANCEL
```

indicating that in the case of an error, the macro is to be terminated.

You can include any number of ISPEXEC CONTROL statements in your macro to turn error handling on and off.

REFERRING TO DATA LINES

You may refer to data lines either by a relative line number or by a symbolic label.

Note: Special lines (MASK line, TABS line, COLS line, BOUNDS line, MSG lines, and others) are not considered data lines and are not assigned relative line numbers, and they may not be assigned labels. These lines cannot be directly referenced by a macro even though they may be displayed in the data portion of the screen.

Relative line numbers are not affected by the presence of sequence numbers within the data, nor are they affected by the current setting of number mode. The first line of data is always treated as line number 1,

the next line is line number 2, and so on. The "top of data" line is considered line number 0.

When lines are inserted or deleted, the lines that follow will have new relative line numbers. If a new line is inserted after line 3, for example, it becomes relative line 4 and all lines that follow will have new relative line numbers. (What was relative line 4 becomes relative line 5, and so on.) Similarly, if line 7 is deleted, the line that was relative line 8 becomes relative line 7, and so on.

REFERRING TO COLUMN POSITIONS

In edit macros, column positions are always referred to relative to the data portion of a line. The data portion excludes the sequence numbers when number mode is on. For example, if "NUMBER COBOL ON" mode is in effect, the first six card image positions of each line contain the sequence number. The first data character is in card image position 7, which is considered relative column 1.

If your macro has to access the sequence numbers as data, first ensure that number mode is off. If desired, your macro may save the current number mode, set number mode off, and then restore the original number mode before returning to the end user.

When a macro retrieves the current cursor position, a relative column number of zero is returned if the cursor is outside the data portion of the line. When a macro sets the cursor column to zero, the cursor is placed in the line command field (left side) of the designated line.

LABELS

There are several special labels that are automatically assigned by the editor. They all begin with the letter "Z". Labels beginning with "Z" are reserved for editor use, and may not be assigned by a macro or an end user.

The editor-assigned labels are:

.ZCSR The data line on which the cursor is currently positioned.

.ZFIRST The first data line (same as relative line number 1). May be abbreviated .ZF.

.ZLAST The last data line. May be abbreviated .ZL.

.ZFRANGE The first line in a range indicated by the user.

.ZLRANGE The last line in a range indicated by the user.

.ZDEST The destination line indicated by the user.

Note: Unlike other labels, .ZCSR, .ZFIRST, and .ZLAST do not stay with

the same line. Label .ZCSR stays with the cursor, and labels .ZFIRST and .ZLAST point to the current first and last lines.

A macro can refer to labels assigned by the user. A lower-level macro (nested macro) is able to refer to all labels assigned by higher-level macros as well as the user. When a macro assigns labels, they are normally associated with that macro and are automatically unassigned when the macro completes execution. The labels "belong" to the macro that assigned them, and may have the same name as labels at a higher level without any conflict.

If desired, a macro may assign labels that are to be passed to the user or to a higher-level macro. Labels to be passed back are indicated with a level operand when the label is assigned:

```
ISREDIT  LABEL lptr = label [level]
```

where level is a numeric value. Level 0 is the end user level, level 1 is the macro invoked by the end user, level 2 is the next lower-level (nested) macro, and so on. The maximum nesting level allowed is 255. If the level operand is omitted, or if its value is equal to or greater than the level at which the macro is executing, the label is assigned to the macro's own level.

A macro has access to any labels that it assigns to the user or to a higher-level macro, but those labels do not "belong" to the macro and are not automatically unassigned when the macro completes.

MACRO LEVELS

Each macro operates on a separate and unique level. The end user always operates at level 0. The macro invoked by an end user always operates at level 1, the macro invoked by a level 1 macro operates at level 2, and so on. The level is the degree of macro nesting. When a macro sets a label without indicating a level, the label is set at the macro level that is currently in control and does not affect any labels set in a higher level. When a macro queries a label without specifying a level, or uses the label as a line pointer, the search for the label starts at the current level and goes up, level by level, until the label defined closest to the current level is found.

If you specify a level parameter that is outside the current active levels, it is adjusted as follows: a value less than zero is set to zero; a value greater than the current nesting level is set to the current nesting level. This means that a higher-level macro cannot set a label at the level of the macro that it is going to invoke.

When the macro terminates, the labels at the current nesting level are deleted. To set a label for the next higher level, the macro can issue the MACRO_LEVEL assignment statement to obtain the current level and decrement the level by 1.

When a label is set on a line, it remains associated with the line even when the relative line number of that line changes as a result of new lines being added to or deleted from the data. Using the same label name as an existing label on the same level moves that label name from one line to another.

A macro may find out the level of a label with the LABEL assignment statement, as follows:

```
ISREDIT  (varname1,varname2) = LABEL lptr
```

The label assigned to the referenced line is stored in the first variable, and its level is stored in the second variable. If there is no label assigned to the line, a blank is stored in both variables.

A macro may find out its own level with the following assignment statement:

```
ISREDIT  (varname) = MACRO_LEVEL
```

The current level number is stored in the specified variable.

Notes:

1. Labels at the end-user level are retained until editing of the current data is terminated.

2. Whenever a line is deleted, any labels associated with it become unassigned.

3. A labeled line may be assigned a new label, which causes the previous label to be unassigned (if both labels are at the same level). If the new label is a blank, the line becomes unlabeled. For example:

   ```
   ISREDIT  LABEL .HERE = ' '
   ```

   causes the line labeled .HERE to no longer have a label.

4. If a label that is in use is assigned to another line, the label is "moved" from the original line to the new line (provided that the new assignment is at the same level as the original).

5. If the label is set on a line that already has a label at the same level associated with it, setting the new label causes the previous label to be deleted.

PASSING MULTIPLE PARAMETERS TO A MACRO

When you invoke a macro with parameters, the editor takes the parameters and puts them into variables, as yet unnamed. Then it invokes the macro without parameters. If the macro allows parameters to be specified, the MACRO command identifies the names of one or more variables, which will contain any passed parameters. These names are enclosed in parentheses. The MACRO command allows parameters to be omitted or entered in any order. This allows the macro to assume default values for parameters that are not supplied, similar to the way edit FIND looks for the next occurrence of string ABC when FIND ABC is entered (NEXT is a default keyword).

It is an error if a macro that does not have parameters is invoked with parameters. If an end user makes this error, edit displays a message. It is not an error if more or fewer parameters are supplied than the number of variables coded on the MACRO command. It is the macro writer's responsibility to check for omissions and the order of parameters.

Multiple parameters are placed into one or more variables based on the number of variables specified on the MACRO command. If only one variable name is coded on the MACRO command, that variable contains all parameters entered after the macro name. If more than one variable name is coded, the editor attempts to parse any parameters, and stores them in order (that is, the first parameter in the first variable, the second in the second, and so on). If there are more parameters entered than there are available variables, the editor stores the remaining parameters as one character string in the last variable.

If there are more variable names than parameters, the unused variables are set to nulls. The parameter is defined as a simple string, separated by a blank or a comma, or a quoted string, separated separated by a blank or comma. Quotes may be single (') or double ("). For example, if your FIXIT macro is to have two parameters, you can code:

```
ISREDIT MACRO (PARM1,PARM2,REST)
```

This means that if the user enters

```
====> FIXIT GOOD BAD AND UGLY
```

variable &PARM1 will be assigned the value GOOD, &PARM2 will be assigned the value BAD, and &REST will be assigned the value AND UGLY. If the parameters passed were GOOD BAD, variable &REST would be null.

If the MACRO statement in TRYIT (Figure 4 on page 14) were coded with two variables, "ISREDIT MACRO (COMMAND,PARM)", "TRYIT RESET" would result in the variable &COMMAND being set to RESET and &PARM being set to null; "TRYIT FIND A" would result in &COMMAND being set to FIND and &PARM being set to A.

TRYIT would have to be further modified to allow edit commands that require parameters to work with the modified MACRO command.

DEFINING MACROS

If you want to establish names for macros that are different from their member names, or use aliases for built-in edit commands, or identify macros as program macros, you must issue a DEFINE command. You will commonly issue DEFINE commands in an initial macro.

Implicit Definitions

When you or your macro issues a command unknown to the editor, the SYSPROC concatenation sequence is searched for a CLIST with that name. If it is found, it is implicitly defined as a CLIST macro. For example, if you enter XXX on the command line or the macro command processed is ISREDIT XXX and XXX exists in SYSPROC, XXX is defined implicitly as a CLIST macro. You may implicitly define a program macro by preceding its name with an exclamation point (!); you can use this method only when the name is seven characters or less. The third way to define a macro implicitly is to define a name as an alias of a name that is unknown to edit. If the unknown name is in the SYSPROC concatenation, it becomes implicitly defined.

Overriding Command Names

To override an existing edit command name, that is, to cause a macro to be executed in place of a built-in edit command, issue a DEFINE command. To issue the built-in edit command in an overriding macro, precede the command with BUILTIN. However, you cannot override an assignment statement. Commands that can only be executed in a macro (like LINE or MACRO) do not require a DEFINE command to override the definition for the end user. Since the command is unknown to an end user, the SYSPROC directory is searched for a macro with that name. To use LINE as a macro (not recommended), issue DEFINE LINE MACRO. Then an ISREDIT LINE command without an equals sign causes the LINE macro to be executed.

Defining an Alias

To establish an alias or alternate invocation name for a command, code the alias name first, followed by the ALIAS keyword and the command name. For example, issuing

```
DEFINE FILE ALIAS SAVE
```

allows you to use the command FILE to save the data currently being edited on disk. If the command following the ALIAS keyword is unknown to edit, the SYSPROC concatenation is searched and, if the name is found, it is defined implicitly as a macro.

Resetting Definitions

To undo the last DEFINE for a command and to return it to its previous status, issue the DEFINE command with the RESET keyword. For example, after the previous definition of FILE as an alias for SAVE, if you issue

```
DEFINE FILE RESET
```

any further attempts to use the FILE command would be flagged as an invalid command.

Scope of Definition

DEFINE commands issued in a macro are in effect while the current member is being edited. DEFINE commands issued by an end user are in effect until the edit session is terminated.

USING THE PROCESS COMMAND

An end user can perform three different kinds of editing with a single interaction. He can:

- Enter a primary command
- Enter one or more line commands
- Overtype data on the screen

If a macro is entered as the primary command, the sequence of events is as follows:

- The macro is executed up to the ISREDIT MACRO command, which must be the first command in the macro.
- Any overtyping that was made on the screen is merged into the file being edited.
- Any line commands are executed.
- The rest of the macro is executed.

It is possible to alter this sequence by using an ISREDIT MACRO command with the NOPROCESS argument, and then code an ISREDIT PROCESS command.

The syntax of the MACRO statement is:

```
ISREDIT MACRO (parm...) PROCESS/NOPROCESS
```

The PROCESS keyword indicates that screen data and line commands are to be processed when the MACRO command is encountered. PROCESS is the default.

The NOPROCESS keyword indicates that processing of the screen data and line commands are to be deferred until an ISREDIT PROCESS command is encountered later in the macro, or the macro terminates.

There are two reasons for coding NOPROCESS. The first is that you want to execute statements before the screen data or line commands are processed. You might want to perform initial verification of parameters or capture lines from the file before they have been changed from the screen.

The second reason is that you want to code an ISREDIT PROCESS command to specify whether or not the macro expects, and will handle, line commands that identify either a range of lines, a destination line, or both. This is the method by which the editor allows a macro command to interact with line commands in the same way that the built-in MOVE or REPLACE commands do. Once the ISREDIT PROCESS command has been executed, the editor can process the line commands that have been entered by the end user, performing meaningful error and consistency checking.

If ISREDIT PROCESS DEST is coded, it indicates that the macro expects a destination line to be specified by the end user. A destination line is always specified using either A (after) or B (before). The dialog variable .ZDEST is set to the line preceding the destination; if neither A nor B is specified, .ZDEST is set to the last data line in the file.

If ISREDIT PROCESS RANGE arg is coded, it indicates that the macro expects a range of lines to be specified by the end user. The argument following the RANGE keyword identifies either one or two commands which are to be accepted. For example, PROCESS RANGE Q Z allows the user to enter either Q or Z line commands in conjunction with this macro. The line commands could take any of the following forms:

- Q or Z, to indicate a single line
- QQ or ZZ, to indicate a block of lines (this form is obtained by doubling the last letter of the single-line command)
- Qn or Zn, where n is a number that indicates a block of n lines

After the PROCESS command is completed, the dialog variable .ZFRANGE is set to the first line of the user-entered range and the dialog variable .ZLRANGE is set to the last line of the user-entered range. The labels may refer to the same line. If no range is entered, the range defaults to the entire file. When a choice may be made between the two line commands, the RANGE_CMD assignment statement is used to return the value of the command entered. The names of line commands used to define the range of processing may be one to six characters, but if the name is six characters long, it may not be used as a block format command. The name may contain any alphabetic or special character except blank, hyphen (-), or apostrophe ('). It may not contain any numeric characters.

An example may clarify some of the above statements.

Example

The NOPROCESS keyword on the MACRO command is used to defer processing of the screen data until the line with the cursor is assigned to a variable. After the PROCESS command, the line contains any changes that were made by overtyping it.

```
ISREDIT MACRO NOPROCESS
ISREDIT (BEFORE) = LINE .ZCSR
ISREDIT PROCESS
ISREDIT (AFTER) = LINE .ZCSR
IF &STR(&BEFORE) = &STR(&AFTER) THEN -
        ... no change to line with the cursor.
ELSE -
        ... the line with the cursor has been changed.
```

PROFILES

The set of default modes for a given record format (like fixed 80) is called a profile. Each user has his own edit profile table (ISREDIT) on disk that contains profiles for each type (the lowest-level qualifier in the data set name) of data edited unless an explicit profile name is specified. The user can explicitly specify a profile name, and thus the modes to use, in three ways:

- Issue a 'PROFILE name' command
- Fill in the PROFILE field on the edit entry panel
- Supply a PROFILE keyword when invoking the EDIT service:

```
ISPEXEC EDIT PROFILE(name) ...
```

In addition to the disk profile, a current profile is maintained. Another form of the PROFILE command (PROFILE LOCK/UNLOCK) controls whether the disk profile is automatically updated to reflect changes to the current profile. The following assignment statements change profile modes or values:

AUTONUM	HEX	NUMBER	RECOVERY
AUTOLIST	IMACRO	NOTE	TABS
AUTOSAVE	MASKLINE	PACK	TABSLINE
BOUNDS	NULLS	PROFILE	

Each time another member is edited, the current profile is reloaded from disk. The data is then examined for caps, number, stats, and pack mode values and warning messages are issued if the current copy of the profile is changed. With a locked profile, the messages occur only when a member has characteristics different from those stated in the profile. Since a locked profile establishes default values that are reinstated when a new member is edited, a change to a profile setting, such as a boundary change, or a difference in the mask line, will be in effect only for the current member being edited.

INITIAL MACROS

An initial macro is executed after a member or sequential data set has been specified and its data read, but before the data is displayed.

An initial macro can be used to set up your edit environment if you want to default to values other than those automatically set up by edit. For example, if you want caps mode on, regardless of whether the data contains lowercase data, create an initial macro with a CAPS ON command. Edit first reads the profile and the data and then sets caps mode to correspond to the data. Then it executes your initial macro, which overrides the setting of caps mode by edit.

You can specify an initial macro in one of the following ways:

- Using the IMACRO command to store the macro name in the edit profile:

  ```
  IMACRO STARTUP
  ```

- Specifying the initial macro name on the edit selection panel:

  ```
  INITIAL MACRO ===>
  ```

- Specifying the initial macro name on the EDIT service invocation:

  ```
  ISPEXEC EDIT DATASET(dsname) MACRO(initmac) ...
  ```

Specifying the parameter on the edit selection panel or on the EDIT service invocation overrides the setting in the edit profile. You can enter NONE to suppress execution of the initial macro defined in the profile.

Some commands you may find useful in an initial macro, either for all members or for new members (members with zero lines):

```
CAPS    - Force caps mode on or off
NUMBER  - Force number mode off
PACK    - Force pack mode on or off
STATS   - Force stats mode on or off
RESET   - Reset unwanted information messages
VERSION - Set version number
LEVEL   - Set or increment modification level
```

Commands that reference display values (DISPLAY_COLS, DISPLAY_LINES, DOWN, LEFT, RIGHT, UP) are invalid in an initial macro.

If the initial macro issues an END command, changes to the data made by the macro are saved and the member is not displayed.

RECOVERY MACROS

When you are recovering from a system failure, you may want to restore the command definitions and aliases that you were using when the system failed, but you don't want to destroy the profile changes that you've made during the edit session before the failure. To allow you to recover, edit provides a recovery macro which, like an initial macro, is executed after the data has been read but before it is displayed. In this case, however, the macro is executed whenever the edit of the data set is occurring as a result of recovery. You can specify a recovery macro in your initial macro, using the RMACRO command.

CHAPTER 5. SAMPLE EDIT MACROS

This chapter presents several edit macros, along with line-by-line comments about their operation. The numbers within parentheses on the left identify the statements described in more detail below and are **not** part of the macro.

FORMAT MACRO

The FORMAT macro (Figure 6) initializes the edit profile values and PF keys for text entry. It may be invoked from the command line or may be used as an initial macro set in the profile used for text entry with the command IMACRO FORMAT; it requires no parameters. It is to be used in conjunction with the BOX macro, which will be described later.

```
     /*                                                              */
     /* FORMAT initializes the profile and PF keys for text work     */
     /*                                                              */
(1)  ISREDIT MACRO                          /*                       */
                                            /*                       */
(2)    ISREDIT NUMBER OFF                   /* Set profile values    */
       ISREDIT TABS OFF                     /*   default bounds      */
       ISREDIT NULLS OFF                    /* Set tabs, nulls, caps */
       ISREDIT BOUNDS                       /*   and number off      */
       ISREDIT CAPS OFF                     /*                       */
       ISREDIT RECOVERY ON                  /* Set recovery on       */
                                            /*                       */
(3)    SET &ZPF24 = BOX                     /* Set PF 12 to BOX      */
(4)    ISPEXEC VPUT (ZPF24) PROFILE         /*   and save in profile */
                                            /*                       */
(5)    ISREDIT DEFINE END    ALIAS PFEND    /* Do DEFINEs to reset   */
       ISREDIT DEFINE CANCEL ALIAS PFCAN    /*   the PF key at exit  */
       ISREDIT DEFINE QUIT   ALIAS CANCEL   /* QUIT = PFCAN          */
(6)  EXIT CODE(0)                           /*                       */
```

(1) The MACRO command identifies this CLIST as a macro.
(2) These six commands set profile values; the boundaries are set to the first and last column numbers of data.
(3) The CLIST SET statement sets &ZPF24 to BOX. This ISPF variable controls the function of PF key 12 (for terminals with 12 program function keys) or PF key 24 (for 24-key terminals). BOX is the command to be executed when PF key 12 or PF key 24 is pressed. Since no native edit command exists with the name BOX, the SYSPROC concatenation will be searched for a CLIST named BOX.
(4) A dialog service sets the PF key variable in the profile pool where the PF key variables are saved.
(5) Macros are defined to be invoked when edit commands are issued. When the user enters a CANCEL or CAN or QUIT command, the macro PFCAN is executed. Similarly, when the user enters an END command or presses the END PF key, the macro PFEND is invoked. Notice that since QUIT is defined after CANCEL was defined as an alias of a macro, it too becomes an alias of the same macro. The PFCAN macro is also shown later as an example.
(6) Exit from the macro, setting a return code of zero.

Figure 6. FORMAT Macro

PFCAN MACRO

The PFCAN macro (Figure 7) may be run in place of the edit CANCEL command. It cancels the edit session, but first it resets PF 12, which was defined by the FORMAT macro.

```
      /* PFCAN  Reset PF 12, which was defined by the FORMAT macro.        */
      /*                                                                   */
       ISREDIT MACRO                              /*                      */
          SET ZPF24 = CURSOR                      /*  Reset PF 12 to its  */
(1)       ISPEXEC VPUT (ZPF24) PROFILE           /*    default value     */
(2)       ISREDIT BUILTIN CANCEL                 /*  Cancel the edit     */
                                                 /*    session           */
        EXIT                                     /*                      */
```

(1) PF 12 is reassigned to its default setting: place the cursor in the command area.

(2) The native edit CANCEL command is executed. If BUILTIN did not precede CANCEL on this statement, PFCAN would issue a CANCEL command which would cause PFCAN to get invoked, and so on.

Figure 7. PFCAN Macro

BOX MACRO

The BOX macro (Figure 8) draws a box whose left corner is at the cursor position. You can prepare to invoke BOX in one of two ways:

- Enter KEYS on the command line, set a PF key to the BOX macro, and enter the END command.
- Use the FORMAT macro, defined earlier, which sets up the PF key for BOX and defines the profile values for text entry.

Then position the cursor on a data line where you want the box drawn and press PF 12 (or another key that you have defined) to invoke BOX. After the box is drawn, the cursor is positioned inside, ready for entering text to fill the box. If any of the macro commands fail, it issues a warning message. To show how the box is drawn, the BOX macro was run placing the cursor on the "&" of &EVAL in the BOX macro definition. The result:

```
ISREDIT LINE &ROW          = LINE + < &COL '+----------------+'>
ISREDIT LINE +-----------------+NE + < &COL '|                |'>
ISREDIT LINE |                 |NE + < &COL '|                |'>
ISREDIT LINE |                 |NE + < &COL '|                |'>
ISREDIT LINE |                 |NE + < &COL '|                |'>
ISREDIT LINE |                 |NE + < &COL '+----------------+'>
             +-----------------+          /*                    */
```

```
      /*                                                                */
      /* BOX  - Draw a box whose left corner is at the cursor position. */
      /*                                                                */
        ISREDIT MACRO
(1)     ISREDIT (ROW,COL) = CURSOR                /* Get cursor position  */
                                                  /*                      */
(2)     ISPEXEC CONTROL ERRORS RETURN             /* No macro error panel */
                                                  /* Draw box over        */
                                                  /*  existing lines      */
                                                  /*                      */
(3)     ISREDIT LINE &ROW          = LINE + < &COL '+----------------+'>
        ISREDIT LINE &EVAL(&ROW+1) = LINE + < &COL '|                |'>
        ISREDIT LINE &EVAL(&ROW+2) = LINE + < &COL '|                |'>
        ISREDIT LINE &EVAL(&ROW+3) = LINE + < &COL '|                |'>
        ISREDIT LINE &EVAL(&ROW+4) = LINE + < &COL '|                |'>
        ISREDIT LINE &EVAL(&ROW+5) = LINE + < &COL '+----------------+'>
                                                  /*                      */
(4)     IF &MAXCC > 0 THEN                        /* If error on          */ -
          DO                                      /*   overlaying lines   */
(5)         SET ZEDSMSG = INCOMPLETE BOX
            SET ZEDLMSG = NOT ENOUGH LINES/COLUMNS TO DRAW COMPLETE BOX
            ISPEXEC SETMSG MSG(ISRZ001)
          END                                     /* Issue error message  */
        SET &COL = &COL + 2                       /* Position cursor      */
        SET &ROW = &ROW + 1                       /*  within the box      */
(6)     ISREDIT CURSOR = (ROW,COL)                /*                      */
        EXIT
```

Figure 8 (Part 1 of 2). BOX Macro

(1) The variables &ROW and &COL are set to the cursor position.
(2) A dialog service allows the macro to handle severe errors. This allows a message to be displayed when the cursor is placed too close to the end of the file. The LINE assignment statement fails if the row it is setting does not exist.
(3) The LINE assignment statements overlay existing data on a line with characters to form a box. LINE uses a merge format to include the existing line data and then a template to put the overlaying data at the cursor column position. The CLIST &EVAL function increments the relative line numbers before the statement is passed to edit.
(4) The CLIST IF statement checks the &MAXCC variable, and if it is nonzero, invokes the dialog service SETMSG to display a message. &MAXCC is a variable updated by the CLIST processor to contain the highest condition code.
(5) The message used in SETMSG is one of two messages (ISRZ000 and ISRZ001) reserved for macro use. Each message uses two variables:
- &ZEDSMSG to set the text for the short message (up to 24 characters) that is displayed when the macro completes.
- &ZEDLMSG to set the text for the long message that appears when the HELP PF key is pressed.

Message ISRZ001 sounds the alarm to indicate an error, message ISRZ000 does not sound the alarm.
(6) This statement positions the cursor within the box to simplify entering text when the screen is redisplayed.

Figure 8 (Part 2 of 2). BOX Macro

ALLMBRS MACRO

The ALLMBRS macro (Figure 9) uses library management services to get each member name in the partitioned data set that is being edited. An inner macro is invoked for each member in the data set except the member currently being edited. The name of the inner macro to execute is passed as the only parameter to ALLMBRS. The inner macro is invoked with the member name as a parameter. To invoke ALLMBRS, edit a new member and invoke ALLMBRS with the name of the macro to be invoked in each member. For example, if the name of the macro is IMBED, issue:

```
COMMAND ====> ALLMBRS IMBED
```

```
      /*                                                              */
      /* ALLMBRS Invokes a macro for every member of the PDS being edited.*/
      /*                                                              */
(1)       ISREDIT MACRO (DOITMAC)                 /* Pass macro name      */
(2)       ISREDIT (DATA1) = DATAID                /* Get the data id      */
(3)       ISREDIT (CURMBR) = MEMBER               /* Get edit member name */
(4)       ISPEXEC LMOPEN DATAID(&DATA1)           /* Open dataid for input*/ -
                         OPTION(INPUT)            /*                      */
(5)       SET LMRC = &LASTCC                      /*                      */

(6)       DO WHILE (&LMRC = 0 )                   /* Build member     (3) */
(7)          ISPEXEC LMMLIST DATAID(&DATA1)       /*  list and return     */ -
                   OPTION(LIST) MEMBER(MEMBER)    /*  next member name    */ -
                   STATS(NO)                      /*                      */
             SET &LMRC = &LASTCC                  /* Capture return code  */

             IF &LMRC = 0 THEN                    /* If a member name     */ -
               DO                                 /*  returned OK         */
                                                  /*  current member      */
(8)              IF &CURMBR = &MEMBER THEN       /*  Skip if the same    */
                 ELSE                             /* Otherwise will       */ -
                   DO                             /*  invoke inner macro  */
(9)                  WRITE PROCESSING MEMBER &MEMBER
                                                  /*  confirm working     */
(10)                 ISREDIT &DOITMAC &MEMBER     /* Invoke macro     (5) */
                   END                            /*    member name       */
               END
          END
(11)      ISPEXEC LMMLIST DATAID(&DATA1)          /* Free member list     */ -
                         OPTION(FREE)             /*                      */
          ISPEXEC LMCLOSE DATAID(&DATA1)          /* Close dataid         */
        EXIT CODE(&MAXCC)
```

Figure 9 (Part 1 of 2). ALLMBRS Macro

(1) The MACRO command identifies &DOITMAC as the variable to
    contain the name of the inner macro. If this macro is invoked
    without a parameter, &DOITMAC will be set to a null value.
    IMBED (Figure 10) is an example of a macro
    that can be invoked by ALLMBRS.
(2) The DATAID assignment statement will return a dataid in variable &DATA1.
    The dataid defines the data set(s) to library management (LMF).
    When this macro is invoked under the ISPF/PDF editor, the dataid
    identifies the concatenation of data sets currently being
    edited.
(3) The name of the member being edited is returned in &CURMBR.
(4) The data set is opened by LMF to allow the LMMLIST service to be
    invoked later.
(5) The condition code is captured in &LMRC. The CLIST processor
    updates the &LASTCC variable after each statement is processed.
(6) The CLIST DO statement is coded to loop as long as no error
    is found by LMOPEN or LMMLIST. &LMRC will be set nonzero
    when the member list is exhausted, ending the loop.
(7) LMMLIST invokes the LMF member list service. It returns the
    next member name in the MEMBER variable.
(8) If the current member being edited is the same as the name
    returned by LMMLIST, nothing is done. This IF statement
    does not have a CLIST continuation character as the last
    character on the line; therefore, no action is taken when the
    IF statement is true.
(9) A CLIST WRITE statement is used to write line-I/O messages.
    As the macro processes each member, the member name will appear on
    the terminal to keep you informed as to what is happening.
    A nicer way to do this is to display a panel showing the member
    name after issuing a ISPEXEC CONTROL DISPLAY LOCK.
(10)The inner macro is invoked. If no macro name was passed
    to ALLMBRS, the member name is used as the macro name.
    If the macro does not exist, a macro error panel is displayed.
(11)At the end of the loop, the dataid is closed and freed.

Figure 9 (Part 2 of 2). ALLMBRS Macro

IMBED MACRO

The IMBED macro (Figure 10) builds a list of imbed (.im) statements found in the member whose name is entered as a parameter. The list is created at the end of the member currently being edited. The imbed statements are indented under a MEMBER identifier line.

```
MEMBER mbrname
  .im imbedname1
  .im imbedname2
MEMBER mbrname
```

You can invoke this macro by editing a new member, such as IMBEDLIST, and then enter ALLMBRS IMBED on the command line.

```
      /* IMBED   -   Creates a list of imbed statements                  */
      /*                                                                 */
       ISREDIT MACRO (MEMBER)                        /* Member name passed  */
                                                     /*   as input          */
(1)      ISREDIT LINE_AFTER .ZL = 'MEMBER &MEMBER'  /* Add member ID line  */
(2)      ISREDIT (LINENBR) = LINENUM .ZL            /* Get its line number */
                                                     /*                     */
(3)      ISREDIT COPY AFTER .ZL &MEMBER             /* Copy member at end  */
(4)      ISREDIT (NEWLL) = LINENUM .ZL              /* Get new last line#  */
                                                     /*                     */
(5)      IF &LINENBR = &NEWLL THEN                  /* If no data copied   */ -
           EXIT CODE(8)                              /*   exit              */
         ELSE                                        /* Else                */
           DO                                        /*                     */
(6)          ISREDIT LABEL &EVAL(&LINENBR + 1)      /* Label first line    */ -
                       = .FIRST                      /*  copied             */
                                                     /*                     */
(7)          ISREDIT EXCLUDE  ALL  .FIRST .ZL       /* Exclude just copied */
                                                     /*  lines              */
(8)          ISREDIT FIND ALL .IM 1 .FIRST .ZL      /* Show lines          */
             SET FINDRC = &LASTCC                    /*  containing .im     */
                                                     /*                     */
(9)          ISREDIT DELETE  ALL X  .FIRST .ZL      /* Delete any lines    */
                                                     /*  still excluded     */
(10)         ISREDIT (NEWLL) = LINENUM .ZL          /* Update new last     */
                                                     /*  line# after delete */
(11)         IF &FINDRC = 0 THEN                    /* If .im was found    */ -
               DO WHILE (&LINENBR < &NEWLL)         /*   loop through      */
                 SET LINENBR = &LINENBR + 1         /*   all .im lines     */
                 ISREDIT SHIFT &LINENBR ) 8         /*   shift right 8     */
               END
           END
       EXIT CODE(&MAXCC)
```

Figure 10 (Part 1 of 2). IMBED Macro

(1) Add a line identifying the member to be searched at the end
of IMBEDLIST. The .ZL label (or .ZLAST) is always
associated with the last line in the file.
(2) Retrieve the line number of the identifier line just added
by statement (1) into &LINENBR.
(3) Now copy, at the end of IMBEDLIST, the member whose name was
passed as an input parameter.
(4) &NEWLL is set to the new last line number of IMBEDLIST.
(5) Check to see if any lines were added by the copy. Exit from the
macro if no lines were added.
(6) Set the .FIRST label on the first line copied. This label
is only available to this macro and is not seen by the end
user.
(7) Exclude all the lines that were just copied: all the lines
in the range .FIRST to .ZL.
(8) The FIND command is used to find all occurrences of .IM
starting in column 1 of the copied lines.
This will show (unexclude) the lines to keep.
If .IM was not found on any line, &FINDRC will be 4.
(9) All the lines still excluded are now deleted.
(10) Reobtain the last line number, because it will have changed if
lines were deleted.
(11) If .IM lines were found, loop using a column shift to indent
them under the member identifier line. Note that &LINENBR
is still associated with the identifier line.

Figure 10 (Part 2 of 2). IMBED Macro

FINDCHGS MACRO

The FINDCHGS macro (Figure 11) identifies the lines most recently changed by showing just those lines and excluding all the others. When no level is passed, the latest level is assumed. A label range may also be passed to FINDCHGS to limit the search. This macro relies on the modification level maintained by edit for members with numbers and ISPF statistics.

For example, to show lines with level 8 or greater on line range:

```
COMMAND ===> FINDCHGS 8 .FIRST .LAST
```

```
      /*                                                              */
      /* FINDCHGS shows the most recent changes to a file.            */
      /*                                                              */
(1)   ISREDIT MACRO (SEARCH,PARMS)                  /* Macro accepts args:  */
                                                    /*  level & label range */
(2)     ISREDIT (SAVE) = USER_STATE                 /*Save user info/csr pos*/
        ISREDIT (NUMBER)= NUMBER                    /* Get number mode      */
        ISREDIT (STATS) = STATS                     /*      stats mode      */
        ISREDIT (LEVEL) = LEVEL                     /*      current level   */
(3)     IF &SEARCH = &STR() | &SUBSTR(1:1,&SEARCH ) = &STR(.) THEN -
          DO                                        /* If first arg omitted */
            SET PARMS = &STR(&SEARCH &PARMS)        /*   or looks like a    */
            SET SEARCH = &LEVEL                     /*   label, keep labels */
          END                                       /*   set current level  */
(4)     IF &STATS = OFF | &NUMBER = OFF THEN -
          DO                                        /* If level not possible*/
            SET ZEDSMSG = INVALID DATA
            SET ZEDLMSG = BOTH NUMBER AND STATS MODE MUST BE ON
            ISPEXEC SETMSG MSG(ISRZ001)             /*  set an error message*/
            EXIT CODE(8)
          END
(5)     IF &DATATYPE(&SEARCH) = CHAR | &SEARCH > &LEVEL THEN -
          DO                                        /* First arg not number */
            SET ZEDSMSG = INVALID ARG
            SET ZEDLMSG = &STR(SEARCH ARGUMENT MUST BE FIRST AND +
                          MUST BE A NUMBER <= CURRENT LEVEL NUMBER)
            ISPEXEC SETMSG MSG(ISRZ001)             /*  set an error message*/
            EXIT CODE(8)
          END
                                                    /*If here no errors     */
(6)     ISREDIT NUMBER = OFF                        /* now numbers are data */
(7)     ISREDIT (RECFM) = RECFM                     /* get file record fmt  */
        IF &RECFM = F THEN -
          DO                                        /*Fixed format file so  */
            ISREDIT (LRECL) = LRECL                 /* get maximum column   */
            SET COL1 = &LRECL - 1                   /* in file and use last */
            SET COL2 = &LRECL                       /* 2 columns to find lvl*/
          END
```

Figure 11 (Part 1 of 2). FINDCHGS Macro

```
        ELSE -
          DO                                        /*Variable format file: */
            SET COL1 = 7                            /* use columns 7 and 8  */
            SET COL2 = 8
          END                                       /*Initialize for loop   */
(8)     ISREDIT EXCLUDE ALL                         /* exclude all lines    */
(9)     DO WHILE (&SEARCH LE &LEVEL)                /*Do for each level     */
                                                    /* find level where     */
          ISREDIT FIND ALL '&SEARCH' &COL1 &COL2 &PARMS
                                                    /* found lines popped   */
          SET SEARCH = &SEARCH + 1                  /* up level to next     */
        END
(10)    ISREDIT USER_STATE = (SAVE)                 /*Restore user values   */
      EXIT CODE(&MAXCC)
```

(1) FINDCHGS allows three optional parameters to be passed. A search level and a label range (two labels). If all three are passed, PARMS will contain two labels.
(2) This macro saves end user information, number mode, last find string, cursor location etc. It also retrieves number, stats mode and the current modification level for parameter checking.
(3) FINDCHGS requires that the level be entered first if it is specified. This checks allow the level to default to the current (highest) modification level. Notice a label range can be specified without a level number; PARMS is reset to capture both labels.
(4) Check if member modification level is maintained. If not issue error message and exit macro.
(5) A CLIST DATATYPE function is used to check if first parameter is valid (a number). If not valid, issue an error message and exit from the macro.
(6) Now we've passed validity checks, so set number mode off. This allows us to treat the number field, which contains the level number, as data.
(7) Set &COL1 and &COL2 to the columns containing the level numbers.
(8) Exclude all lines in the member.
(9) For each level, find that level number. If a label range was specified it will be in the PARMS variable. All lines with matching levels will be unexcluded.
(10)Restore user values and especially number mode.

Figure 11 (Part 2 of 2). FINDCHGS Macro

MASKDATA MACRO

The MASKDATA macro (Figure 12) allows data in the mask line to overlay lines. It can be used to place a comment area over existing lines in a member.

To invoke, specify MASKDATA in the command area and also indicate the range of lines to be overlaid a O or $ line command. Then press ENTER. You can use O, OO, On, or $, $$, $n, where n is the number of lines. If you specify an O line command, data on that line is nondestructively overlaid with mask line data: only blanks are replaced by the mask line data. If you specify a $ line command, nonblank mask line data overlays data existing on the line.

```
      /*                                                              */
      /* MASKDATA   -   Overlays a line with data from the mask line. */
      /*                                                              */
(1)    ISREDIT MACRO NOPROCESS                    /* Wait to process  */
(2)      ISREDIT PROCESS RANGE O $                /* "O" and "$" reserved */
(3)      IF &LASTCC = 0 THEN                      /*   for macro      */ +
           DO                                     /* If specified get */
(4)          ISREDIT (CMD) = RANGE_CMD            /*  command entered (3) */
(5)          ISREDIT (LINE1) = LINENUM .ZFRANGE   /*  and line number (4) */
             ISREDIT (LINE2) = LINENUM .ZLRANGE   /*  range           */
             DO WHILE &LINE1 LE &LINE2            /* Loop merging data */
               ISREDIT (LINE) = LINE &LINE1       /*  based on which  */
                                                  /*  line command was */
(6)            IF &CMD = $ THEN                   /*  entered. If $   */ +
                 ISREDIT LINE &LINE1 = (LINE) + MASKLINE
               ELSE                               /*  overlay data- else */ +
                 ISREDIT LINE &LINE1 = MASKLINE + (LINE)
                                                  /*   do not overlay */
               SET LINE1 = &LINE1 + 1             /*  Increment line num */
             END                                  /*                  */
             SET RC = 0                           /*                  */
           END                                    /*                  */
         ELSE                                     /* Else give prompt */ +
           DO                                     /*  message         */
(7)          SET ZEDSMSG = &STR(ENTER "O"/"$" LINE CMD)
             SET ZEDLMSG = &STR("MASKDATA" REQUIRES AN "O" OR +
                "$" CMD TO INDICATE LINE(S) MERGED WITH MASKLINE
             ISPEXEC SETMSG MSG(ISRZ001)          /*                  */
             SET RC = 12                          /* Return code >= 12 */
           END                                    /*  keeps command in */
       EXIT CODE(&RC)                             /*  command area    */
```

Figure 12 (Part 1 of 2). MASKDATA Macro

(1) The NOPROCESS keyword on the MACRO command allows the macro to control when end-user input (changes to data and line commands) is processed.
(2) Now process user input; also notice if certain line commands are entered. The 0 and $ following the RANGE keyword indicate the line commands to be processed by this macro.
(3) A zero return code indicates the user entered a 0 or $ in any of its valid forms: 00-00, 0n, etc.
(4) &CMD is set to 0 or $, whichever command was entered.
(5) &LINE1 and &LINE2 contain the first and last line numbers of the lines indicated by the user line commands.
(6) Each line indicated by the user is merged with data from the mask line. The &LINE variable contains line data. The line command entered controls how the data is merged. A $ indicates nonblank mask line data will overlay line data. An 0 indicates the mask line data will only overlay where the line contains blanks.
(7) When no line command is entered issue a prompt message. Set a return code of 12 to keep MASKDATA displayed in the command line.

Figure 12 (Part 2 of 2). MASKDATA Macro

CHAPTER 6. WRITING PROGRAM MACROS

In addition to writing edit macros as TSO CLISTs, you can also write edit macros in programming languages, just as you write programming dialogs. There are three basic reasons to go to the additional work of debugging a program macro:

1. A macro that is executed many times will execute faster in a language which can be precompiled than in the CLIST interpretive language.

2. A macro that has to deal with data containing symbols can confuse the CLIST processor. Ampersands in data can cause problems.

3. A macro that has complex logic may better be handled in a programming language.

There are some differences in the way program macros are handled:

- Variables are not self defining in a program macro. The VDEFINE dialog service must be invoked to identify variables looked at or set by the program.

- Variables are not automatically converted to uppercase. A macro invoked accepting parameter input must be aware that the input may be in lowercase.

- When an unknown name is typed on the command line, the editor automatically checks SYSPROC to see if a member with the same name exists. If it does, it is assumed to be a macro. This is not done for program macros. There are two ways to tell edit to invoke a program macro: precede the name with a "!" if it is less than eight characters, or use the DEFINE command to define the name as a program macro.

- Program macros may be executed without being verified as a macro; the macro statement may be preceded by calls to dialog services.

- The editor will scan edit statements to do variable substitution. Only one level of scanning is done. This may simplify using variables set by edit in a subsequent command. Scanning edit commands for ampersands is a default; use the SCAN assignment statement to prevent this step.

Edit commands are executed from a program macro using the ISPLINK (or ISPLNK for FORTRAN) or ISPEXEC interface. The appropriate program must be link edited with your macro program. Parameters are passed to the ISREDIT service as follows:

```
CALL ISPLINK ('ISREDIT',length,buffer)

CALL ISPEXEC (length,'ISREDIT command')
```

'ISREDIT'
identifies the service name

length
must be a fullword integer that contains the length of the command buffer

buffer
is the command buffer that may contain any edit command that is valid from a macro, coded with the same syntax that would be used in a CLIST.

command
is any PDF edit command that is valid from a macro, coded with the same syntax that would be used in a CLIST. No CLIST variables or functions may be coded in the command field.

The following examples show three different methods of coding a FIND command. They are coded using PL/I syntax:

```
1.  CALL ISPLINK ('ISREDIT',LEN0,'¢FIND XYZ¢')
2.  CALL ISPLINK ('ISREDIT',LEN8,'FIND XYZ')
3.  CALL ISPEXEC (LEN16,'ISREDIT FIND XYZ')
```

LEN0
is a fullword program variable containing a length of 0.

LEN8
is a fullword program variable containing a length of 8.

LEN16
is a fullword program variable containing a length of 16.

In each example, the remainder of the command is coded as a literal value.

The first two examples show the ISPLINK format. In the ISPLINK call, ISREDIT is coded as the first parameter and is omitted from the command buffer.

The first example uses a special interface. A zero length may be passed only when the command is delimited by a special character. A special character cannot be A-Z or 0-9. If the length is zero and if a valid delimiter is the first character in the command buffer, a scan of the

command is done to find the next occurrence of that character. The command length is the number of characters between the two delimiters. In this case, the cent sign (¢) is used as a delimiter.

In the second example, an explicit length of 8 is coded and the command buffer contains the command without delimiters.

The third example shows the ISPEXEC format. This format always requires the length of the command buffer to be passed. The command buffer in this case includes the ISREDIT prefix, in the same way the CLIST command is coded.

WRITING PROGRAM MACROS

It is often helpful when writing a program macro to first code it as a CLIST macro to help debug the logic and the command statements. This was done with SEPLINE, a simple macro that separates each line in a file with a line of dashes. The CLIST syntax is shown in Figure 13, the PL/I program is shown in Figure 14, and the COBOL program is shown in Figure 15. Notice that, in the program, a VDEFINE is not required for the variable &SAVE, which is only referenced by edit.

```
ISREDIT MACRO

  ISREDIT (SAVE) = USER_STATE
  ISREDIT RESET
  ISREDIT EXCLUDE ----- 1 ALL
  ISREDIT DELETE ALL X
  SET &LASTL = 1
  SET &LINE = 0
  SET &LINX = &STR(----------------------------------+
              ------------------------------------)

  DO WHILE (&LINE < (&LASTL + 1) )
    ISREDIT LINE_AFTER &LINE = (LINX)
    ISREDIT (LASTL) = LINENUM .ZLAST
    SET &LINE = &LINE + 2
  END
  ISREDIT USER_STATE = (SAVE)
EXIT
```

Figure 13. SEPLINE CLIST Macro

```
/*                                                                    */
/*  SEPLINE- EDIT MACRO PROGRAM TO INSERT SEPARATOR LINES             */
/*                                                                    */
SEPLINE: PROC OPTIONS(MAIN);
/*                                                                    */
  DECLARE                                 /*                          */
      LINEX CHAR (70) INIT ((70)'-'),     /* SEPARATOR LINE -------   */
      LASTL FIXED BIN(31,0) INIT (0),     /* LAST LINE OF TEXT        */
      LINE  FIXED BIN(31,0) INIT (0),     /* CURRENT LINE NUMBER      */
      LEN0  FIXED BIN(31,0) INIT (0),     /* LENGTHS - 0              */
      LEN1  FIXED BIN(31,0) INIT (1),     /* LENGTHS - 1              */
      LEN4  FIXED BIN(31,0) INIT (4),     /* LENGTHS - 4              */
      LEN70 FIXED BIN(31,0) INIT (70); /* LENGTHS - 70                */
                                          /*                          */
  DECLARE                                 /*                          */
      ISPLINK ENTRY OPTIONS(ASM,INTER,RETCODE);/* LINK TO ISPF        */
                                          /*                          */
      CALL ISPLINK ('VDEFINE','(LASTL)',LASTL,'FIXED',LEN4);
      CALL ISPLINK ('VDEFINE','(LINE)', LINE, 'FIXED',LEN4);
      CALL ISPLINK ('VDEFINE','(LINEX)',LINEX,'CHAR', LEN70);

      CALL ISPLINK('ISREDIT',LEN0,'¢  MACRO ¢');
      CALL ISPLINK('ISREDIT',LEN0,'¢  (SAVE) = USER_STATE ¢');
      CALL ISPLINK('ISREDIT',LEN0,'¢  RESET ¢');
      CALL ISPLINK('ISREDIT',LEN0,'¢  EXCLUDE ------ 1 ALL ¢');
      CALL ISPLINK('ISREDIT',LEN0,'¢  DELETE ALL X  ¢');

      LASTL = 1;
      LINE = 0;

      DO WHILE (LINE < (LASTL + 1));
        CALL ISPLINK ('ISREDIT',LEN0,'¢ LINE_AFTER &LINE = (LINEX) ¢');
        CALL ISPLINK ('ISREDIT',LEN0,'¢ (LASTL) = LINENUM .ZLAST ¢');
        LINE = LINE + 2;
      END;

      CALL ISPLINK('ISREDIT',LEN0,'¢ USER_STATE = (SAVE) ¢');

 END SEPLINE;
```

Figure 14. SEPLINE PL/I Macro

```
 ID DIVISION.
 PROGRAM-ID.  SEPLINE.
*
*           EDIT MACRO PROGRAM TO INSERT SEPARATOR LINES
*
 ENVIRONMENT DIVISION.
 DATA DIVISION.
 WORKING-STORAGE SECTION.

 01  LINEX    PIC  X(70) VALUE  ALL "-".
*    SEPARATOR LINE -------

 01  LASTL    PIC  9(6)  VALUE   0  COMP.
* LAST LINE OF TEXT

 01  LYNE     PIC  9(6)  VALUE   0  COMP.
* CURRENT LINE NUMBER

 01  ISREDIT  PIC  X(8)  VALUE "ISREDIT ".
 01  VDEFINE  PIC  X(8)  VALUE "VDEFINE ".
 01  ZLASTL   PIC  X(8)  VALUE "(LASTL )".
 01  ZLINE    PIC  X(8)  VALUE "(LINE  )".
 01  ZLINEX   PIC  X(8)  VALUE "(LINEX )".
 01  FIXED    PIC  X(8)  VALUE "FIXED   ".
 01  CHAR     PIC  X(8)  VALUE "CHAR    ".
 01  LEN0     PIC  9(6)  VALUE   0  COMP.
 01  LEN4     PIC  9(6)  VALUE   4  COMP.
 01  LEN70    PIC  9(6)  VALUE  70  COMP.

 01  EM1      PIC  X(10) VALUE "¢  MACRO ¢".
 01  EM2      PIC  X(24) VALUE "¢  (SAVE) = USER_STATE ¢".
 01  EM3      PIC  X(10) VALUE "¢  RESET ¢".
 01  EM4      PIC  X(25) VALUE "¢  EXCLUDE ------ 1 ALL ¢".
 01  EM5      PIC  X(18) VALUE "¢  DELETE ALL X  ¢".
 01  EM6      PIC  X(30) VALUE "¢ LINE_AFTER &LINE = (LINEX) ¢".
 01  EM7      PIC  X(28) VALUE "¢ (LASTL) = LINENUM .ZLAST ¢".
 01  EM8      PIC  X(23) VALUE "¢ USER_STATE = (SAVE) ¢".
```

Figure 15 (Part 1 of 2). SEPLINE COBOL Macro

```
PROCEDURE DIVISION.
    CALL  "ISPLINK"  USING  VDEFINE  ZLASTL  LASTL  FIXED LEN4
    CALL  "ISPLINK"  USING  VDEFINE  ZLINE   LYNE   FIXED LEN4
    CALL  "ISPLINK"  USING  VDEFINE  ZLINEX  LINEX  CHAR  LEN70

    CALL  "ISPLINK"  USING  ISREDIT  LEN0    EM1
    CALL  "ISPLINK"  USING  ISREDIT  LEN0    EM2
    CALL  "ISPLINK"  USING  ISREDIT  LEN0    EM3
    CALL  "ISPLINK"  USING  ISREDIT  LEN0    EM4
    CALL  "ISPLINK"  USING  ISREDIT  LEN0    EM5

    MOVE 1 TO LASTL
    MOVE 0 TO LYNE
    PERFORM LOOP UNTIL LYNE IS NOT LESS THAN (LASTL + 1)
    CALL  "ISPLINK"  USING  ISREDIT  LEN0    EM8
    GOBACK.
LOOP.
    CALL  "ISPLINK"  USING  ISREDIT  LEN0    EM6
    CALL  "ISPLINK"  USING  ISREDIT  LEN0    EM7
    ADD 2 TO LYNE.
```

Figure 15 (Part 2 of 2). SEPLINE COBOL Macro

INVOKING PROGRAM MACROS

Edit assumes that any primary command that is unknown is a macro, and it normally assumes that the macro has been implemented as a CLIST. You can define a macro to edit as a program macro, either by executing a DEFINE command or by prefixing the invocation of the macro with the special character "!".

If a macro named FINDIT were coded as a CLIST, for example, it would be invoked by entering FINDIT. If it were coded as a program, it could be invoked by entering !FINDIT, or it could be invoked by entering FINDIT if it had previously been defined as a program macro by means of the DEFINE command.

The first invocation of a program macro with a "!" prefix implicitly defines that macro as a program macro. Thereafter, the prefix may be omitted.

To use the DEFINE command to define a macro as a program, code DEFINE name PGM MACRO.

Note: The keywords may be coded in either order. The following, for example, is valid:

DEFINE name MACRO PGM

You can explicitly define a built-in command, such as FIND, as a macro (causing the macro to override the built-in command) by using the DEFINE command.

VARIABLE SUBSTITUTION

The SCAN assignment statement is used either to set the current value of scan mode (for variable substitution), or to retrieve the current value of scan mode and place it in a variable. Scan mode controls the automatic replacement of variables in command lines passed to edit.

When scan mode is on, edit command lines are scanned for ampersands (&). If an & followed by a nonblank character is found, the name following the ampersand (terminated by a blank or period) is assumed to be a variable name, and the value of the variable is substituted in the command for the '&name' or '&name.' (the period allows concatenation of the variable value without an intervening blank delimiter) before the command is processed.

CHAPTER 7. MACRO COMMAND REFERENCE

This chapter contains information about the edit macro commands available for ISPF/PDF.

Each command description consists of the following information:

Description A description of the function and operation of the command. This description also refers to other commands that may be used with this command.

Syntax A syntax diagram for coding the macro command.

Operands A description of any required or optional keywords or parameters.

Return Codes A description of the codes returned by the macro command. For all commands a return code of 20 implies a severe error. This error is normally a command syntax error, but may be any severe error detected when using dialog service routines. For example, if a severe error is detected in attempting to access a variable or a system table, a code of 20 is returned.

Some commands may return additional codes for specific errors.

Examples Sample usage of the macro command.

A list of all edit macro commands with their syntax can be found in Appendix B. A list of all abbreviations for command and keyword operand names can be found in Appendix C. Note, however, that it is recommended that you not code abbreviations for command names or operands, for ease in reading and maintenance.

AUTOLIST - Set or Query Autolist Mode

The AUTOLIST assignment statement is used either to set the current autolist mode, or to retrieve the current setting of autolist mode and place it in a variable. Autolist mode controls whether edit generates a source listing in the ISPF list data set when the edit session is terminated with data that was changed.

```
ISREDIT (varname)  = AUTOLIST
ISREDIT AUTOLIST = mode
ISREDIT AUTOLIST mode
```

---

varname — is the name of a variable containing the setting of autolist mode, either ON or OFF.

mode — is the setting of autolist mode, either ON or OFF:

- ON — When the edit session is terminated with data that was changed, PDF edit generates a source listing in the ISPF list data set.
- OFF — No source listing is generated.

If no value is specified when setting autolist mode, ON is assumed.

The following return codes may be issued:

```
 0 - Normal completion
20 - Severe error
```

Examples:

To store the autolist mode setting in variable &ALMODE:

```
ISREDIT (ALMODE) = AUTOLIST
```

To set autolist mode on with the assignment statement:

```
ISREDIT AUTOLIST = ON
```

To set autolist mode on with the end user command:

```
ISREDIT AUTOLIST ON
```

To set autolist mode from the variable &ALMODE:

```
ISREDIT AUTOLIST = (ALMODE)
```

AUTONUM - Set or Query Autonum Mode

The AUTONUM assignment statement is used either to set the current autonum mode, or to retrieve the current setting of autonum mode and place it in a variable. Autonum mode controls the automatic renumbering of data when it is saved.

```
ISREDIT (varname) = AUTONUM
ISREDIT AUTONUM = mode
ISREDIT AUTONUM mode
```

---

varname — is the name of a variable containing the setting of autonum mode, either ON or OFF.

mode — is the setting of autonum mode, either ON or OFF:

- ON — When number mode is also on, the data is automatically renumbered when it is saved.
- OFF — Data is not renumbered.

If no value is specified when setting autonum mode, ON is the default.

The following return codes may be issued:

- 0 - Normal completion
- 20 - Severe error

Examples:

To put the value of autonum mode into variable &ANUM:

```
ISREDIT (ANUM) = AUTONUM
```

To set autonum mode from variable &ANUM1:

```
ISREDIT AUTONUM = &ANUM1
```

To set autonum mode off with the end user command:

```
ISREDIT AUTONUM OFF
```

AUTOSAVE - Set or Query Autosave Mode

The AUTOSAVE assignment statement is used either to set the current autosave mode, or to retrieve the current setting of autosave mode and place it in variables. Autosave mode controls the saving of data when the END command is issued.

```
ISREDIT (varname1,varname2) = AUTOSAVE
ISREDIT AUTOSAVE = mode
ISREDIT AUTOSAVE mode
```

---

varname1 is the name of a variable to contain the setting of autosave mode, ON or OFF.

varname2 is the name of a variable to contain the prompt value, PROMPT or NOPROMPT.

mode is the setting of autosave mode.

- ON — When coded, the changed data is saved when END is entered. PROMPT or NOPROMPT are ignored, if coded.
- OFF PROMPT — When coded, the user is notified that changes have been made and that either SAVE (followed by END) or CANCEL must be used.
- OFF NOPROMPT — When coded, the user is not notified and data is not saved when an END command is issued. The END command becomes an equivalent to the CANCEL command. Use this option with caution.

If no value is supplied when setting autosave mode, ON NOPROMPT is assumed. If ON is supplied for the first value, NOPROMPT is assumed for the second value. If OFF is supplied for the first value, PROMPT is assumed for the second value.

The following return codes may be issued:

- 0 - Normal completion
- 4 - OFF NOPROMPT specified
- 20 - Severe error

Examples:

To put the value of autosave mode into variables &ASAV1 and &ASAV2:

```
ISREDIT (ASAV1,ASAV2) = AUTOSAVE
```

To set autosave mode from variables &ASAV1 and &ASAV2:

```
ISREDIT AUTOSAVE = (ASAV1,ASAV2)
```

To set autosave mode on:

```
ISREDIT AUTOSAVE ON
```

BLKSIZE - Query the Block Size

The BLKSIZE assignment statement returns the block size of the data set being edited in a specified variable.

```
ISREDIT (varname) = BLKSIZE
```

---

varname is the name of a variable to contain the block size of the data set being edited, a 6-character value, left-padded with zeros.

The following return codes may be issued:

```
 0 - Normal completion
20 - Severe error
```

Example:

To check the block size of the data set and process the data if the blocksize is greater than 4096:

```
ISREDIT (BSIZE) = BLKSIZE
IF &BSIZE > 4096 THEN -
   ...
```

BOUNDS - Set or Query the Current Boundaries

The BOUNDS command is used to set or retrieve the current left boundary, the current right boundary, or both boundaries and place these values in variables. The column numbers are always data column numbers. Thus, for a variable format data set with number mode on, data column 1 is column 9 in the record.

```
ISREDIT (varname1,varname2) = BOUNDS
ISREDIT BOUNDS = left right
ISREDIT BOUNDS left right
```

---

varname1 is a variable containing the left boundary.
varname2 is a variable containing the right boundary.
left is the left boundary to be set.
right is the right boundary to be set.

To set one boundary while leaving the other value unchanged, enter an asterisk (*) for the boundary to be unchanged. To set the boundaries to their default values, enter the BOUNDS command without arguments.

The following return codes may be issued:

```
 0 - Normal completion
 4 - Right boundary greater than default, default right boundary used
12 - Invalid boundaries specified
20 - Severe error
```

Examples:

To save the value of the left boundary in the variable &LEFT:

```
ISREDIT (LEFT) = BOUNDS
```

To save the value of the right boundary in the variable &RIGHT:

```
ISREDIT (,RIGHT) = BOUNDS
```

To set the left boundary to 1, leaving the right boundary unchanged:

```
ISREDIT BOUNDS = 1 *
```

To set the boundaries to their default values:

```
ISREDIT BOUNDS
```

To set the left boundary from the variable &LEFT, leaving the right boundary unchanged:

```
ISREDIT BOUNDS &LEFT *
```

Note: The following commands work within the column range specified by the current boundary setting: CHANGE, EXCLUDE, FIND, SEEK, SHIFT, SORT, TFLOW, TSPLIT, and TENTER. This range is in effect unless overriding boundaries can be specified with the command. Refer to the individual command descriptions for the effect of the BOUNDS command.

BUILTIN - Execute a Built-in Command

The BUILTIN command is used within a macro to execute a built-in edit command, even though a macro or a macro statement with the same name may have been defined.

For example, a DEFINE END ALIAS MACEND could be issued so that when an END command is processed by edit, the user-defined MACEND macro is executed. Within the macro, logic could be performed, and a built-in END command could be issued to actually terminate edit.

Note: If the END command is issued in the user-defined MACEND macro without being preceded by BUILTIN, the MACEND macro would be executed, resulting in an endless loop.

```
ISREDIT BUILTIN cmdname
```

---

cmdname is the built-in command to be executed.

The following return codes may be issued:

```
 n - Return code from the built-in command
20 - Severe error
```

Examples: To execute the built-in END command:

```
ISREDIT BUILTIN END
```

To execute the built-in CHANGE command:

```
ISREDIT BUILTIN CHANGE ALL " " "-"
```

CANCEL - Cancel the Edit Session

The CANCEL command is used to cancel an edit session without saving the current data on the disk.

```
ISREDIT CANCEL
```

The following return codes may be issued:

```
 0 - Normal completion
20 - Severe error
```

Example:

To cancel the current edit session:

```
ISREDIT CANCEL
```

CAPS - Set or Query Caps Mode

The CAPS assignment statement is used either to set the current caps mode, or to retrieve the current setting of caps mode and place it in a variable. Caps mode controls the translation of input to uppercase.

```
ISREDIT (varname) = CAPS
ISREDIT CAPS = mode
ISREDIT CAPS mode
```

---

varname — is the name of a variable containing the setting of caps mode, either ON or OFF.

mode — is the setting of caps mode, either ON or OFF:

- ON — Input is translated to uppercase when entered.
- OFF — Input is not translated, but is left as entered.

If no value is specified when setting caps mode, ON is assumed.

The following return codes may be issued:

```
 0 - Normal completion
20 - Severe error
```

Examples:

To save the value of caps mode in variable &CAPMODE:

```
ISREDIT (CAPMODE) = CAPS
```

To set caps mode off:

```
ISREDIT CAPS = OFF
```

To set the value of caps mode from variable &CAPMODE:

```
ISREDIT CAPS &CAPMODE
```

CHANGE - Change a Data String

The CHANGE command is used to change one or more occurrences of one data string to another. The arguments on the CHANGE command are exactly the same as those available to an end user.

```
ISREDIT CHANGE str-1 str-2 [label-range] [NEXT ] [CHARS ] [X ][col-1 [col-2]]
                                         [ALL  ] [PREFIX] [NX]
                                         [FIRST] [SUFFIX]
                                         [LAST ] [WORD  ]
                                         [PREV ]
```

---

label-range indicates that two labels are required to indicate a range of lines; one label is invalid. If relative line numbers are coded, they are interpreted as column numbers or a syntax error is detected.

args For information about specifying the following arguments for the CHANGE command, see the ISPF/PDF for MVS/XA Program Reference:

NEXT	CHAR	X
ALL	PREFIX	NX
FIRST	SUFFIX	
LAST	WORD	
PREV		

If your macro is working with text containing uppercase and lowercase data, str-1 is found independent of case, unless coded as a character string. Therefore, a CHANGE 'first' 'next' will change any of the following strings - first, FIRST, or First - to next. To limit the search to a specific form of 'first', code C'first', C'FIRST', or C'First'.

The boundary settings limit the columns searched to the columns between the boundary columns. When writing a general macro, you may want to override the boundary settings with specific columns.

When the cursor is within the line range, the search for the string starts at the cursor position for a NEXT or PREV change request; when the cursor is outside the range, the search starts at the top (if NEXT) or bottom (if PREV) of the range. Therefore, if labels are included to limit the lines searched and the cursor is within those lines, the search begins at the cursor position and goes to the end of the range.

If a string to be changed is found and can be changed, the cursor position is changed based on the direction of search. For a forward

search (FIRST, NEXT, or ALL), the cursor is placed at the line and column position of the character following the end of the first changed string. For a backward search (PREV or LAST), the cursor is placed preceding the first character of the first changed string. If labels are included to limit the lines searched and the cursor is within those lines, CHANGE searches from the cursor position to the end of the range. When a string is not found, the cursor is not moved.

The following return codes may be issued:

```
 0 - Normal completion
 4 - String not found
 8 - Change error ('to' string longer than 'from' string and
     substitution was not performed on at least one change)
20 - Severe error
```

Example:

After putting the current member name in variable &MEMNAME, add an identifier to the name if it is found in columns 1 to 10 in lines between the first line and the line labeled .XLAB:

```
ISREDIT  (MEMNAME) = MEMBER
ISREDIT CHANGE WORD &MEMNAME "MEMBER: &MEMNAME" 1 10 .ZFIRST .XLAB
```

Notes:

1. CHANGE shows all lines meeting the search criteria. Use the SEEK command in combination with the XSTATUS command to preserve the exclude status of a line.
2. When a CHANGE ALL is done, the lines changed are flagged with a ==CHG>, and lines that cannot be changed are flagged with an ==ERR>. The status of these lines can be used by the LOCATE command and changed by the RESET command.

CHANGE_COUNTS - Query Change Counts

The CHANGE_COUNTS assignment statement is used to retrieve values set by the most recently executed CHANGE command and place these values in variables.

```
ISREDIT (varname1,varname2) = CHANGE_COUNTS
```

---

varname1 is the name of a variable to contain the number of strings changed, an 8-character value, left-padded with zeros.
varname2 is the name of a variable to contain the number of strings that could not be changed, an 8-character value, left-padded with zeros.

The following return codes may be issued:

```
  0 - Normal completion
 20 - Severe error
```

Examples:

To put the number of changes resulting from the most recent CHANGE command into variable &CHGED:

```
ISREDIT (CHGED) = CHANGE_COUNTS
```

To put the number of change errors into variable &ERRS:

```
ISREDIT (,ERRS) = CHANGE_COUNTS
```

To put the number of changes and change errors into variables &CHG and &ERR:

```
ISREDIT (CHG,ERR) = CHANGE_COUNTS
```

COPY - Copy a Member

The COPY command is used to copy a member of the current library into the member being edited.

```
ISREDIT COPY member {AFTER } lptr [linenum-range]
                   {BEFORE}
```

---

member indicates the name of the member of the current library to be copied.

AFTER or BEFORE indicates the relative position of the new data to the insertion point.

lptr indicates a line pointer must be used to indicate where the data is to be copied. A line pointer can be a label or a relative line number.

linenum-range indicates the data line numbers of the member being copied. Two line numbers are required to indicate a range of lines; specifying one line number is invalid.

The following return codes may be issued:

```
 0 - Normal completion
 8 - End of file reached before last record read
12 - Invalid line pointer (lptr); member not found or BLDL error
16 - End of file reached before first record of specified range read
20 - Syntax error (invalid name, incomplete range)
   - I/O error
```

Examples:

To copy all of member MEM1 at the end of the data:

```
ISREDIT COPY MEM1 AFTER .ZLAST
```

To copy all of member MEM1 before the first line of data:

```
ISREDIT COPY MEM1 BEFORE .ZFIRST
```

To copy the first three lines of member MEM1 at the beginning of the current member:

```
ISREDIT COPY MEM1 BEFORE .ZF 1 3
```

CREATE - Create a Member

The CREATE command is used to create a new member in the library that is currently being edited.

```
ISREDIT CREATE member lptr-range
```

---

member is the name of the new member to be created.
lptr-range indicates that two line pointers are required to indicate a range of lines in the current member to be used to create the new member. A line pointer may be a label or a relative line number. Specifying one line pointer is invalid.

The following return codes may be issued:

```
 0 - Normal completion
 8 - Member already exists, member not created
12 - Invalid line pointer (lptr); member not found or BLDL error
20 - Syntax error (invalid name or incomplete lptr range)
   - I/O error
```

Example:

To create a new 10-line member from the first 10 lines of member being edited:

```
ISREDIT CREATE MEM1 1 10
```

CTL_LIBRARY - Query Controlled Library Status

The CTL_LIBRARY assignment statement is used to retrieve the status of a controlled library and place the status in variables. CTL_LIBRARY is normally used in initial macros to define the use of controlled library members.

```
ISREDIT (varname1,varname2) = CTL_LIBRARY
```

---

varname1 is the name of a variable to contain the lock status of the member.
varname2 is the name of a variable to contain additional information about the status.

The following table summarizes the information contained in varname1 and varname2. The table entries are defined following the table.

varname1	varname2
OBTAINED	Userid that obtained member
UNAVAILABLE	{Userid } {DEACTIVATED} {*LOCKED* }
ERROR	blanks
NOCHECK	{FIRSTLIB} {blanks }

The value placed in varname1 is one of the following:

OBTAINED indicates that the lock has been obtained for the member being edited. The member was found in a controlled library. If the member is modified and saved in your library, the next time this statement is executed, NOCHECK will be returned as the lock status. If the member is not saved in your library, the lock for this member is freed.
UNAVAILABLE indicates that the lock could not be obtained for the member being edited. The member was found in a controlled library.
ERROR indicates that the library access service was unable to determine whether or not the member was locked because of an error or unusual condition.

NOCHECK indicates that no check was done to determine the status of the member. NOCHECK is returned in the following cases:

- The member is new
- The member was obtained from the first library in the concatenation sequence
- An ISRCFIL file name is not allocated to the user

If OBTAINED is placed in varname1, varname2 contains your userid, the userid that locked the member.

If UNAVAILABLE is placed in varname1, indicating the lock is not available, the value placed in varname2 is one of the following:

userid The userid that has locked the member
DEACTIVATED Library controls have been deactivated
LOCKED The member is available but exists in a lower level of the library structure that did not precede the library where the member was found in the edit concatenation sequence. This is called a pseudo-lock.

If ERROR is placed in varname1, varname2 is set to blanks.

If NOCHECK is placed in varname1, indicating that no checking was done, the value placed in varname2 is the reason, one of the following:

FIRSTLIB The member was found in the first library of the concatenation sequence used for editing, or the member is new.
blank The file allocation indicates that library management should not be invoked; no ISRCFIL file id is allocated.

The following return codes may be issued:

```
  0 - Normal completion
 20 - Severe error
```

Example:

To get the control and lock status of the current member:

```
ISREDIT (CSTATUS,LSTATUS) = CTL_LIBRARY
```

CURSOR - Set or Query the Cursor Location

The CURSOR assignment statement is used to set the relative line number and the relative column number of the cursor, or retrieve the relative line number and the relative column number of the cursor and place these values in variables. The position of the cursor is used as the starting location for the search argument for SEEK, FIND, CHANGE, and EXCLUDE; or as the text split point for the TSPLIT command.

```
ISREDIT (varname1,varname2) = CURSOR
ISREDIT CURSOR = line col
```

---

varname1 is the name of a variable containing the relative line number, a 6-character value, left-padded with zeros.
varname2 is the name of a variable containing the data column number, a 3-character value, left-padded with zeros.
line is the relative line number
col is the data column number

The following return codes may be issued:

- 0 - Normal completion
- 4 - Column number beyond data, line number incremented
- 12 - Invalid line number
- 20 - Severe error

Examples:

To put the line number of the current cursor position into variable &LINE:

```
ISREDIT (LINE) = CURSOR
```

To set the cursor position to data line 1, column 1:

```
ISREDIT CURSOR = 1 1
```

To set the cursor position to column 1 of the last data line:

```
ISREDIT CURSOR = .ZLAST 1
```

To set the cursor position to the line with the label .LAB, without changing the column position:

```
ISREDIT CURSOR = .LAB
```

Notes:

1. When a macro is invoked by the end user, the cursor value is the cursor position on the screen at invocation time.
2. If the cursor is in the command area, the cursor value is the relative number of the first data line on the screen, column 0.
3. If the column number is beyond the data when setting the cursor, the cursor is positioned to the next line, column 0, which is equivalent to the first position of the line command area.
4. When setting the cursor to a line number, the line number must exist.
5. When retrieving the cursor position in an empty member, the line number and column number are both set to 0.
6. The following statements may change the cursor position:

   CHANGE SEEK
   EXCLUDE TSPLIT
   FIND

   See the individual statement descriptions for their effect on cursor position. No other commands have any effect on cursor position.

DATA_CHANGED - Query the Data Changed Status

The DATA_CHANGED assignment statement is used to retrieve the current data changed status and to place it in a variable.

Note: Data may be saved without data being changed if there is a change to number, stats, or pack mode. This command does not indicate whether data will be saved.

```
ISREDIT (varname) = DATA_CHANGED
```

---

varname is the name of a variable containing the value of data changed status, either YES or NO. The data changed status is initially set to NO at the beginning of an edit session, and is reset to NO whenever a save is done. When data in the current file is changed, or if a command is issued which might have changed the data, the changed status is set to YES.

The following return codes may be issued:

```
 0 - Normal completion
20 - Severe error
```

Example:

To determine whether data has been changed, and, if it has, issue the built-in SAVE command:

```
ISREDIT (CHGST) = DATA_CHANGED
IF &CHGST = YES THEN ISREDIT BUILTIN SAVE
```

DATA_WIDTH - Query Data Width

The DATA_WIDTH assignment statement is used to retrieve the current logical data width and place it in a variable. For data without sequence numbers, the logical data width is the same as the logical record length (LRECL) of the data set being edited. For data with sequence numbers, the logical data width is:

Sequence Number Type	Logical Data Width
STD	LRECL - 8
COBOL	LRECL - 6
STD COBOL	LRECL - 14

```
ISREDIT (varname) = DATA_WIDTH
```

---

varname is the name of the variable to contain the logical data width, a 3-character value, left-padded with zeros.

The following return codes may be issued:

```
 0 - Normal completion
20 - Severe error
```

Example:

To put the data width in variable &MAXCOL and override the boundary setting for the SEEK command:

```
ISREDIT (MAXCOL) = DATA_WIDTH
ISREDIT SEEK 1 &MAXCOL &ARGSTR
```

DATAID - Query Dataid

The DATAID assignment statement is used to retrieve the dataid for the data set currently allocated for editing and place it in a variable. The dataid is created by the LMINIT service to identify a data set.

If edit was invoked with a dataid, the dataid is returned when this command is invoked. If edit was invoked without a dataid, then an LMINIT service is performed by edit and the dataid thus obtained is returned. On return from a top-level macro, edit releases any dataid it has obtained.

For further information about the use of library access services, refer to ISPF/PDF for MVS/XA Services.

```
ISREDIT (varname) = DATAID
```

---

varname is the name of a variable containing the dataid for the data set currently allocated for editing.

The following return codes may be issued:

```
 0 - The dataid returned was passed to edit
 4 - Dataid was generated by edit and will be freed by edit
 8 - A previously generated dataid was returned
20 - Severe error
```

Example:

To store the dataid in variable &DID, and then find member MEM1 of that data set using the LMMFIND library access service:

```
ISREDIT (DID) = DATAID
ISPEXEC LMMFIND DATAID(DID) MEMBER(MEM1)
IF &LASTCC = 0 THEN ...
```

DATASET - Query the Current Data Set Name

The DATASET assignment statement is used to retrieve the name of the data set into which the data currently being edited will be stored and place it in a variable.

```
ISREDIT (varname) = DATASET
```

---

varname is the name of a variable to contain the name of the data set currently being edited. The data set name is fully qualified, without quotation marks (').

The following return codes may be issued:

```
 0 - Normal completion
20 - Severe error
```

Example:

To determine if you are editing a data set with a prefix of PROJ:

```
ISREDIT (DSNAME) = DATASET
IF &SUBSTR(1:4,&DSNAME  ) = PROJ THEN -
   ....
```

DEFINE - Define a Name

The DEFINE command is used to identify a command name to edit, or to cancel the effect of a previous DEFINE command. A command name is identified as a CLIST or program macro, an alias of another command name, or a NOP.

DEFINE commands issued in a macro are in effect while the current member is being edited. DEFINE commands issued by an end user are in effect until the edit session is terminated.

DEFINE commands can be nested. There should be a DEFINE xxx RESET command for every DEFINE xxx command, unless the definition is intended to be permanent for the member or the edit session.

```
ISREDIT DEFINE name  {CMD MACRO  }
                     {PGM MACRO  }
                     {ALIAS name2}
                     {NOP        }
                     {RESET      }
```

---

name is the user invocation name.
CMD MACRO identifies 'name' as a command language macro, which is to be invoked by the SELECT CMD service.
PGM MACRO identifies 'name' as a program (load module) macro, which is to be invoked by the SELECT PGM service.
ALIAS name2 identifies 'name' as an alias of 'name2' with the same characteristics
NOP identifies 'name' as a NOP. When 'name' is invoked, nothing is executed. Any aliases are set to NOP, also.
RESET resets the most recent definition of 'name' to the status in effect prior to that definition.

CMD MACRO, MACRO CMD, and MACRO are equivalent. PGM MACRO and MACRO PGM are equivalent.

The following return codes may be issued:

- 0 - Normal completion
- 8 - RESET was attempted for a name not currently defined, or DEFINE name ALIAS name2 requested and name2 is a NOP
- 12 - DEFINE was attempted for a name not currently defined
- 20 - Severe error (unknown command)

Examples:

To define the name IJKDOIT as a program macro:

```
ISREDIT DEFINE IJKDOIT PGM MACRO
```

To define the name DOIT as an alias of the macro IJKDOIT:

```
ISREDIT DEFINE DOIT    ALIAS IJKDOIT
```

To define the name SAVE to have no effect:

```
ISREDIT DEFINE SAVE    NOP
```

To define the name SETITUP to be a program macro:

```
ISREDIT DEFINE SETITUP PGM MACRO
```

To define the name FINDIT to be a CLIST macro:

```
ISREDIT DEFINE FINDIT MACRO CMD
```

To reset the definition of the name SAVE:

```
ISREDIT DEFINE SAVE RESET
```

DELETE - Delete Lines from the Current Data Set

The DELETE command deletes lines from the current file. DELETE may specify a single line or a range of lines, or it may limit the lines to be deleted to all excluded or non-excluded lines in the file or all excluded or non-excluded lines within a line pointer range.

```
ISREDIT DELETE {ALL X|NX [lptr-range]}
               {[ALL] X|NX lptr-range}
               {lptr                 }
               {lptr-range           }
```

ALL indicates that all specified lines are to be deleted.
X indicates that only excluded lines are to be deleted.
NX indicates that only non-excluded lines are to be deleted.
lptr indicates a line pointer must be used to identify a line to be deleted. A line pointer can be a label or a relative line number.
lptr-range indicates that two line pointers are required to indicate a range of lines to be deleted. A line pointer may be a label or a relative line number. Specifying one line pointer is invalid.

The following return codes may be issued:

0 - Normal (lines deleted successfully)
4 - No lines deleted
8 - No standard records exist
12 - Invalid line number
20 - Severe error

Examples:

To delete all non-excluded lines:

```
ISREDIT DELETE ALL NX
```

To delete all lines in the range of labels .A and .B with a blank in column 1:

```
ISREDIT RESET X .A .B
ISREDIT EXCLUDE ALL ' ' 1 .A .B
ISREDIT DELETE ALL X .A .B
```

To delete the last line of data in the current file:

```
ISREDIT DELETE .ZLAST
```

To delete the first 10 lines of data in the current file:

```
ISREDIT DELETE 1 10
```

DISPLAY_COLS - Query Display Columns

The DISPLAY_COLS assignment statement retrieves the column number of the first and last data columns that are being seen by the end user and places them in variables. Columns that contain sequence numbers are not considered data columns. This assignment statement is invalid in initial macros, since the columns displayed are not known until the data is first displayed.

```
ISREDIT (varname1,varname2) = DISPLAY_COLS
```

---

varname1 is the name of a variable containing the column number of the first data column visible to the end user, a 3-character value, left-padded with zeros.

varname2 is the name of a variable containing the column number of the last data column visible to the end user, a 3-character value, left-padded with zeros.

The following return codes may be issued:

```
 0 - Normal completion
20 - Severe error
```

Example:

To put the leftmost and rightmost column values displayed to the user in variables &LEFT and &RIGHT:

```
ISREDIT (LEFT,RIGHT) = DISPLAY_COLS
```

DISPLAY_LINES - Query Display Lines

The DISPLAY_LINES assignment statement retrieves the relative line numbers of the first and last data lines that would be displayed, if the macro terminated, at this point and places them in variables. Other non-data lines might be on the display. This assignment statement is invalid in an initial macro, since the lines displayed are not known until the data is first displayed.

```
ISREDIT (varname1,varname2) = DISPLAY_LINES
```

---

varname1 is the name of a variable containing the relative line number of the first data line that would be visible to the end user if the macro terminated at this point, a 6-character value, left-padded with zeros.

varname2 is the name of a variable containing the relative line number of the last data line that would be visible to the end user if the macro terminated at this point, a 6-character value, left-padded with zeros.

The following return codes may be issued:

```
 0 - Normal completion
 4 - No visible data lines
 8 - No existing data lines
20 - Severe error
```

Example:

To place the top and bottom line numbers in variables &TOP and &BOT:

```
ISREDIT (TOP,BOT) = DISPLAY_LINES
```

DOWN - Scroll Down

The DOWN command causes a scroll down from the current screen position. To scroll down using the screen position when the macro was issued, use USER_STATE assignment statements to save and then restore the screen position parameters.

If you define a macro named DOWN, it overrides the DOWN command when used from another macro, but has no effect for the end user. The DOWN command does not change the cursor position and cannot be used in an initial macro.

```
ISREDIT DOWN amt
```

---

amt is the scroll amount, the number of lines or one of the following keywords: MAX, HALF, PAGE, CURSOR, or DATA.

The following return codes may be issued:

```
 0 - Normal completion
 4 - No visible lines
 8 - No data to display
12 - Amount not specified
20 - Severe error
```

Example:

To make the line where the cursor is placed the first one on the display:

```
ISREDIT DOWN CURSOR
```

Notes:

1. The first line that would be displayed is determined by:
   a. Whether the cursor was set explicitly by a CURSOR assignment statement or implicitly by a SEEK, FIND, CHANGE, or TSPLIT command. Since the cursor must be on the screen, the line that is the first line on the screen may be different from the line that was first when the user invoked the macro.
   b. A LOCATE command setting the line to be first on the screen.
2. The number of lines scrolled are affected by non-data lines on the screen.
3. The number of lines on the screen is determined by:
   a. The number of lines excluded from the display.
   b. The number of non-data lines displayed: profile, message, note, bounds, tabs, or mask lines.
   c. The terminal display size and the split-screen line.

EDIT - Edit a Member

The EDIT command is used to edit another member in the partitioned data set you are editing. This is called recursive editing. The current library concatenation sequence is used to find the member. Your initial edit session is suspended until the second-level edit session is complete. To exit from the second-level edit session, an END or CANCEL command must be executed by the macro or by the end user. The edit service invocation ISPEXEC EDIT ... is the recommended method to recursively invoke edit. It allows the option of editing another data set and specifying an initial macro.

→ see page 36

```
ISREDIT EDIT member
```

---

member is the name of a member in the partitioned data set you are currently using.

The following return codes may be issued:

```
 0 - Normal completion
12 - User error (invalid member name, recovery pending, sequential data
     set being edited)
20 - Severe error
```

Example:

To recursively edit the member OLDMEM in your current partitioned data set:

```
ISREDIT EDIT OLDMEM
```

END - End the Edit Session

The END command terminates the editing of the current data set. If the data has been changed, END automatically saves the data on disk unless the profile in effect has autosave mode set off. In this case, to end the edit session, either a SAVE command must precede the END command to save data, or a CANCEL command must be issued to terminate without saving data.

```
ISREDIT END
```

The following return codes may be issued:

```
 0 - Existing member saved
 4 - New member saved
12 - END not done, AUTOSAVE OFF set
   - Data not saved (not enough PDS space or directory space)
20 - Severe error
```

Example:

To end the current edit session:

```
ISREDIT END
```

EXCLUDE - Exclude Lines from the Display

The EXCLUDE command marks lines in the current data as excluded.

```
ISREDIT EXCLUDE str-1 [label-range] [NEXT ] [CHARS ] [X ] [col-1 [col-2]]
                                    [ALL  ] [PREFIX] [NX]
                                    [FIRST] [SUFFIX]
                                    [LAST ] [WORD  ]
                                    [PREV ]
```

---

str-1 indicates the string used to identify those line to be excluded.

label-range indicates that two labels are required to indicate a range of lines to be searched. Specifying one label is invalid.

args For information about the following EXCLUDE arguments, see the ISPF/PDF for MVS/XA Program Reference:

NEXT	CHARS	X
ALL	PREFIX	NX
FIRST	SUFFIX	
LAST	WORD	
PREV		

If EXCLUDE is coded with a search argument, not the keyword ALL, the cursor position affects the search. When the cursor is within the line range, the search for the string starts at the cursor position for a NEXT or PREV EXCLUDE request. When the cursor is outside the range, the search starts at the top (for NEXT) or bottom (for PREV) of the range. If the string is found, the cursor position is changed to the line and column position of the first character of the first string found. When a string is not found, the cursor is not moved. The boundaries affect the columns searched unless explicitly overridden on this command.

IF EXCLUDE is coded with the keyword ALL, it excludes all lines in the file or all lines within a specified line pointer range.

The following return codes may be issued:

- 0 - Normal completion
- 4 - String not found
- 20 - Severe error

Examples:

To exclude all lines in the file and show only those lines containing "IF":

```
ISREDIT EXCLUDE ALL
ISREDIT FIND ALL IF
```

To exclude all lines with a blank in column 1:

```
ISREDIT EXCLUDE ALL " " 1
```

To exclude all lines with an alphabetic character in column 1:

```
ISREDIT EXCLUDE P"@" 1 ALL
```

EXCLUDE_COUNTS - Query Exclude Counts

The EXCLUDE_COUNTS assignment statement is used to retrieve values set by the most recently executed EXCLUDE command and place them in variables.

```
ISREDIT (varname1,varname2) = EXCLUDE_COUNTS
```

---

varname1 is the name of a variable to contain the number of strings that were found, an 8-character value, left-padded with zeros.
varname2 is the name of a variable to contain the number of lines excluded, an 8-character value, left-padded with zeros.

The following return codes may be issued:

```
 0 - Normal completion
20 - Severe error
```

Example:

To determine the number of lines containing the word 'BOX':

```
ISREDIT EXCLUDE BOX
ISREDIT (,BOXLINES) = EXCLUDE_COUNTS
```

FIND - Find a Data String

The FIND command is used to find one or more occurrences of a data string. The arguments on the FIND command are the same arguments available to the end user.

```
ISREDIT FIND str-1 [label-range] [NEXT ] [CHARS ] [X ] [col-1 [col-2]]
                                 [ALL  ] [PREFIX] [NX]
                                 [FIRST] [SUFFIX] [EX]
                                 [LAST ] [WORD  ]
                                 [PREV ]
```

---

str-1 indicates the string to be located.

label-range indicates that two labels are required to indicate a range of lines to limit the FIND operation. If relative line pointers are coded, they are interpreted as column numbers or a syntax error is detected. Specifying one label is invalid.

args For complete information about the following FIND arguments, see the ISPF/PDF for MVS/XA Program Reference:

NEXT	CHAR	X
ALL	PREFIX	NX
FIRST	SUFFIX	EX
LAST	WORD	
PREV		

When the cursor is within the line range, the search for string-1 starts at the cursor position for a NEXT or PREV FIND request; when the cursor is outside the range, the search starts at the top (for NEXT) or bottom (for PREV) of the range. If string-1 is found, the cursor position is changed to the line and column position of the first character of the first occurrence of the string found. When a string is not found, the cursor is not moved.

The boundary settings limit the columns searched to the columns between the current boundary columns. When writing a general macro, you may want to override a boundary setting by supplying specific column numbers.

The following return codes may be issued:

- 0 - Normal completion
- 4 - String not found
- 20 - Severe error

Example:

To find the next blank character on the line with label .LAB1 and show the line if it was excluded:

```
ISREDIT FIND NEXT " " .LAB1 .LAB1
```

Note: The FIND command is not recommended for use in a macro, since any excluded string found is shown on the display. Use the SEEK command to perform the identical function without changing the lines' exclude status.

FIND_COUNTS - Query Find Counts

The FIND_COUNTS assignment statement is used to retrieve values that were set by the most recently executed FIND or RFIND command and place these values in variables.

```
ISREDIT (varname1,varname2) = FIND_COUNTS
```

---

varname1 is the name of a variable to contain the number of strings found, an 8-character value, left-padded with zeros.
varname2 is the name of a variable to contain the number of lines on which strings were found, an 8-character value, left-padded with zeros. strings were found.

The following return codes may be issued:

```
 0 - Normal completion
20 - Severe error
```

Example:

To find all occurrences of '&' in the line labeled .A and loop through all occurrences, processing them:

```
ISREDIT FIND .A .A && ALL
ISREDIT (FINDS) = FIND_COUNTS
DO WHILE &FINDS > 0
  ...
END
```

FLOW_COUNTS - Query Flow Counts

The FLOW_COUNTS assignment statement is used to retrieve values that were set by the most recently executed TFLOW command and place these values in variables.

```
ISREDIT (varname1,varname2) = FLOW_COUNTS
```

---

varname1 is the name of a variable to contain the number of original lines that participated in the text flow operation, an 8-character value, left-padded with zeros.

varname2 is the name of a variable to contain the number of lines that were generated by the text flow operation, an 8-character value, left-padded with zeros.

If the value in varname1 is larger than the value in varname2, the difference is the number of lines that were deleted from the current data because of the text flow operation. If the value in varname1 is less than the value in varname2, the difference is the number of lines that were added to the current data because of the text flow operation.

The following return codes may be issued:

```
 0 - Normal completion
20 - Severe error
```

Example:

To get the value of the rightmost column displayed, allow a margin of 8 for the text flow, and then take action if lines were added because of the text flow operation:

```
ISREDIT (,MAXCOL) = DISPLAY_COLS
ISREDIT TFLOW .ZCSR &EVAL(&MAXCOL - 8)
ISREDIT (INLINE,OUTLIN) = FLOW_COUNTS
  IF &OUTLIN > &INLINE THEN -
  ...
```

HEX - Set or Query Hex Mode

The HEX assignment statement is used either to set the current hex mode, or to retrieve the current values of hex mode and place them in variables. Hex mode controls the display of data in hexadecimal format, with vertical or data format.

```
ISREDIT (varname1,varname2) = HEX
ISREDIT HEX = mode
ISREDIT HEX mode
```

---

varname1 is the name of a variable to contain ON or OFF.
varname2 is the name of a variable to contain DATA, VERT, or blanks.
mode Hex mode can have values of ON DATA, ON VERT, or OFF:

ON DATA	Causes the hexadecimal representation of the data to be displayed as a string of hexadecimal characters (two per byte) under the characters.
ON VERT	Causes the hexadecimal representation of the data to be displayed vertically (two rows per byte) under each character.
OFF	Causes no hexadecimal representation of the data to be displayed.

The following return codes may be issued:

```
 0 - Normal completion
20 - Severe error
```

Examples:

To put the value of hex mode (ON or OFF) in variable &HEXMODE and to do processing if hex mode is on:

```
ISREDIT (HEXMODE) = HEX
IF &HEXMODE = ON THEN -
  DO ....
```

To turn hex mode off:

```
ISREDIT HEX OFF
```

IMACRO - Set or Query Initial Macro

The IMACRO assignment statement is used to set or retrieve the value for the initial macro in the current profile and place it in a variable.

```
ISREDIT (varname) = IMACRO
ISREDIT IMACRO = name
ISREDIT IMACRO   name
```

---

varname is the name of a variable to contain the name of the initial macro.

name the name of the initial macro to be executed when edit is next invoked. The name NONE can be used to eliminate the initial macro from the profile; conversely, a value of NONE is returned when no initial macro has been specified.

The following return codes may be issued:

```
 0 - Normal completion
 4 - IMACRO set not accepted; profile is locked
12 - Invalid name specified
20 - Severe error
```

Examples:

To set the initial macro name to ISCRIPT:

```
ISREDIT IMACRO ISCRIPT
```

To set no initial macro:

```
ISREDIT IMACRO NONE
```

To store the name of the initial macro in the variable &IMACNAM:

```
ISREDIT (IMACNAM) = IMACRO
```

INSERT - Prepare Display for Data Insertion

The INSERT command is used, as it is in the editor, to prepare the display for data input by the user by displaying one or more blank input lines and allowing the user to fill them with data.

Inserted lines are initialized with data from the mask line. Inserted lines are not data lines and cannot be referred to by any macro. If you do not enter data on an inserted line, it is deleted from the file.

This command is not used for adding lines with specific data; the LINE_BEFORE and LINE_AFTER assignment statements should be used for that purpose.

```
ISREDIT INSERT lptr [numlines]
```

---

lptr	indicates a line pointer must be used. A line pointer can be a label or a relative line number.
numlines	indicates the number of lines to be displayed for data input; these lines will not be saved unless data is entered on them. If numlines is not entered, one data input line is displayed.

The following return codes may be issued:

```
 0 - Normal completion
12 - Invalid line number
20 - Severe error
```

Example:

To open a 5-line area for data input after the line with the label .POINT, locate .POINT to position it to the top of the display, then issue INSERT:

```
ISREDIT LOCATE .POINT
ISREDIT INSERT .POINT 5
```

Note: You must ensure that the line referenced on the INSERT command will be displayed; otherwise, the inserted line will not be seen. The LOCATE command causes a line to be positioned at the top of the display.

LABEL - Set or Query a Line Label

The LABEL assignment statement is used to set or retrieve the values for the label on the specified line and place the values in variables.

```
ISREDIT (varname1,varname2) = LABEL lptr
ISREDIT LABEL lptr = labelname  [level]
```

---

varname1 is the name of a variable to contain the name of the label.
varname2 is the name of the variable to contain the nesting level of the label, a 3-character value, left-padded with zeros.
lptr indicates a line pointer must be used to identify the line for which a label is being set or retrieved. A line pointer can be a label or a relative line number.
labelname indicates the name of the label. It must begin with a period, followed by from one to eight alphabetic characters, the first of which must not be Z. (Z is reserved for editor-defined labels.) No special characters or numeric characters are allowed. If the label is to be seen by the end user, it must be five characters or less. To delete a label, set the label name to blank (' ').
level indicates the highest nesting level at which this label is visible to a user or macro. Level 0 is the highest level and labels at this level are visible to the user and all levels of nested macro; level 1 is not visible to the end user but to all macros, and so on. The level can never exceed the current nesting level. The maximum nesting level is 255.

The following return codes may be issued:

```
 0 - Normal completion
 4 - Label not found
 8 - Label set, but an existing label at the same level was deleted
20 - Severe error
```

Example:

To get the name of the label at the cursor, find the contents of variable &ARG and, if found, label the line so that the end user can see it:

```
ISREDIT (NAME) = LINE .ZCSR
ISREDIT FIND &ARG
IF &LASTCC = 0 THEN -
  ISREDIT LABEL .ZCSR = .POINT 0
```

Notes:

1. Use the LINENUM assignment statement to obtain the current relative line number of a line with a label. See the LOCATE and RESET command descriptions in which a label can be coded as a keyword.
2. The following commands operate on a range of lines; for these commands, a range of labels is particularly useful:

CHANGE	EXCLUDE	REPLACE	SORT
CREATE	FIND	RESET	SUBMIT
DELETE	LOCATE	SEEK	

LEFT - Scroll Left

The LEFT command causes a scroll left from the current screen position. The current setting of the boundaries may affect the amount actually scrolled. To scroll left using the screen position when the macro was issued, use USER_STATE assignment statements to save and then restore the screen position parameters.

If you define a macro named LEFT, it overrides the LEFT command when used from another macro, but has no effect for the end user. The LEFT command does not change the cursor position and cannot be used in an initial macro. For further information, see the BOUNDS and DISPLAY_COLUMNS descriptions.

```
ISREDIT LEFT amt
```

---

amt — indicates the scroll amount, the number of columns or one of the following keywords: MAX, HALF, PAGE, CURSOR, or DATA.

The following return codes may be issued:

```
 0 - Normal completion
 4 - No visible lines
 8 - No data to display
12 - Amount not coded
20 - Severe error
```

Example:

To scroll the display left and put the column specified in variable &COL in column 1:

```
ISREDIT LEFT &COL
```

Notes:

1. The first line that would be displayed is determined by:
   a. Whether the cursor was set explicitly by a CURSOR assignment statement or implicitly by a SEEK, FIND, CHANGE, or TSPLIT command. Since the cursor must be on the screen, the line that is the first line on the screen may be different from the line that was first when the user invoked the macro.
   b. A LOCATE command setting the line to be first on the screen.
2. The number of lines scrolled are affected by non-data lines on the screen.
3. The number of lines on the screen is determined by:
   a. The number of lines excluded from the display.

b. The number of non-data lines displayed: profile, message, note, bounds, tabs, or mask lines.
c. The terminal display size and the split-screen line.

LEVEL - Set or Query Modification Level

The LEVEL assignment statement is used either to set the modification level, or to retrieve the current modification level and place it in a variable.

```
ISREDIT (varname) = LEVEL
ISREDIT LEVEL = num
ISREDIT LEVEL num
```

---

varname The name of a variable to contain the modification level, a 2-character value, left-padded with zeros.
num The modification level, any number from 0 to 99

The following return codes may be issued:

```
 0 - Normal completion
 4 - Statistics mode is off
12 - Invalid value specified
20 - Severe error
```

Examples:

To reset the modification level to 1:

```
ISREDIT LEVEL = 1
```

To save the value of the modification level in variable &MODLVL:

```
ISREDIT (MODLVL) = LEVEL
```

LINE - Set or Query a Line from the File

The LINE assignment statement is used either to set or to retrieve the data from the data line specified by a line pointer and place it in a variable. The logical data width of the line determines how many characters are retrieved or set. Refer to the DATA_WIDTH description. The line pointer must be specified for setting or retrieving a line. To set data on a line, you may use a variety of data formats: (variable), templates, or merging a line with other data. The data on the line is completely overlaid with the data specified on this command.

```
ISREDIT (varname) = LINE lptr
ISREDIT LINE lptr = data
```

---

varname — is the name of a variable to contain the contents of the specified data line.

lptr — indicates a line pointer must be used. A line pointer can be a label or a relative line number.

data — indicates that the following forms may be used:

- simple string
- delimited string
- variable
- keyword:

LINE	Data from this line is used
LINE lptr	Data from the line with the given lptr
MASKLINE	Data from the mask line
TABSLINE	Data from the tabs line

- template (< col,string >)
- merge format (string-1 + string-2, keyword + string-2, string-1 + keyword)

The following return codes may be issued:

```
 0 - Normal completion
 4 - Data truncated (line shorter than data supplied)
 8 - Variable not found
12 - Invalid line number
16 - Variable data truncated
20 - Severe error
```

Examples:

To set comment delimiters in columns 40 and 70, blanking the rest of the line:

```
ISREDIT LINE 1 = < 40 '/*' 70 '*/' >
```

To overlay the first two columns of line 2 with '//':

```
ISREDIT LINE 2 = LINE +   //
```

To merge mask line data with data from variable &VAR:

```
ISREDIT LINE 3 = MASKLINE + (VAR)
```

LINE_AFTER - Add a Line to the Current File

The LINE_AFTER assignment command is used to add a line to the current file after the specified line.

```
ISREDIT LINE_AFTER lptr = [DATALINE] data
                          [NOTELINE]
                          [MSGLINE ]
```

---

lptr — indicates a line pointer must be used to identify the line after which the new line is to be inserted. A line pointer of zero causes the new line to be inserted at the beginning of the current data set. The line pointer can be either a label or a relative line number.

DATALINE — if coded, or if the operand is omitted, the line inserted is a data line.

NOTELINE — if coded, the line inserted is a temporary line (a non-data line). The line command area shows =NOTE= in high intensity and the data on the line is in low intensity. A note line has a data length of 72 characters, regardless of the data width.

MSGLINE — if coded, the line inserted is also a temporary line. The line command area contains ==MSG> in high intensity and the data on the line is also in high intensity. After NOTE and MSG lines have been added to the data, they cannot be referenced; they are not data lines. A message line has a data length of 72 characters, regardless of the data width.

data — indicates that the following data formats may be used:

- simple string
- delimited string
- variable
- keyword:

LINE	Data from the line preceding this line is used
LINE linenum	Data from the line with the given line number
MASKLINE	Data from the mask line
TABSLINE	Data from the tabs line

- template (< col,string >)
- merge format (string-1 + string-2, keyword + string-2, string-1 + keyword)

The following return codes may be issued:

0 - Normal completion
4 - Data truncated
12 - Invalid line number
20 - Severe error

Examples:

To put a new line as the first line of the file containing the string "This is the new top line of the file":

```
ISREDIT LINE_AFTER 0 = "This is the new top line of the file"
```

To put the contents of the line labeled .START on a new line following the line labeled .END:

```
ISREDIT LINE_AFTER .END = LINE .START
```

To put the contents of the mask line modified by the variable &DATA after the line whose number is in variable &N:

```
ISREDIT LINE_AFTER &N = MASKLINE + &DATA
```

LINE_BEFORE - Add a Line to the Current File

The LINE_BEFORE assignment command is used to add a line to the current file before the specified line.

```
ISREDIT LINE_BEFORE lptr = [DATALINE] data
                           [NOTELINE]
                           [MSGLINE ]
```

---

lptr — indicates a line pointer must be used to identify the line before which the new line is to be inserted. A line pointer of zero is invalid. The line pointer can be either a label or a relative line number.

DATALINE — if coded, or if the operand is omitted, the line inserted is a data line.

NOTELINE — if coded, the line inserted is a temporary line (a non-data line). The line command area shows =NOTE= in high intensity and the data on the line is in low intensity. A note line has a data length of 72 characters, regardless of the data width.

MSGLINE — if coded, the line inserted is also a temporary line. The line command area contains ==MSG> in high intensity and the data on the line is also in high intensity. After NOTE and MSG lines have been added to the data, they cannot be referenced; they are not data lines. A message line has a data length of 72 characters, regardless of the data width.

data — indicates that the following data formats may be used:

- simple string
- delimited string
- variable
- keyword:

LINE	Data from the line following this line is used
LINE lptr	Data from the line with the given line lptr
MASKLINE	Data from the mask line
TABSLINE	Data from the tabs line

- template (< col,string >)
- merge format (string-1 + string-2, keyword + string-2, string-1 + keyword)

The following return codes may be issued:

0 - Normal completion
4 - Data truncated
12 - Invalid line number
20 - Severe error

Examples:

To put the contents of the line labeled .START on a new line preceding the line labeled .END:

```
ISREDIT LINE_BEFORE .END = LINE .START
```

To put the contents of the mask line modified by the variable &DATA before the line whose number is in variable &N:

```
ISREDIT LINE_BEFORE &N = MASKLINE + &DATA
```

LINENUM - Query the Line Number of a Labeled Line

The LINENUM assignment statement is used to retrieve the current relative line number of a specified label and place it in a variable. The line number can then be used in CLIST arithmetic operations.

Note: It may not be sufficient to get the line number of a line once. If lines are added or deleted before this line, you may need to get the line number each time a change might occur.

```
ISREDIT (varname) = LINENUM label
```

---

varname is the name of the variable to contain the line number of the line with the specified label, a 6-character value, left-padded with zeros.

label is the name of the label for the line whose line number is needed. a label or a relative line number.

The following return codes may be issued:

```
 0 - Normal completion
 4 - Line 0 specified
 8 - Label specified, but not found (variable set to 0)
12 - Invalid line number
20 - Severe error
```

Examples:

To set variable &VAR to the last line number in the file:

```
ISREDIT (NUM) = LINENUM .ZLAST
```

To set variable NUM to the line number containing the label .MYLAB:

```
ISREDIT (NUM) = LINENUM .MYLAB
```

LOCATE - Locate a Line or Type of Line

The LOCATE command identifies a line to be displayed as the first line on the screen.

```
ISREDIT LOCATE {lptr                        }
               {[dir] lineid  [lptr-range]}
```

---

lptr indicates a line pointer must be used for the target. A line pointer can be a label or a relative line number.

dir indicates the direction of searching. NEXT is the default.
- NEXT to search from the cursor line, proceeding forward.
- PREV to search from the cursor line, proceeding backward.
- FIRST to search from the first line, proceeding forward.
- LAST to search from the last line, proceeding backward.

lineid generic line identifier, which may be:
- LABEL any line with a label
- CHANGE any line flagged with ==CHG> as a result of a CHANGE ALL command
- ERROR any line flagged with ==ERR> as a result of a CHANGE or SHIFT data command that failed because data would not fit on a line
- SPECIAL any special non-data line:
  - Profile lines flagged as =PROF>
  - Mask lines flagged as =MASK>
  - Bounds line flagged as =BNDS>
  - Tabs line flagged as =TABS>
  - Message lines flagged as ==MSG>
  - Note lines flagged with =NOTE=
- EXCLUDED any excluded line
- COMMAND any line with a pending line command in the line command field

lptr-range indicates that two line pointers are required to indicate a range of lines in which to search. A line pointer may be a label or a relative line number. Specifying one line pointer is invalid.

The following return codes may be issued:

- 0 - Normal completion
- 4 - Line not located
- 20 - Severe error

Examples:

To locate the next occurrence of a line with a label:

```
ISREDIT LOCATE NEXT LABEL
```

To locate the first occurrence of a special (non-data) line:

```
ISREDIT LOCATE FIRST SPECIAL
```

To locate the last excluded line:

```
ISREDIT LOCATE LAST X
```

To locate the previous line with an unexecuted line command:

```
ISREDIT LOCATE PREV CMD
```

LRECL - Query the Logical Record Length

The LRECL assignment statement returns the logical record length of the data being edited in a specified variable. This length includes the sequence number field, if there is one. Use the DATA_WIDTH command to get the length of the data columns without regard to whether the data is numbered.

```
ISREDIT (varname) = LRECL
```

---

varname is the name of a variable to contain the logical record length of the data being edited, a 3-character value, left-padded with zeros.

The following return codes may be issued:

```
 0 - Normal completion
20 - Severe error
```

Example:

To check the logical record length of the data and process the data if the LRECL is 80:

```
ISREDIT (RECLEN) = LRECL
IF &RECLEN = 80 THEN -
   ...
```

MACRO - Identify an Edit Macro

The MACRO command identifies the command as a macro. It is required in all macros. It must be the first command in a CLIST macro that is not a CLIST statement. It must be the first edit command in a program macro.

```
ISREDIT MACRO [(varname1 [,varname2,...])] [PROCESS   ]
                                           [NOPROCESS]
```

---

varname1, 2, ... if a macro allows parameters to be specified, the names of the variables to contain the parameters. Parameters are parsed and placed into the named variables in the order coded. The last variable will contain any remaining parameters. Variables that do not receive a parameter are set to a null string. A parameter is a simple or quoted string, separated by blanks or commas. Quotes may be single (') or double ("), but must be matched at the beginning and end of the string.

PROCESS processes all user-entered changes and line commands immediately

NOPROCESS processes user-entered changes and line commands when the macro completes processing or a PROCESS statement is encountered. NOPROCESS must be coded if the macro is to use line commands as input to its processing.

For more information, refer to the PROCESS statement description.

The following return codes may be returned:

```
 0 - Normal completion
20 - Severe error
```

Examples:

To begin a macro, accepting a member name and optionally a line number range to be placed in variable &PARM:

```
ISREDIT MACRO (PARM)
ISREDIT COPY AFTER .ZCSR &PARM
```

To begin a macro, checking parameters before processing screen information, testing for input omitted, non-numeric input, and too many parameters:

```
ISREDIT MACRO NOPROCESS (COL,X)
IF &STR(&COL) = &STR () THEN -
   ISREDIT (,COL) = DISPLAY_COLS
ELSE -
   IF &DATATYPE(&COL) = CHAR THEN -
      GOTO MSG
IF &STR(&X) ¬= &STR() THEN -
      GOTO MSG
ISREDIT PROCESS
```

MACRO_LEVEL - Query the Current Macro Nesting Level

The MACRO_LEVEL assignment statement is used to retrieve the current nesting level of the macro being executed and place the nesting level in a variable. The nesting level can be any number between 1 (user-invoked macro) and 255. MACRO_LEVEL is used to adjust processing based on whether the macro is invoked by an end user or by another macro. It is required if labels are to be set for the invoker of this macro. See the LABEL statement description.

```
ISREDIT (varname) = MACRO_LEVEL
```

---

varname is the name of a variable to contain the macro nesting level, a 3-character value, left-padded with zeros.

The following return codes may be issued:

```
 0 - Normal completion
20 - Severe error
```

Example:

To set the label for the caller of the macro at one less than the current level:

```
ISREDIT (NESTLEV) = MACRO_LEVEL
ISREDIT LABEL .ZCSR = .XSTR &EVAL(&NESTLEV -1)
```

MASKLINE - Set or Query the Mask Line

The MASKLINE assignment statement is used to set or retrieve the value of the mask line, used to control the formatting of input from the end user on the display. It places the mask line contents in a variable or sets the mask line from a variable. The mask line can contain any characters and serves to initialize inserted lines to the value of the mask line. See the description of templates in Chapter 2 to simplify the setting of a mask line.

```
ISREDIT (varname) = MASKLINE
ISREDIT MASKLINE = data
```

---

varname — is the name of a variable to contain the contents of the mask line.

data — indicates that the following forms may be used:

- simple string
- delimited string
- variable
- keyword:

LINE lptr	Data from the line with the given lptr
MASKLINE	Data from the mask line
TABSLINE	Data from the tabs line

- template (< col,string >)
- merge format (string-1 + string-2, keyword + string-2, string-1 + keyword)

The following return codes may be issued:

```
 0 - Normal completion
 4 - Data truncated
16 - Variable data truncated
20 - Severe error
```

Examples:

To set the mask line to place comment delimiters starting at lines 40 and 70:

```
ISREDIT MASKLINE = <40 '/*' 70 '*/'>
```

To set the mask line to blanks:

```
ISREDIT MASKLINE = ' '
```

MEMBER - Query the Current Member Name

The MEMBER assignment statement is used to retrieve the name of the member currently being edited and place it in a variable. If a sequential file is being edited, the variable is set to blanks.

```
ISREDIT (varname) = MEMBER
```

---

varname is the name of a variable to contain the name of the member currently being edited.

The following return codes may be issued:

```
 0 - Normal completion
20 - Severe error
```

Example:

To determine if you are editing a member with a prefix of MIN:

```
ISREDIT (MEMNAME) = MEMBER
IF &SUBSTR(1:3,&MEMNAME  ) = MIN THEN -
   ....
```

MODEL - Copy a Model into the Current Data Set

The MODEL command is used to copy a specified dialog development model before or after a specified line. If note mode is set on, note lines are also copied, but they cannot be directly referenced by a macro.

```
ISREDIT MODEL modelname {AFTER }  lptr  [NOTE  ]
                        {BEFORE}        [NONOTE]

ISREDIT MODEL CLASS classname
```

BEFORE or AFTER	indicates whether the model is to be copied before or after the line indicated by lptr.
modelname	indicates the name of the model to be copied. Refer to ISPF/PDF for MVS/XA Program Reference for a list of models and model names.
lptr	indicates a line pointer must be used to indicate where the model should be copied. A line pointer can be a label or a relative line number.
NOTE	indicates that explanatory notes will be displayed when a model is copied. NOTE is the default if neither NOTE nor NONOTE is specified.
NONOTE	indicates that no explanatory notes will be displayed.
classname	indicates the model class to be set for following MODEL command invocations.

The following return codes may be issued:

```
 0 - Normal completion
20 - Severe error
```

Example:

To copy the VGET model at the end of the current file:

```
ISREDIT MODEL VGET AFTER .ZL
```

MOVE - Move a Member into the Current File

The MOVE command is used to move a member of the partitioned data set being edited into the current member after the specified line number. The entire member is moved, then deleted. The AFTER or BEFORE keyword must be followed by a line pointer to indicate the insertion point; data will be copied after or before that line.

```
ISREDIT MOVE member {AFTER } lptr
                   {BEFORE}
```

BEFORE or AFTER indicates whether the member is to be moved before or after the target specified by lptr.

lptr indicates a line pointer must be used to identify the target of the move. A line pointer can be a label or a relative line number.

The following return codes may be issued:

```
 0 - Normal completion
 8 - End of file before last record read
12 - Invalid line pointer (lptr); member not found or BLDL error
16 - End of data before first record read
20 - Syntax error (invalid name, incomplete range)
   - I/O error
```

Examples:

To move the contents of member ABC after the first line in the current file:

```
ISREDIT MOVE ABC AFTER .ZF
```

To move the contents of member DEF before the line where the cursor is currently positioned:

```
ISREDIT MOVE DEF BEFORE .ZCSR
```

NOTE - Set or Query Note Mode

The NOTE assignment statement is used either to set the current note mode, or to retrieve the current setting of the note mode and place it in a variable. Note mode controls whether notes are to be displayed when a dialog development model is copied into the data. Refer to the MODEL command description.

```
ISREDIT (varname) = NOTE
ISREDIT NOTE = mode
ISREDIT NOTE mode
```

---

varname — is the name of a variable to contain the value of note mode, either ON or OFF.

mode — is the value that note mode can have, either ON or OFF.

- ON — indicates that explanatory notes will be displayed when a model is copied into the data being edited.
- OFF — indicates that no explanatory notes will be displayed.

The following return codes may be issued:

```
 0 - Normal completion
20 - Severe error
```

Examples:

To set note mode off:

```
ISREDIT NOTE = OFF
```

To store the value of note mode in variable &NOTEMODE:

```
ISREDIT (NOTEMODE) = NOTE
```

NULLS - Set or Query Nulls Mode

The NULLS assignment statement is used either to set the current nulls mode, or to retrieve the current setting of the nulls mode and place it in a variable. Nulls mode controls whether data fields on the screen are padded with blank or null characters.

```
ISREDIT (varname1,varname2) = NULLS
ISREDIT NULLS = mode
ISREDIT NULLS mode
```

---

varname1 is the name of a variable to contain either ON or OFF
varname2 is the name of a variable to contain ALL, STD, or blanks.
mode is the setting of nulls mode, ON, OFF, ON STD, or ON ALL:

- ON indicates that trailing blanks in each data field will be written as nulls.
- OFF indicates that trailing blanks in each data field will be written as blanks.
- ALL indicates that all trailing blanks and all-blank fields will be written as nulls.
- STD indicates that one trailing blank will be preserved.

The following return codes may be issued:

```
 0 - Normal completion
20 - Severe error
```

Examples:

To set nulls mode to ON STD:

```
ISREDIT NULLS = ON STD
```

To set nulls mode off:

```
ISREDIT NULLS = OFF
```

NUMBER - Set or Query Number Mode

The NUMBER assignment statement is used either to set the current number mode, or to retrieve the current setting of number mode and place it in variables. Number mode controls the numbering of lines in the current file.

```
ISREDIT (varname1,varname2) = NUMBER
ISREDIT NUMBER = mode
ISREDIT NUMBER mode
```

---

varname1 is the name of a variable to contain either ON or OFF.

varname2 is the name of a variable to contain STD, COBOL, or DISPLAY, or a combination.

mode is the setting of number mode, either ON or OFF and STD, COBOL, DISPLAY, or a combination:

- ON — automatically verifies that all lines have valid numbers in ascending sequence and renumbers any lines that are either unnumbered or out of sequence. ON is the default if no operands are coded.
- OFF — turns off number mode, indicating that the data does not contain sequence numbers.
- STD — when number mode is on, indicates that the data is numbered in the standard sequence field. If you omit both the STD and COBOL operands, the default is STD unless number mode was already on, in which case the data is numbered in whichever field(s) were previously specified.
- COBOL — when number mode is on, indicates that the data is numbered in the COBOL sequence field.
- STD COBOL — when number mode is on, indicates that the data is numbered in both the standard and the COBOL sequence number fields.
- DISPLAY — indicates that the width of the data window includes the sequence number fields.

The value STD, COBOL, or DISPLAY can be placed in varname2 even when varname1 is set to OFF. This allows the macro to save and restore number mode or set number mode off with defaults to be used when number mode is set on by the end user without specifying STD or COBOL.

The following return codes may be issued:

```
 0 - Normal completion
20 - Severe error
```

Example:

To save the current value of number mode, set number mode off for processing, and then restore the value of number mode:

```
ISREDIT (STAT,VALUE) = NUMBER
ISREDIT NUMBER OFF
        ...
ISREDIT NUMBER = (STAT,VALUE)
```

PACK - Set or Query Pack Mode

The PACK assignment statement is used either to set the current pack mode, or to retrieve the current setting of pack mode and place it in a variable. Pack mode controls whether the data is to be stored in packed format.

```
ISREDIT (varname) = PACK
ISREDIT PACK = mode
ISREDIT PACK mode
```

---

varname — is the name of a variable to contain the setting of pack mode, either ON or OFF.

mode — indicates the value of pack mode, either ON or OFF:

- ON — indicates that data will be saved in packed format.
- OFF — indicates that data will be saved in normal, unpacked format.

If this mode is changed, data will be written when an END command is issued.

The following return codes may be issued:

```
 0 - Normal completion
20 - Severe error
```

Example:

To set pack mode off:

```
ISREDIT PACK OFF
```

PROCESS - Process the Display Screen

The PROCESS command allows the macro to control when user-entered line commands or data changes are to be processed. If a line is retrieved before the PROCESS statement has been executed, changes made by the user on this interaction will not be seen. The DEST and RANGE keywords allow the macro to identify what line commands a user may enter as additional input to the macro.

Refer to "Controlling User Input" for a more complete description of this command.

```
ISREDIT PROCESS [DEST] [RANGE cmd1 [cmd2]]
```

DEST — indicates that the macro can capture an AFTER (A) or a BEFORE (B) line command entered by the user. The .ZDEST label is set to the line preceding the insertion point. If A or B is not entered, .ZDEST points to the last line in the file.

RANGE — must be followed by the names of one or two line commands, either of which the user may enter. This permits the macro to define and then capture a line command entered by the user. It can also modify its processing based on which of the two commands was entered.

When a choice may be made between two line commands, the RANGE_CMD assignment statement is used to return the value of the line command entered.

cmd1, cmd2 — indicates one or two line command names, which may be one to six characters, but if the name is six characters long it may not be used as a block format command. The name may contain any alphabetic or special character except blank, hyphen (-), or apostrophe ('). It may not contain any numeric character.

The .ZFRANGE label is set to the first line identified by the user-entered line command and .ZLRANGE is set to the last line. They may refer to the same line. If the expected RANGE line command was not entered, .ZFRANGE points to the first line in the file and .ZLRANGE points to the last line in the file.

The following return codes may be issued:

```
  0 - Normal completion
  4 - Range expected by macro but not entered by user; defaults set
  8 - Destination expected by macro but not entered by user; defaults
      set
 12 - Both range and destination expected by macro but neither range nor
      destination entered by user; defaults set
 16 - Incomplete or conflicting line commands entered by user
 20 - Severe error
```

Examples:

To set up the macro to process the line commands * and # (defined by the macro writer):

```
ISREDIT MACRO NOPROCESS
ISPEXEC CONTROL ERRORS RETURN
ISREDIT PROCESS RANGE * #
IF &LASTCC > = 16 THEN EXIT CODE(&LASTCC)
ISREDIT (CMD) = RANGE_CMD
ISREDIT (FIRST) = LINENUM .ZFRANGE
ISREDIT (LAST)  = LINENUM .ZLRANGE
IF &CMD = &STR(*) THEN -
   ...
```

To place data depending on the location of the A or B line command:

```
ISREDIT MACRO NOPROCESS
ISREDIT PROCESS DEST
ISREDIT LINE_AFTER .ZDEST = "&DATA"
```

To allow processing of the A and B destination line commands and the specification of a range using the * line command (defined by the macro writer):

```
ISREDIT MACRO NOPROCESS
ISREDIT PROCESS DEST RANGE *
```

Note: To specify this command, the MACRO command must have been coded with a NOPROCESS keyword.

PROFILE - Set or Query the Current Profile

PROFILE allows the user to view or switch the default modes in the current edit session. PROFILE has two forms: a command and an assignment statement. The profile name cannot be set by an assignment statement. Use the PROFILE command to change the profile name, thereby changing the current edit profile and the edit profile values.

```
ISREDIT (varname1,varname2) = PROFILE
ISREDIT PROFILE name [number]
ISREDIT PROFILE number
ISREDIT PROFILE LOCK
ISREDIT PROFILE UNLOCK
```

---

varname1 is the name of a variable to contain the name of the current profile.

varname2 is the name of a variable to contain the profile status, LOCK or UNLOCK.

name indicates the profile name. The edit profile table is searched for an existing entry with the same name. That profile is then read from disk and used. If one is not found, a new entry is created in the profile table.

number indicates the number of lines, from 0 through 8, of profile data to be displayed. When you code the number as 0, no profile data is displayed. When you code no number, the profile modes are displayed; the mask and tabs lines are displayed if they contain data.

LOCK indicates that the current values in the profile are saved in the edit profile table and are not modified on disk until the profile is unlocked. The current copy of the profile can be changed, either as a result of user commands that modify profile values (BOUNDS and NUMBER, for example) or as a result of differences in the data from the current profile settings. Caps, number, stats, or pack mode are automatically changed to fit the data. These changes occur when the file is first read or when data is copied into the file.

Note: To force caps, number, stats, or pack mode to a particular setting, use an initial macro. Be aware, however, that if you set number mode on, data may be overlaid.

UNLOCK indicates that changes to profile values are automatically saved on disk.

The following return codes may be issued:

```
 0 - Normal completion
20 - Severe error
```

Example:

To check the lock status of the profile and perform processing if the profile is locked:

```
ISREDIT (,STATUS) = PROFILE
IF &STATUS = LOCK THEN -
  ...
```

RANGE_CMD - Query the Command Entered by the User

The RANGE_CMD assignment statement is used to identify the name of a line command entered by the end user and processed by this macro. The macro first issues a PROCESS command to identify line commands to be processed by this macro. The RANGE_CMD statement returns either a Q or $ when the following PROCESS command has been issued by the macro:

```
PROCESS RANGE Q $
```

If the end user enters Q5, just Q is returned.

```
ISREDIT (varname) = RANGE_CMD
```

---

varname is the name of a variable to contain the line command entered by the user.

The following return codes may be issued:

```
 0 - Normal completion
 4 - Line command not set
 8 - Line command setting not acceptable
20 - Severe error
```

Example:

To determine which line command (* or #) the user entered and to process the line command (defined by the macro writer):

```
ISREDIT MACRO NOPROCESS
ISREDIT PROCESS RANGE * #
ISREDIT (CMD) = RANGE_CMD
IF &CMD = &STR(*) THEN -
        .....
ELSE IF &CMD = &STR(#) THEN -
        .....
```

RCHANGE - Repeat a Change

The RCHANGE command repeats the change requested by the most recent CHANGE command. It is recommended that the CHANGE command be reissued for clarity.

The RCHANGE command can be used repeatedly to change other occurrences of the search string. After a "string not found" condition is encountered, the next RCHANGE issued starts at the first line of the current range for a forward search (FIRST or NEXT specified) or the last line of the current range for a backward search (LAST or PREV specified).

```
ISREDIT RCHANGE
```

The following return codes may be issued:

```
 0 - Normal completion
 4 - String not found
 8 - Change error ('to' string longer than 'from' string and
     substitution was not performed on at least one change)
12 - Syntax error
20 - Severe error (no previous change)
```

Example:

To perform a single-line change and then repeat it if there are more lines to be searched:

```
ISREDIT CHANGE C'.  the' C'.  The' 1 8
IF &LASTLN ¬= &END THEN -
  ISREDIT RCHANGE
```

RECFM - Query the Record Format

The RECFM assignment statement is used to retrieve the record format of the data set being edited and to place the value in a variable.

```
ISREDIT (varname) = RECFM
```

---

varname is the name of a variable to contain the record format of the data set being edited, either F or V:
F indicates fixed-length records.
V indicates variable-length records.

The following return codes may be issued:

```
 0 - Normal completion
20 - Severe error
```

Example:

To place the record format in variable &RECFORM and then use the logical data width, if the file is fixed, or the right display column, if the file is variable:

```
ISREDIT (RECFORM) = RECFM
IF &RECFORM = F THEN -
  ISREDIT (WIDTH) = DATA_WIDTH
ELSE -
  ISREDIT (,WIDTH) = DISPLAY_COLS
```

RECOVERY - Set or Query Recovery Mode

The RECOVERY assignment statement is used either to set the current value of recovery mode, or to retrieve the current value and place it in a variable.

```
ISREDIT (varname) = RECOVERY
ISREDIT RECOVERY = mode
ISREDIT RECOVERY mode
```

---

varname is the name of a variable to contain the setting of recovery mode, either ON or OFF.

mode is the setting of recovery mode, either ON or OFF:

- ON indicates that a recovery file is created when the first change is made to the data and updated for each change thereafter.
- OFF indicates that no recovery file is created.

The following return codes may be issued:

```
 0 - Normal completion
20 - Severe error
```

Examples:

To save the value of recovery mode in variable &RECOV:

```
ISREDIT (RECOV) = RECOVERY
```

To set recovery mode off:

```
ISREDIT RECOVERY = OFF
```

RENUM - Resequence and Number the Data

The RENUM command sets number mode on and renumbers all data lines.

```
ISREDIT RENUM [STD] [COBOL] [DISPLAY]
```

The STD, COBOL, and DISPLAY keywords are the same as those used on the NUMBER command.

The following return codes may be issued:

```
 0 - Normal completion
20 - Severe error
```

Example:

To renumber all data lines with COBOL numbering:

```
ISREDIT RENUM COBOL
```

REPLACE - Replace a Member

The REPLACE command is used to replace the specified member in the library currently being edited with the data in the current member as limited by the range of line pointers.

```
ISREDIT REPLACE member lptr-range
```

---

member indicates the member name of the member in the current library to be replaced. If the member does not exist, it is created.

lptr-range indicates that two line pointers are required to indicate a range of lines. A line pointer may be a label or a relative line number. Specifying one line pointer is invalid.

The following return codes may be issued:

```
 0 - Normal completion
12 - Invalid line pointer; member not found or BLDL error
20 - Syntax error (invalid name, incomplete line pointer value)
   - I/O error
```

Example:

To replace member MEM1 with the first 10 lines of the current file:

```
ISREDIT REPLACE MEM1 1 10
```

RESET - Reset Lines

The RESET command is used to reset the status of lines or delete special (non-data) lines. If all operands are omitted, the RESET command resets everything except labels.

Note: The LOCATE command positions the first line on the screen with the same generic line identifiers.

```
ISREDIT RESET lineid [lptr-range]
```

---

lineid — indicates a generic line identifier, one or more of the following:

- LABEL — to delete labels from any line with a label
- CHANGE — to delete ==CHG> flag from the line command area of any lines flagged as a result of a CHANGE ALL command
- ERROR — to delete ==ERR> flag from the line command area of any line flagged as a result of a CHANGE or SHIFT data command that failed because data would not fit on a line
- SPECIAL — to delete from the display any special or non-data line:
  - Profile lines flagged as =PROF>
  - Mask lines flagged as =MASK>
  - Bounds line flagged as =BNDS>
  - Tabs line flagged as =TABS>
  - Message lines flagged as ==MSG>
  - Note lines flagged with =NOTE=
- EXCLUDED — to show any excluded line
- COMMAND — to delete any pending line commands from the line command field

lptr-range — indicates that two line pointers are required to indicate a range of lines for the reset operation. A line pointer may be a label or a relative line number. Specifying one line pointer is invalid.

The following return codes may be issued:

```
 0 - Normal completion
20 - Severe error
```

Example:

To reset all special lines in the current file:

```
ISREDIT RESET SPECIAL
```

RFIND - Repeat Find

The RFIND command finds the search string defined by the most recent SEEK, FIND, or CHANGE command, or excludes a line containing the search string defined by the previous EXCLUDE command.

The RFIND command can be used repeatedly to find other occurrences of the search string. After a "string not found" condition is encountered, the next RFIND issued starts at the first line of the current range for a forward search (FIRST or NEXT specified) or the last line of the current range for a backward search (LAST or PREV specified).

```
ISREDIT RFIND
```

The following return codes may be issued:

```
 0 - Normal completion
 4 - String not found
12 - Syntax error
20 - Severe error (string not defined)
```

Example:

To find a character string, process it, and repeat the find operation if there are more lines:

```
ISREDIT FIND NEXT C'.  the'
    .....(process the line)....
IF &LASTLN ¬= &END THEN -
   ISREDIT RFIND
```

RIGHT - Scroll Right

The RIGHT command causes a scroll right from the current screen position. The current setting of the boundaries may affect the amount actually scrolled. To scroll right using the screen position when the macro was issued, use USER_STATE assignment statements to save and then restore the screen position parameters.

If you define a macro named RIGHT, it overrides the RIGHT command when used from another macro, but has no effect for the end user. The RIGHT command does not change the cursor position and cannot be used in an initial macro. For further information, see the BOUNDS and DISPLAY_COLUMNS descriptions.

```
ISREDIT RIGHT amt
```

---

amt indicates the scroll amount, the number of columns or one of the following keywords: MAX, HALF, PAGE, CURSOR, or DATA.

The following return codes may be issued:

```
 0 - Normal completion
 4 - No visible lines
 8 - No data to display
12 - Amount not coded
20 - Severe error
```

Example:

To scroll the display right to put the column specified in variable &RCOL in column 1:

```
ISREDIT RIGHT &RCOL
```

Notes:

1. The first line that would be displayed is determined by:
   a. Whether the cursor was set explicitly by a CURSOR assignment statement or implicitly by a SEEK, FIND, CHANGE, or TSPLIT command. Since the cursor must be on the screen, the line that is the first line on the screen may be different from the line that was first when the user invoked the macro.
   b. A LOCATE command setting the line to be first on the screen.
2. The number of lines scrolled are affected by non-data lines on the screen.
3. The number of lines on the screen is determined by:
   a. The number of lines excluded from the display.

b. The number of non-data lines displayed: profile, message, note, bounds, tabs, or mask lines.
c. The terminal display size and the split-screen line.

RMACRO - Set or Query the Recovery Macro

The RMACRO assignment statement is used to set or retrieve the name of the macro set in this edit session. This macro is to be executed after the file has been recovered but before the edit data panel is displayed for editing. Commands that refer to display values are invalid in a recovery macro.

Recovery may occur if this edit session terminates abnormally, when a SAVE, END, or CANCEL command has not been issued, and when recovery mode is on.

```
ISREDIT (varname) = RMACRO
ISREDIT RMACRO = name
ISREDIT RMACRO name
```

---

varname — is the name of a variable to contain the name of the recovery macro.

name — indicates the name of the macro to be executed after a file has been recovered. The name NONE can be used to prevent a recovery macro from being used; conversely, a value of NONE is returned when no recovery macro has been specified.

The following return codes may be issued:

```
 0 - Normal completion
12 - Invalid name specified
20 - Severe error
```

Example:

To set the RMACRO name from the variable &RMAC:

```
ISREDIT RMACRO = &RMAC
```

SAVE - Save the Current Data on Disk

The SAVE command saves the current data on disk. See the DATA_CHANGED, AUTOSAVE, CANCEL, and END commands, which are interrelated, for further information about saving data.

```
ISREDIT SAVE
```

The following return codes may be issued:

```
 0 - Normal completion
12 - Data not saved; not enough PDS space or directory space
20 - Severe error
```

Example:

To check autosave mode and, if off, ensure that changes are saved:

```
ISREDIT (VAR) = AUTOSAVE
IF &VAR = OFF THEN -
  ISREDIT SAVE
```

SCAN - Set Command Scan Mode

The SCAN assignment statement is used either to set the current value of scan mode (for variable substitution), or to retrieve the current value of scan mode and place it in a variable. Scan mode controls the automatic replacement of variables in command lines passed to edit.

SCAN is valid only within a macro.

For further information, refer to "Variable Substitution" on page 61.

```
ISREDIT (varname) = mode
ISREDIT SCAN = mode
ISREDIT SCAN  [mode]
```

---

varname is the name of a variable to contain the setting of scan mode, either ON or OFF:

mode is the setting of scan mode, either ON or OFF:

- ON indicates that edit automatically replaces variables in command lines passed to edit.
- OFF indicates that edit does not automatically replace variables.

If mode is omitted, the default is ON. Scan mode is initialized to ON when a macro is invoked.

The following return codes may be issued:

```
 0 - Normal completion
20 - Severe error
```

Example:

To set the line whose number is in variable &LNUM to "&SYSDATE is a CLIST built-in function", set scan mode off and issue the LINE command with the CLIST function name coded as &&SYSDATE. The CLIST processor strips off the first &, but, because scan mode is off, edit does not remove the second &:

```
ISREDIT SCAN OFF
ISREDIT LINE &LNUM = "&&SYSDATE is a CLIST built-in function"
ISREDIT SCAN ON
```

SEEK - Seek a Data String, Positioning the Cursor

The SEEK command is used to find one or more occurrences of a data string. The SEEK command is exactly like the FIND command except that when a string is found, the exclude status of the line is not affected. The SEEK command is provided so that the cursor can be positioned without affecting the end user's view of the screen. Therefore, SEEK should be used in place of FIND within macros.

```
ISREDIT SEEK str-1 [label-range] [NEXT ] [CHARS ] [X ] [col-1 [col-2]]
                                 [ALL  ] [PREFIX] [NX]
                                 [FIRST] [SUFFIX]
                                 [LAST ] [WORD  ]
                                 [PREV ]
```

---

str-1 indicates the string to be located.

label-range indicates that two labels are required to indicate a range of lines; one label is invalid. If relative line numbers are coded, they are interpreted as column numbers, or a syntax error is detected.

args For complete information about the following SEEK arguments, see the ISPF/PDF for MVS/XA Program Reference:

NEXT	CHAR	X
ALL	PREFIX	NX
FIRST	SUFFIX	
LAST	WORD	
PREV		

When the cursor is within the line range, the search for the string starts at the current cursor position for a SEEK NEXT or SEEK PREV request; when the cursor is outside the range, the search begins at the top (for NEXT) or bottom (for PREV) of the range. If str-1 is found, the cursor position is changed to the line and column position of the first character of the first string found. If a string is not found, the cursor is not moved.

The boundary settings limit the columns searched to the columns within the boundary columns. When writing a general macro, you may want to override the boundary setting by supplying specific columns.

The following return codes may be issued:

```
  0 - Normal completion
  4 - String not found
 12 - Syntax error
 20 - Severe error
```

Examples:

To position the cursor to the first 'abc':

```
ISREDIT SEEK FIRST 'abc'
```

To position the cursor to the previous numeric field:

```
ISREDIT SEEK PREV P'###'
```

SEEK_COUNTS - Query Seek Counts

The SEEK_COUNTS assignment statement is used to retrieve values that were set by the most recently executed SEEK command and to place them in variables.

```
ISREDIT (varname1,varname2) = SEEK_COUNTS
```

---

varname1 is the name of a variable to contain the number of strings found, an 8-character value, left-padded with zeros.
varname2 is the name of a variable to contain the number of lines on which strings were found, an 8-character value, left-padded with zeros.

The following return codes may be set:

```
 0 - Normal completion
20 - Severe error
```

Example:

To seek all lines with a blank in column 1 and store the number of such lines in variable &BLNKS:

```
ISREDIT SEEK ALL " " 1
ISREDIT (BLNKS) = SEEK_COUNTS
```

SHIFT (- Shift Columns Left

The SHIFT (command is used to shift columns of data left, in the same manner that the (edit line command shifts data. The current setting of the boundaries affects the columns within which data is shifted.

The SHIFT (command is limited to shifting data on a single line. If you want to shift data on several lines, you have to loop through the lines, shifting data on each line individually.

```
ISREDIT SHIFT ( lptr [n]
                     [2]
```

---

lptr — indicates a line pointer must be used. A line pointer can be a label or a relative line number.

n — indicates the number of columns to shift. If this operand is omitted, the default is two columns.

The following return codes may be issued:

```
 0 - Normal completion
12 - invalid line number
20 - Severe error
```

Examples:

To shift columns of data on the line containing the cursor to the left 10 columns:

```
ISREDIT SHIFT ( .ZCSR 10
```

To shift columns of data on the line with the label .LAB to the left 2 columns:

```
ISREDIT SHIFT ( .LAB
```

SHIFT) - Shift Columns Right

The SHIFT) command is used to shift columns of data right, in the same manner that the) edit line command shifts data. The current setting of the boundaries affects the columns within which data is shifted.

The SHIFT) command is limited to shifting data on a single line. If you want to shift data on several lines, you have to loop through the lines, shifting data on each line individually.

```
ISREDIT SHIFT ) lptr [n]
                     [2]
```

---

lptr — indicates a line pointer must be used. A line pointer can be a label or a relative line number.

n — indicates the number of columns to shift. If this operand is omitted, the default is two columns.

The following return codes may be issued:

```
 0 - Normal completion
12 - invalid line number
20 - Severe error
```

Examples:

To shift columns of data on the line containing the cursor to the right 4 columns:

```
ISREDIT SHIFT ) .ZCSR 4
```

To shift columns of data on the line with the label .LAB to the right 2 columns:

```
ISREDIT SHIFT ) .LAB
```

SHIFT < - Shift Data Left

The SHIFT < command is used to shift data left, in the same manner that the < edit line command shifts data. The current setting of the boundaries affects the columns within which data is shifted.

The SHIFT < command is limited to shifting data on a single line. If you want to shift data on several lines, you have to loop through the lines, shifting data on each line individually.

If the data cannot be shifted, the line is flagged in the line command area with ==ERR>. You can use the LOCATE and RESET commands, which are sensitive to this flag.

```
ISREDIT SHIFT < lptr [n]
                     [2]
```

---

lptr — indicates a line pointer must be used. A line pointer can be a label or a relative line number.

n — indicates the number of columns to shift. If this operand is omitted, the default is two columns.

The following return codes may be issued:

```
 0 - Normal completion
12 - invalid line number
20 - Severe error
```

Examples:

To shift data on the line containing the cursor to the left 4 columns:

```
ISREDIT SHIFT < .ZCSR 4
```

To shift data on the line with the label .LAB to the left 2 columns:

```
ISREDIT SHIFT < .LAB
```

SHIFT > - Shift Data Right

The SHIFT > command is used to shift data right, in the same manner that the > edit line command shifts data. The current setting of the boundaries affects the columns within which data is shifted.

The SHIFT > command is limited to shifting data on a single line. If you want to shift data on several lines, you have to loop through the lines, shifting data on each line individually.

If the data cannot be shifted, the line is flagged in the line command area with ==ERR>. You can use the LOCATE and RESET commands, which are sensitive to this flag.

```
ISREDIT SHIFT > lptr [n]
                     [2]
```

---

lptr — indicates a line pointer must be used. A line pointer can be a label or a relative line number.

n — indicates the number of columns to shift. If this operand is omitted, the default is two columns.

The following return codes may be issued:

```
 0 - Normal completion
12 - invalid line number
20 - Severe error
```

Examples:

To shift data on the line containing the cursor to the right 4 columns:

```
ISREDIT SHIFT > .ZCSR 4
```

To shift data on the line with the label .LAB to the right 2 columns:

```
ISREDIT SHIFT > .LAB
```

SORT - Sort Data

The SORT command is used to put data in a specified order.

The data columns to be sorted are defined by the current boundaries. The columns to be used for sort keys are specified as arguments, identical to those specified by the end user.

```
ISREDIT SORT [X ] [sortfield-1 [sortfield-2, ...]] [label-range]
             [NX]
```

X	indicates that only excluded lines are to be sorted.
NX	indicates that only non-excluded lines are to be sorted.
sortfield-1, 2, ...	indicates the fields that define the sort operation. For further information about sort fields, refer to ISPF/PDF for MVS/XA Program Reference.
label-range	indicates that two labels are required to indicate a range of lines for the sort operation; one label is invalid.

The following return codes may be issued:

```
 0 - Normal completion
 4 - Lines were already in sort order
20 - Severe error
```

Examples:

To sort the entire file in ascending order:

```
ISREDIT SORT
```

To sort the file in descending order, using the sort key in columns 15 through 20:

```
ISREDIT SORT D 15 20
```

To sort all excluded lines in ascending order:

```
ISREDIT SORT X A
```

To sort lines between labels .A and .B in ascending order:

```
ISREDIT SORT .A .B
```

To sort lines between labels .A and .B, using the sort key in column 5:

```
ISREDIT SORT .A .B  5
```

STATS - Set or Query Stats Mode

The STATS assignment statement is used either to set the current stats mode or to retrieve the current setting of the stats mode and place it in a variable.

```
ISREDIT (varname) = STATS
ISREDIT STATS = mode
ISREDIT STATS mode
```

---

varname is the name of a variable to contain the setting of stats mode, either ON or OFF.

mode is the setting of stats mode, either ON or OFF:

- ON indicates that library statistics will be created or updated when the data is saved.
- OFF indicates that no library statistics are to be created or saved.

The following return codes may be issued:

```
 0 - Normal completion
20 - Severe error
```

Examples:

To put the value of stats mode in variable &LIBSTAT:

```
ISREDIT (LIBSTAT) = STATS
```

To set stats mode on:

```
ISREDIT STATS = ON
```

To set stats mode off:

```
ISREDIT STATS OFF
```

To reset stats mode from the mode saved in variable &LIBSTAT:

```
ISREDIT STATS = (LIBSTAT)
```

SUBMIT - Submit a Job for Batch Execution

The SUBMIT command is used to submit the current data set, or that part of the data set defined by the range of line pointers, to be executed as a batch job.

```
ISREDIT SUBMIT [lptr-range]
```

---

lptr-range indicates that two line pointers are required to indicate a range of lines. A line pointer may be a label or a relative line number. Specifying one line pointer is invalid.

The following return codes may be issued:

```
 0 - Normal completion
20 - Severe error (submit failed)
```

Examples:

To submit the first 20 lines of the JCL file:

```
ISREDIT SUBMIT 1 20
```

To submit the entire file of JCL:

```
ISREDIT SUBMIT
```

TABS - Set or Query Tabs Mode

The TABS assignment statement is used either to set the current tabs mode or to retrieve the current setting of the tabs mode and place it in a variable.

```
ISREDIT (varname1,varname2) = TABS
ISREDIT TABS = mode [tabchar] [ALL|STD]
ISREDIT TABS   mode [tabchar] [ALL|STD]
```

---

varname1 is the name of a variable to contain the setting of tabs mode, either ON or OFF.

varname2 is the name of a variable to contain the tab character and either ALL or STD. This variable may be blank.

mode is the setting of tabs mode, either ON or OFF:

- ON indicates that logical tabs can be used to break up strings of data.
- OFF indicates that logical tabs cannot be used.

tabchar indicates the character to be defined as the logical tab character.

ALL indicates that nonblank characters are not replaced with hardware attribute bytes.

STD indicates that all characters, including nonblank characters, are replaced with hardware attribute bytes, thus overlaying data.

The following return codes may be issued:

```
 0 - Normal completion
20 - Severe error
```

Examples:

To store the setting of tabs mode in variable &TABVAL:

```
ISREDIT (TABVAL) = TABS
```

To set tabs mode on and set the tab character to \:

```
ISREDIT TABS ON \
```

To set the value of tabs mode from variable &TABVAL:

```
ISREDIT TABS = (TABVAL)
```

TABSLINE - Set or Query Tabs Line

The TABSLINE assignment statement is used either to set the current tabs line or to retrieve the current tabs line and place it in a variable.

```
ISREDIT (varname) = TABSLINE
ISREDIT TABSLINE = data
```

---

varname — is the name of a variable to contain the contents of the current tabs line.

data — indicates the data used to set the tabs line. The only valid characters for this data are blanks and valid tab characters: asterisk (*), hyphen (-), or underscore (_). The following forms may be used:

- simple string
- delimited string
- variable
- keyword:

LINE lptr	Data from the line with the given lptr
MASKLINE	Data from the mask line
TABSLINE	Data from the tabs line

- template (< col,string >)
- merge format (string-1 + string-2, keyword + string-2, string-1 + keyword)

The following return codes may be issued:

```
 0 - Normal completion
 4 - Data truncated
 8 - Invalid data detected and ignored
20 - Severe error (invalid input)
```

Examples:

To store the value of the tabs line in variable &OLDTABS:

```
ISREDIT (OLDTABS) = TABSLINE
```

To set the tabs line to '*___* *':

```
ISREDIT TABSLINE = '*___*  *'
```

To clear the tabs line:

```
ISREDIT TABSLINE = ' '
```

To set tabs in columns 1 and 35:

```
ISREDIT TABSLINE = <1,*,35,*>
```

To add a tab in column 36:

```
ISREDIT TABSLINE = TABSLINE + <36,*>
```

To remove a tab from column 36:

```
ISREDIT TABSLINE = TABSLINE + <36,' '>
```

TENTER - Set Up Display Screen for Text Entry

The TENTER command performs the function of the TE edit line command for the end user, preparing the screen to allow power typing. While in text entry mode, no macros may be executed. Text entry is affected by the boundary settings.

```
ISREDIT TENTER lptr [n]
```

---

lptr — indicates a line pointer must be used. A line pointer can be a label or a relative line number.

n — indicates the number of lines displayed for text entry. If n is not specified, the remainder of the screen is displayed for text entry.

The following return codes may be issued:

```
 0 - Normal completion
12 - Invalid line number
20 - Severe error
```

Example:

To find the last line in the file and set up the display for text entry following the last line:

```
ISREDIT LOCATE .ZL
ISREDIT TENTER .ZL
```

Note: You must ensure that the line pointer referenced on the TENTER command is displayed; otherwise, the text area will not be visible to the end user. The LOCATE command causes a line to be placed at the top of the display.

TFLOW - Text Flow a Paragraph

The TFLOW command performs the functions of the TF line command for the end user, restructuring paragraphs to smooth display line endings. Data is flowed starting at the specified line to the specified column within the current right and left boundaries. Data is flowed until a paragraph end is reached. A paragraph end is indicated by a blank line, a change in indentation, or a special character (period (.), colon (:), or ampersand (&)) in column 1.

```
ISREDIT TFLOW lptr [col]
```

---

lptr — indicates a line pointer must be used. A line pointer can be a label or a relative line number.

col — indicates the column to which the text should be flowed. If the column number is omitted, it defaults to the right boundary, unlike the TF line command, which defaults to the screen width when default boundaries are in effect.

The following return codes may be issued:

```
 0 - Normal completion
12 - Invalid line number
20 - Severe error
```

Example:

To limit the flowing of text to the displayed columns:

```
ISREDIT (,RCOL) = DISPLAY_COLS
ISREDIT TFLOW .PP &RCOL
```

TSPLIT - Text Split a Line

The TSPLIT command performs the functions of a TS line command for the end user, splitting a line so that data can be added. It is affected by the current setting of the boundaries; data beyond the right boundary is not moved to the line added by TSPLIT and the data between the split column and the right boundary is moved to the new line starting at the left boundary. The cursor position is set to the split point.

```
ISREDIT TSPLIT [lptr col]
```

---

lptr — indicates a line pointer is used to identify the line where the split is to occur. A line pointer can be a label or a relative line number.

col — indicates the column at which the text is to be split.

If no operands are coded, the split point is assumed to be the current cursor position.

The following return codes may be issued:

```
 0 - Normal completion
12 - Invalid line number
20 - Severe error
```

Example:

To split the line labeled .TOP at column 15:

```
ISREDIT (LINENBR) = LINENUM .TOP
ISREDIT TSPLIT &LINENBR 15
```

UNNUM - Unnumber the Current File

The UNNUM command removes the numbers from the current file and turns number mode off.

```
ISREDIT UNNUM
```

The following return codes may be issued:

```
 0 - Normal completion
12 - Number mode not on
20 - Severe error
```

Example:

To remove the line numbers from the current file and turn number mode off:

```
ISREDIT UNNUM
```

UP - Scroll Up

The UP command causes a scroll up from the current screen position. To scroll up using the screen position when the macro was issued, use USER_STATE assignment statements to save and then restore the screen position parameters.

If you define a macro named UP, it overrides the UP command when used from another macro, but has no effect for the end user. The UP command does not change the cursor position and cannot be used in an initial macro.

```
ISREDIT UP amt
```

---

amt indicates the scroll amount, the number of lines or one of the following keywords: MAX, HALF, PAGE, CURSOR, or DATA.

The following return codes may be issued:

```
 0 - Normal completion
 4 - No visible lines
 8 - No data to display
12 - Amount not coded
20 - Severe error
```

Example:

To make the line where the cursor is placed the first one on the display:

```
ISREDIT UP CURSOR
```

Notes:

1. The first line that would be displayed is determined by:
   a. Whether the cursor was set explicitly by a CURSOR assignment statement or implicitly by a SEEK, FIND, CHANGE, or TSPLIT command. Since the cursor must be on the screen, the line that is the first line on the screen may be different from the line that was first when the user invoked the macro.
   b. A LOCATE command setting the line to be first on the screen.
2. The number of lines scrolled are affected by non-data lines on the screen.
3. The number of lines on the screen is determined by:
   a. The number of lines excluded from the display.
   b. The number of non-data lines displayed: profile, message, note, bounds, tabs, or mask lines.
   c. The terminal display size and the split-screen line.

USER_STATE - Save or Restore User State

The USER_STATE assignment statement is used to save or restore the state of edit profile values, FIND and CHANGE values, and screen and cursor values. USER_STATE can be used at the beginning of a macro to save conditions and at the end of a macro to restore the conditions which may have changed during the execution. Many of the values saved by USER_STATE can be saved and restored individually. The USER_STATE assignment statement is a simple way of saving a large number of values with a single statement.

```
ISREDIT (varname) = USER_STATE
ISREDIT USER_STATE = (varname)
```

---

varname is the name of a variable to contain the user status information.

The following return codes may be issued:

```
 0 - Normal completion
20 - Severe error
```

Examples:

To save the user state in variable &STATUS:

```
ISREDIT (STATUS) = USER_STATE
```

To restore the user state from variable &STATUS:

```
ISREDIT USER_STATE = (STATUS)
```

Notes:

1. The following edit modes and values are saved and restored by USER_STATE:

AUTOLIST	HEX	NUMBER	TABSLINE
AUTONUM	IMACRO	PACK	
AUTOSAVE	MASKLINE	PROFILE	
BOUNDS	MODEL CLASS	RECOVERY	
CAPS	NOTE	STATS	
CURSOR	NULLS	TABS	

   current FIND/CHANGE/SEEK/EXCLUDE parameters

2. The information in the variable is saved in an internal format that is subject to change. Dependence on the format may lead to macro errors.

VERSION - Set or Query Version Number

The VERSION assignment statement is used either to set the current version number or to retrieve the current version number and place it in a variable.

Version numbers may be used to identify classes of members. For example, members associated with the first release of a product could be set to version 1 and members associated with the second release of a product could be set to version 2, while members not associated with any release could be set to version 99.

```
ISREDIT (varname) = VERSION
ISREDIT VERSION = num
ISREDIT VERSION num
```

---

varname is the name of a variable to contain the version number, a 3-character value, left-padded with zeros.
num is the version number, any number from 1 to 99.

The following return codes may be issued:

```
 0 - Normal completion
 8 - Stats mode is off, the command is ignored
12 - Invalid value specified (the version must be 1 to 99)
20 - Severe error
```

Examples:

To save the version number in variable &VERS:

```
ISREDIT (VERS) = VERSION
```

To set the version number to 1:

```
ISREDIT VERSION 1
```

To set the version number from variable &VERS:

```
ISREDIT VERSION = &VERS
```

XSTATUS - Set or Query Exclude Status of a Line

The XSTATUS assignment statement is used either to set the exclude status of the specified data line or to retrieve the exclude status of the specified data line and place it in a variable. Exclude status determines whether or not the line is excluded.

```
ISREDIT (varname) = XSTATUS lptr
ISREDIT XSTATUS lptr = X | NX
```

---

varname	is the name of a variable to contain the exclude status, either X or NX.
lptr	indicates a line pointer must be used. A line pointer can be a label or a relative line number.
X	indicates that the specified line is to be excluded.
NX	indicates that the specified line is to be shown (non-excluded).

The following return codes may be set:

- 0 - Normal completion
- 8 - An attempt to set a line status to 'NX' could not be performed. (The line had an interpreted but unexecuted line command in it). For example, if an excluded line is overtyped with the M command "MOVE/COPY IS PENDING" and the line(s) cannot be shown.
- 12 - Line number is not an existing line
- 20 - Severe error

Examples:

To store the exclude status of the line whose number is in variable &N in variable &LINEX:

```
ISREDIT (LINEX) = XSTATUS &N
```

To exclude line 1:

```
ISREDIT XSTATUS 1 = X
```

To locate a string and change it, saving and then restoring the exclude status:

```
ISREDIT SEEK &DATA
IF &LASTCC = 0 THEN -
  DO
    ISREDIT (XLINE) = XSTATUS .ZCSR
    ISREDIT CHANGE &DATA &NEWDATA .ZCSR .ZCSR
    ISREDIT XSTATUS .ZCSR = (XLINE)
  END
```

Note: To exclude a number of lines at one time, use the EXCLUDE command. To show a number of lines at one time, use the FIND command. Use XSTATUS together with SEEK and CHANGE to preserve the exclude status of a line.

APPENDIX A. CLIST CONSIDERATIONS

If you are not an experienced CLIST writer, several considerations are likely to cause problems initially. You may want to note the following items, which frequently are the cause of errors for new users. For additional information about writing CLISTs, refer to TSO Command Language Reference and TSO Terminal User's Guide.

- A '-' (hyphen) or a '+' (plus sign) at the end of a line serves as a continuation character. In Figure 2 on page 4, the IF statement and the ELSE statement both have a hyphen (-) following the comment. The plus sign deletes leading blanks from the following line; a minus sign leaves the leading blanks when continuing the line. The plus sign is preferred when dealing with data in which blanks are not wanted; the minus sign preserves the indentation of programming statements for debugging purposes.

  Three common errors occur with continuation characters. See if you can spot the errors in the examples that follow:

  Example

```
SET &A = 1
IF &A ¬= 1 THEN
  DO
    WRITE   THIS LINE SHOULD NEVER BE WRITTEN.
  END
```

  In the above example, the line that should never be written is in fact written, because the continuation character on the IF ... THEN statement was omitted. The IF statement could be interpreted as:

  "IF the value of &A is not equal to 1 THEN do nothing"

  Example

```
SET &A = 1
WRITE ---- TITLE LINE ----
SET &A = 9
```

  In the above example, because the final '-' (hyphen) on the WRITE statement was taken as a continuation character, the CLIST writes:

  "---- TITLE LINE ---   SET 1 = 9"

Example

```
SET &A = 1
IF &A ¬= 1 THEN -          /* test for the end of the loop */
DO
   WRITE   THIS LINE SHOULD NEVER BE WRITTEN.
END
```

Once again, the line that should never be written is in fact written. The '-' following the word THEN is not treated as a continuation character, because it is not the last nonblank character on the line. This type of error can occur if comments are added after the original statements are written.

- A symbolic variable must be given a value before it can be used in a statement. The variable can be set with a SET statement or with an edit assignment statement. For example:

  ```
  SET &NAME = CLEANUP
  ISREDIT (NAME) = MEMBER
  SET &NAME = &STR(        )
  SET &NAME = &STR()
  ```

  are all valid ways to set the variable NAME. The first sets &NAME to the word CLEANUP. The second sets &NAME to the name of the member that is being edited. The third sets &NAME to a string of eight blanks. The fourth sets &NAME to a null string.

  Once set, &NAME can be included in a statement, and the value assigned to &NAME is substituted before the statement is executed. You may use a variable set to a null value, but it is an error if you use a variable that has never been set.

  If you attempt to execute a statement that contains a variable that has never been set, a CLIST error message is displayed, and the CLIST is abnormally terminated.

- CLIST statements must be written in uppercase. Lowercase letters in statement names, variables, or operators will cause unpredictable results. Of course, quoted string variables and operands on edit macro statements can be uppercase or lowercase, since they are passed to the PDF editor for processing.

- To check the return code of a command, you must check the CLIST control variable &LASTCC in the very next statement following the command. Its value is reset at the end of each command, including CLIST WRITE, IF, and DO statements. You should capture its value in another variable, using a statement like SET &RETCODE = &LASTCC. Then the variable can be tested where appropriate to the logic.

- When a variable that contains leading and trailing blanks is used in a CLIST comparison, the blanks are ignored.

- Use "&STR(" before data that may contain arithmetic or logical operators (= + - * / // ¬= <= ...) to suppress evaluation of the expression. To determine whether a variable is set to nulls:

```
IF &STR(&VAR) = &STR() THEN ...
```

- Remember that the CLIST processor reads and interprets each statement in the CLIST before the statement is passed to the editor for macro processing. CLIST control statements such as SET, IF, and DO are not passed to the editor so those statements cannot contain any macro language syntax. For example, a CLIST error results if a SET statement contains a label.

- To parse a statement into "words" (character strings, quoted strings ('), parenthesized strings, or a null string indicated by two adjacent commas (,,)), you can do the following:

```
SET &SYSDVAL = &STR(&DATA
READDVAL &VAR1 &VAR2 &VAR3 ...
```

If &DATA contained HE SAID 'TRY IT', &VAR1 would be set to HE, &VAR2 would be set to SAID, and &VAR3 would be set to TRY IT.

Note: "TRY IT" is not considered a quoted string by the CLIST language so, if &DATA contained HE SAID "TRY IT", &VAR3 would be set to "TRY.

- The CLIST ERROR statement can be used to capture any nonzero return code. Since edit returns a code of 4 and sometimes 8 for information, it is suggested that, if used, it be coded as follows:

```
ERROR -
    IF &MAXCC < 12 THEN       /* If the last code is         */-
      DO                      /*   less than 12              */
        SET &MAXCC = 0        /*   continue processing       */
        RETURN                /*    (reset &MAXCC)           */
      END                     /* Otherwise go to severe      */
    ELSE                      /*    error processing         */-
      GOTO SEVERE             /*    to issue messages & exit */
```

APPENDIX B. SUMMARY OF MACRO STATEMENTS

This appendix lists all the forms of the PDF edit macro statements. For a description of the operands and usage, see the command descriptions in Chapter 7.

```
AUTOLIST          ISREDIT (varname) = AUTOLIST
                  ISREDIT AUTOLIST = mode
                  ISREDIT AUTOLIST mode
AUTONUM           ISREDIT (varname) = AUTONUM
                  ISREDIT AUTONUM = mode
                  ISREDIT AUTONUM mode
AUTOSAVE          ISREDIT (varname1,varname2) = AUTOSAVE
                  ISREDIT AUTOSAVE = mode
                  ISREDIT AUTOSAVE mode
BLKSIZE           ISREDIT (varname) = BLKSIZE
BOUNDS            ISREDIT (varname1,varname2) = BOUNDS
                  ISREDIT BOUNDS = left right
                  ISREDIT BOUNDS left right
BUILTIN           ISREDIT BUILTIN cmdname
CANCEL            ISREDIT CANCEL
CAPS              ISREDIT (varname) = CAPS
                  ISREDIT CAPS = mode
                  ISREDIT CAPS mode
CHANGE
  ISREDIT CHANGE str-1 str-2 [label-range] [NEXT ] [CHARS ] [X ][col-1 [col-2]]
                                           [ALL  ] [PREFIX] [NX]
                                           [FIRST] [SUFFIX]
                                           [LAST ] [WORD  ]
                                           [PREV ]
CHANGE_COUNTS     ISREDIT (varname1,varname2) = CHANGE_COUNTS
COPY              ISREDIT COPY member AFTER|BEFORE lptr [linenum-range]
CREATE            ISREDIT CREATE member lptr-range
CTL_LIBRARY       ISREDIT (varname1,varname2) = CTL_LIBRARY
CURSOR            ISREDIT (varname1,varname2) = CURSOR
                  ISREDIT CURSOR = line  col
DATA_CHANGED      ISREDIT (varname) = DATA_CHANGED
DATA_WIDTH        ISREDIT (varname) = DATA_WIDTH
DATAID            ISREDIT (varname) = DATAID
DEFINE            ISREDIT DEFINE name PGM MACRO|CMD MACRO
                  ISREDIT DEFINE name ALIAS name2
                  ISREDIT DEFINE name NOP
                  ISREDIT DEFINE name RESET
DELETE            ISREDIT DELETE lptr
                  ISREDIT DELETE lptr-range
                  ISREDIT DELETE ALL X|NX [lptr-range]
                  ISREDIT DELETE [ALL] X|NX lptr-range
DISPLAY_COLS      ISREDIT (varname1,varname2) = DISPLAY_COLS
DISPLAY_LINES     ISREDIT (varname1,varname2) = DISPLAY_LINES
DOWN              ISREDIT DOWN amt
EDIT              ISREDIT EDIT member
```

```
END                ISREDIT END
EXCLUDE
  ISREDIT EXCLUDE str-1 [label-range] [NEXT ] [CHARS ] [X ] [col-1 [col-2]]
                                      [ALL  ] [PREFIX] [NX]
                                      [FIRST] [SUFFIX]
                                      [LAST ] [WORD  ]
                                      [PREV ]
EXCLUDE_COUNTS     ISREDIT (varname1,varname2) = EXCLUDE_COUNTS
FIND
  ISREDIT FIND str-1 [label-range] [NEXT ] [CHARS ] [X ] [col-1 [col-2]]
                                   [ALL  ] [PREFIX] [NX]
                                   [FIRST] [SUFFIX] [EX]
                                   [LAST ] [WORD  ]
                                   [PREV ]
FIND_COUNTS        ISREDIT (varname1,varname2) = FIND_COUNTS
FLOW_COUNTS        ISREDIT (varname1,varname2) = FLOW_COUNTS
HEX                ISREDIT (varname1,varname2) = HEX
                   ISREDIT HEX = mode
                   ISREDIT HEX mode
IMACRO             ISREDIT (var) = IMACRO
                   ISREDIT IMACRO = name
                   ISREDIT IMACRO  name
INSERT             ISREDIT INSERT lptr [numlines]
LABEL              ISREDIT (varname1,varname2) = LABEL lptr
                   ISREDIT LABEL lptr = .labelname [level]
LEFT               ISREDIT LEFT amt
LEVEL              ISREDIT (varname) = LEVEL
                   ISREDIT LEVEL = num
                   ISREDIT LEVEL num
LINE               ISREDIT (varname) = LINE lptr
                   ISREDIT LINE lptr = data
LINE_AFTER         ISREDIT LINE_AFTER lptr = DATALINE|NOTELINE|MSGLINE data
LINE_BEFORE        ISREDIT LINE_BEFORE lptr = DATALINE|NOTELINE|MSGLINE data
LINENUM            ISREDIT (varname) = LINENUM lptr
LOCATE             ISREDIT LOCATE lptr
                   ISREDIT LOCATE [dir] lineid [lptr-range]
LRECL              ISREDIT (varname) = LRECL
MACRO              ISREDIT MACRO [(variables)] [PROCESS|NOPROCESS]
MACRO LEVEL        ISREDIT (varname) = MACRO_LEVEL
MASKLINE           ISREDIT (varname) = MASKLINE
                   ISREDIT MASKLINE = data
MEMBER             ISREDIT (varname) = MEMBER
MODEL              ISREDIT MODEL modelname AFTER|BEFORE lptr [NOTE|NONOTE]
                   ISREDIT MODEL CLASS classname
MOVE               ISREDIT MOVE member AFTER|BEFORE lptr
NOTE               ISREDIT (varname) = NOTE
                   ISREDIT NOTE = mode
                   ISREDIT NOTE mode
NULLS              ISREDIT (varname1,varname2) = NULLS
                   ISREDIT NULLS = mode
                   ISREDIT NULLS mode
```

```
NUMBER              ISREDIT (varname1,varname2) = NUMBER
                    ISREDIT NUMBER = mode
                    ISREDIT NUMBER mode
PACK                ISREDIT (varname) = PACK
                    ISREDIT PACK = mode
                    ISREDIT PACK mode
PROCESS             ISREDIT PROCESS [DEST] [RANGE cmd1 [cmd2]]
PROFILE             ISREDIT (varname1,varname2) = PROFILE
                    ISREDIT PROFILE name [number]
                    ISREDIT PROFILE number
                    ISREDIT PROFILE LOCK
                    ISREDIT PROFILE UNLOCK
RANGE_CMD           ISREDIT (varname) = RANGE_CMD
RCHANGE             ISREDIT RCHANGE
RECFM               ISREDIT (varname) = RECFM
RECOVERY            ISREDIT (varname) = RECOVERY
                    ISREDIT RECOVERY = mode
                    ISREDIT RECOVERY mode
RENUM               ISREDIT RENUM [STD|COBOL|DISPLAY]
REPLACE             ISREDIT REPLACE member lptr-range
RESET               ISREDIT RESET lineid [lptr-range]
RFIND               ISREDIT RFIND
RIGHT               ISREDIT RIGHT amt
RMACRO              ISREDIT (varname) = RMACRO
                    ISREDIT RMACRO = name
                    ISREDIT RMACRO name
SAVE                ISREDIT SAVE
SCAN                ISREDIT SCAN  mode
SEEK
  ISREDIT SEEK str-1 [label1-range] [NEXT ] [CHARS ] [X ] [col-1 [col-2]]
                                    [ALL  ] [PREFIX] [NX]
                                    [FIRST] [SUFFIX]
                                    [LAST ] [WORD  ]
                                    [PREV ]
SEEK_COUNTS         ISREDIT (varname1,varname2) = SEEK_COUNTS
SHIFT (             ISREDIT SHIFT ( linenum [n]
SHIFT )             ISREDIT SHIFT ) linenum [n]
SHIFT <             ISREDIT SHIFT < linenum [n]
SHIFT >             ISREDIT SHIFT > linenum [n]
SORT                ISREDIT SORT [X|NX] sortfield-1 [sortfield-2, ...] [label-range]
STATS               ISREDIT (varname) = STATS
                    ISREDIT STATS = mode
                    ISREDIT STATS mode
SUBMIT              ISREDIT SUBMIT [lptr-range]
TABS                ISREDIT (varname) = TABS
                    ISREDIT TABS = mode
                    ISREDIT TABS mode
TABSLINE            ISREDIT (varname) = TABSLINE
                    ISREDIT TABSLINE = data
TENTER              ISREDIT TENTER lptr [n]
TFLOW               ISREDIT TFLOW lptr [col]
TSPLIT              ISREDIT TSPLIT [lptr col]
UNNUM               ISREDIT UNNUM
```

```
UP              ISREDIT UP amt
USER_STATE      ISREDIT (varname) = USER_STATE
                ISREDIT USER_STATE = (varname)
VERSION         ISREDIT (varname) = VERSION
                ISREDIT VERSION = num
                ISREDIT VERSION num
XSTATUS         ISREDIT (varname) = XSTATUS lptr
                ISREDIT XSTATUS lptr = X|NX
```

APPENDIX C. LIST OF ABBREVIATIONS

The following list includes the command names and keywords that can be abbreviated, followed by the allowable abbreviation(s). It is recommended that, in order to improve readability, abbreviations not be used in edit macros.

AFTER	AFT			
BEFORE	BEF			
BOUNDS	BOUND	BNDS	BND	
BROWSE	BRO			
CANCEL	CAN			
CHANGE	CHA	CHG	C	
CHARS	CHAR			
COMMAND	CMD	COM		
COBOL	COB			
COLUMNS	COLS	COL		
CREATE	CRE			
CURSOR	CUR	CSR		
DATA	D			
DEFINE	DEF			
DELETE	DEL			
DISPLAY	DIS	DISP	DISPL	
ERR	ERROR			
EXCLUDED	EXCLUDE	EXC	EX	X
FIND	F			
HALF	H			
LABEL	LABELS	LAB		
LEVEL	LEV			
LOCATE	LOC	L		
MAX	M			
MODEL	MOD			
NOCOBOL	NOCOB			
NOCOLS	NOCOL			
NONOTES	NONOTE	NONOT		
NONULLS	NONULL	NONUL		
NONUM	NONUMBR	NONUMB	NONUMBER	
NOPROCESS	NOPROC			
NOTABS	NOTAB			
NOTES	NOTE			
NULLS	NULL	NUL		
NUMBER	NUMB	NUM		
NX	NON X'ED	NONX		
PAGE	P			
PROGRAM	PGM			
PREFIX	PRE			
PROFILE	PROF	PRO	PR	
RECOVERY	RECOVER	RECOVRY	RECVRY	RECVR
RECOVERY	RECOV	REC		
RENUM	REN			

REPLACE	REPL	REP	
RESET	RES		
SPECIAL	SPE		
STANDARD	STD		
SUBMIT	SUB		
SUFFIX	SUF		
TABS	TAB		
UNNUMBER	UNNUMB	UNNUM	UNN
VERSION	VERS	VER	
VERTICAL	VERT		

INDEX

M

N

Interactive System Productivity Facility/
Program Development Facility
Version 2 for MVS/Extended Architecture SC34-4018-0
Edit Macros

READER'S
COMMENT
FORM

This manual is part of a library that serves as a reference source for systems analysts, programmers, and operators of IBM systems. This form may be used to communicate your views about this publication. It will be sent to the author's department for whatever review and action, if any, is deemed appropriate. Comments may be written in your own language; use of English is not required.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

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

Possible topics for comments are:

Clarity Accuracy Completeness Organization Coding Retrieval Legibility

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

_____

_____

_____

Note: Staples can cause problems with automated mail sorting equipment.
Please use pressure sensitive or other gumed tape to seal this form.

Cut or Fold Along Line

What is your occupation? _____

Number of latest Technical Newsletter (if any) concerning this publication: _____

Thank you for your cooperation. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments.)

Reader's Comment Form

Cut or Fold Along Line

Fold and tape | **Please Do Not Staple** | Fold and tape

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

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

POSTAGE WILL BE PAID BY ADDRESSEE:

International Business Machines Corporation
Department T46
P. O. Box 60000
Cary, North Carolina 27511

Fold and tape | **Please Do Not Staple** | Fold and tape

IBM®

ISPF/PDF for MVS/Extended Architecture Edit Macros

Printed in U.S.A.

SC34-4018-0

Interactive System Productivity Facility/
Program Development Facility
Version 2 for MVS/Extended Architecture SC34-4018-0
Edit Macros

READER'S
COMMENT
FORM

This manual is part of a library that serves as a reference source for systems analysts, programmers, and operators of IBM systems. This form may be used to communicate your views about this publication. It will be sent to the author's department for whatever review and action, if any, is deemed appropriate. Comments may be written in your own language; use of English is not required.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

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

Possible topics for comments are:

Clarity Accuracy Completeness Organization Coding Retrieval Legibility

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

_____

_____

_____

Note: Staples can cause problems with automated mail sorting equipment.
Please use pressure sensitive or other gumed tape to seal this form.

Cut or Fold Along Line

What is your occupation? _____

Number of latest Technical Newsletter (if any) concerning this publication: _____

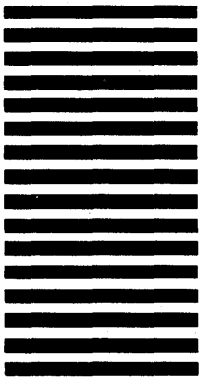
Thank you for your cooperation. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments.)

Reader's Comment Form

Cut or Fold Along Line

Fold and tape | **Please Do Not Staple** | Fold and tape

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES



BUSINESS REPLY MAIL

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

POSTAGE WILL BE PAID BY ADDRESSEE:

International Business Machines Corporation
Department T46
P. O. Box 60000
Cary, North Carolina 27511

Fold and tape | **Please Do Not Staple** | Fold and tape

ISPF/PDF for MVS/Extended Architecture Edit Macros
Printed in U.S.A.
SC34-4018-0

Publication Number
SC34-4018-0

File Number
S370/4300-39

Program Number
5665-317

Printed in
U.S.A.

IBM

SC34-4018-0