

ATTN: CHARLIE GIBBS

01111
CAV208M45541 UP 8806-B

UAS

SPERRY UNIVAC
1 - 1018 CORNWALL STREET
VANCOUVER B C

V6J 1C7

**PUBLICATIONS
UPDATE**

Operating System/3 (OS/3)

Dialog Specification
Language

User Guide/Programmer
Reference

UP-8806-B

This Library Memo announces the release and availability of Updating Package B to "SPERRY UNIVAC Operating System/3 (OS/3) Dialog Specification Language User Guide/Programmer Reference", UP-8806.

This update incorporates minor changes and corrections to the manual for release 8.0:

- Nested COPY commands
- Compile bigger dialogs
- Output command

Copies of Updating Package B are now available for requisitioning. Either the updating package only or the complete manual with the updating package may be requisitioned by your local Sperry Univac representative. To receive only the updating package, order UP-8806-B. To receive the complete manual, order UP-8806.

LIBRARY MEMO ONLY	LIBRARY MEMO AND ATTACHMENTS	THIS SHEET IS
Mailing Lists BZ, CZ and MZ	Mailing Lists A00, A02, A06, B00, B02, 18, 18U, 19, 19U, 20, 20U, 21, 21U, 28U, 29U, 75, 75U, 76, and 76U (Package B to UP-8806, 42 pages plus Memo)	Library Memo for UP-8806-B RELEASE DATE: September, 1982



PAGE STATUS SUMMARY

ISSUE: Update B – UP-8806
RELEASE LEVEL: 8.0 Forward

Part/Section	Page Number	Update Level	Part/Section	Page Number	Update Level	Part/Section	Page Number	Update Level
Cover/Disclaimer		Orig.	8	1, 2 3 4, 5	B A B			
PSS	1	B	Appendix A	1	Orig.			
Preface	1, 2	Orig.	Appendix B	1 2 thru 5	A Orig.			
Contents	1, 2 3, 4 5, 6 7	Orig. B Orig. A	Appendix C	1 thru 4 5 6 thru 8	Orig. B Orig.			
1	1 2 3 thru 5 6 6a 7 thru 9	Orig. B Orig. B B* B	Appendix D	1 thru 3 4 5 6	Orig. B Orig. B			
2	1 thru 4	Orig.	Appendix E	1 thru 7	Orig.			
3	1 thru 3	Orig.	Glossary	1, 2 3 4	Orig. A Orig.			
4	1, 2 3 4 thru 15 16, 17 18 thru 26	Orig. A Orig. A Orig.	Index	1 2 3 4 5, 6 7, 8	Orig. B Orig. A B Orig.			
5	1 2 thru 6 7 8 thru 10 10a 11, 12 13 thru 54	Orig. A Orig. A A A Orig.	User Comment Sheet					
6	1 thru 3 4 5 thru 7 8 9 thru 20 21 22 23 thru 26 26a 27 28 thru 47 48 48a 49, 50 51	Orig. B Orig. B Orig. A Orig. B B* A Orig. B B* Orig. A						
7	1 2 thru 28	B Orig.						

*New pages

All the technical changes are denoted by an arrow (→) in the margin. A downward pointing arrow (↓) next to a line indicates that technical changes begin at this line and continue until an upward pointing arrow (↑) is found. A horizontal arrow (→) pointing to a line indicates a technical change in only that line. A horizontal arrow located between two consecutive lines indicates technical changes in both lines or deletions.



5. DECLARATIONS

5.1.	INTRODUCTION		5-1
5.2.	NAMING AND DEFINING AN ARRAY - BASIC CONCEPTS	(DATA)	5-1
5.2.1.	How to Name and Define Variable Arrays (DATA Format 1)		5-2
5.2.2.	How to Name and Define Message Arrays (DATA Format 2)		5-12
5.3.	HOW TO NAME AND DEFINE CONTROL MASKS	(MASK)	5-15
5.4.	HOW TO NAME AND DEFINE BLOCKS	(DO...END)	5-18
5.5.	NAMING AND DEFINING TREES - BASIC CONCEPTS	(TREE)	5-25
5.5.1.	How to Name and Define Simple Trees (TREE Format 1)		5-27
5.5.2.	How to Name and Define Nested Trees (TREE Format 2)		5-42
5.6.	NAMING AND DEFINING SEPARATE BRANCHES - BASIC CONCEPTS	(BRANCH...END)	5-49
5.6.1.	How to Name and Define Simple Separate Branches (BRANCH...END Format 1)		5-49
5.6.2.	How to Name and Define Separate Branches Containing Nested Trees (BRANCH...END Format 2)		5-51
5.7.	HOW TO NAME AND DEFINE SUBSTITUTION TEXT	(MEANS)	5-53

6. COMMANDS

6.1.	INTRODUCTION		6-1
6.2.	FORMING UNNAMED BLOCKS - BASIC CONCEPTS	(DO...END)	6-4
6.2.1.	How to Form Blocks (DO...END Format 1)		6-4
6.2.2.	How to Form Trunk Blocks (DO...END Format 2)		6-6
6.2.3.	How to Form Branch Blocks (DO...END Format 3)		6-7
6.3.	HOW TO CALL NAMED BLOCKS	(CALL)	6-8
6.4.	FORMING UNNAMED TREES - BASIC CONCEPTS	(TREE)	6-9
6.4.1.	How to Form Simple Trees (TREE Format 1)		6-9
6.4.2.	How to Form Nested Trees (TREE Format 2)		6-11
6.5.	HOW TO DISPLAY TREES	(PRESENT)	6-13
6.6.	HOW TO SOLICIT RESPONSE FROM THE DIALOG USER (ENTER)		6-15
6.7.	HOW TO DISPLAY MESSAGES	(DISPLAY)	6-17
6.8.	HOW TO PRINT MESSAGES IN THE SUMMARY REPORT (PRINT)		6-20
6.9.	STORING DIALOG USER RESPONSES IN AN INPUT BUFFER - BASIC CONCEPTS	(OUTPUT)	6-23
6.9.1.	How to Store Dialog User Responses (OUTPUT Format 1)		6-23
6.9.2.	How to Route Stored Responses to Input Buffer and Return Control to Program (OUTPUT Format 2)		6-25



6.10.	CHANGING ARRAY VALUES - BASIC CONCEPTS	(LOAD)	6-26
6.10.1.	How to Change All Array Elements (LOAD Format 1)		6-26
6.10.2.	How to Change Current Array Element (LOAD Format 2)		6-28
6.11.	CHANGING CONTROL MASK VALUES - BASIC CONCEPTS	(SET)	6-30
6.11.1.	How to Change Control Mask to a New Value (SET Format 1)		6-31
6.11.2.	How to Reset Control Masks to Initial Value (SET Format 2)		6-34
6.11.3.	How to Reset Arrays and Their Control Masks to Initial Value (SET Format 3)		6-35
6.12.	HOW TO USE A CONTROL MASK TO CHANGE A CONTROL MASK	(MASK)	6-36
6.13.	PERFORMING CONDITIONAL OPERATIONS - BASIC CONCEPTS	(IF)	6-40
6.13.1.	How to Perform Conditional Operations Outside Trees (IF Format 1)		6-40
6.13.2.	How to Perform Trunk Conditional Operations (IF Format 2)		6-43
6.13.3.	How to Perform Branch Conditional Operations (IF Format 3)		6-44
6.14.	HOW TO INDICATE THE END OF YOUR PROGRAM	(EOF)	6-46
6.15.	HOW TO START MESSAGES ON A NEW LINE ON SCREEN	(NEWLINE)	6-47
6.16.	HOW TO COPY SOURCE CODE FROM LIBRARY FILE INTO YOUR PROGRAM	(COPY)	6-48
6.17.	HOW TO ADVANCE THE PRINTER FORM TO A NEW PAGE	(EJECT)	6-49
6.18.	HOW TO ADVANCE THE PRINTER FORM TO A NEW LINE	(SPACE)	6-50
6.19.	HOW TO ADVANCE THE PRINTER FORM TO A NEW PAGE AND PRINT A TITLE	(TITLE)	6-51
7. SAMPLE PROGRAM			
7.1.	GENERATION OF DIALOG SCREENS		7-1
7.2.	SUMMARY REPORT		7-27
7.3.	RESULTS OF OUTPUT COMMAND		7-28
8. JOB CONTROL STATEMENTS			
8.1.	INTRODUCTION		8-1
8.2.	PARAM STATEMENT		8-1
8.3.	SAMPLE JOB CONTROL STREAM TO EXECUTE THE TRANSLATOR		8-4

1. Introduction

1.1. PROCESSING IN AN INTERACTIVE ENVIRONMENT

You use the *dialog specification language* to write programs that produce interactive dialogs displayed to a dialog user on a workstation screen. A *workstation* is an interactive device with a screen for displaying dialog text and a keyboard for entering user input. The *interactive dialogs* consist of a series of questions and answers between the dialog user and the dialogs you write. These interactive dialog sessions solicit input from the dialog user that is used as data for an application program. *Interactive processing* allows the dialog user to provide data to the application program by using the workstation instead of entering data by punched cards, disk, diskette, and magnetic tape.

Figure 1-1 illustrates the interactive environment.

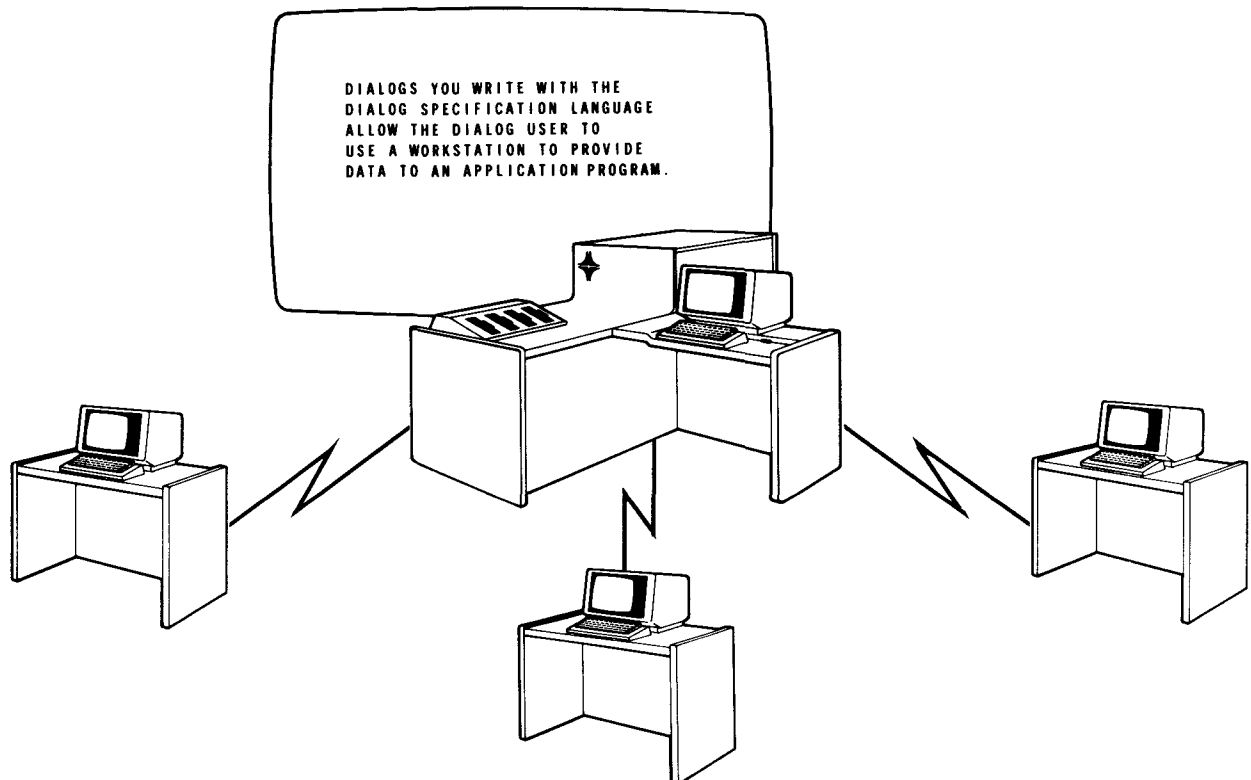
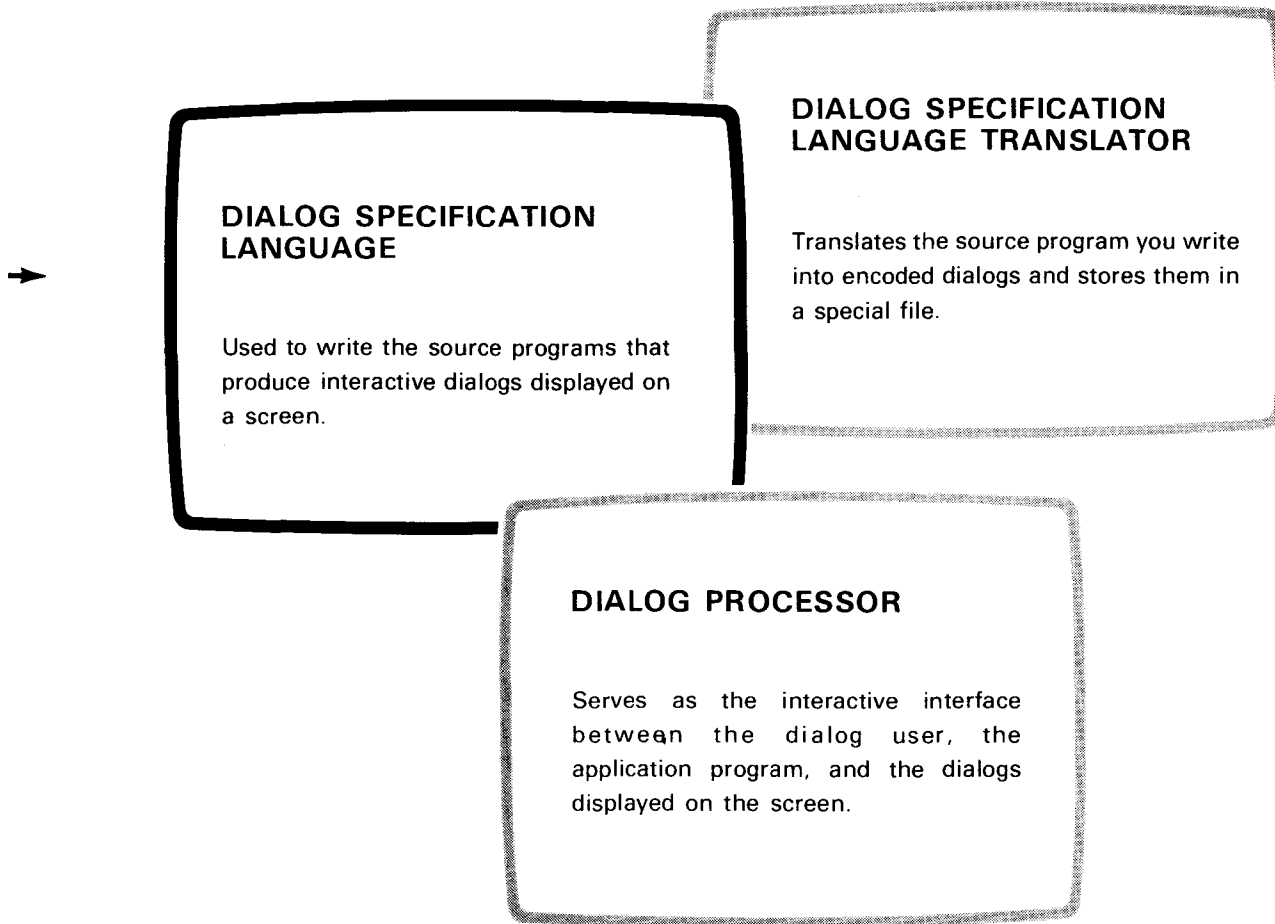


Figure 1-1. Interactive Environment

Three elements work together to form interactive dialog sessions:



1.2. THE DIALOG SPECIFICATION LANGUAGE AND ITS USE

You use the dialog specification language to write a program that displays an interactive dialog on a workstation screen. These interactive dialogs are used by any application program in your operation that would be enhanced by using a dialog to solicit data from a dialog user. You write your dialog to satisfy the requirements of the application program. Using the high-level dialog specification language, you specify the dialog structure, the messages displayed on the screen, the input entered by the dialog user, and the content and format of output records and printed reports.

1.2.1.2. Dialogs Written by You

Using the dialog specification language, you can write a dialog for any application program that lends itself to an interactive environment.

Why should you write your own dialogs?

1. **Dialogs are suited to your needs.** When you write your own dialogs, you create an interactive interface to new or existing application programs that suits your needs.
2. **Dialogs make data entry easy.** The dialog user can use the dialog to enter data in a conversational manner. The dialog user can be an experienced or inexperienced person.
3. **Dialogs reduce data entry errors.** Because data entry is so simple, data errors are reduced. You can write the dialog so that invalid responses are rejected. For example, if the dialog asks the user to enter a division code, you can write the dialog to accept only 5-digit numeric data. Although this doesn't ensure that the dialog user will enter the correct code, it does help reduce errors.
4. **Dialogs enforce uniform data entry** Using a dialog to solicit input enforces uniform data entry and eliminates incomplete entries.
5. **Dialogs are compatible with your existing programs.** Existing programs that use another medium for input, such as punched cards, are easily changed to accept input from a dialog session. You can use any language to write an application program that calls a dialog.
6. **Dialogs can be used by experienced and inexperienced personnel.** You can write the dialog so that users can choose the version of the dialog that is at their level of experience. The dialog user can switch back and forth between the versions at any time.

After you write the dialog, you store the source program in a library file. Figure 1-2 illustrates the process of writing a program with the dialog specification language.

You write source dialogs by using the dialog specification language and store them in a SAT library file either by using the workstation with the general editor (EDT) or by using batch methods and standard job control statements.

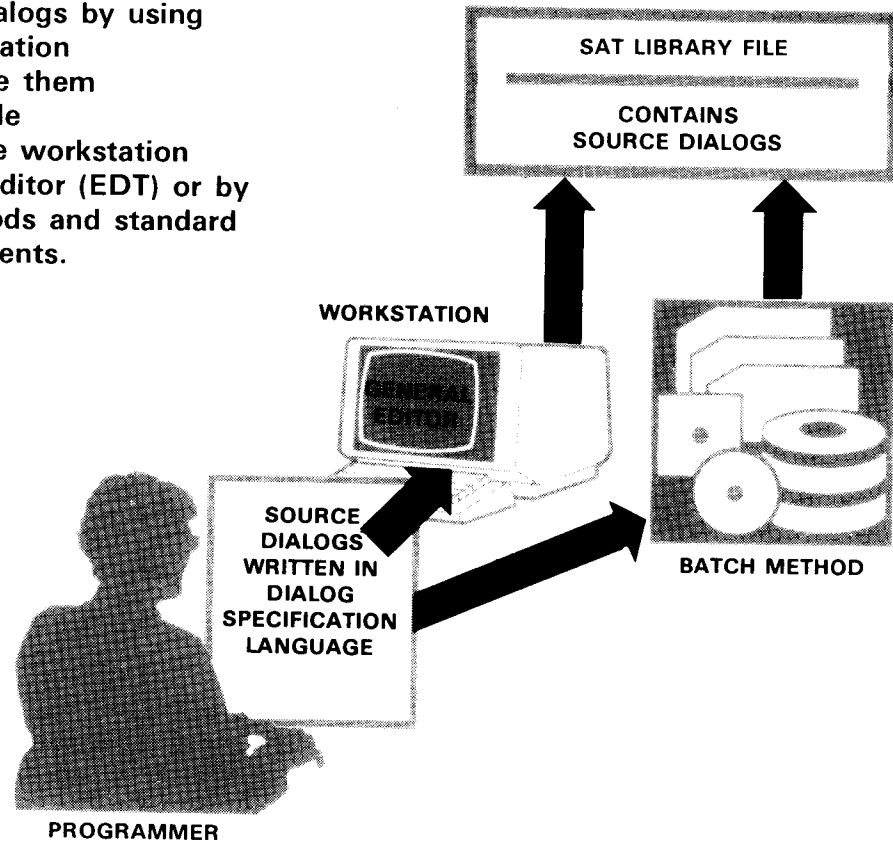


Figure 1—2. Use of Dialog Specification Language

1.3. THE DIALOG SPECIFICATION LANGUAGE TRANSLATOR AND ITS USE

After you store your source program (dialog) in a SAT library file, the dialog specification language translator, referred to as the *translator* in the rest of this manual, takes over. The translator operates only in consolidated data management (CDM) mode. The translator uses the source program stored in the SAT library file as its main input. It takes the source program from the library file, translates it into encoded dialogs, automatically stores the dialog in a MIRAM file, and generates two output files:

1. Encoded MIRAM Dialog File

This file is the primary output of the translator. It is a MIRAM system file and contains the encoded dialogs generated by the translator. You must give each encoded dialog stored in this file a unique name. The dialog user specifies that name in the job control stream of the program that calls that dialog. The encoded dialogs are the input to the dialog processor. Because they're specially encoded dialogs, you can't access them any way other than through the dialog processor. You can change your dialog indirectly, however. First, you must change your code and then retranslate it.

When insufficient contiguous main storage exists in the system to process the dialog file, the message

DP082 INSUFFICIENT MEMORY

is displayed and the job is terminated. You must wait until sufficient contiguous space becomes available to retry the job.



2. Output Listing File

In addition to the encoded dialog file, the translator can, at your option, generate an output listing file. This file contains:

Source Listing

A listing of the source program translated by the dialog specification language translator.

Error Listing

A listing of the errors detected during translation. Error messages (Appendix E) are categorized into warning diagnostics and fatal diagnostics.

The translator requires at least 70K of main storage and runs under the minimum operating system. You can't call the translator from other software as a subroutine – it is a stand-alone product.

Figure 1-3 summarizes the use of the translator.

Source dialogs stored in the SAT library file are compiled by the translator, which generates an encoded MIRAM dialog file containing encoded dialogs, and an output listing file containing listings and errors in your source program.

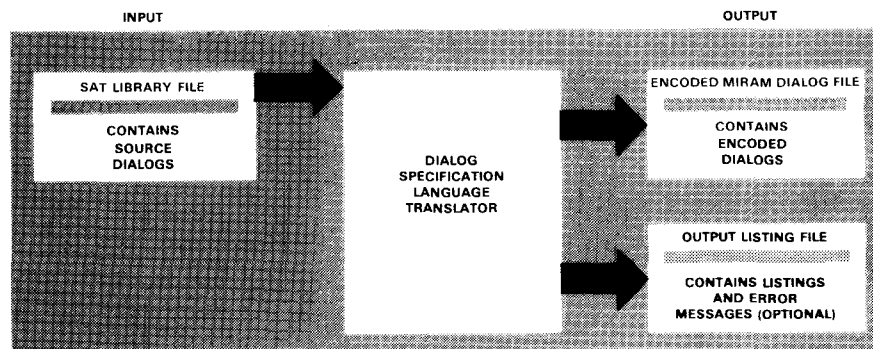


Figure 1-3. Dialog Specification Language Translator

1.4. THE DIALOG PROCESSOR AND ITS USE

The dialog processor supervises the interactive exchanges between the dialog user and the encoded dialogs stored in the encoded dialog file. The application program calls data management, which, if so directed by job control, calls the dialog processor. The dialog processor then displays the explanations and questions contained in the encoded dialog on the workstation screen. When the dialog user keys in a response, the dialog processor analyzes each response, controls succeeding interactions, and routes the response to a file used by the application program that called the dialog.

NOTE:

When the application program is written in basic assembly language (BAL), the dialog processor only supports the following data management macros: OPEN, CLOSE, DMINP, and RIB. (The TRUNC parameter, however, is not supported, so report truncation by setting the CD\$TRUNC bit in the CDIB macroinstruction.)

The dialog processor handles the display of standard control and error messages, the positioning of headings, the tabulation of choices, and error recovery procedures, so you don't have to include commands for these in your program.

For more information about the dialog processor, see the dialog processor user guide/programmer reference, UP-8858 (current version).

Figure 1-4 summarizes the use of the dialog processor, and Figure 1-5 gives an overview of interactive processing using dialogs.

→ Encoded dialogs stored in the encoded MIRAM dialog file are managed by the dialog processor, which serves as the interface between the dialog user, the encoded dialog, and the application program in an interactive environment.

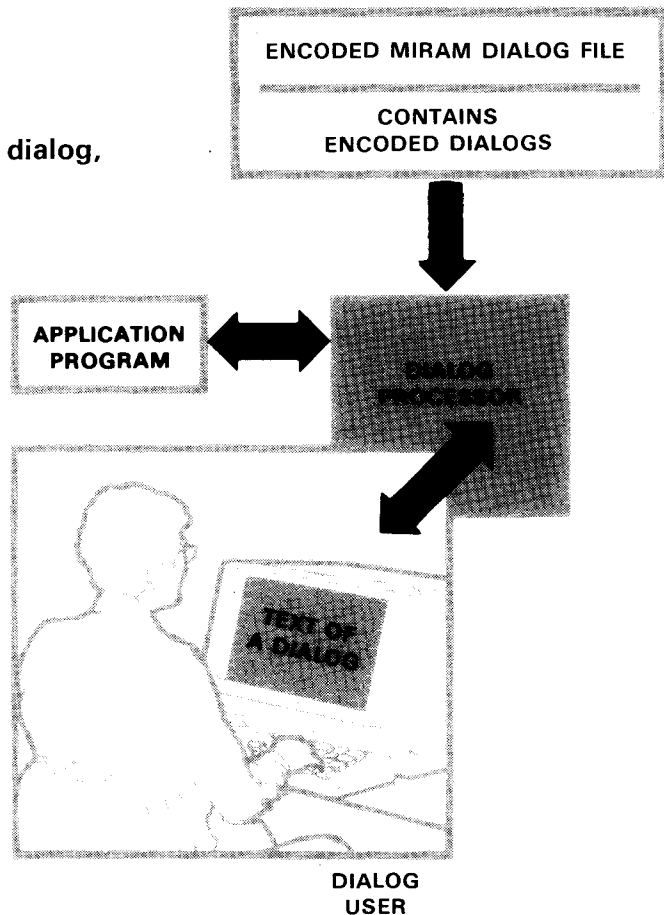


Figure 1-4. Dialog Processor

Dialogs that you write in the dialog specification language are stored in a SAT library file. The translator compiles the dialog and generates encoded MIRAM dialogs stored in the encoded dialog file. The dialog processor supervises the interactive dialog session between the dialog user, the encoded dialog, and the application program.

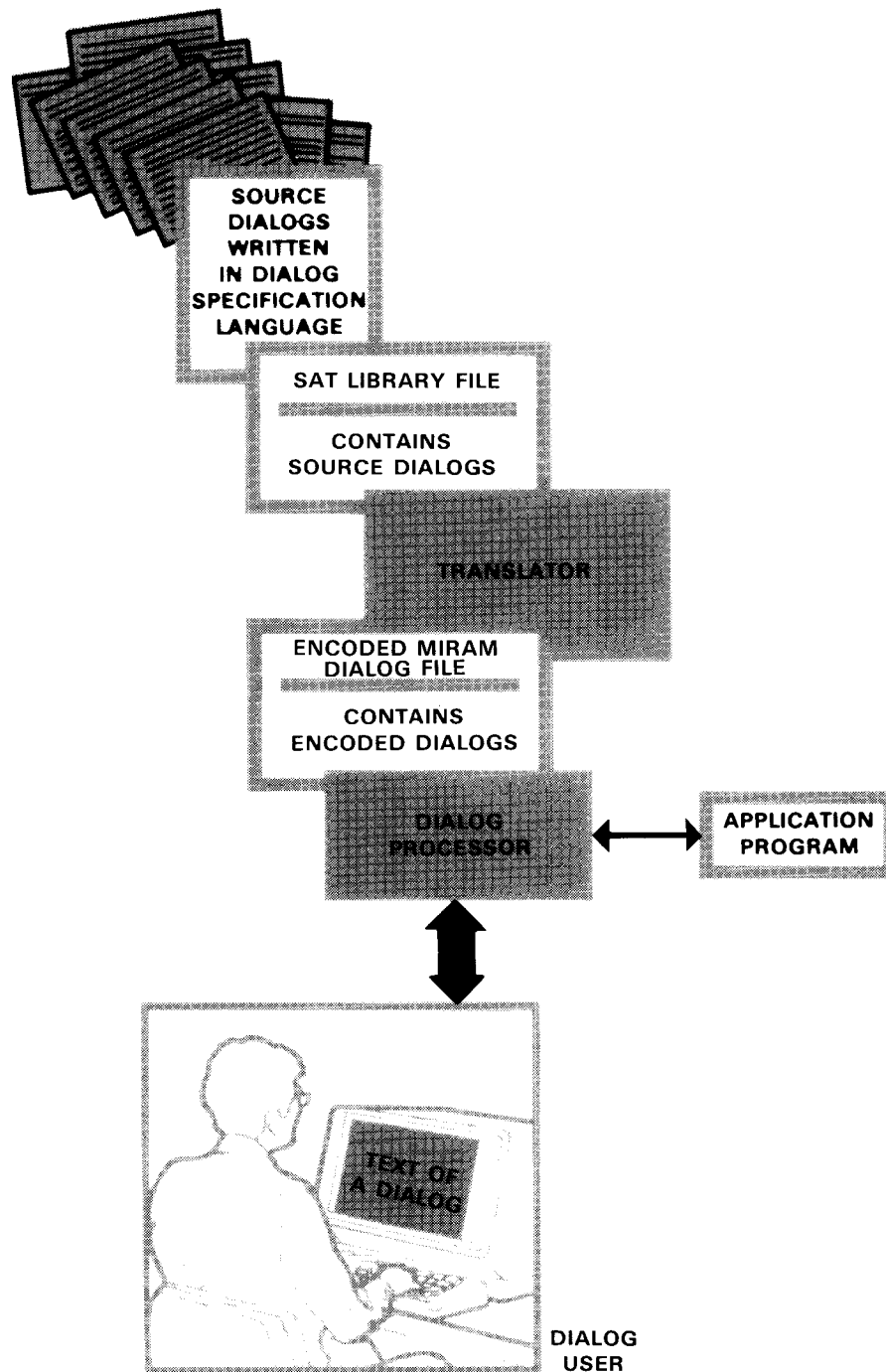


Figure 1—5. Overview of Interactive Processing Using Dialogs



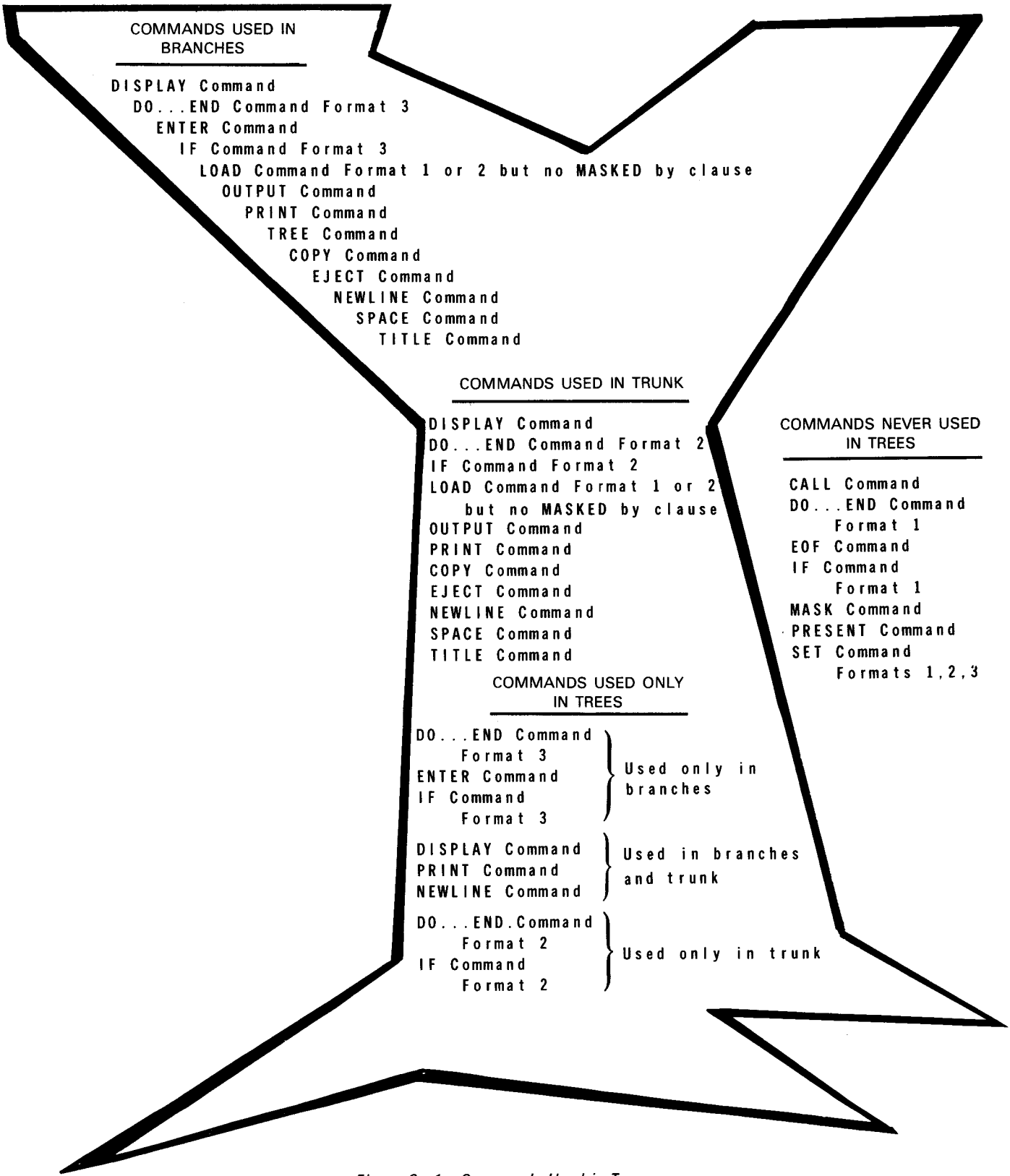


Figure 6-1. Commands Used in Trees

6.2. FORMING UNNAMED BLOCKS - BASIC CONCEPTS (DO...END)

You use the DO...END command to form unnamed blocks that are executed inline. Unnamed blocks consist of commands and declarations delimited by the reserved words DO and END. An unnamed block must contain a command or a declaration; it can't be null.

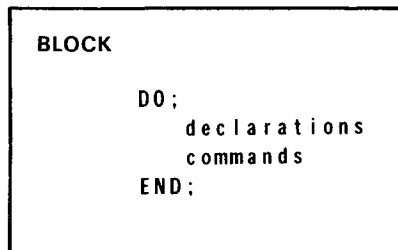
Unnamed blocks are categorized into three types:

1. **Blocks**, which specify actions and control the repetition of block commands, the transfer of control between block commands, the reinitialization of array and mask variables, and the scope (local or global) of declarations and commands
2. **Trunk blocks**, which are used only in the trunk of a tree when you want to execute more than one command conditionally with the IF command
3. **Branch blocks**, which are used only in the branches of a tree when you want to execute more than one command conditionally with the IF command

6.2.1. How to Form Blocks (DO...END Format 1)

You use the format 1 DO...END command to form unnamed blocks that are executed inline. They specify actions to be performed and control the repetition of block commands, the transfer of control between block commands, the reinitialization of array and mask variables, and the scope (local or global) of commands and declarations. You can't use a format 1 DO...END command in the trunk or branches of a tree.

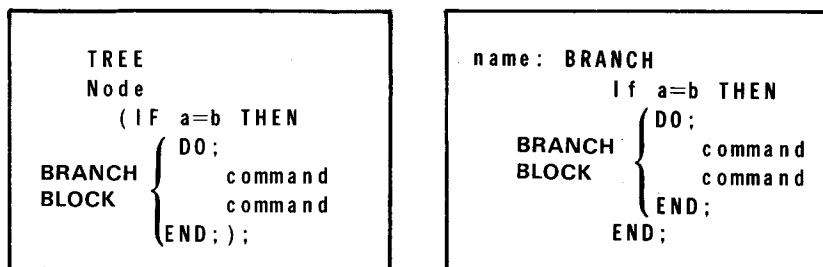
A graphic representation of an unnamed block formed by a format 1 DO...END command is:



6.2.3. How to Form Branch Blocks (DO...END Format 3)

You use the format 3 DO...END command to form unnamed branch blocks used only in the branches of a tree. You use this format when you want to execute more than one command conditionally in the branch. In other words, when you use the format 3 IF command in a branch to execute two or more commands, you must use a branch block.

A graphic representation of an unnamed branch block formed by the format 3 DO...END command is:



The format 3 DO...END command is:

```

DO;
{
  DISPLAY Command;
  DO...END Command Format 3;
  ENTER Command;
  IF Command Format 3;
  LOAD Command Format 1 or 2 but
  no MASKED BY clause;
  OUTPUT Command;
  PRINT Command;
  NEWLINE Command;
}
END;

```

The following two examples show how a branch block is used when you want to execute more than one command conditionally:

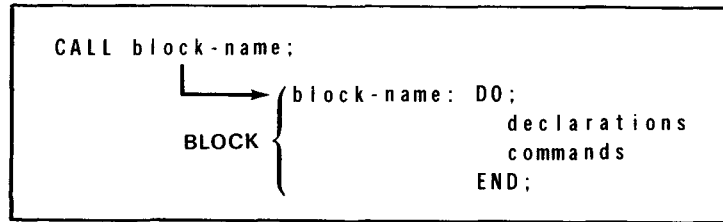
<pre> FINALBR: BRANCH IF LEVEL='2' THEN DO; DISPLAY TOTO; ENTER OBJT; END; END; </pre>	<pre> PRESENT TREE PARALLEL (IF LEVEL='2' THEN DO; DISPLAY TOTO; ENTER OBJT; END;); </pre>
--	--

In both examples, the branch block formed by the format 3 DO...END command is used to execute a DISPLAY command and an ENTER command if the value of LEVEL is equal to 2.

6.3. HOW TO CALL NAMED BLOCKS (CALL)

You use the CALL command to execute a named block defined by a DO...END declaration. You may nest CALL commands as long as you don't repeat the name of the block in a nested CALL command. You can't use the CALL command anywhere in the trunk or branches of a tree.

A graphic representation of the use of a CALL command is:



The format of the CALL command is:

```
CALL block-name;
```

The explanation of the parameter is:

block-name

Name of a block defined by a DO...END declaration.

The following example illustrates the use of the CALL command:

```
CALL PROCSTMT;
PROCSTMT: DO;
    OUTPUT 'PROCSTMT WAS CALLED.';
END;
```

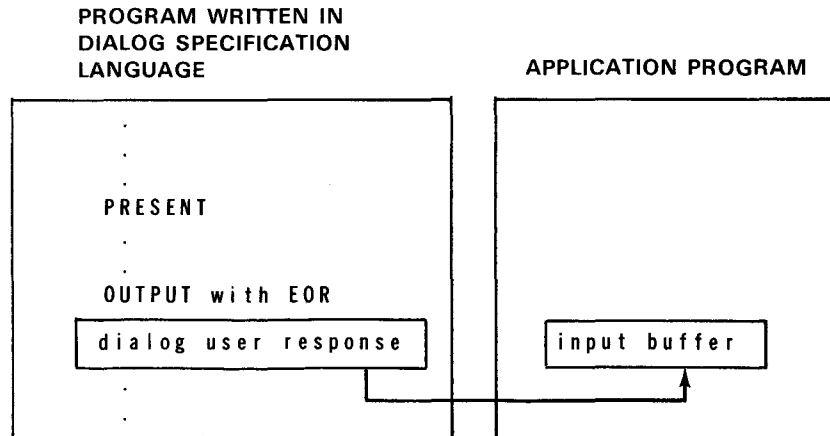
In this example, the CALL command references a block named PROCSTMT. When the CALL command is executed, the PROCSTMT block provides the CALL command with the data defined in the block. A named block can't be null. Some executable statement must appear within each named block.

Another example of the CALL command is:

```
DO UNTIL TIMESW = '1';
    CALL GETDATE;
    CALL GETTIME;
END;
GETDATE: DO;
    PRESENT DATE;
END;
GETTIME: DO;
    PRESENT TREE2;
    LOAD TIMESW = 'D';
END;
```

6.9. STORING DIALOG USER RESPONSES IN AN INPUT BUFFER - BASIC CONCEPTS (OUTPUT)

Use the OUTPUT command to store dialog user responses in an output file and then route them to an input buffer identified in the application program that called the dialog. Editing in the DATA declarations controls the way data is stored. Use the OUTPUT command anywhere in your program. A graphic representation of the use of the OUTPUT command is:



6.9.1. How to Store Dialog User Responses (OUTPUT Format 1)

The format of the format 1 OUTPUT command is:

```
OUTPUT { 'message'
        { array-name
        { string-expression } } [EOR];
```

First, we'll cover the required parameters of this command.

'message'

The message or value that is used as input by the application program. The following example shows how an OUTPUT command is used:

```
PROCSTMT: DO;
           OUTPUT 'PROCSTMT WAS CALLED';
           END;
```

In this example, the message PROCSTMT WAS CALLED is routed to an input buffer for use by an application program.

array-name

Name of an array defined by a DATA declaration. It is the elements in the array that are stored.

With the OUTPUT command, no underscores are generated, and the dialog user can't enter data. Instead, the initialization value is displayed on the screen, padded if necessary with spaces to the length of the element. The dialog user just views the message.

This example illustrates an OUTPUT command that references an array name:

```
ARRAY1: DATA \A5\ ;
PRESENT TREE PARALLEL
      (ENTER ARRAY1; OUTPUT ARRAY1;);
```

When this tree is executed, ARRAY1 is displayed, and whatever response is entered by the dialog user into ARRAY1 is stored for use by the application program.

string-expression

Expression mainly used to concatenate messages and arrays so that one OUTPUT command can process them at the same time. You must enclose the concatenated value within brackets.

See Appendix D for a complete description of string expressions.

The following example shows how the OUTPUT command uses a string expression to obtain output used by the application program:

```
C1: DATA \A3\ 'GAS';
C2: DATA \A3\ 'OIL';
C3: DATA \A5\ 'SOLAR';
POWERNAME: DATA \A5\ ;
COMP: TREE EXCLUSIVE
      (C1; OUTPUT '5' EOR;)
      (C2; OUTPUT '6' EOR;)
      (C3; ENTER POWERNAME; OUTPUT['7, ', POWERNAME] EOR;);
PRESENT COMP;
```

Assume that the application program that called this dialog has the following input format:

```
n[ , powername ]
```

where:

```
n = 5
    For gas.

n = 6
    For oil.

n = 7
    For solar.
```

When this tree is presented, there is dual output, which is an important feature of the dialog processor. The summary report is designed for user readability, while the output is designed for the requirements of an application program.

The dialog user sees:

```

1.  GAS
2.  OIL
3.  SOLAR_____
SELECT ITEM BY ENTERING A NUMBER --

```

If the dialog user picks GAS, 5 is stored for use by the application program. If OIL, 6 is stored, and if SOLAR, 7 is stored along with the value entered by the dialog user. See the dialog processor user guide/programmer reference, UP-8858 (current version).

Now, we'll discuss the optional parameter.

EOR

End of the stored record.

Application programs request the input at the record level. The data is not available to the application program until an EOR is output.

This example shows how EOR is used in a format 1 OUTPUT command:

```

DAY: DATA \A3\ 'MON' ;
SPACE: DATA \A2\ ;
MONTH: DATA \A3\ 'JAN' ;
OUTPUT [DAY, SPACE \A3\, MONTH, ' ', '77'] EOR;

```

When the OUTPUT command with EOR is executed, the following is routed to the input buffer and control returns to the program. (The EOR indicates the end of the record.)

```
MON JAN 77
```

6.9.2. How to Route Stored Responses to Input Buffer and Return Control to Program (OUTPUT Format 2)

The format of the format 2 OUTPUT command is:

```
OUTPUT EOR;
```

When this command executes, the user responses stored in the output file are routed to the input buffer and control returns to the program. You don't need this command when your last format 1 OUTPUT command ends with the optional parameter EOR.

6.10. CHANGING ARRAY VALUES - BASIC CONCEPTS (LOAD)

You use the LOAD command to change an array element to another value. The new value replaces the original value of the element. Only the array elements whose corresponding control mask bits are set to 1 or T are changed. The LOAD command doesn't change the control mask associated with each array. You can use the LOAD command anywhere in the program, but when you use it in the branch of a tree, you can't use the MASKED BY clause with it.

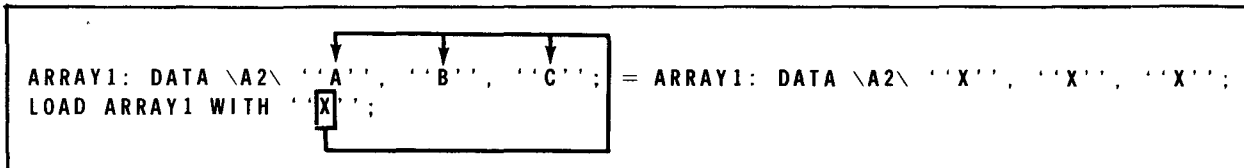
LOAD commands are categorized into two types:

1. Change all the array elements of an uncontrolled array to a new value.
2. Change the current array element of a controlled array to a new value.

6.10.1. How to Change All Array Elements (LOAD Format 1)

You use the format 1 LOAD command to change all array elements of an uncontrolled array to a new value. The LOAD command only affects the elements whose corresponding control mask bit is set to 1 or T. The elements are changed (loaded) one for one with the results of successive evaluations of the value by using the corresponding elements. The new value replaces the original value of the elements. The LOAD command doesn't change the control mask for the array.

A graphic representation of the use of the format 1 LOAD command is:



The format of the format 1 LOAD command is:

```

LOAD uncontrolled-array-name WITH { 'value'
                                   array-name
                                   string-expression }
[ MASKED BY { 'bit-string'
             mask-name
             array-name
             logical-expression } ];

```

The parameters are:

uncontrolled-array-name

Name of an uncontrolled array defined by a DATA declaration whose elements are replaced with new value. It is the destination array.

An uncontrolled array is not referenced in a FOR ALL/EACH clause in a block. Because the array is uncontrolled, the LOAD command moves the new value into all the array elements whose corresponding control mask bit is set to 1 or T. This value replaces the original value of that element.

''value''

The LOAD command moves the value you specify into the elements of the destination array.

In the following example, the value of the element in the destination array is replaced by a new value.

```
TIMESW: DATA \D2\ ''21'';  
LOAD TIMESW WITH ''12'';
```

The implicit control mask for TIMESW is 1, so when the LOAD command references TIMESW, it replaces the value 21 with the value 12. TIMESW now contains the value 12.



The format of the EOF command is:

```
EOF;
```

If you omit the EOF command, the translator provides it and prints an error message on the source listing.

6.15. HOW TO START MESSAGES ON A NEW LINE ON SCREEN (NEWLINE)

The NEWLINE command is a formatting command. You use it to terminate the current line on the screen to begin a message on a new line. The message itself is specified by the DISPLAY or ENTER command and is concatenated on a single line until a NEWLINE command is encountered. At that point, a new line is started. You only use the NEWLINE command in the trunk or branches of a tree.

The format of the NEWLINE command is:

```
NEWLINE;
```

The following example illustrates the use of the NEWLINE command:

```
JOBNAME: DATA \A8\;
JOBIDENT: BRANCH
    IF LEVEL = '2' THEN DO;
        'THE NAME IS A 1 TO 8 ALPHANUMERIC';
        NEWLINE;
        'STRING, BEGINNING WITH AN ALPHANUMERIC';
        NEWLINE;
        'CHARACTER.';
        NEWLINE; NEWLINE;
    END;
    'ENTER NAME: '; ENTER JOBNAME;
END;
PRESENT TREE PARALLEL (JOBIDENT;);
```

When this branch is executed, the dialog user sees the following message displayed on the screen and also printed:

```
THE NAME IS A 1 TO 8 ALPHANUMERIC
STRING, BEGINNING WITH AN ALPHANUMERIC
CHARACTER.
ENTER NAME: _____
```

Notice that there is double spacing between the two messages since two NEWLINE commands were specified at that point.

6.16. HOW TO COPY SOURCE CODE FROM LIBRARY FILE INTO YOUR PROGRAM (COPY)

The COPY command saves programming time by allowing you to store commonly used coding in a library file and then reference it with a COPY command. The COPY command also permits you to support dialogs written in more than one language. You can use the COPY command anywhere in your program.

Use the COPY command to insert source code copied from a library file into your program. You can use COPY commands to seven levels; that is, the sixth level of source code copied from the library file can contain COPY commands.

You can reference COPY modules from no more than two libraries.

When the translator encounters a COPY command, all succeeding source lines are taken from the library file until the EOF command in that file is reached. Control then returns to the source line immediately following the COPY command last executed.

The format of the COPY command is:

```
COPY 'module-name';
```

The parameter is:

```
'module-name'
```

Name of a source library file module that contains source code written in the dialog specification language.

The module name you specify must adhere to the naming conventions defined by the library utilities. The library file name in which the module resides is specified by the PARAM COPY job control statement (8.2).

A graphic representation of the use of the COPY command:

PROGRAM BEFORE TRANSLATION	LIBRARY FILE 1	PROGRAM AFTER TRANSLATION
<pre> declarations . . . commands . . . COPY 'module-nameA'; . . . EOF;</pre>	<pre> module-nameA ARRAY1: DATA \A1\ 'X'; ARRAY2: DATA \A1\ 'Y'; ARRAY3: DATA \A1\ 'Z'; COPY 'module-nameB';</pre> <hr/> <pre> LIBRARY FILE 2 module-nameB ARRAYA: DATA \D1\ '1'; ARRAYB: DATA \D1\ '2';</pre>	<pre> 0001 declarations . . . 0010 commands . . . 0015 C10016 ARRAY1: DATA \A1\ 'X' C10017 ARRAY2: DATA \A1\ 'Y' C11018 ARRAY3: DATA \A1\ 'Z' C20019 ARRAYA: DATA \D1\ '1' C20020 ARRAYB: DATA \D1\ '2' 021 EOF;</pre>

When you process a module from one copy library (COP in the example), the nested copy module must come from the other copy library (SOURCE).

Example:

```
// JOB DSLTST,,12000,,,,J219
// DVC 20 // LFD PRNTR
// DVC 50 // VOL D01906 // LBL DPSRC // LFD SOURCE
// DVC 50 // VOL D01906 // LBL PMTRANS // LFD DSLTOUT
// DVC 50 // VOL D01906 // LBL DPSRC // LFD SRC
// DVC 50 // VOL D01906 // LBL DPSRC // LFD COP
// WORK1
// EXEC DSLT
// PARAM IN=CFILE1/SRC
// PARAM OUT=DSLTOU
// PARAM COPY=COP/SOURCE
/&
```

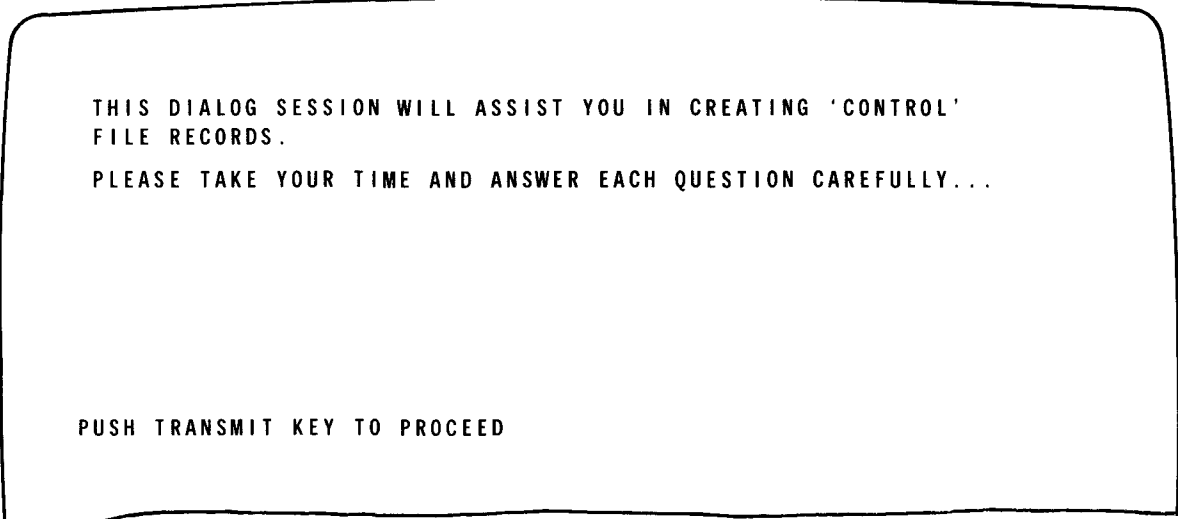


7. Sample Program

7.1. GENERATION OF DIALOG SCREENS

Figures 7-1 and 7-2 illustrate how dialogs are generated onto a screen from a sample program written in the dialog specification language. Figure 7-1 shows the dialog screen questions and the answers chosen for each question, and Figure 7-2 shows the program that produced the screens. The number of each screen in Figure 7-1 relates the screen to the corresponding coding elements in the program (Figure 7-2). The items in reverse type (white letters on black background) in Figure 7-1 indicate responses from the dialog user.

SCREEN 1



```
THIS DIALOG SESSION WILL ASSIST YOU IN CREATING 'CONTROL'  
FILE RECORDS.  
PLEASE TAKE YOUR TIME AND ANSWER EACH QUESTION CAREFULLY...  
  
PUSH TRANSMIT KEY TO PROCEED
```

Figure 7-1. Screen Displays for Sample Program (Part 1 of 10)

SCREEN 2

***** COMPANY ENVIRONMENT SPECIFICATIONS *****

SELECT THE COMPANY ENVIRONMENT FOR THIS USER FROM THE CHOICES BELOW:

1. A SINGLE COMPANY WITH NO DIVISIONS EXISTS FOR THIS USER.
2. A SINGLE COMPANY WITH MORE THAN ONE DIVISION EXISTS FOR THIS USER.
3. MORE THAN ONE COMPANY EXISTS FOR THIS USER.

*** NOTE: *** FOR MULTI-COMPANY ENVIRONMENTS, THE NUMBER OF DIVISIONS WILL BE SPECIFIED FOR EACH COMPANY AS THE COMPANY IS DEFINED.

SELECT ITEM BY ENTERING A NUMBER

SCREEN 3

***** COMPANY IDENTIFICATION *****

PLEASE ENTER THE APPROPRIATE INFORMATION BELOW:

ENTER THE COMPANY NAME AS IT SHOULD APPEAR ON REPORTS AND SCREENS:

ENTER THE 1 CHARACTER DEDUCTION CODE FOR BONDS FOR USE IN THE PAYROLL APPLICATION SYSTEM:

ENTER THE CHARACTER TO BE USED AS A NUMERIC DATE DELIMITER (I.E. '/', ':', OR '.'):

8. Job Control Statements

8.1. INTRODUCTION

After you write the dialog, using the commands and declarations of the dialog specification language, you enter the source program into a SAT library file. You can do this in either of two ways:

1. Using the workstation terminal with the general editor (EDT). For a complete explanation of job control, see the interactive job control user guide, UP-8822 (current version). For an explanation of the general editor (EDT), see the general editor (EDT) user guide/programmer reference, UP-8828 (current version).
2. Using diskette, punched cards, disk, or magnetic tape along with the appropriate job control language

In the dialog specification language, you use the characters `/*` to indicate the start of a comment. Since `/*` is also used by the job control language to indicate the end of file for user input, use `// OPTION EOD=/x` in the job control stream to change the end-of-file indicator. You can specify any character for `x`. For example, `// OPTION EOD=/A` establishes that `/A` is used to indicate the end of the file.

Library facilities and the general editor help you build and update the source programs stored in the library file.

8.2. PARAM STATEMENT

You use the PARAM job control statement to specify input, output, and COPY files used during the execution of the translator. You can use one PARAM statement to specify all three files, or you can use separate PARAM statements for each file.

The format of the PARAM statement is:

```
// PARAM IN=modulename [/filename]
      [,OUT=filename]
      [ ,COPY= { filename1[/filename2]
                { filename1/(N)
                  (N)/filename2
                  (N)
                }
            }
      [ ,SEQ={column-number}
            { 73
            }
      [ , {BACK STROKE}=character
        { BS
        }
      [ , {RIGHT BRACKET}=character
        { RB
        }
```

or:

```
// PARAM IN=modulename[/filename]
// PARAM OUT=filename
// PARAM COPY={filename1[/filename2]
              { filename1/(N)
                (N)/filename2
                (N)
              }
// PARAM SEQ={column-number}
            { 73
            }
// PARAM {BACK STROKE}=character
        { BS
        }
// PARAM {RIGHT BRACKET}=character
        { RB
        }
```

Here is what you need to know about the keyword parameters for the PARAM statement.

IN=modulename

→ You must use the IN parameter to specify the name of the source module that you want the translator to translate.

The translator searches the \$Y\$SRC library file for the input module you specify. You don't need to specify DVC and LFD statements for the \$Y\$SRC library file.

IN=modulename/filename

→ This specifies the name of the source module containing your code and the name of the source library file in which it resides.

The name of the source library file must be the same as the file name you specified in the LFD statement for the input file.

OUT=filename

↓

This specifies the name of the output MIRAM file you want the translator to create. (MIRAM is a disk access method that handles sequential, relative, and indexed files.) The translator automatically stores the translated dialog in the output MIRAM file.

↑

This file must be the same as the file name you specify in the LFD statement for the output file. If you omit this parameter, the translator doesn't create an output MIRAM file (an encoded object module file).

COPY=filename

This specifies the name of the first library file that stores modules to be copied by the COPY command.

First, the translator searches this library file for the specified module referenced in a COPY command in your source program. If the module is not there, the translator searches the \$Y\$SRC library file.

COPY=filename1/filename2

This specifies that the file specified by filename1 is to be searched first by the translator for source modules referenced in COPY commands.

Then, the file specified by filename2 is to be searched by the translator. The \$Y\$SRC library file is not searched by the translator.

COPY=filename1/(N)

Only the file specified by filename1 is searched by the translator for the module.

COPY=(N)/filename2

Only the file specified by filename2 is searched by the translator for the module.

COPY=(N)

No files, not even the \$Y\$SRC library file, are searched by the translator for the module. This implies that you didn't use the COPY command in your source program.

SEQ=column-number

The translator doesn't process the data from the column number you specify to the end of the line, but it does print those columns in the source listing.

SEQ=73

The translator doesn't process the data from column 73 to the end of the line, but it does print those columns in the source listing. This is the default value if you omit a column number.

BACK STROKE=character

This allows the translator to accept another character in place of the back stroke (\) character. You can't use any character allowed in the dialog specification language syntax (4.2) to replace the back stroke character. When you specify an invalid character, an error message occurs and the job is terminated. For example, **BACK STROKE=&** indicates that the ampersand character (&) will replace the back stroke character in the syntax.

RIGHT BRACKET=character

This allows the translator to accept another character in place of the right bracket (]) character. You can't use any character allowed in the dialog specification language syntax (4.2) to replace the right bracket character. When you specify an invalid character, an error message occurs and the job is terminated. For example, **RIGHT BRACKET=!** indicates that the exclamation character (!) will replace the right bracket character in the syntax.

8.3. SAMPLE JOB CONTROL STREAM TO EXECUTE THE TRANSLATOR

After the source program is stored in the SAT library file, you can compile it and execute the translator.

The following sample job control stream compiles the source program and executes the translator. Assume that the private disk volume D01234 contains a source program named DIALOG1 stored in an input source library named DLGLIBIN and that the translator uses an output MIRAM file named DLGLBOUT. Also assume that space for the output file is already allocated prior to the execution of the translator. The job control stream is:

```
1. // JOB COMPILE,,18000
2. // DVC 20 // LFD PRNTR
3. // DVC 51 // VOL D01234 // LBL DLGLIBIN // LFD DSLTIN
4. // DVC 51 // VOL D01234 // LBL DLGLBOUT // LFD DSLTOUT
5. // DVC 51 // VOL D01234 // LBL COPYFIL1 // LFD COPYLIB1
6. // DVC 51 // VOL D01234 // LBL COPYFIL2 // LFD COPYLIB2
7. // WORK1
8. // EXEC DSLT
9. // PARAM IN=DIALOG1/DSLTIN
10. // PARAM OUT=DSLTOU
11. // PARAM COPY=COPYLIB1/COPYLIB2
12. /&
```

Step by step, this means:

1. The name of the job is COMPILE. A main storage space of X'18000' allows you to run large dialogs. Smaller dialogs don't need main storage allocation.
2. This defines the printer file for output listings produced by the translator.
3. The input SAT source library has a file identifier of DLGLIBIN and a file name of DSLTIN. Notice that the file name must be the same as the file name specified in the PARAM IN statement described in 9. DLGLTBIN and DSLTOU are user-specified names so you can name them anything you want for these files.
4. The output MIRAM file created by the translator has a file identifier (LBL) of DLGLBOUT and a file name (LFD) of DSLTOU. Notice that the file name must be the same as the file name specified in the PARAM OUT statement described in 10. DLGLBOUT and DSLTOU are user-specified names so you can specify any name you want for these files.
5. The alternate source library file (from which input records are copied by the translator) has a file identifier (LBL) of COPYFIL1 and a file name (LFD) of COPYLIB1. Notice that the file name must be the same as the first file name specified in the PARAM COPY statement described in 11.
6. Another alternate source library file has a file identifier of COPYFIL2 and a file name of COPYLIB2. Notice that the file name must be the same as the second file name specified in the PARAM COPY statement described in 11.
7. A MIRAM work file for translator internal processing is allocated. This file is erased when the job step is completed.

8. The translator, which resides in the system load library, is executed.
9. The input source file has a source module name of DIALOG1 and a file name of DSLTIN (same as the file name in 3). ←
10. The output encoded MIRAM dialog file created by the translator has a file name of DSLTOUT (same as the file name in 4). If you omit this parameter, the translator does not create an output file. ←
11. The two alternate source library files have file names of COPYLIB1 and COPYLIB2. These are the files from which source modules are copied by the translator when you use a COPY command in your program.
12. This indicates the end of the job named COMPILE. ↓

You can't change the encoded dialog file directly. First, you must change your code and then retranslate it.

Only the dialog processor can access the encoded MIRAM file. ↑



C.6. DISPLAY COMMAND (DISPLAY MESSAGE)

```
[DISPLAY] { 'message'
            { array-name
            { string-expression }
            };
```

C.7. PRINT COMMAND (PRINT MESSAGE)

```
[PRINT] { 'message'
          { array-name
          { string-expression }
          };
```

C.8. OUTPUT COMMAND (STORE MESSAGE)

Format 1 (Store Dialog User Responses in Input Buffer):

```
OUTPUT { 'message'
         { array-name
         { string-expression }
         } [EOR];
```

Format 2 (Route Output File to Input Buffer and Return Control to Program):

```
OUTPUT EOR;
```

C.9. LOAD COMMAND (CHANGE ARRAY ELEMENT VALUE)

Format 1 (Change Element of Uncontrolled Array):

```
LOAD uncontrolled-array-name WITH { 'value'
                                    { array-name
                                    { string-expression }
                                    };

[MASKED BY { 'bit-string'
             { mask-name
             { array-name
             { logical-expression }
             }
             }];
```

Format 2 (Change Element of Controlled Array):

```
LOAD controlled-array-name WITH { 'value'
                                  { array-name
                                  { string-expression }
                                  };
```

C.10. SET COMMAND (CHANGE CONTROL MASK VALUES)

Format 1 (Change Control Mask to New Value):

```
SET { mask-name1
     { uncontrolled-array-name }
     TO { 'bit-string'
          { mask-name2
          { array-name
          { logical-expression }
          }
          };
```

Format 2 (Reset Control Mask to Initial Value):

```
SET mask-name TO INITIAL ;
```

Format 3 (Reset Array and Control Mask to Initial Value):

```
SET uncontrolled-array-name TO INITIAL;
```

C.11. MASK COMMAND (MASK A CONTROL MASK)

```
MASK { mask-name-1
      { uncontrolled-array-name } } WITH { 'bit-string'
      { mask-name2
      { array-name
      { logical-expression } } } ;
```

C.12. IF COMMAND (CONDITIONAL OPERATIONS)**Format 1 (Conditional Operations Outside Trees):**

```
IF { array-name
    { numeric-constant
    { conditional-expression } } { <
    { < =
    { =
    { < >
    { > =
    { > } } { array-name
    { numeric-constant
    { conditional-expression } } ;
```

```
THEN
```

```
{ CALL Command;
  DO...END Format 1 Command;
  IF Format 1 Command;
  LOAD Command;
  MASK Command;
  OUTPUT Command;
  PRESENT Command;
  SET Command;
ELSE
{ CALL Command;
  DO...END Format 1 Command;
  IF Format 1 Command;
  LOAD Command;
  MASK Command;
  OUTPUT Command;
  PRESENT Command;
  SET Command;
};
```


Conditional expression is the same as:

$$\text{conditional element} \left[\begin{array}{c} \text{AND} \\ \text{OR} \\ \text{XOR} \end{array} \right] \text{conditional element} \dots$$

Conditional element is the same as:

$$[\text{NOT}] \text{ expression} \left\{ \begin{array}{l} < \\ < = \\ = \\ < > \\ > = \\ > \end{array} \right\} \text{ expression}$$

Expression is the same as:

$$\text{term} \left[\begin{array}{c} + \\ - \end{array} \right] \text{term} \dots$$

Term is the same as:

$$\text{element} \left[\begin{array}{c} (\text{REM}) \\ (*) \\ (/) \end{array} \right] \text{element} \dots$$

Element is the same as:

$$[-] \left\{ \begin{array}{l} \text{array-name} \\ \text{numeric-constant} \\ (\text{expression}) \end{array} \right\}$$

We'll briefly cover these parameters.

array-name

Array defined by a DATA declaration.

numeric-constant

Positive integer.

REM

An operator that stands for remainder.

An operator that stands for multiplication.

/

An operator that stands for division.

AND

See D.2 for an explanation.

- OR
See D.2 for an explanation.
- XOR
See D.2 for an explanation.
- <
An operator that stands for less than.
- <=
An operator that stands for less than or equal to.
- =
An operator that stands for equal to.
- #
An operator that stands for not equal to.
- >=
An operator that stands for greater than or equal to.
- >
An operator that stands for greater than.

An example of a command that uses a conditional expression is:

```
IF DATA1*25+DATA2 = DATA3/DATA4 THEN
  DO UNTIL FLAG1 < FLAG2;
    CALL TIME;
    LOAD FLAG1 WITH FLAG1-1;
  END;
ELSE CALL MOD;
```

In this example, the IF command uses a conditional expression. DATA1, DATA2, DATA3, and DATA4 are arrays, and 25 is a numeric constant. This conditional expression is evaluated by first computing the term DATA1*25+DATA2 and then computing the term DATA3/DATA4 and then comparing the two. If the result is true, then the block formed by the DO...END command is executed until the condition FLAG1 < FLAG2 is true. If the result of the IF command is false, the CALL MOD command is executed.

The following example shows another command that uses a conditional expression:

```
IF X=Y+Z AND A=B OR A=C THEN
  PRESENT COMPTREE;
```

D.4. STRING EXPRESSIONS

You use string expressions in PRINT, DISPLAY, OUTPUT, and LOAD commands mainly to concatenate arrays and messages onto one line so that one command can process the data. You must enclose the concatenated string expression within brackets and separate them by a comma or a space.

String expression is the same as:

```
expression [format description]
```

Expression is the same as:

```
term[ {+} term ] ...
```

Format description is the same as:

$$\left\{ \begin{array}{l} \text{A size} \\ \text{B size} \\ \text{X size} \\ \text{D size[.size]} \end{array} \right\} \left[\begin{array}{l} \text{C} \\ \text{L} \\ \text{R} \\ \text{S} \\ \text{Z} \end{array} \right] \backslash$$

Term is the same as:

```
element [ { (REM) } element ] ...
```

Element is the same as:

$$[-] \left\{ \begin{array}{l} \text{array-name} \\ \text{numeric-constant} \\ \text{(expression)} \end{array} \right\}$$

We'll cover the *format description* parameter here:

```
format description
```

You use the format description to assign a data type, size, and editing rule to an array referenced in the string expression. This format description overrides any format description you specified for the array in its DATA declaration. If you use a format description with a concatenated string expression, concatenation is performed before the formatting occurs, and then the data is processed in the format you define.

See D.3 for an explanation of other parameters.

An example of a string expression used in a command is:

```
DAY: DATA \A3\ 'MON';
SPASE: DATA \A2\ ' ';
MONTH: DATA \A3\ 'JAN';
OUTPUT [DAY, SPASE \ A4 \ , MONTH, '77'];
```

The OUTPUT command uses a string expression to concatenate arrays and a message on one line. The message MON JAN77 is written to the output file. Notice that the format description used with SPASE in the OUTPUT command overrides the format description used in the DATA declaration.

Other examples are:

```
OUTPUT ['ADDRESS:', STNAME, CITYNAME, STATE, ZIP] EOR;
PRINT [ID, LABEL2, 'OTHERS', HOURS1 \D3.2R\];
PRINT DATA12 \A10L\ ;
```

D.5. EXPRESSION

You use *expression* in the TIMES and CASE clauses in a block to create a number that affects the execution of the block.

Expression is the same as:

$$\text{term} \left[\left\{ \begin{array}{l} + \\ - \end{array} \right\} \text{term} \right] \dots$$

Term is the same as:

$$\text{element} \left[\left\{ \begin{array}{l} (REM) \\ (\cdot) \\ (/) \end{array} \right\} \text{element} \right] \dots$$

Element is the same as:

$$[-] \left\{ \begin{array}{l} \text{array-name} \\ \text{numeric-expression} \\ (\text{expression}) \end{array} \right\}$$

For an explanation of these parameters, see D.3.

An example of a command that uses an expression is:

```
DO CASE MOD+LEVEL;
  CALL ARRAY1;
  CALL ARRAY2;
  CALL ARRAY3;
END;
```

The CASE clause in a block uses an expression. The arrays in the expression MOD+LEVEL are evaluated to obtain a number. This number determines which of the CALL commands in the block are executed. If the result is 1, the first CALL command is executed. If the result is 2, the second CALL command is executed, and so on.

Index

Term	Reference	Page	Term	Reference	Page
A			B		
Array			Bit string		
arrays referenced in parallel	See arrays referenced in parallel.		control masks	5.3	5—16
controlled arrays	See controlled arrays.		description	4.6	4—15
description	4.6	4—14	Blocks		
editing rules	See editing rules.		branch	6.2.3	6—7
illustration	Fig. 4—9	4—14	description	4.4	4—3
implicit control mask	4.6	4—15	global declarations and commands	4.4.3	4—6
initialization of array values	See initialization of array values.		illustration	Fig. 4—1	4—4
logical AND operation	See logical AND operation.		local declarations and commands	4.4.2	4—5
message arrays	5.2.2.	5—12	named	5.4	5—18
reinitialization of array values	See reinitialization of array and control mask values.		nested	4.4.1	4—4
uncontrolled arrays	See uncontrolled arrays.		reinitialization of array and control mask values	See reinitialization of array and control mask values.	
uncontrolled arrays referenced in trees	See uncontrolled arrays.		trunk	6.2.2	6—6
variable arrays	5.2.1	5—2	unnamed	6.2	6—4
Array control mask	See control mask.		Branch		
Arrays referenced in parallel			available branches	5.5	5—26
description	4.6.4	4—19	choice mode	5.5	5—26
illustration	Fig. 4—12	4—20	chosen branches	5.5	5—26
Available branches	5.5	5—26	description	4.5	4—8
			display mode	5.5	5—26
			expansion of array elements	4.6.3	4—18
			implied branches	4.6.2	4—17
			inline branches	5.5	5—26
			rejected branches	5.5	5—26
			separate named branches	See separate branches.	

Term	Reference	Page	Term	Reference	Page
Branch blocks	6.23	6—7			
BRANCH...END declaration					
description	5.6	5—49			
separate branches containing nested trees	5.6.2	5—51			
simple separate branches	5.6.1	5—49			
			C		
			CALL command	6.3	6—8
			CASE clause	5.4	5—22
			Character set	4.2	4—2
			Choice mode	5.5	5—26
			Chosen branches	5.5	5—26
			Coding rules		
			colon	3.4	3—2
			columns	3.2	3—1
			comma	3.4	3—2
			comments	3.7	3—3
			description	3.1	3—1
			lines	3.3	3—1
			punctuation	3.4	3—2
			quotation marks	3.6	3—3
			semicolon	3.4	3—2
			space	3.5	3—2
			Commands		
			CALL	6.3	6—8
			commands used in tree structures	Fig. 6—1	6—3
			COPY	6.16	6—48
			description	4.3	4—2
			DISPLAY	6.7	6—17
			DO...END	6.2	6—4
			EJECT	6.17	6—49
			ENTER	6.6	6—15
			EOF	6.14	6—46
			IF	6.13	6—40
			LOAD	6.10	6—26
			MASK	6.12	6—36
			NEWLINE	6.15	6—47
			OUTPUT	6.9	6—23
			PRESENT	6.5	6—13
			PRINT	6.8	6—20
			Sequence of execution	6.5	6—13
			SET	6.11	6—30
			SPACE	6.18	6—50
			summary	Appendix C	
			TITLE	6.19	6—51
			TREE	6.4	6—9
			use	Table 6—1	6—2
			Concatenated control mask	4.63	4—18
			Conditional expressions	D.3	D—2

Term	Reference	Page	Term	Reference	Page
M			O		
MASK command	6.12	6—36	OUTPUT command		
MASK declaration	5.3	5—15	description	6.9	6—23
MASKED BY clause			results of OUTPUT command	7.3	7—28
blocks	5.4	5—24	returning control to program	6.9.2	6—25
LOAD command	6.10.1	6—28	Output listing file	1.3	1—7
PRESENT command	6.5	6—14			
trees	5.5	5—26			
MEANS declaration	5.7	5—53			
Message arrays	5.2.2	5—12			
MIRAM files	8.2	8—2			
N			P		
Nested blocks	4.4.1	4—4	PARALLEL node	5.5.1	5—30
Nested trees			PARAM statement		
description	4.5.2	4—12	BACK STROKE	8.2	8—3
illustration	Fig. 4—8	4—13	COPY keyword	8.2	8—3
named trees	5.5.2	5—42	description	8.2	8—1
unnamed trees	6.4.2	6—11	IN keyword	8.2	8—2
NEWLINE command	6.15	6—47	OUT keyword	8.2	8—2
Nodes			RIGHT BRACKET	8.2	8—3
description	4.5	4—8	SEQ keyword	8.2	8—3
EXCLUSIVE	5.5	5—26	PRESENT command	6.5	6—13
INCLUSIVE	5.5.1	5—28	PRINT command	6.8	6—20
PARALLEL	5.5.1	5—29			
SEQUENTIAL	5.5.1	5—30			
	5.5.1	5—31			

USER COMMENT SHEET

Your comments concerning this document will be welcomed by Sperry Univac for use in improving subsequent editions.

Please note: This form is not intended to be used as an order blank.

(Document Title)

(Document No.)

(Revision No.)

(Update No.)

Comments:

Cut along line.

From:

(Name of User)

(Business Address)

Fold on dotted lines, and mail. (No postage stamp is necessary if mailed in the U.S.A.)
Thank you for your cooperation

FOLD



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

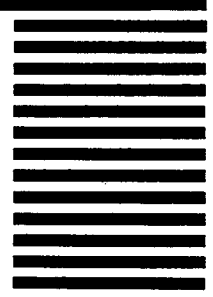
FIRST CLASS PERMIT NO. 21 BLUE BELL, PA.

POSTAGE WILL BE PAID BY ADDRESSEE

SPERRY UNIVAC

ATTN.: SYSTEMS PUBLICATIONS

P.O. BOX 500
BLUE BELL, PENNSYLVANIA 19424



CUT

FOLD