Burroughs

Reference Manual

A Series ALGOL

(Relative to the Mark 3.6 System Software Release)

Distribution Code SE

Priced Item
Printed in U.S.A.
December 1985

1169844

Burroughs cannot accept any financial or other responsibilities that may be the result of your use of this information or software material, including direct, indirect, special or consequential damages. There are no warranties extended or granted by this document or software material.

You should be very careful to ensure that the use of this software material and/or information complies with the laws, rules, and regulations of the jurisdictions with respect to which it is used.

The information contained herein is subject to change without notice. Revisions may be issued to advise of such changes and/or additions.

Comments or suggestions regarding this document should be submitted on a Field Communication Form (FCF) with the Class specified as "2" (System Software), the Type specified as "1" (F.T.R.), and the Product specified as the seven-digit form number of the manual (for example, "1169844"). The FCF should be sent to the following address:

Burroughs Corporation
PA&S/Orange County
19 Morgan
Irvine, CA 92718

CONTENTS

1 INTRODUCTION

Purpose of this Manual

This reference manual is intended for use by the programmer who is familiar with ALGOL. Both the organization and the presentation of the material are designed to supply answers to well-conceived questions regarding the syntax, semantics, and pragmatics of ALGOL as implemented on A Series systems. This manual is not a tutorial text.

Burroughs Extended ALGOL

Burroughs Extended ALGOL is a high-level, structured programming language designed for A Series and B 5000/B 6000/B 7000 Series systems. In addition to implementing virtually all of ALGOL 60, Burroughs Extended ALGOL has provisions for communication between programs and input/output (I/O) devices, the editing of data, and the implementation of diagnostic facilities for program debugging.

The Structure of ALGOL

The fundamental constituents of ALGOL are the language components. These are the building blocks of the language and include, among other things, letters, digits, and special characters such as the semicolon (;).

At a level of complexity higher than language components are declarations, statements, and expressions. These are the building blocks of ALGOL programs. A declaration associates identifiers with specific properties. For example, an identifier can be associated with the properties of a real number. A statement indicates an operation to be performed, such as the assignment of a numerical value to an array element or the transfer of program flow to a location in the program out of the normal sequence. An expression describes operations that are performed on specified quantities and return a value. For example, the expression "SQRT(100)" returns 10.0, the square root of 100.

Note: Burroughs Extended ALGOL is based on the "Revised Report on the Algorithmic Language ALGOL 60" (Communications of the ACM, Vol. 6, No. 1; January, 1963).

At the highest level are program units. A program unit is any group of ALGOL constructs that can be compiled as a whole by the ALGOL compiler. An ALGOL program is, by definition, a program unit.

This manual describes the language components, declarations, statements, expressions, and program units of Burroughs Extended ALGOL. Unless otherwise stated, the word ALGOL refers to Burroughs Extended ALGOL.

ORGANIZATION OF THIS MANUAL

The earlier chapters describe the fundamentals of ALGOL: the structure of programs and the basic components of the language. The middle chapters describe the major constructs of ALGOL: declarations, statements, and expressions. The later chapters describe topics related to compiling ALGOL programs, and interfaces between ALGOL and other facilities such as libraries and Data Management System II (DMSII). The appendixes contain reference information about reserved words and about the format used internally to store data.

The manual contains the following chapters and appendixes.

Chapters

2 PROGRAM STRUCTURE

This chapter defines the basic structure of an ALGOL program and the scope of variables.

3 LANGUAGE COMPONENTS

This chapter defines the most elemental constructs in the ALGOL language.

4 DECLARATIONS

This chapter defines the constructs that establish data structures in an ALGOL program and associate identifiers with those data structures. These constructs are ordered alphabetically by declaration name.

5 STATEMENTS

This chapter defines the constructs that describe operations to be performed in an ALGOL program. These constructs are ordered alphabetically by statement name.

6 EXPRESSIONS

This chapter defines the constructs used to describe operations that are performed on specified quantities and return a value. The first part of the chapter describes the types of expressions. These types are ordered alphabetically by expression name. The second part of the chapter describes functions that are intrinsic to ALGOL. These functions are ordered alphabetically by their names.

7 COMPILING PROGRAMS

This chapter describes the various input and output files used by the ALGOL compiler and the compiler control options that control the compiler's processing of ALGOL source input.

8 INTERFACE TO THE LIBRARY FACILITY

This chapter describes library creation, use, sharing, and initiation.

9 DMSII INTERFACE

This chapter describes the ALGOL interface with Data Management System II (DMSII).

10 COMPILE-TIME FACILITY

This chapter describes how ALGOL source data can be compiled conditionally and iteratively.

11 BATCH FACILITY

This chapter describes how the cost of system overhead may be reduced by grouping programs together in a single run.

Appendixes

A RESERVED WORDS

This appendix lists the identifiers that need not be declared in an ALGOL program before they are used, if they appear in recognized contexts.

B DATA REPRESENTATION

This appendix describes the internal form of the various operands, the descriptor, the pointer, and the various character sets.

C RUN-TIME FORMAT-ERROR MESSAGES

This appendix interprets the error numbers given at run time when an error occurs in a READ or WRITE statement.

An explanation of railroad diagrams, a glossary, and an index appear at the end of this manual.

Using the Manual

The chapters that describe declarations, statements, and expressions are each divided into sections describing constructs appropriate to the chapter. In the table of contents, sections are indicated by the major unnumbered subheadings under the chapter titles. Within the chapters that describe declarations and statements, and within the first half of the chapter on expressions, the sections are ordered alphabetically and follow the same general format:

- The syntax for the construct is presented in a railroad diagram. This is a diagrammatic description of the acceptable ways of using the construct in a program. For those unfamiliar with railroad diagrams, a description of how they are read can be found in "Understanding Railroad Diagrams."

- Under the heading "Semantics" appears a description of the function of the construct. If further explanation of the syntax is required, it occurs here.

- Under the heading "Pragmatics" appears information about the implementation of the construct on A Series systems.

- Examples of use of the construct usually appear at the end of the section.

The chapter on expressions is divided into two sub-chapters. The first sub-chapter describes the types of expressions. The second sub-chapter describes functions that are intrinsic to ALGOL. The two sub-chapters are ordered alphabetically by expression name and function name, respectively. In the sub-chapter on functions, the syntax for each function is presented, and the action of the function and data type returned by the function are given.

Finding Information

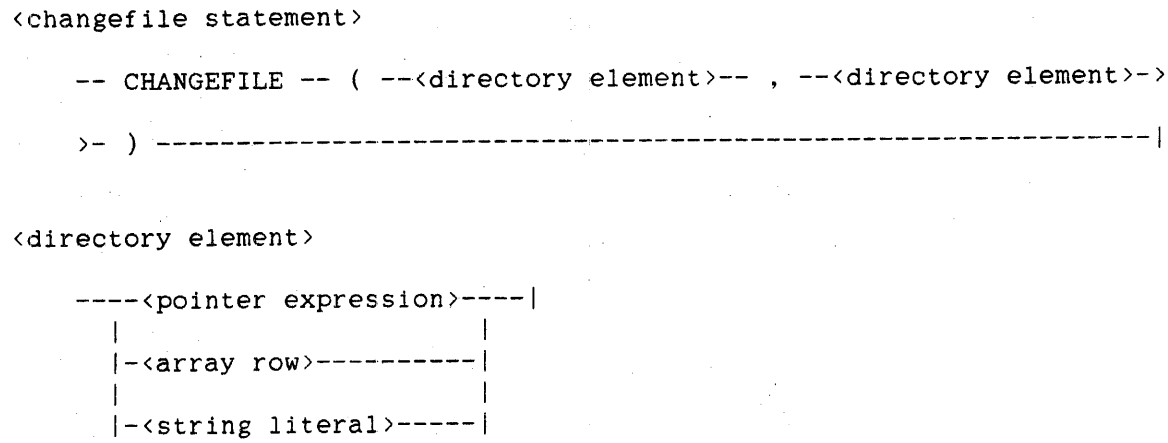
Because of the alphabetical ordering of the chapters on declarations, statements, and expressions, the reader can quickly find the description of any of those language constructs.

While using the manual, the reader will find "See also" references. These point to information related to the subject under discussion. "See also" references do not include references to whole chapters or to alphabetically ordered sections. These can easily be found using the chapter tabs and the section headings that appear at the top of each page in the alphabetized chapters. Neither do "See also" references include the fundamental language constructs that appear in the "Language Components" chapter. It is assumed that a reader will have an

understanding of the contents of that chapter before trying to use the rest of the manual.

"See also" references do include metatokens (language constructs) below the level of declarations, statements, and expressions that appear in a section but that are not defined in that section. "See also" references also point the reader to other, less easily found, related information.

For example, the following is the syntax for the CHANGEFILE statement, a statement used to change the names of directories and files without opening the files:



See also
<array row> 43

A cross reference to other syntax appears only for <array row>. It is assumed the reader is familiar with the syntax for <string literal> because it is a basic construct. The syntax for <pointer expression> can be found quickly by using the index tab for the "Expressions" chapter, and then by paging through the alphabetical listing of expressions to the "Pointer Expression" section.

RELATED DOCUMENTS

Document	Form No.
ALGOL Test and Debug System (TADS) User's Guide	1169539
Binder Reference Manual	5014582
CANDE Reference Manual	1169869
DMSII DASDL Reference Manual	1163805
DMSII Transaction Processing System (TPS) Programmer's Manual	1164043
DMSII User Language Interface Software Operation Guide	1180536
Editor User's Guide	1169976
I/O Subsystem Reference Manual	1169984
Message Translation Utility User's Guide	1169554
Operator Display Terminal (ODT) Reference Manual	1169612
System Software Utilities Reference Manual	1170024
Work Flow Language (WFL) Reference Manual	1169802

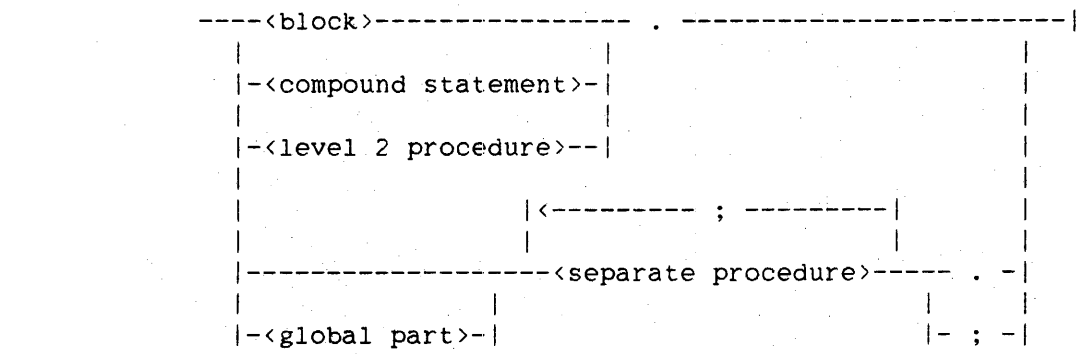
2 PROGRAM STRUCTURE

PROGRAM UNIT

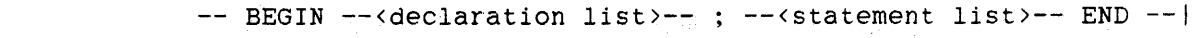
A program unit is a group of ALGOL constructs that can be compiled as a whole.

Syntax

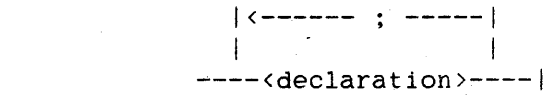
<program unit>



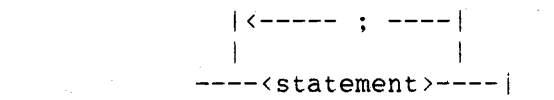
<block>



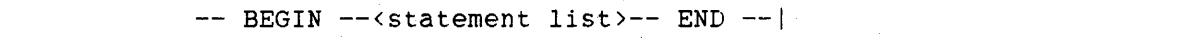
<declaration list>



<statement list>



<compound statement>



<level 2 procedure>

--<procedure declaration>--|

<global part>

-- [--<declaration list>--] --|

<separate procedure>

--<procedure declaration>--|

Semantics

Program units can be blocks, compound statements, level 2 procedures, or separate procedures that have a lexical (lex) level of three or greater and that can have global declarations.

A block is a statement that groups one or more declarations and statements into a logical unit by using a BEGIN-END pair. A compound statement is a statement that groups one or more statements into a logical unit by using a BEGIN-END pair. A compound statement is a block without any declarations.

The definitions of a compound statement and a block are recursive: both compound statements and blocks are made, in part, of statements. A statement can itself be a compound statement or a block.

The structures of compound statements and blocks are illustrated below.

Compound Statements

```
BEGIN
   <statement>;
   <statement>;
        .
        .
        .
   <statement>;
END
```

```
BEGIN
   <statement>;
   <statement>;
   BEGIN
      <declaration>;
      BEGIN
         <statement>;
         <statement>;
      END;
   END;
   <statement>;
END
```

Blocks

```
BEGIN
   <declaration>;
   <declaration>;
         .
         .
         .
   <declaration>;
   <statement>;
   <statement>;
         .
         .
         .
   <statement>;
END
```

```
BEGIN
   <declaration>;
   <declaration>;
   <statement>;
   BEGIN
      <declaration>;
      <statement>;
   END;
   BEGIN
      <statement>;
      <statement>;
      <statement>;
   END;
END
```

A program unit that is a separate procedure is typically bound to a host program to produce a more complete program.

The <global part> construct allows global identifiers to be referenced within a separate procedure. Any program unit that has a global part is valid only for binding to a host.

A program unit can be preceded, but not followed, by a remark.

Pragmatics

A compound statement is executed in-line and does not require a procedure entrance and exit. A block, however, is executed like a procedure and requires a procedure entrance and exit. Entering a block costs extra processor resources; entering a compound statement does not.

Examples

Compound Statement

```
BEGIN
   DISPLAY("HI THERE");
   DISPLAY("THAT'S ALL FOLKS");
END.
```

Block

```
BEGIN
   REAL X;
   X := 100;
END.
```

Level 2 Procedure

```
PROCEDURE S;
BEGIN
   REAL X;
   X := SQRT(4956);
END.
```

Separate Procedure with Global Part

```
[REAL S;
 ARRAY B[0:255];
 FILE LINE;]
REAL PROCEDURE Q;
BEGIN
   Q := S*B[4];
   WRITE(LINE,/,"DONE");
END.
```

Note that, according to the syntax, the last statement of a block or compound statement is not followed by a semicolon (;). However, in the above examples (and throughout this manual), the last statement is always followed by a semicolon. This is valid because the statement before the END is the "null statement."

SCOPE

The scope of an identifier is defined to be the portion of an ALGOL program in which the identifier can successfully be used to denote its corresponding values and characteristics.

In one part of an ALGOL program, an identifier can be used to denote one set of values and characteristics, while in another part of the program, the same identifier can be used to denote a different set of values and characteristics.

For example, in one block the identifier EXAMPLE_IDENT can be declared as a REAL variable. That is, the identifier can be used to store single-precision, floating-point arithmetic values. Such an identifier could be assigned the value "3.14159". In another block of the same program, EXAMPLE_IDENT can be declared as a STRING variable. In this block, EXAMPLE_IDENT could be assigned the value "BURROUGHS ALGOL IS A HIGH-LEVEL, BLOCK-STRUCTURED LANGUAGE".

Although EXAMPLE_INDENT can be of type real and of type string in the same program, within a specific block, EXAMPLE_IDENT has only one type associated with it. In general, the scope of an identifier is always such that within a given block, the identifier has associated with it at most one set of values and characteristics.

The scope of an identifier is described by rules that define which parts of the program are included by the scope, which parts of the program are excluded by the scope, and the requirements for uniqueness placed on the choice of identifiers. These general rules are described in the following paragraphs.

Local Identifiers

An identifier that is declared within a block is referred to as "local" to that block. The value or values associated with that identifier inside the block are not associated with that identifier outside the block. In other words, on entry to a block, the values of local identifiers are undefined; on exit from the block, the values of local identifiers are lost. An identifier that is local to a block is "global" to blocks occurring within the block. When a block is exited, identifiers that are global to that block do not lose the the values associated with them. The properties of global identifiers are described more completely below.

Global Identifiers

An identifier that appears within a block and that is not declared within the block, but is declared in an outer block, is referred to as "global" to that block. A global identifier retains its values and characteristics as the blocks to which it is global are entered and exited.

As the following program illustrates, an identifier can be local to one block but global to another block.

```
BEGIN
   FILE PRTR(KIND = PRINTER);
   REAL A;
   A := 4.2 @ -1; % FIRST STATEMENT OF OUTER BLOCK
   BEGIN
      LIST L1 (A);
      INTEGER A,
      LIST L2 (A);
      A := 3; % FIRST STATEMENT OF INNER BLOCK
      WRITE (PRTR, */, L1);
      WRITE (PRTR, */, L2);
   END; % OF INNER BLOCK
   A := A*A;
   WRITE (PRTR, */, A);
END. % OF PROGRAM
```

In the preceding example, the identifier A that is declared REAL is global to the inner block. The A declared as type INTEGER in the inner block is local to the inner block, so when the inner block is exited, the integer A and its value, 3, are lost. Within the scope of integer A, a reference to A is a reference to the integer A, not to the global, real A. At the time the declaration for list L1 is compiled, the declaration for local A has not been seen, so list L1 contains the global, real A. However, the list L2 contains the local, integer A. The A referenced in the outer block is the A that was declared REAL and assigned the value 4.2 @ -1. The result of the first WRITE statement is "A=0.42". The result of the second WRITE statement is "A=3". The result of the third WRITE statement is "A=0.1764", which equals 4.2 @ -1 * 4.2 @ -1.

Global identifiers are used in inner blocks for the following reasons:

1. To carry values that have been calculated in an outer block into the inner block

2. To carry a value calculated inside the block to an outer block

3. To preserve a value calculated within a block for use in a later entry to the same block

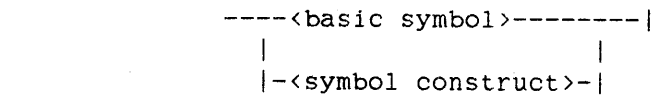
4. To transmit a value from one block to another block that does not contain and is not contained by the first block

3 LANGUAGE COMPONENTS

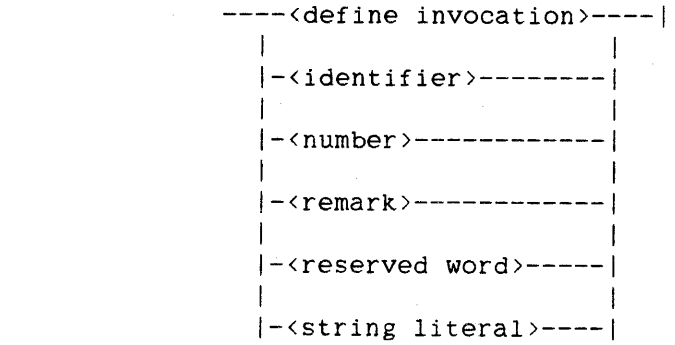
Language components are the building blocks of ALGOL. They consist of basic symbols, such as digits and letters, and symbol constructs, which are those groups of basic symbols that are recognized by the ALGOL compiler.

Syntax

<language component>



<symbol construct>



Semantics

Basic symbols, identifiers, numbers, remarks, and string literals are described under separate headings in this chapter.

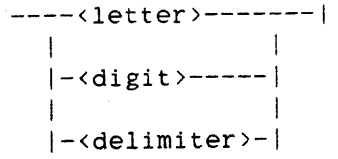
Because the define invocation is closely linked to the DEFINE declaration, the define invocation is explained under "DEFINE Declaration" in the chapter "Declarations."

Reserved words are described and listed in the appendix "Reserved Words."

BASIC SYMBOL

Syntax

<basic symbol>



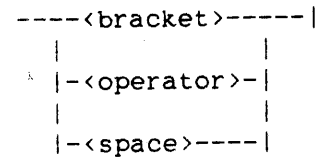
<letter>

Any one of the uppercase (capital) letters A through Z.

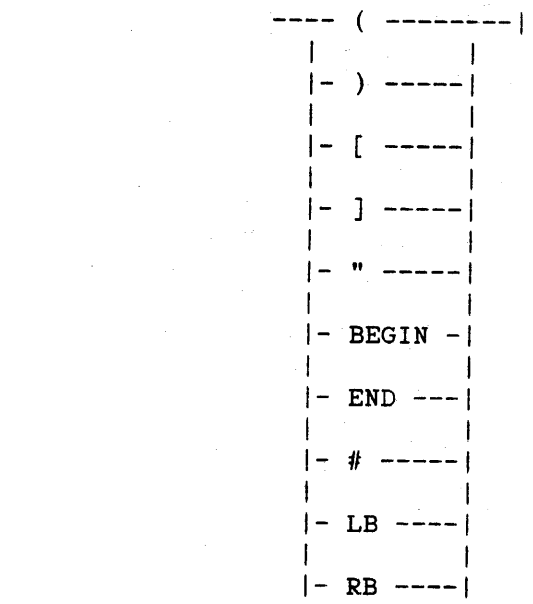
<digit>

Any one of the Arabic numerals 0 through 9.

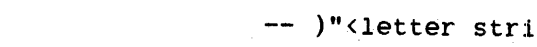
<delimiter>



<bracket>



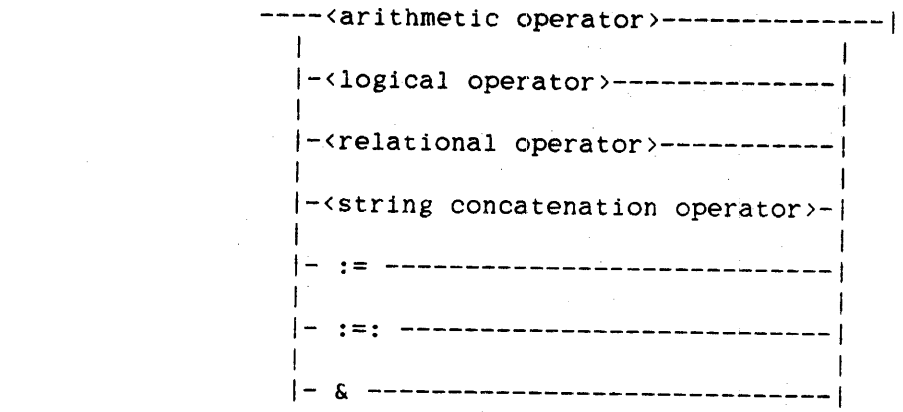
<parameter delimiter>



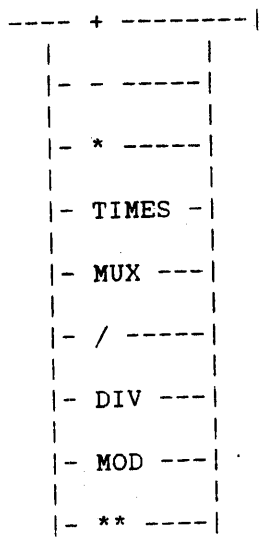
<letter string>

Any character string not containing a quotation mark (").

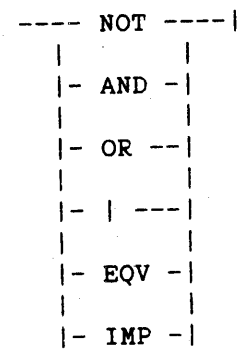
<operator>



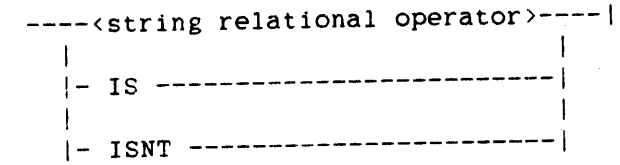
<arithmetic operator>



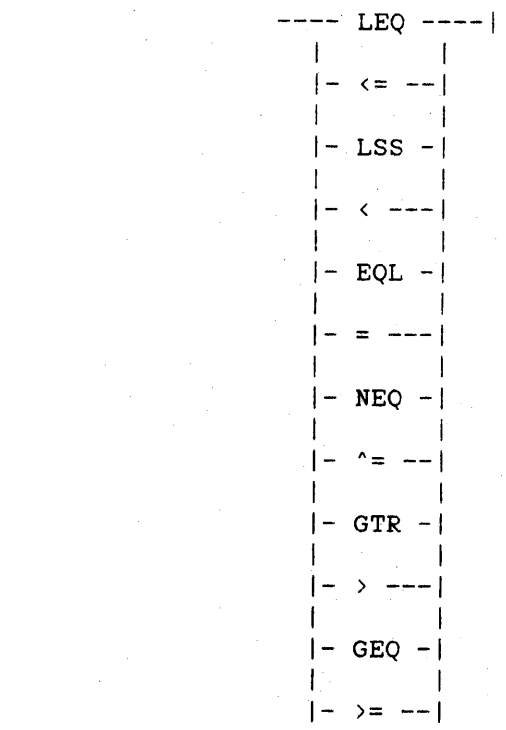
<logical operator>



<relational operator>

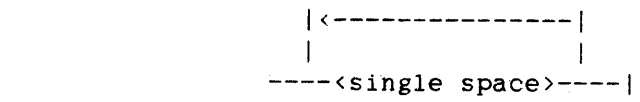


<string relational operator>



<string concatenation operator>

<space>



<single space>

One blank character.

Semantics

Only uppercase letters are permitted. Lowercase letters are specifically disallowed. Individual letters do not have particular meanings except as used in pictures and formats.

Digits are used to form numbers, identifiers, and string literals.

Delimiters include operators, spaces, and brackets. An important function of these elements is to delimit the various entities that make up a program. Each delimiter has a fixed meaning, which, if not obvious, is explained elsewhere in this manual in the syntax of appropriate constructs. Basic symbols that are words, such as some delimiters and operators, are reserved for specific use in the language. A complete list of these words, called reserved words, and details of the applicable restrictions are given in the appendix "Reserved Words."

In ALGOL 60, spaces have no significance, because such language components as BEGIN are construed as one basic symbol. However, in a machine implementation of such a language, this approach is not convenient for programmers. In ALGOL, for example, BEGIN is composed of five letters, TRUE of four letters, and PROCEDURE of nine letters. No space can appear between the letters of a reserved word; otherwise, the reserved word is interpreted as two or more elements.

Reserved words and basic symbols are used, together with variables and numbers, to form expressions, statements, and declarations. Because some of these constructs place programmer-defined identifiers next to delimiters composed of letters, these identifiers and delimiters must be separated. Therefore, a space must separate any two language components of the following forms:

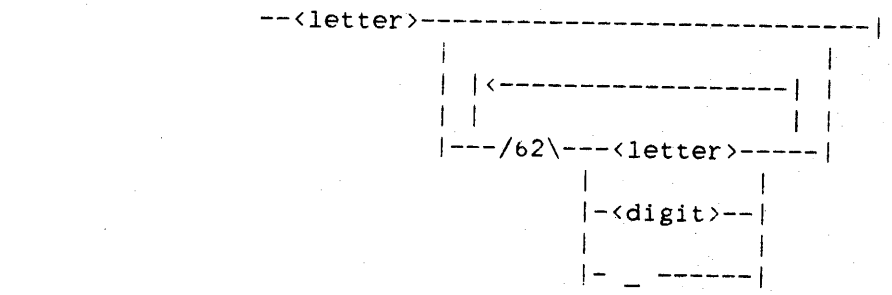
1. Delimiter composed of letters
2. Identifier
3. Boolean value
4. Unsigned number

Aside from these requirements, the use of a space between any two language components is optional. The meanings of the two language components are not affected by the presence or absence of the space.

IDENTIFIER

Syntax

<identifier>



Semantics

Identifiers have no intrinsic meaning. They are names for variables, arrays, procedures, and so forth. An identifier must start with a letter, which can be followed by any combination of letters, digits, and underscore characters (_).

Pragmatics

The scopes of identifiers are described in "Scope" in the chapter "Program Structure."

See also
Scope 12

Examples

Valid Identifiers

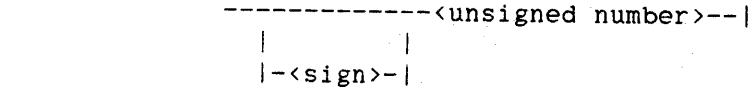
```
A
I
B5
YSQUARE
EQUITY
RETURN_RATE
D2R271GL
TEST_1
```

Invalid Identifiers	Reason
1776	Does not begin with a letter
2BAD	Does not begin with a letter
$	"$" not an allowed character
X-Y	"-" not an allowed character
NET GAINS	Blank spaces not allowed
NO.	"." not an allowed character
_TEST	Does not begin with a letter
BEGIN	Reserved word

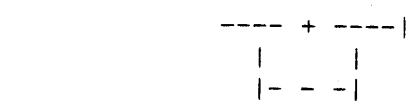
NUMBER

Syntax

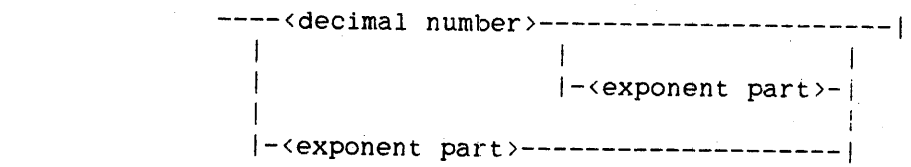
<number>



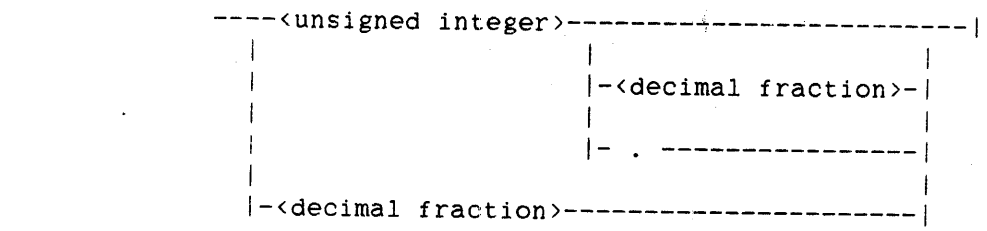
<sign>



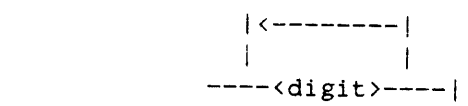
<unsigned number>



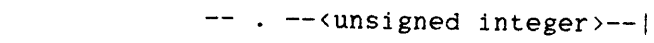
<decimal number>



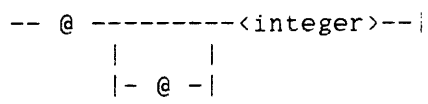
<unsigned integer>



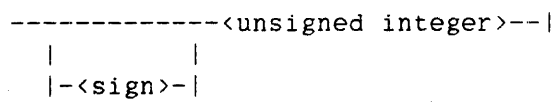
<decimal fraction>



<exponent part>



<integer>



Semantics

No space can appear within a decimal number. All numbers that do not contain the double-precision exponent delimiter "@@" are considered to be single-precision numbers.

Examples

Unsigned Integers	Decimal Fractions	Decimal Numbers
5	.5	69.
69	.69	.546
	.013	3.98
		25

Integers	Exponent Parts	Unsigned Numbers
1776	@8	99.44
-62256	@-06	@-11
+548	@+54	1354.543@48
	@@16	.1864@4

Valid Numbers

```
0
+545627657893
1.75@-46
-4.31468
-@2
.375
```

Invalid Numbers	Reason
50 00.5@8 8	Blanks spaces are not allowed.
1,505,278	Commas are not allowed.
@63.4	Exponent part must be an integer.
1.667E-01	"E" is not allowed for exponent part.

Number Ranges

The sets of numbers that can be represented in ALGOL are symmetrical with respect to zero; that is, the negative number corresponding to any valid positive number can also be expressed in the language and the object program.

The largest and smallest integers and numbers that can be represented are as follows (decimal versions are approximate):

1. Any integer between plus and minus 549755813887 = 8**13 - 1 = 4"007FFFFFFFFF", inclusive, can be represented in integer form.

2. For single-precision numbers:

   a. The largest, positive, normalized, single-precision number that can be represented is 4.31359146674@68 = (8**13 - 1) * 8**63 = 4"1FFFFFFFFFFF".

   b. The smallest, positive, normalized, single-precision number that can be represented is 8.75811540204@-47 = 8**(-51) = 4"3F9000000000".

   Zero and numbers with absolute values between the largest and smallest values given above can be represented as single-precision real numbers.

3. For double-precision numbers:

   a. The largest, positive, normalized, double-precision number that can be represented is 1.9488283820502807912446 7@@29603 = (8**13 - 8**(-13)) * 8**32767 = 4"1FFFFFFFFFFFFFFFFFFFFFFF".

   b. The smallest, positive, normalized, double-precision number that can be represented is 1.9385458571375858335564@@-29581 = 8**(-32755) = 4"3F9000000000FF8000000000".

   Zero and numbers with absolute values between the largest and smallest values given above can be represented as double-precision numbers.

Compiler Number Conversion

The ALGOL compiler can convert into internal format a maximum of 24 significant decimal digits of mantissa in double precision. The "effective exponent," which is the explicit exponent value following the "@@" sign minus the number of digits to the right of the decimal point, must be less than 29604 in absolute value. For example, the final fractional zero cannot be specified in the smallest, positive, normalized, double-precision number shown above: -29581 - (23 fractional digits) = -29604. Leading zeros are not counted in determining the number of significant digits. For example, 0.0002 has one significant digit, but 1.0002 has five significant digits.

The compiler accepts any value that can be represented in double precision (not more than 24 significant decimal digits) as an unsigned number. If this unsigned number does not contain an exponent part with "@@" (specifying a double-precision value), then the single-precision representation of that value is used. If the value represented by the significant digits of such an unsigned number, when disregarding the placement of the decimal point, is greater than 549755813887, then some precision is lost if the unsigned number is converted to single precision.

Exponents

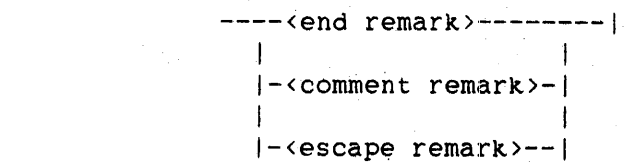
The exponent part is a scale factor expressed as an integer power of 10. The exponent part "@@ <integer>" signifies that the entire number is a double-precision value.

If the form of the unsigned number used includes only an exponent part, a decimal number of 1 is assumed. For example, @-11 is interpreted as 1@-11.

REMARK

Syntax

<remark>



<end remark>

Any sequence of letters, digits, and spaces not containing the reserved words END, ELSE, or UNTIL.

<comment remark>

-- COMMENT --<comment characters>-- ; --|

<comment characters>

Any sequence of EBCDIC characters not containing a semicolon (;).

<escape remark>

-- % --<escape text>--|

<escape text>

Any sequence of EBCDIC characters.

Semantics

Remarks are provided as methods of inserting program documentation throughout an ALGOL source file.

The end remark can follow the language component END. The compiler recognizes the termination of the end remark when it encounters one of the reserved words END, ELSE, or UNTIL, or any nonalphabetic, nonnumeric EBCDIC character. Defines are not expanded within an end remark.

The comment remark is delimited by the word "COMMENT" at the beginning and a semicolon (;) at the end. The comment remark can appear between any two language components except within editing specifications.

Because remarks, string literals, and define invocations are language components, a comment remark is not recognized within a string literal, a define invocation, or another remark. Comment remarks can contain the dollar sign ($), but the comment remark must not contain a dollar sign as the first nonblank character on a source record. If a dollar sign is the first nonblank character on a source record, the compiler interprets the source record as a compiler control record.

The percent sign (%) preceding escape text in an escape remark can follow any language component that is not contained in editing specifications. The escape remark begins with the percent sign and extends to the beginning of the sequence number field of the record. The compiler does not examine the escape remark. When the percent sign that precedes an escape remark is encountered, the compiler skips immediately to the next record of the source file before continuing the compilation.

Examples

The following program illustrates some syntactically correct uses of the remark.

```
BEGIN
   FILE F(KIND=PRINTER COMMENT;);
   FORMAT COMMENT; FMT COMMENT; (A4,I6);
   PROCEDURE P(X,COMMENT;Y,Z);
      REAL X,Y COMMENT; ,Z;       % PERCENT SIGN CAN BE USED HERE
      X := Y + COMMENT: Z;        % HERE TOO
   IF COMMENT; 7 > 5 THEN
      WRITE(F,<"OK">);
   IF 4 COMMENT; > 2 THEN
      WRITE(F,<"OK">);
   IF 8 > 5 THEN
      WRITE COMMENT;(F,<"OK">);
END OF PROGRAM.
```

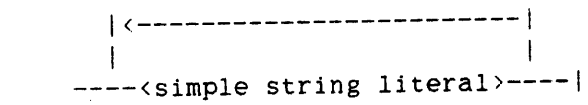
The following program illustrates some invalid uses of the remark.

```
BEGIN
   FILE F(KIND=PRINTER);
   FORMAT FMT(13,F10.3 COMMENT; ,A4);
   ARRAY A[0:99];
   REAL X;
   FORMAT ("ABC", % CANNOT BE USED. "DEE");
   WRITE(F,<"INVALID USE" COMMENT;>);
   REPLACE POINTER(A) BY "ABCD COMMENT;EFGHIJ";
   X := "AB,COMMENT;C";
   COMMENT CANNOT BE USED HERE COMMENT; EITHER;
END.
```

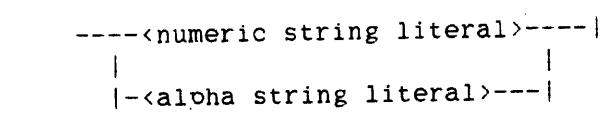
STRING LITERAL

Syntax

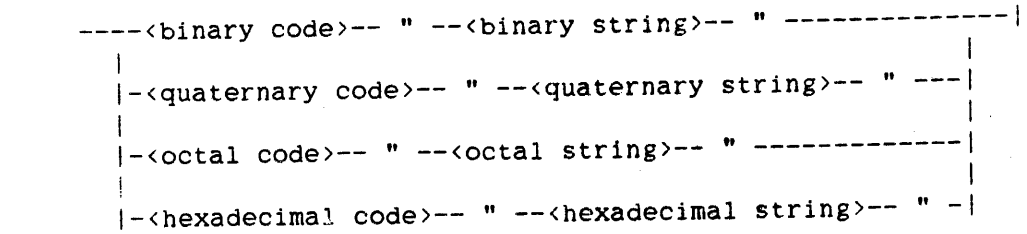
<string literal>



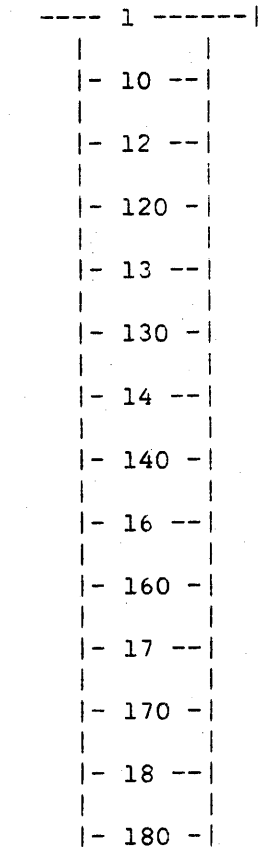
<simple string literal>



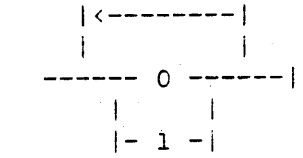
<numeric string literal>



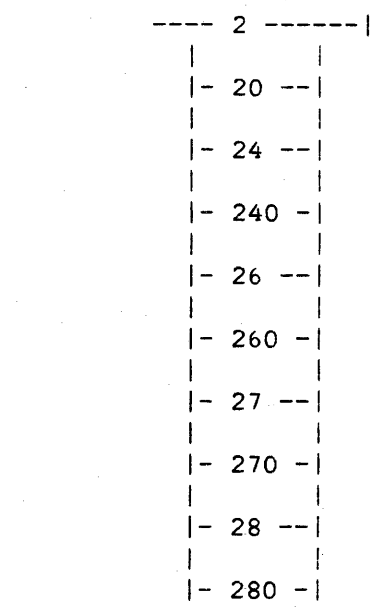
<binary code>



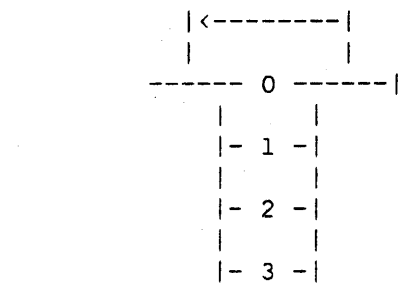
<binary string>



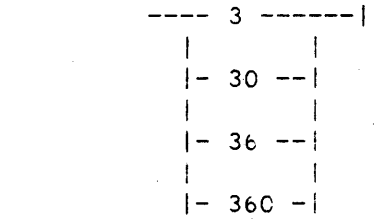
<quaternary code>



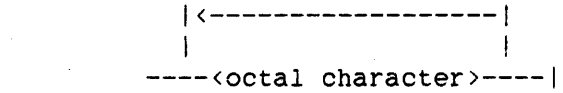
<quaternary string>



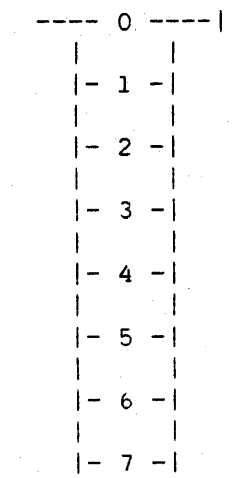
<octal code>



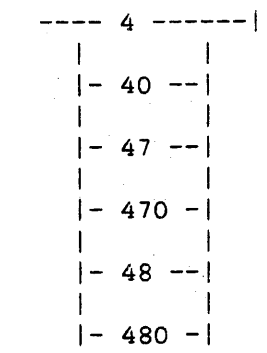
<octal string>



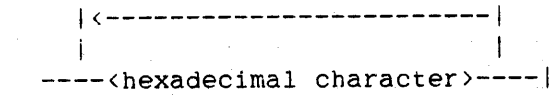
<octal character>



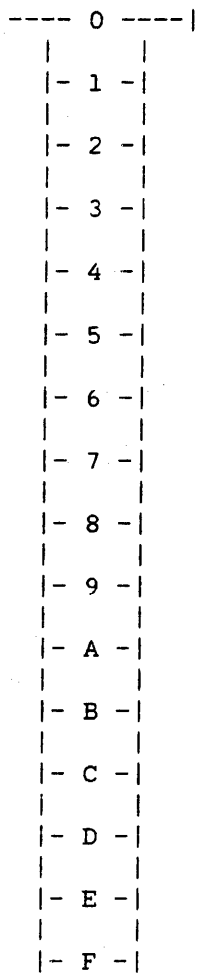
<hexadecimal code>



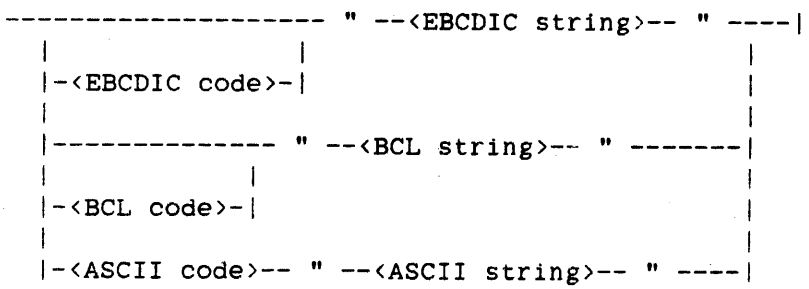
<hexadecimal string>



<hexadecimal character>

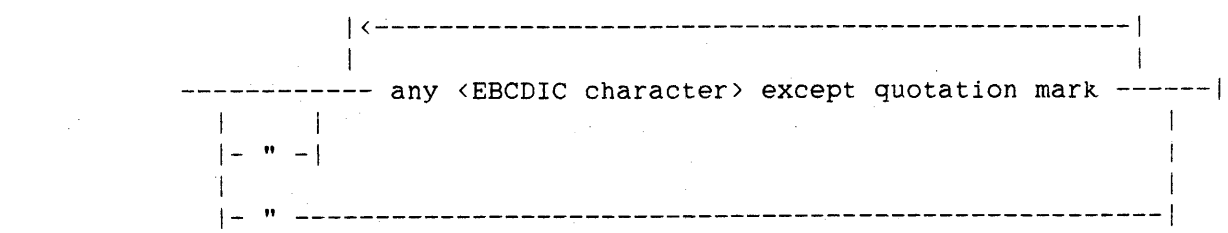


<alpha string literal>



<EBCDIC code>

<EBCDIC string>

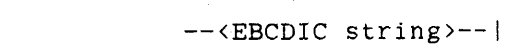


<EBCDIC character>

Any one of the 256 possible EBCDIC characters.

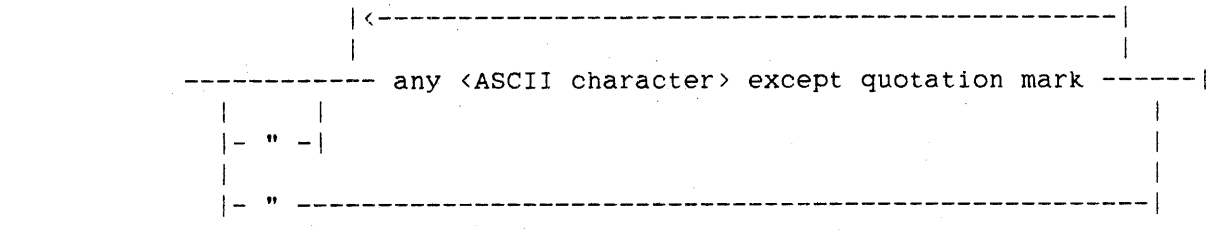
<BCL code>

<BCL string>



<ASCII code>

<ASCII string>



<ASCII character>

Any one of the 128 possible ASCII characters.

Semantics

Character Size

Strings can be composed of binary (1-bit) characters, quaternary (2-bit) characters, octal (3-bit) characters, hexadecimal (4-bit) characters, BCL (6-bit) characters, ASCII (7-bit in 8-bit format) characters, or EBCDIC (8-bit) characters. The word formats of various character types are described under "Character Representation" in the appendix "Data Representation."

See also
Character Representation. 808

String Code

The string code determines the interpretation of the characters between the quotation marks (") of a string literal. The string code specifies the character set and, for strings of less than 48 bits, the justification. The first digit of the string code specifies the character set in which the source string is written. The next nonzero digit (if any) specifies the internal character size of the string to be created by the compiler. If no nonzero digit is specified, the internal size is the same as the source size. If the internal size is different from the source size, the length of the string must be an integral number of internal characters. For example, the string literal 48"C1C2C3C4" is an EBCDIC string expressed in terms of hexadecimal characters.

If the string literal contains fewer than 48 bits, a trailing zero in the string code specifies that the string literal is to be left-justified within the word and that trailing zeros are to fill out the remainder of the word.

If the string literal contains fewer than 48 bits, the absence of a trailing zero in the string code specifies that the string literal is to be right-justified within the word and that leading zeros are to fill out the remainder of the word.

If the string literal contains 48 or more bits, the presence or absence of a trailing zero in the string code has no effect.

If the string code is not specified, the source string and the internal representation of the string are of the default character type. For more information, refer to "Default Character Type" in the appendix "Data Representation."

See also
Default Character Type. 817

String Length

The maximum length permitted for a simple string literal is 256 characters; the maximum length permitted for a string literal is 4095 characters. However, when a string literal is used as an arithmetic primary, it must not exceed 48 bits in length.

Internally, a string literal of 48 bits or less is represented in the object code as an 8-bit, 16-bit, or 48-bit literal. A string literal more than 48 bits long is stored in a "pool array" created by the compiler. An internal pointer carries the character size and address of the string within the pool array.

BCL Strings

BCL strings can contain any EBCDIC character. However, any EBCDIC character that does not have a BCL equivalent is translated by the compiler into a BCL question mark (36"14").

NOTE

The BCL data type is not supported on all A Series and B 5000/B 6000/B 7000 Series systems. The appearance of a BCL construct that may cause the creation of a BCL descriptor, such as a BCL string literal more than 96 bits long, will cause the program to get a compile-time warning message.

ASCII Strings

The ASCII string code can be used only with ASCII strings composed entirely of characters that have corresponding EBCDIC graphics. This is because the compiler recognizes only ASCII characters that have corresponding EBCDIC graphics.

The compiler translates each ASCII character into an 8-bit character. The rightmost seven bits are the ASCII representation of that character; the leftmost bit is 0.

ASCII characters that are not in the EBCDIC character set must be written as a hexadecimal string in which each pair of hexadecimal characters represents the internal code of one ASCII character, right-justified with a leading 0 bit.

Quotation Mark

The quotation mark (") can appear only as the first character of a simple string literal. Strings with internal quotation marks must be broken into separate simple strings by using three quotation marks in succession. For example, the string literal ""ABC" represents the string "ABC, and the string literal "A"""BC" represents the string A"BC.

Dollar Sign

String literals can contain the dollar sign ($). The dollar sign must not be the first nonblank character on a source record. If a dollar sign is the first nonblank character on a source record, the compiler interprets the source record as a compiler control record.

4 DECLARATIONS

A declaration associates certain characteristics and structures with an identifier. In an ALGOL program, every identifier must be declared before it is used. The compiler ensures that subsequent usage of an identifier in a program is consistent with its declaration.

Syntax

<declaration>

Each of the following metatokens represents a valid ALGOL declaration.

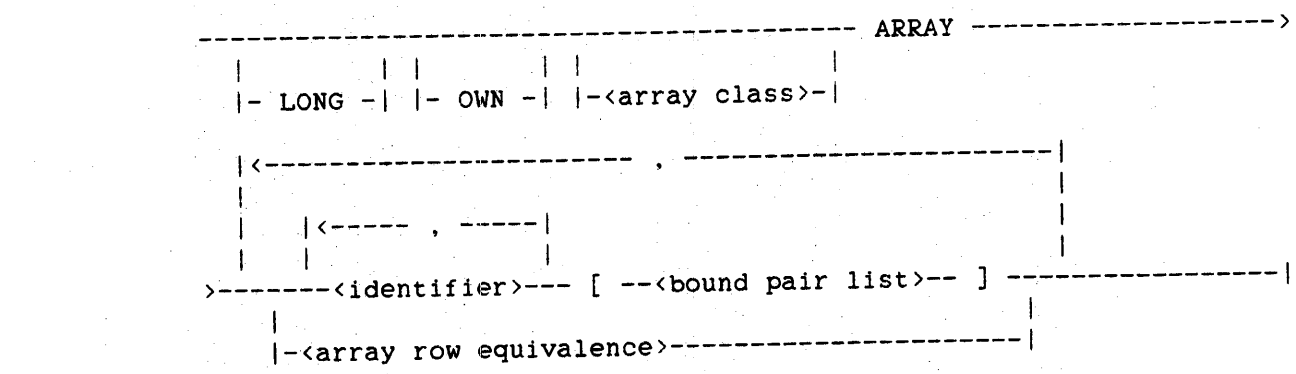
<array declaration>
<array reference declaration>
<Boolean declaration>
<complex declaration>
<define declaration>
<direct array declaration>
<double declaration>
<dump declaration>
<event array declaration>
<event declaration>
<export declaration>
<file declaration>
<format declaration>
<forward interrupt declaration>
<forward procedure declaration>
<forward switch label declaration>
<integer declaration>
<interrupt declaration>
<label declaration>
<library declaration>
<list declaration>
<monitor declaration>
<output message array declaration>
<picture declaration>
<pointer declaration>
<procedure declaration>
<real declaration>
<string array declaration>
<string declaration>
<switch file declaration>
<switch format declaration>
<switch label declaration>
<switch list declaration>
<task array declaration>
<task declaration>
<translate table declaration>
<truth set declaration>
<value array declaration>

ARRAY DECLARATION

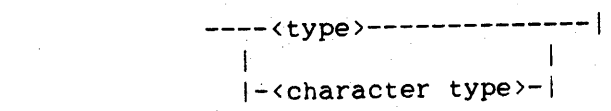
An ARRAY declaration declares one or more identifiers to represent arrays of specified fixed dimensions.

Syntax

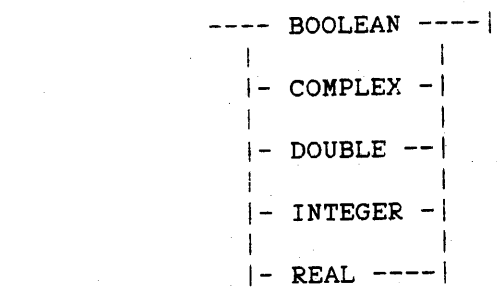
<array declaration>



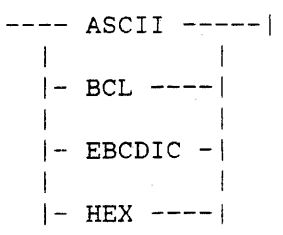
<array class>



<type>



<character type>



<array identifier>

An <identifier> that is associated with an array in an ARRAY declaration.

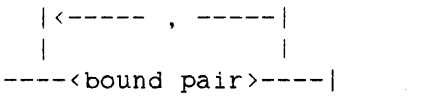
<character array identifier>

An <array identifier>, <array reference identifier>, <direct array identifier>, or <value array identifier> that was declared with a <character type>.

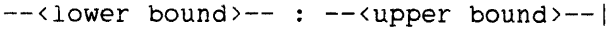
<word array identifier>

An <array identifier>, <array reference identifier>, <direct array identifier>, or <value array identifier> that was declared with a <type>.

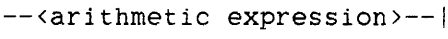
<bound pair list>



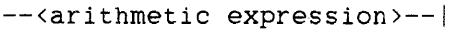
<bound pair>



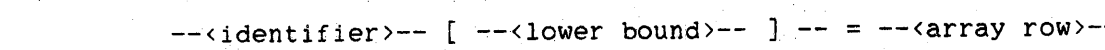
<lower bound>



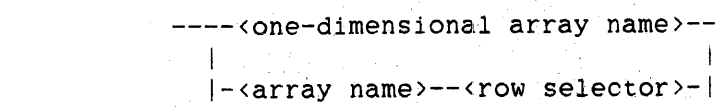
<upper bound>



<array row equivalence>



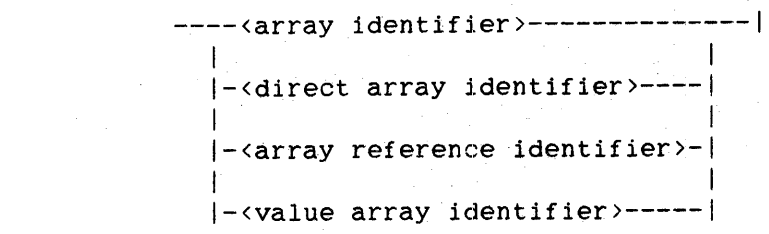
<array row>



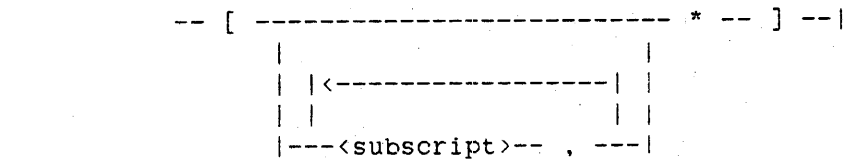
<one-dimensional array name>

An <array name> whose identifier was declared with one dimension.

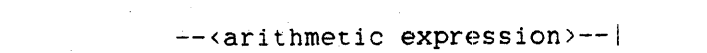
<array name>



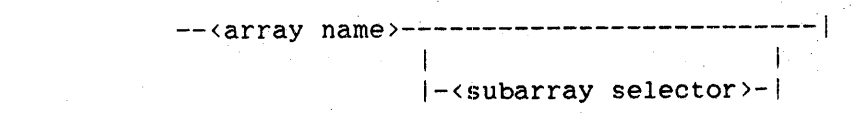
<row selector>



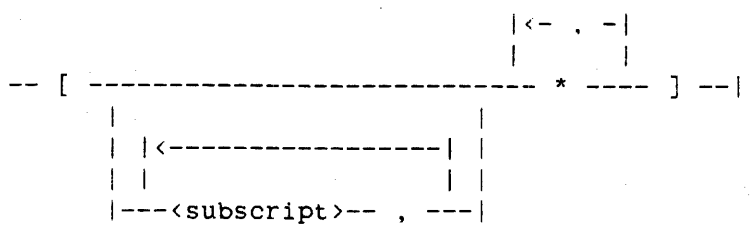
<subscript>



<array designator>



<subarray selector>



See also

Semantics

After an array has been declared in an ARRAY declaration, values can be stored in and retrieved from the elements of the array by the use of subscripted variables, which are comprised of the array identifier and a subscript list.

LONG Arrays

The LONG specification affects only array rows (see the description of array row below). Normally, an array row longer than 1024 words is automatically paged (segmented) at run time into segments of 256 words each. "LONG" specifies that the array is not to be paged regardless of its length.

The array size at which an array row is automatically paged can be changed with the ODT command SEGARRAYSTART. (For more information on the SEGARRAYSTART command, see the "Operator Display Terminal (ODT) Reference Manual.") Arrays smaller than 1024 words are never paged.

OWN Arrays

If an array is declared to be OWN, the array and its contents are retained on exit from the block in which the array is declared and are available on subsequent re-entry into the block.

Own arrays are allocated only once, regardless of how many times entry is made into the block in which the array is declared. If the own array is declared with variable bounds, these bounds are evaluated once when

the array is allocated, and the affected dimension retains these bounds for the remainder of the execution of the program (unless the array is resized; refer to "RESIZE Statement").

Arrays not declared as own are deallocated on exit from the block in which they are declared and are reallocated on every entry into the block in which they are declared.

Array Class

Arrays declared in the same ARRAY declaration are of the same array class. If the array class is omitted, REAL is assumed.

Arrays not declared with a character type are called "word arrays." Arrays declared with a character type are called "character arrays."

Word and character arrays can be passed as parameters and used as array rows. Character arrays can be used as simple pointer expressions.

NOTE

> The BCL data type is not supported on all A Series and B 5000/B 6000/B 7000 Series systems. The appearance of a BCL construct that may cause the creation of a BCL descriptor, such as a BCL array, will cause the program to get a compile-time warning message.

Element Width

The element width of an array is the number of bits used to contain each element of the array. The element width is determined by the array class, as follows:

```
DOUBLE, COMPLEX:          96 bits (double word)
INTEGER, REAL, BOOLEAN:   48 bits (single word)
EBCDIC, ASCII:             8 bits (6 characters per word)
BCL:                       6 bits (8 characters per word)
HEX:                       4 bits (12 characters per word)
```

Within the computer, arrays are manipulated by means of descriptors; each descriptor specifies an element width appropriate to the array class: single- and double-word descriptors are used for word arrays; 4-, 6-, and 8-bit descriptors are used for character arrays. Note that 6-bit descriptors are obsolete and are not supported on some systems.

Because complex and double array elements are composed of two 48-bit words, the two words are allocated contiguously. The layout of a complex array is as follows: the real part of the first element, the imaginary part of the first element, the real part of the second element, the imaginary part of the second element, and so on. Similarly, the layout of a double array is as follows: the first word of the first element. the second word of the first element, the first word of the second element, the second word of the second element, and so on. For information on the internal representation of double and complex operands, refer to "Two-Word Operand" in the appendix "Data Representation."

See also
Two-Word Operand. 824

Bound Pair List

The subscript bounds for an array are given in the first bound pair list following the array identifier. The bound pair list gives the lower and upper bounds of all dimensions taken in order from left to right. In all cases, upper bounds must not be less than their associated lower bounds.

Arithmetic expressions used as array dimension bounds are evaluated once (from left to right) on entering the block in which the array is declared. These expressions can depend only on values that are global to that block or passed in as actual parameters. The results of the arithmetic expressions are integerized. Arrays declared in the outermost block must use constant or constant expression bounds.

Original and Referred Arrays

Every array identifier that is declared with a bound pair list is an original array, which is distinct from all other original arrays.

There are three other ways to associate an identifier with an array: array row equivalence, array reference assignment, and array specification in a PROCEDURE declaration. In each of these cases, the identifier refers to the same data as some original array. Such an

identifier is called a "referred array." An array row equivalence or array reference assignment can cause an array identifier of one array class to refer to data in an original array of another array class.

Dimensionality

The dimensionality (number of dimensions) of an original array is the number of bound pairs in the bound pair list with which the array is declared. Arrays cannot have more than 16 dimensions.

The size (number of elements) of each dimension of an array declared with a given bound pair is given by the following expression:

    <upper bound> - <lower bound> + 1

The maximum size of a dimension is 2**20-1 elements.

Array Row Equivalence

An array row equivalence causes the declared array identifier to refer to the same data as the specified array row. That array row can be an original array or another referred array. The declared identifier is an equivalent array.

The size of the declared array is determined by the size and element width of the array row and the element width for the array class of this declaration. Let Sa and Wa be the size and element width of the array row, and We be the element width for the equivalent array. The size of the equivalent array, Se, is then

    Se := (Sa * Wa) DIV We

Because of the truncation implicit in the DIV operation, Se * We might be less than Sa * Wa. In this case, indexing the equivalent array by Se + <lower bound> causes an invalid index fault. Nevertheless, pointer operations that use the equivalent array can access the entire area of memory allocated to the original array to which the array identifier ultimately refers; the memory area may hold more than Se elements of width We.

The array row equivalence allows the programmer to reference the same array row with two or more identifiers. Each identifier can reference the same data with different type, character type, or lower bound specifications. For example, in the following program, after the assignment I[2] := 25.234 is executed, both I[2] and R[0] contain the value 25.0, but after the assignment R[0] := 25.234 is executed, both I[2] and R[0] contain the value 25.234.

```
BEGIN
   REAL ARRAY R[0:9];
   INTEGER ARRAY I[2] = R;  % Array row equivalence. The INTEGER
                            % array I refers to the same data as
                            % the REAL array R.
   I[2] := 25.234;
   R[0] := 25.234;
END.
```

The array row equivalence part cannot appear in an ARRAY declaration that declares an own array. For example,

```
OWN ARRAY A[0] = B
```

is an invalid declaration. An array declared with an array row equivalence part is own if and only if the array to which it is equated is own.

Note: There are subtle restrictions on the correct declaration and use of an array row equivalence where the array row of the declaration is a row of an array reference, because the default state of an array reference variable is uninitialized.

If the array reference is one-dimensional and has the same element width as the new array, then the two identifiers become synonyms: whenever the array reference variable is assigned a value, the equivalent array describes the same data.

Otherwise, the array row equivalence is established from the value of the array reference variable at the time of entry into the block containing the array row equivalence declaration; later assignments to the array reference variable do not affect the array row equivalence. Therefore, for the declaration to be useful, the array reference variable must have been declared and initialized in a scope global to the block declaring the array row equivalence.

Subarray Selector

A subarray selector selects part of an array by specifying subscripts for high-order dimensions and leaving others unspecified. The unspecified dimensions are indicated by an asterisk (*). The dimensionality of the subarray is the number of asterisks in the subarray selector.

The total number of subscripts and asterisks in a subarray selector must equal the dimensionality of the array identifier to which the subarray selector is suffixed. In the degenerate case of no subscripts, the number of asterisks equals that dimensionality, and the subarray is the whole array. In all other cases, the subarray selector specifies a subarray of reduced dimensionality.

For example, given the declarations

```
ARRAY A[0:9,1:40,0:99];
INTEGER I,J; % (ASSUME 0 <= I <= 9 AND 1 <= J <= 40)
```

then

A and A[*,*,*]	Denote the entire three-dimensional array
A[I,*,*]	Denotes one of the ten two-dimensional arrays that constitute A
A[I,J,*]	Denotes one of the 40 one-dimensional arrays (array rows) that constitute A[I,*,*], and one of the 400 one-dimensional arrays that constitute A

Array Row

An array row is a one-dimensional array designator.

Row Selector

A row selector is the limiting case of a subarray selector, with only one asterisk.

Pragmatics

The maximum value of <lower bound> is 131,071; the minimum value of <lower bound> is -131,071.

The maximum length of an array is 2**20-1.

When "LONG" is specified, the maximum size of an array row is determined by the overlay row size of the system, which is specified at cold-start time.

For non-own arrays, an array is unreferenced from the time the program enters the block in which the array is declared until the first execution of a statement that refers to the array. Once such a statement is encountered, the array is referenced or "touched" until the program exits the block. For an array declared OWN, the array is unreferenced from the time the program begins execution until the first execution of a statement referencing the array is encountered. Once such a statement is encountered, the array is referenced for the remainder of program execution.

For character arrays, the actual storage area allocated is the number of whole words sufficient to contain the specified number of characters. Using pointer operations, the last portion of the last word in the area can be referenced, even if this portion is beyond the valid subscript range. For example, if array A is declared "EBCDIC ARRAY A[0:3]", the characters corresponding to A[4] and A[5] can be referenced using a pointer operation.

Examples

```
INTEGER ARRAY DOG[0:5,0:25,1:7,4:16]
```

Declares DOG, a four-dimensional array made up of 6 * 26 * 7 * 13 = 14196 integer elements.

```
OWN REAL ARRAY STUB[0:9]
```

Declares STUB, a one-dimensional own array made up of 10 real elements.

```
REAL ARRAY GROUP_REAL[0:17], CAD[400:500,1:50]
```

Declares two real arrays: GROUP_REAL, which is a one-dimensional array, and CAD, which is a two-dimensional array.

```
EBCDIC ARRAY GROUP_EBCDIC[0] = GROUP_REAL[*]
```

Declares the EBCDIC array GROUP_EBCDIC. Array row equivalence causes GROUP_EBCDIC to refer to the same data as the previously declared real array GROUP_REAL. Note that the element width of GROUP_REAL is 48 bits, whereas the element width of GROUP_EBCDIC is eight bits. This means that a reference to a single element in GROUP_REAL refers to 48 bits, and a reference to a single element in GROUP_EBCDIC refers to eight bits.

```
ARRAY XRAY[X+Y+Z:3*A+B]
```

Declares XRAY, a one-dimensional array. Because no array class is specified, the array class of XRAY is real. The lower bound is the integerized value of X + Y + Z, and the upper bound is the integerized value of 3 * A + B.

```
LONG BOOLEAN ARRAY BIG_ARRAY[0:9999]
```

Declares BIG_ARRAY, a one-dimensional array made up of 10000 Boolean elements. Because BIG_ARRAY is declared LONG, the array is not paged (segmented). Because it is not paged, the array occupies 10000 contiguous words in memory.

```
ARRAY SEGARRAY[0:50000]
```

Declares SEGARRAY, a one-dimensional array made up of 50001 real elements. Because SEGARRAY is not declared LONG and the array row is longer than 1024 words, SEGARRAY is automatically divided at run time into segments 256 words long.

```
COMPLEX ARRAY C[0:2,0:60]
```

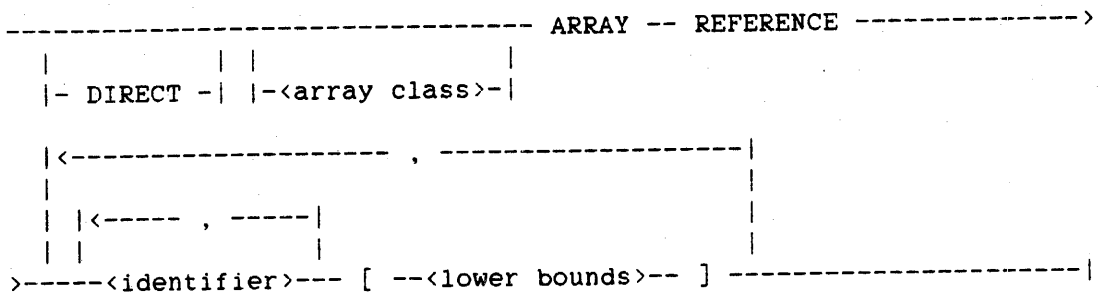
Declares C, a two-dimensional array made up of 3 * 61 = 183 complex elements. Note that the element width of a complex array is 96 bits (two words).

ARRAY REFERENCE DECLARATION

An ARRAY REFERENCE declaration is used to establish an array reference variable. The array reference assignment statement can then be used to assign an array or part of an array to this variable.

Syntax

<array reference declaration>



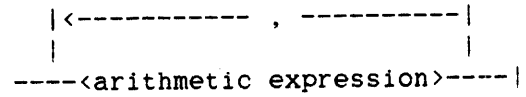
<array reference identifier>

An <identifier> that is associated with an array reference in an ARRAY REFERENCE declaration.

<direct array reference identifier>

An <identifier> that is associated with an array reference that is declared DIRECT in an ARRAY REFERENCE declaration.

<lower bounds>



See also
<array class> 41

Semantics

Following an array reference assignment, any subsequent use of the array reference identifier acts as a reference to the array assigned to it.

If the array class is not specified as COMPLEX, the array reference variable can be declared DIRECT. This allows the array reference variable to be used in direct I/O operations.

NOTE

The BCL data type is not supported on all A Series and B 5000/B 6000/B 7000 Series systems. The appearance of a BCL construct that may cause the creation of a BCL descriptor, such as a BCL array reference, will cause the program to get a compile-time warning message.

If an array class is not specified, REAL is assumed.

The number of dimensions of the array reference variable is determined by the number of lower bounds in its declaration. No more than 16 dimensions are allowed.

The initial state of an array reference variable is uninitialized. Any attempt to use an uninitialized array reference variable as an array results in a fault at run time.

See also
Array Reference Assignment. 231
Direct I/O. 316

Examples

```
ARRAY REFERENCE REFARRAY[3]
```

Declares REFARRAY, an array reference variable with a lower bound of 3. Because an array class is not specified, REFARRAY is a real array reference variable.

DIRECT ARRAY REFERENCE DIRREFARRAY[N]

Declares DIRREFARRAY, a direct, real array reference variable with a lower bound equal to the value of N. Because this array reference variable is declared to be direct, it can be used in direct I/O.

COMPLEX ARRAY REFERENCE CREF1[0], CREF2[0,10,10]

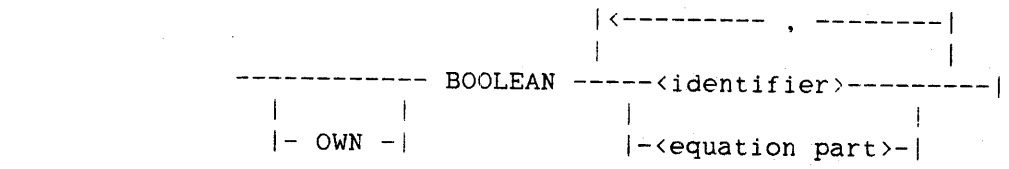
Declares two complex array reference variables. CREF1 is a one-dimensional array reference variable with a lower bound of zero, and CREF2 is three-dimensional with lower bounds of 0, 10, and 10.

BOOLEAN DECLARATION

A BOOLEAN declaration declares simple variables that can have Boolean values of TRUE or FALSE.

Syntax

<Boolean declaration>



<Boolean identifier>

An <identifier> that is associated with the BOOLEAN data type in a BOOLEAN declaration.

<equation part>

--<identifier>-- = --<identifier>--|

Semantics

A simple variable declared to be OWN retains its value when the program exits the block in which the variable is declared, and that value is again available when the program re-enters the block in which the variable is declared.

The equation part causes the simple variable being declared to have the same address as the simple variable associated with the second identifier. This action is called "address equation." An identifier can be address-equated only to a previously declared local identifier or to a global identifier. The first identifier must not have been previously declared within the block of the equation part.

Address equation is allowed only between integer, real, and Boolean variables. Because both identifiers of the equation part have the same address, altering the value of either variable affects the value of both variables.

The following example demonstrates the effects of address-equating Boolean and real variables.

```
BEGIN
   REAL R;
   BOOLEAN B = R;
   R := 4;              % B = FALSE  R = 4.0
   B := B OR B.[2:1];   % B = TRUE   R = 5.0
   B := TRUE;           % B = TRUE   R = 1.0
END.
```

The OWN specification has no effect on an address-equated identifier. The first identifier of an equation part is own only if the second identifier of the equation part is own.

A BOOLEAN declaration with an equation part is not allowed in the global part of a program unit.

Pragmatics

The TRUE or FALSE value of a Boolean simple variable (and the value of any other Boolean operand) depends only on the low-order bit (bit zero) of the word. Each of the 48 bits of a Boolean simple variable contains a Boolean value that can be interrogated or altered by using the partial word part or concatenation.

When a Boolean simple variable is allocated, it is initialized to FALSE (a 48-bit word with all bits equal to zero). However, to ensure compatibility with ALGOL 60, programmers should explicitly initialize Boolean simple variables with appropriate assignment statements.

The appendix "Data Representation" contains additional information on the internal structure of a Boolean operand as implemented on A Series and B 5000/B 6000/B 7000 Series systems.

See also
Boolean Operand 823

Examples

```
BOOLEAN BOOL
```

Declares BOOL as a Boolean simple variable.

```
OWN BOOLEAN DONE, ENDOFIT
```

Declares DONE and ENDOFIT as Boolean simple variables. Because they are declared OWN, these simple variables retain their values when the program exits the block in which they are declared.

```
BOOLEAN FLAG, BINT = INTGR
```

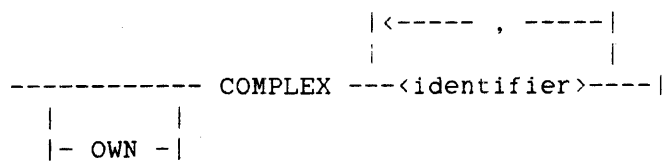
Declares FLAG and BINT as Boolean simple variables, and address-equates BINT to the previously declared simple variable INTGR. BINT and INTGR share the same address.

COMPLEX DECLARATION

A COMPLEX declaration declares a simple variable that can have complex values.

Syntax

<complex declaration>



<complex identifier>

An <identifier> that is associated with the COMPLEX data type in a COMPLEX declaration.

Semantics

Complex variables allow for the storage and manipulation of complex values in a program. The interpretation of complex values is the usual mathematical one. The real and imaginary parts of complex values are always stored separately as single-precision real values.

Because a real value is a complex value with an imaginary part equal to zero, the set of real values is a subset of the set of complex values. Therefore, arithmetic values can be assigned to complex variables, but complex values cannot be assigned to arithmetic variables.

A simple variable declared to be OWN retains its value when the program exits the block in which it is declared. That value is again available when the program re-enters the block in which the variable is declared.

Pragmatics

The appendix "Data Representation" contains additional information on the internal structure of a complex operand as implemented on A Series and B 5000/B 6000/B 7000 Series systems.

See also
Complex Operand 826

Examples

```
COMPLEX C1, C2
```

Declares C1 and C2 as complex simple variables.

```
OWN COMPLEX CURRENT, VOLTAGE, IMP
```

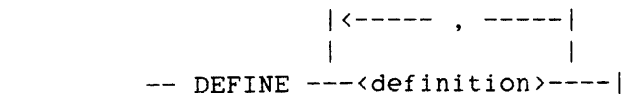
Declares CURRENT, VOLTAGE, and IMP as complex simple variables. Because they are declared OWN, these simple variables retain their values when the program exits the block in which they are declared.

DEFINE DECLARATION

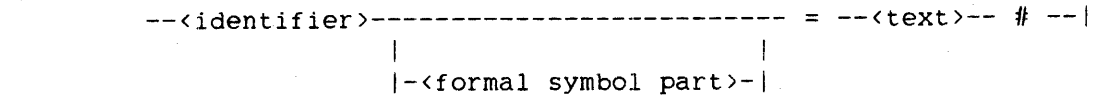
The DEFINE declaration causes the compiler to save the specified text until the associated define identifier is encountered in a define invocation. At that point, the saved text is retrieved and compiled as if the text were located at the position of the define invocation.

Syntax

<define declaration>



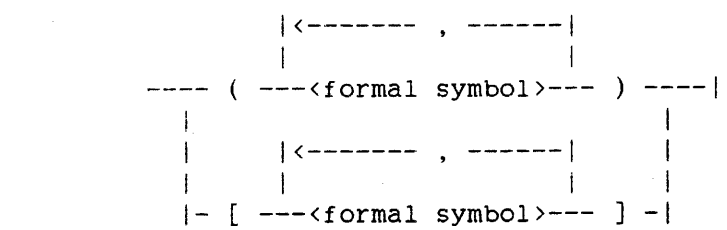
<definition>



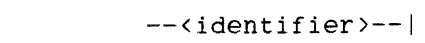
<define identifier>

An <identifier> that is associated with <text> in a DEFINE declaration.

<formal symbol part>



<formal symbol>



<text>

Any sequence of valid characters not including a free pound sign (#) character.

Semantics

A define has two forms: simple and parametric. These forms are readily differentiated because parametric defines have a series of parameters (called formal symbols) enclosed in matching parentheses or brackets.

The formal symbols constitute the essential part of a parametric define. Formal symbols function similarly to the formal parameters of a PROCEDURE declaration. When a parametric define is invoked, wherever formal symbols appear in the text, a substitution of the corresponding closed text of the define invocation is made before that part of the text is compiled. References to formal symbols cannot appear outside the text of the corresponding parametric define. No more than nine formal symbols are allowed in a parametric define.

Text is bracketed on the left by the equal sign (=) and on the right by the pound sign (#). The equal sign is said to be "matched" with the pound sign. The text can be any sequence of characters not containing a "free" pound sign. A free pound sign is one that is not in a string literal, not in a remark, and not matched with an equal sign in a define declaration within the text. The compiler interprets the first free pound sign as signaling the end of the text. That is, the first free pound sign is matched with the equal sign that started the text.

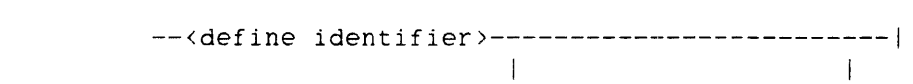
Compiler control records occurring within the text are processed normally if the dollar sign ($) is in column 1 or 2. If the dollar sign is in column 3 or beyond, a syntax error is generated whenever the define is invoked.

Define Invocation

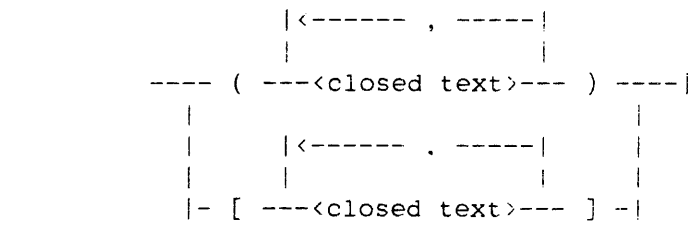
A define invocation causes the occurrence of a define identifier to be replaced by the text associated with the define identifier.

Syntax

<define invocation>



<actual text part>



<closed text>

Program text not containing mismatched or unmatched parentheses, brackets, or quotation marks, and not containing any comma outside of these bracketing symbols.

Semantics

The invocation of a parametric define causes textual substitution of the closed text into the positions of the associated text indicated by the corresponding formal symbol. The closed text need not be simple; for example, given the DEFINE declaration

```
DEFINE FORJ(A,B,C) = FOR J := A STEP B UNTIL C #
```

the define invocation

```
FORJ(0,B*3,MAX(X,Y,Z))
```

expands to

```
FOR J := 0 STEP B*3 UNTIL MAX(X,Y,Z)
```

The closed text can be empty in a define invocation. In this case, all occurrences of the corresponding formal symbol in the text are replaced by no text. For example, given the DEFINE declaration

```
DEFINE F(M, N) = M + N #
```

the define invocation

```
R := F(, 1);
```

expands to

```
R := +1;
```

which is syntactically correct. However, the statement

```
R:=F(2,);
```

expands to

```
R:=2+;
```

which is syntactically incorrect.

A define identifier cannot be invoked as a part, rather than the whole, of a language component such as a string literal or a number. For example, given the declarations

```
EBCDIC STRING S;
DEFINE EBCDIC_STR = 8 #;
```

the statement

```
S : = EBCDIC_STR"ABC";
```

is NOT interpreted by the compiler to be equivalent to

```
S := 8"ABC":
```

The invocation of define EBCDIC_STR is interpreted by the compiler as a whole language component, specifically a number, and not as an EBCDIC code preceding a quoted EBCDIC string. Thus, it appears that a number is being assigned to a string variable, which is illegal, and the compiler flags the statement with a syntax error.

In the same manner, given the declarations

```
REAL R;
DEFINE ITEM = 15 #;
```

the statement

```
R := ITEM;
```

is legal, but the statement

```
R := ITEM.30;
```

is syntactically equivalent to

```
R := (15).30;
```

and is illegal.

In the following contexts, the appearance of a define identifier does NOT cause the define to be expanded:

1. Defines are not expanded in an end remark, a comment remark, or an escape remark.

2. Defines are not expanded within quoted strings. For example, given the declaration

   ```
   DEFINE ONE = THE FIRST #;
   ```

   the string

   ```
   "ONE WEEK"
   ```

   is not equivalent to the string

   ```
   "THE FIRST WEEK"
   ```

3. Defines are not expanded within identifiers. For example, given the declaration

   ```
   DEFINE A = PREFIX #;
   ```

   the identifiers

   ```
   A_B and ABC
   ```

   are not expanded to

   ```
   PREFIX_B and PREFIXBC
   ```

4. Define identifiers are not always expanded when they occur in declarations. If the define identifier occurs in a position where an identifier can appear, the define identifier is not expanded. If the define identifier occurs in a position where an identifier is not expected, the define identifier is expanded. The following examples illustrate this rule:

```
DEFINE A = ARRAY #;
A B[0:10];          % A IS EXPANDED
REAL A B[0:10];     % A CAN BE INTERPRETED AS AN IDENTIFIER
                    % IN A REAL DECLARATION. A IS NOT
                    % EXPANDED. A SYNTAX ERROR RESULTS.
EBCDIC A B[0:10]; % A IS EXPANDED
```

5. A define identifier is not expanded either in the format part of a FORMAT declaration or in the editing specifications of a READ statement or WRITE statement. Furthermore, if a FORMAT declaration or editing specifications are located within the text of a parametric define, they cannot reference the formal symbols of that define.

6. A define identifier is not expanded when used in place of a file or task attribute mnemonic. (Refer to the "I/O Subsystem Reference Manual" for file attribute mnemonics and the "Work Flow Language (WFL) Reference Manual" for task attribute mnemonics.) In the following example, the define identifiers are not expanded in the FILE declaration or in the VALUE function.

```
DEFINE NEVERUSED = NEWTASK #,
       PRINTER = REMOTE #;

FILE F(KIND = PRINTER);          % INTERPRETED AS PRINTER,
                                 % NOT REMOTE
T.STATUS := VALUE(NEVERUSED); % INTERPRETED AS NEVERUSED,
                                 % NOT NEWTASK
```

Pragmatics

If the ALGOL compiler encounters a syntax error while compiling the combination of the text, actual text part, and formal symbol part at the occurrence of a define invocation, some or all of the expanded define is given along with the appropriate error message.

To avoid problems with expanding a define, particularly when an expression is passed in as actual text, each occurrence of a formal symbol in the text of a parametric define should be enclosed in parentheses. For example, consider the following program:

```
BEGIN
   BOOLEAN BOOL;
   DEFINE
      LOGIC1(A,B) = A AND B #,
      LOGIC2(A,B) = (A) AND (B) #;
   BOOL := LOGIC1(TRUE OR TRUE, FALSE);  % INVOCATION OF LOGIC1
   BOOL := LOGIC2(TRUE OR TRUE, FALSE);  % INVOCATION OF LOGIC2
END.
```

The invocation of LOGIC1 above expands to

```
BOOL := TRUE OR TRUE AND FALSE;
```

which, because of the order of precedence of Boolean operators, evaluates as

```
BOOL := TRUE OR (TRUE AND FALSE);
```

assigning a value of TRUE to BOOL. In contrast, the invocation of LOGIC2 expands to

```
BOOL := (TRUE OR TRUE) AND (FALSE);
```

which assigns a value of FALSE to BOOL.

Passing an updating expression to a parametric define should be done cautiously. Multiple uses of the corresponding formal symbol cause multiple updates. For example, given the DEFINE declaration

```
DEFINE Q(E) = E + 2 * E #
```

the define invocation

```
Q(X := X + 1)
```

expands to

```
X := X + 1 + 2 * X := X + 1
```

Examples

```
DEFINE BLANKIT = REPLACE POINTER(LINEOUT) BY " " FOR 22 WORDS #
```

Declares BLANKIT as a define identifier. Where BLANKIT appears as an allowable define invocation, it is expanded to

```
REPLACE POINTER(LINEOUT) BY " " FOR 22 WORDS
```

when the program is compiled.

```
DEFINE SEC(X) = 1 / COS(X) #
```

Declares SEC as a define identifier with a formal symbol X. If SEC(N) appears as an allowable define invocation, it is expanded to

```
1 / COS(N)
```

when the program is compiled.

```
DEFINE LENGTH(X,Y) = SQRT(X**2 + Y**2)#
```

Declares LENGTH as a define identifier with two formal symbols, X and Y. If LENGTH(3,4) appears as an allowable define invocation, it is expanded to

```
SQRT(3**2 + 4**2)
```

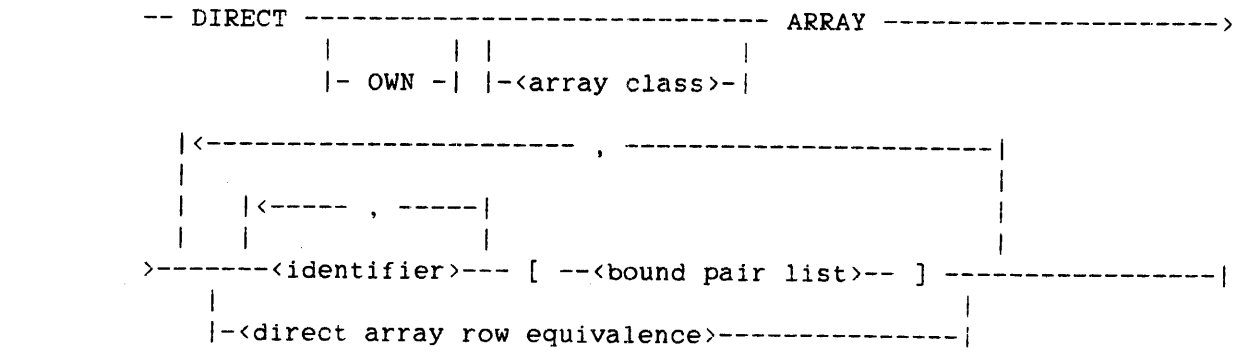
when the program is compiled.

DIRECT ARRAY DECLARATION

A DIRECT ARRAY declaration declares arrays that can be used in direct input/output (I/O) operations.

Syntax

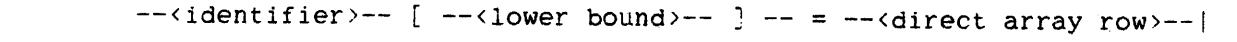
<direct array declaration>



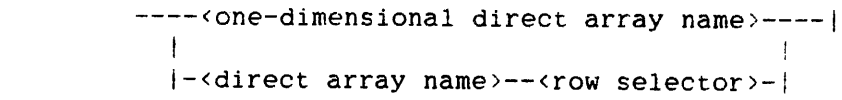
<direct array identifier>

An <identifier> that is associated with a direct array in a DIRECT ARRAY declaration.

<direct array row equivalence>



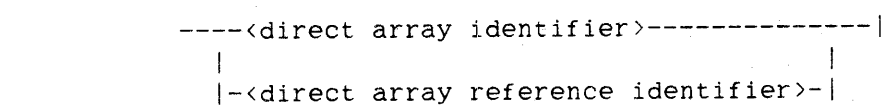
<direct array row>



<one-dimensional direct array name>

A <direct array name> whose identifier was declared with one dimension.

<direct array name>



See also

Semantics

A direct array can be a word array or a character array. Direct arrays of type COMPLEX are not allowed.

NOTE

> The BCL data type is not supported on all A Series and B 5000/B 6000/B 7000 Series systems. The appearance of a BCL construct that may cause the creation of a BCL descriptor, such as a direct BCL array, will cause the program to get a compile-time warning message.

A direct array can be used in any way that a non-direct array can be used. However, arbitrary use of direct arrays instead of normal arrays can seriously degrade overall system efficiency.

A direct array has certain attributes, which can be programmatically interrogated and altered before, during, and after an actual I/O operation that uses the array.

The semantics of the OWN specification are discussed under the "ARRAY Declaration."

The dimensionality of a direct array is the number of bound pairs in its declaration. No more than 16 dimensions are allowed.

The warning about an array row equivalence declared using an array reference identifier applies with equal force to the case of a direct array row equivalence declared using a direct array reference identifier.

Pragmatics

Because a direct array can be used in performing direct I/O operations, a direct array is automatically unpaged (unsegmented).

See also
Direct I/O. 316

Examples

```
DIRECT ARRAY DIRARY[0:29]
```

Declares DIRARY, a one-dimensional direct array. Because no array class is specified, the array class of DIRARY is REAL.

```
DIRECT INTEGER ARRAY DIREQVARAY[5] = DIRARY
```

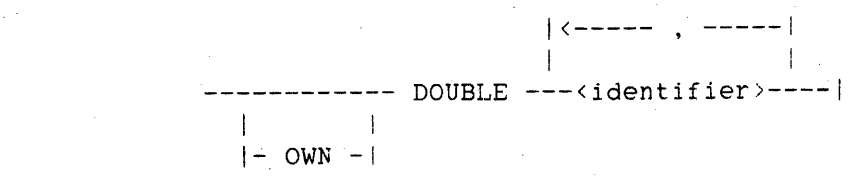
Declares the direct integer array DIREQVARAY. Array row equivalence causes the array DIREQVARAY to refer to the same data as the previously declared direct real array DIRARY.

DOUBLE DECLARATION

A DOUBLE declaration declares simple variables that can have double-precision values (that is, 96-bit arithmetic entities).

Syntax

<double declaration>



<double identifier>

An <identifier> that is associated with the DOUBLE data type in a DOUBLE declaration.

Semantics

A simple variable declared to be OWN retains its value when the program exits the block in which the variable is declared, and that value is again available when the program re-enters the block in which the variable is declared.

Pragmatics

When a double-precision simple variable is allocated, it is initialized to a double-precision zero (two 48-bit words with all bits equal to zero). However, to ensure compatibility with ALGOL 60, programmers should explicitly initialize double-precision simple variables with appropriate assignment statements.

The appendix "Data Representation" contains additional information on the internal structure of a double-precision operand as implemented on A Series and B 5000/B 6000/B 7000 Series systems.

See also
Double-Precision Operand. 824

Examples

```
DOUBLE DUBL
```

Declares DUBL, a double-precision simple variable.

```
DOUBLE BIGNUMBER, GIGUNDOUS, DUBLPRECISION
```

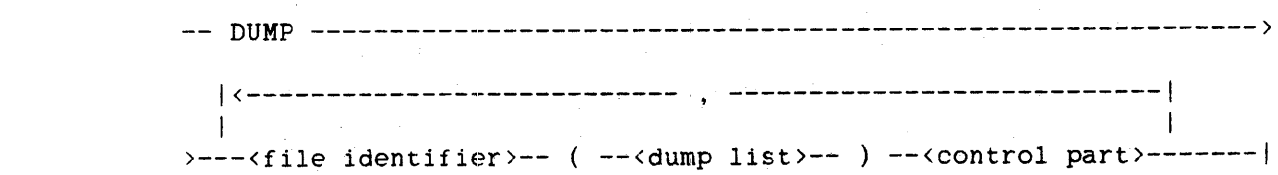
Declares three double-precision variables: BIGNUMBER, GIGUNDOUS, and DUBLPRECISION.

DUMP DECLARATION

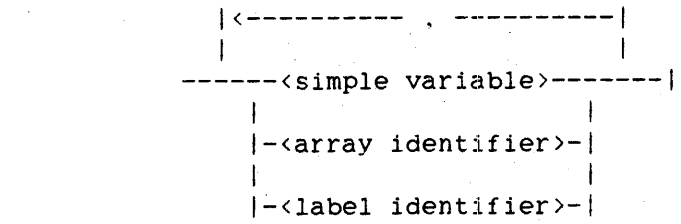
The DUMP declaration allows the display of the values of selected items during the execution of a program.

Syntax

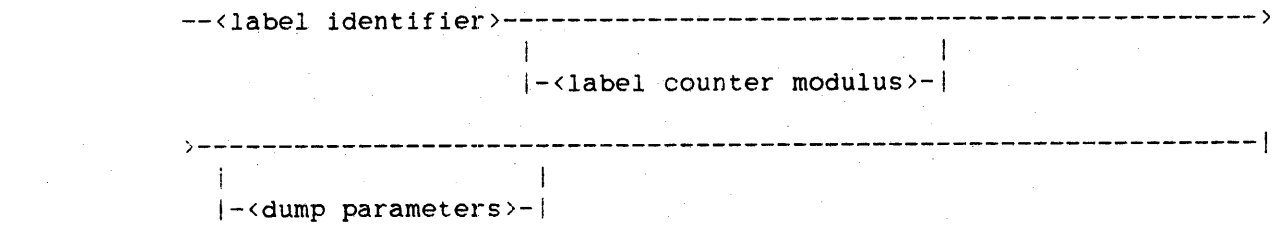
<dump declaration>



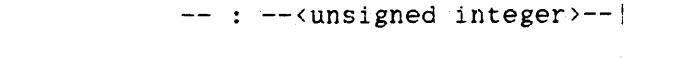
<dump list>



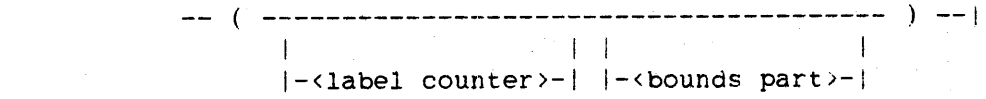
<control part>



<label counter modulus>

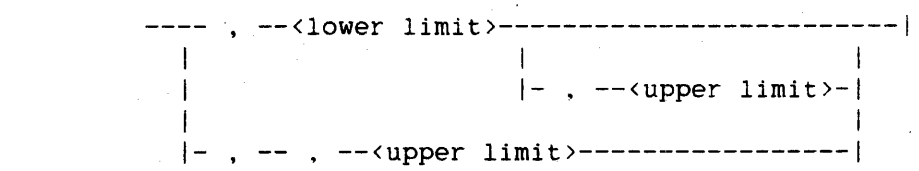


<dump parameters>



<label counter>

<bounds part>



<lower limit>

<upper limit>

See also

Semantics

The file identifier specifies the name of the file to which the displayed information is to be written, and the dump list specifies the items whose values are to be displayed. The following types of variables and arrays must NOT appear in the dump list:

1. Arrays with multiple dimensions
2. Character arrays
3. String variables
4. String arrays

The control part determines when the items are to be displayed. The possible forms of the control part are described in the following paragraphs.

<label identifier>

If the control part is simply a label identifier, the items in the dump list are dumped each time program execution encounters the statement labeled by the specified label identifier.

<label identifier> <label counter modulus>

If a label counter modulus appears, the items in the dump list are dumped every <label counter modulus> times that the statement labeled by the label identifier is encountered. Specifically, if N is the label counter modulus and E is the number of times that the labeled statement has been encountered, then the items in the dump list are dumped whenever E MOD N is equal to zero.

<label identifier> <dump parameters>

Dump parameters are used to restrict the dumping to a specified range of encounters. All three parameters (the label counter, the lower limit, and the upper limit) are optional.

If a label counter is given, this variable is used to count the number of times that the labeled statement has been encountered. The specified variable is incremented automatically each time the labeled statement is encountered; changing the value of this variable elsewhere in the program affects the dumping process.

The items in the dump list are dumped when the number of times the labeled statement is encountered (or the value of the label counter variable, if specified) is greater than or equal to the lower limit and less than or equal to the upper limit. If the lower limit is not specified, it has a default value of zero. If the upper limit is not specified, it has a default value of infinity (no limit).

<label identifiers> <label counter modulus> <dump parameters>

When both a label counter modulus and dump parameters are specified, both the modulus check and the range check are performed. The items in the dump list are dumped when all the following conditions are true for the number of times that the labeled statement has been encountered (or the value of the label counter variable, if specified):

1. It is greater than or equal to the lower limit and less than or equal to the upper limit.

2. It is evenly divisible by the label counter modulus.

Form of Output

The information produced when a dump occurs depends on the declared types of the items to be dumped. When a dump occurs, the symbolic name (up to six characters) of each item in the dump list is produced, along with the following information:

Dumped Simple Variables

1. If the simple variable is of type REAL or DOUBLE, a real value is printed (for example, "REEL = .10000000000" or "DUBL = 0.0").

2. If the simple variable is of type INTEGER, an integer value is printed (for example, "I = 2").

3. If the simple variable is of type BOOLEAN, the Boolean value is printed (for example, "BOOL = .FALSE.").

4. If the simple variable is of type COMPLEX, it is printed as a pair of numbers. The format consists of a left parenthesis, the real part in REAL format, a comma, the imaginary part in REAL format, and a right parenthesis (for example, "COMP = (3.0000000000, 5.0000000000)").

Dumped Arrays

1. If the array is of type REAL, each element is printed as if the value were operated on by an R editing phrase. (For more information, refer to "FORMAT Declaration.")

2. If the array is of type BOOLEAN, the value of each element is shown as ".TRUE." or ".FALSE.".

3. If the array is of type INTEGER, each element is printed as an integer value.

4. If the array is of type COMPLEX, each element is printed in the form used for complex variables (for example, "CA = (2.0000000000, 3.0000000000), (5.0000000000, 7.0000000000)").

Dumped Labels

A dumped label shows the number of times execution control has passed the specified label (for example, "L2 = 3").

Examples

```
DUMP FYLE (A) LBL
```

Dumps the value of variable A to a file named FYLE each time the statement labeled LBL is encountered during execution of the program.

```
DUMP PRNTR (I,INFO,INDX) NEXT (DMPCOUNT, ,DPHIGH)
```

Dumps the values of I, INFO, and INDX to a file named PRNTR when the statement labeled NEXT is encountered. A label counter, DMPCOUNT, counts the number of times the statement labeled NEXT is encountered. Dumps occur until the value of DMPCOUNT exceeds DPHIGH. Note that when a label counter is specified, the programmer has the option of altering this counter elsewhere in the program.

```
DUMP FID (X,Y,ARRAYV,COUNTER) LOUP : 3
```

Dumps the values of X, Y, ARRAYV, and COUNTER to a file named FID. Because a label counter modulus of 3 is specified, a dump of these items occurs only every third time the label LOUP is encountered during execution of the program.

```
DUMP LP (A,B,LBL1,ARRAYV) AGAIN : 5 (TALY.20,50)
```

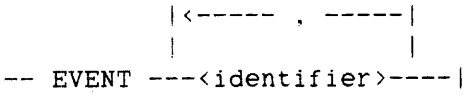
Dumps the values of A, B, LBL1, and ARRAYV to a file named LP. Because a label counter modulus of 5 is specified, a dump of these items occurs only every fifth time the label AGAIN is encountered during execution of the program. Dumps are further restricted to those times when the label counter TALY has a value between 20 and 50, inclusive. Because the dump occurs each time TALY MOD 5 = 0, dumps occur when TALY has the values 20, 25, 30, 35, 40, 45, and 50. Note that the programmer has the option of altering TALY elsewhere in the program.

EVENT AND EVENT ARRAY DECLARATIONS

An event provides a means to synchronize simultaneously executing processes. An event can be used either to indicate the completion of an activity (for example, the completion of a direct I/O read or write operation) or as an interlock between participating programs over the use of a shared resource.

Syntax

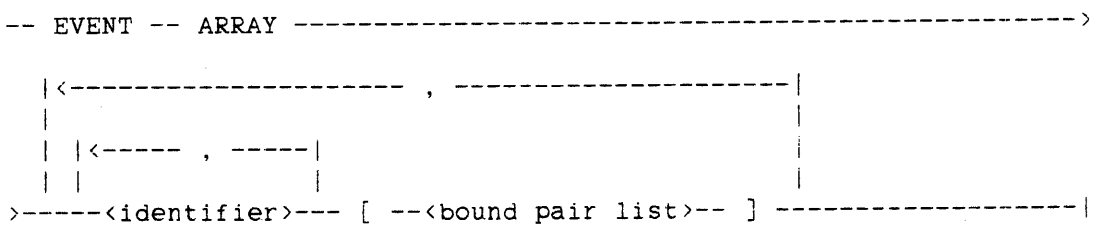
<event declaration>



<event identifier>

An <identifier> that is associated with an event in an EVENT declaration.

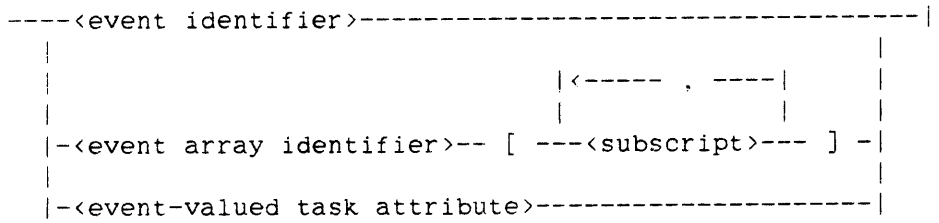
<event array declaration>



<event array identifier>

An <identifier> that is associated with an event array in an EVENT ARRAY declaration.

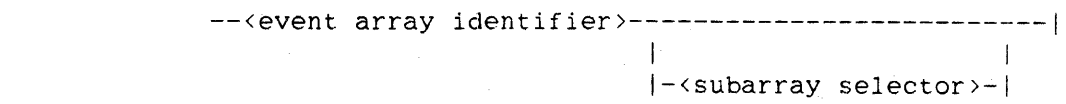
<event designator>



<event-valued task attribute>

<event-valued task attribute name>

<event array designator>



See also

Semantics

An event array is an array whose elements are events. An event array can have no more than 16 dimensions.

An event designator represents a single event. An event array designator represents an array of events.

Events can be used synchronously by explicitly testing the state of an event at various programmer-defined points during execution, or they can be used asynchronously by using the software interrupt facility.

Events have two Boolean characteristics, happened and available. Each characteristic can be either TRUE or FALSE. Language constructs such as the SET, RESET, and CAUSE statements can be used to change the happened state of an event. The HAPPENED function returns the value of the happened state of an event. The FIX, FREE, and LIBERATE statements can be used to change the available state of an event. The AVAILABLE function returns the available state of an event.

The initial available state of an event is TRUE (available), and the initial happened state of an event is FALSE (not happened). For more information on events, refer to "Event Statement." For more information on interrupts, refer to "INTERRUPT Declaration."

See also

Examples

EVENT FILEA

Declares an event, FILEA.

EVENT ARRAY SWAPPEE[0:5]

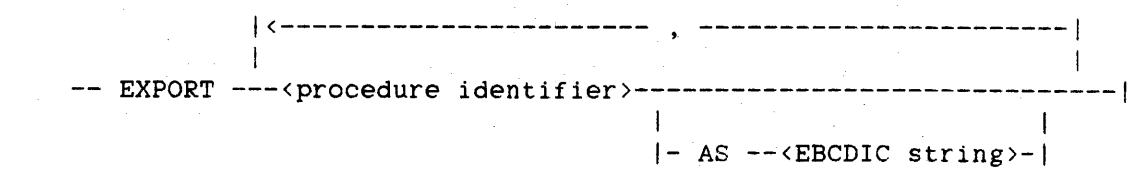
Declares an event array, SWAPPEE, which can store up to six events.

EXPORT DECLARATION

The EXPORT declaration declares procedures in a library program to be entry points into that library. A procedure that is declared as an entry point into a library can be accessed by programs external to the library.

Syntax

<export declaration>



See also
<procedure identifier>. 165

Semantics

All procedure identifiers to be exported must be declared before the appearance of the EXPORT declaration and must be declared in the same block as the EXPORT declaration.

A program becomes a library by exporting procedures and then executing a FREEZE statement. The code file for that program contains a structure called a library directory, which describes the library and its entry points. The directory's description of an entry point includes the entry point's name, a description of the procedure's type, if any, and descriptions of its parameters.

When a program calls a library entry point, the description of the entry point in the library template of the calling program is compared to the description of the entry point of the same name in the library directory of the library. If the called entry point does not exist in the library or if the two entry point descriptions are not compatible, a run-time error is given and the calling program is terminated.

The name given for an exported entry point in a library directory is the procedure identifier from the EXPORT declaration, unless an AS clause appears, in which case it is given by the EBCDIC string.

The EBCDIC string in the AS clause cannot contain any leading, trailing, or embedded blanks and must be a "valid identifier," where "valid identifier" is defined to be any sequence of characters beginning with a letter and consisting of letters, digits, hyphens (-), and underscores (_).

A library entry point can be any of the following:

Untyped procedure
Boolean procedure
Double procedure
Real procedure
Integer procedure
Complex procedure
EBCDIC string procedure
ASCII string procedure
Hexadecimal string procedure

The parameters to a library entry point can be any of the following types:

Boolean variable or array
Double variable or array
Real variable or array
Integer variable or array
Complex variable or array
EBCDIC string variable or array
ASCII string variable or array
Hexadecimal string variable or array
Task variable or array
Event variable or array
EBCDIC character array
ASCII character array
Hexadecimal character array
File
Pointer

A parameter to a library entry point can also be a fully specified formal procedure (the <formal parameter specifier> construct of the PROCEDURE declaration must be used) with the above restrictions on its type and parameters.

Pragmatics

If a library exports a procedure that is declared to be an entry point in yet another library, then when a program calls this entry point, the template of the library to which the procedure is declared to belong is searched for an entry point with the same name as that of the called entry point in the directory for this library. For example, assume the following declarations have been compiled:

```
LIBRARY L;
PROCEDURE LIBPROC; LIBRARY L;
EXPORT LIBPROC;
```

When another program calls entry point LIBPROC of this library, the template for library L is searched for an entry point named "LIBPROC". When found, the entry point LIBPROC of library L is then called.

On the other hand, if the following declarations have been compiled:

```
LIBRARY L;
PROCEDURE LIBPROC; LIBRARY L;
EXPORT LIBPROC AS "P";
```

and another program calls entry point P of this library, the template for library L is searched for an entry point named "P". If it is found, that entry point is called. If it is not found, a run-time message is given and the calling program is terminated. In either case, procedure LIBPROC of library L is not executed. For more information on libraries, refer to the chapter "Interface to the Library Facility."

Examples

EXPORT EXPROC

Declares the procedure EXPROC as an entry point in this library program.

EXPORT PROC1 AS "LIBPROC3"

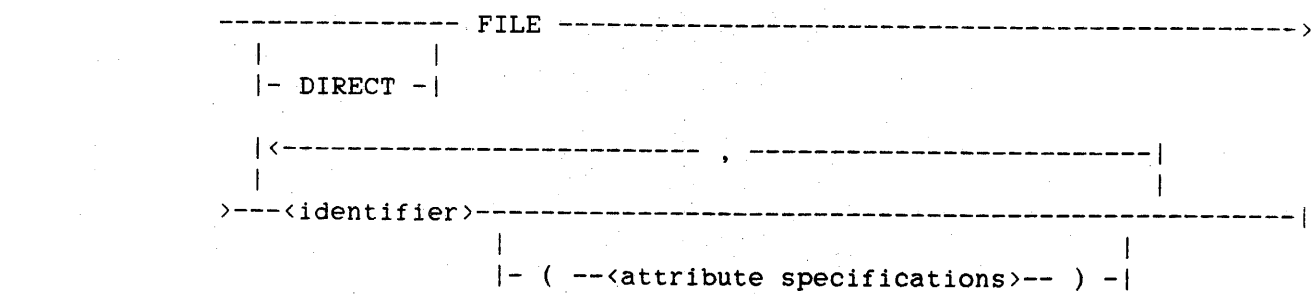
Declares the procedure PROC1 as an entry point in this library program. The name exported for this procedure is LIBPROC3, so a program calls PROC1 in this library by using the name LIBPROC3.

FILE DECLARATION

A FILE declaration associates a file identifier with a file and assigns values to the file attributes of the file.

Syntax

<file declaration>



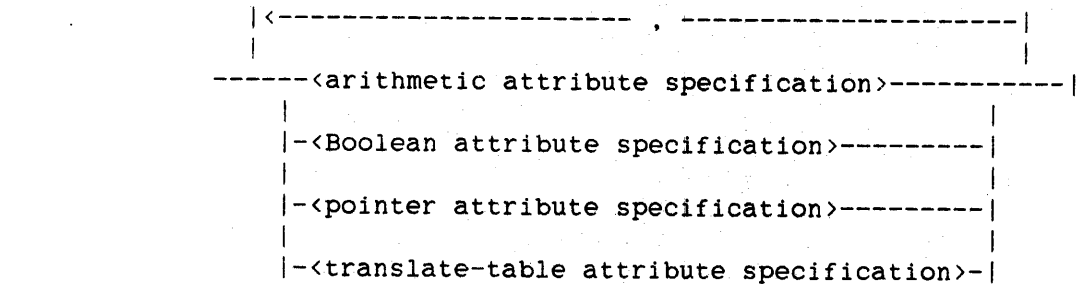
<file identifier>

An <identifier> that is associated with a file in a FILE declaration.

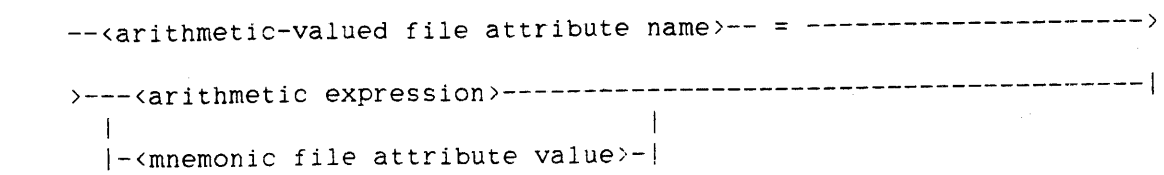
<direct file identifier>

An <identifier> that is associated with a file declared DIRECT in a FILE declaration.

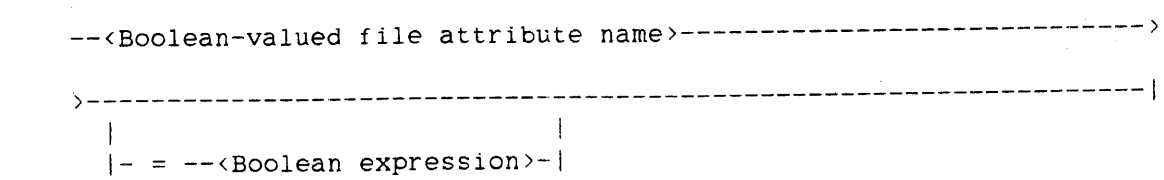
<attribute specifications>



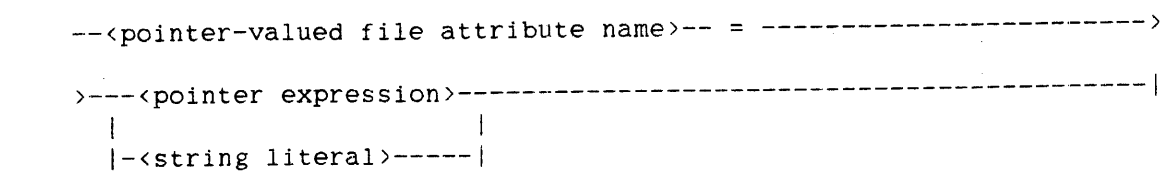
<arithmetic attribute specification>



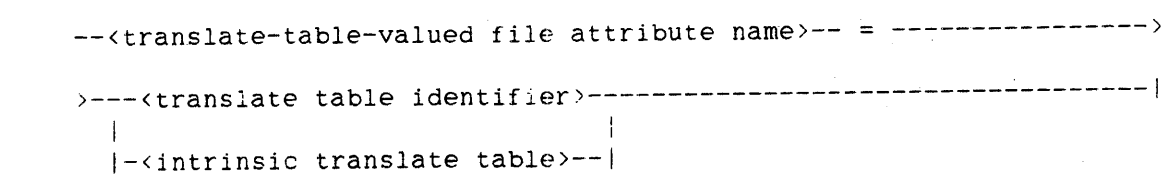
<Boolean attribute specification>



<pointer attribute specification>



<translate-table attribute specification>



<arithmetic-valued file attribute name>
<Boolean-valued file attribute name>
<pointer-valued file attribute name>
<translate-table-valued file attribute name>
<mnemonic file attribute value>

ALGOL supports all file attributes and file attribute values described in the "I/O Subsystem Reference Manual."

See also

Semantics

If "DIRECT" is specified, the file is declared as a direct file to be used for direct I/O.

The attributes for a particular file need not be specified in the FILE declaration. Attributes can be assigned values by using an appropriate assignment statement, by using the multiple attribute assignment statement, by using compile-time or run-time file equation, or (by default) by the I/O subsystem. Refer to the "Work Flow Language (WFL) Reference Manual" for file equation syntax.

Note that although the syntax allows more than one file identifier to precede the optional attribute specifications, only the identifier immediately before the attribute specifications is assigned the specified file attribute values. The other identifiers are assigned default file attribute values.

For example, the result of the declaration

```
FILE A,B,C(KIND=DISK)
```

is that the KIND attribute of file C is assigned the value DISK, and the KIND attributes of files A and B are assigned the default value for the KIND attribute, which may or may not be DISK. For more information on file attributes and their default values, refer to the "I/O Subsystem Reference Manual."

A Boolean-valued file attribute whose name appears in a Boolean attribute specification without the "= <Boolean expression>" part is assigned the value TRUE.

A translate table identifier assigned to a translate-table-valued file attribute name must have been declared previously and must reference the first (or only) translate table declared in that particular TRANSLATETABLE declaration.

Pragmatics

In a FILE declaration, the attribute specifications cannot reference the file identifier of the file being declared. For example, the following is not valid:

```
FILE F(MAXRECSIZE=90, BLOCKSIZE=F.MAXRECSIZE*10)
```

Examples

```
FILE F
```

Declares a file named F.

```
FILE NEWFILE(KIND=DISK, MAXRECSIZE=14, BLOCKSIZE=420, NEWFILE,
         FILEUSE=OUT, AREAS=20, AREASIZE=450,
         TITLE="DATA ON PACK.");
```

Declares a file named NEWFILE. This FILE declaration is the first step in creating a new disk file with the title DATA on a pack named PACK.

```
FILE SCREEN_OUTPUT(KIND=REMOTE)
```

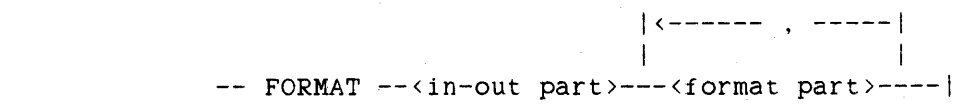
Declares a file, SCREEN_OUTPUT, to be a remote file. Typically, using this declaration in conjunction with a WRITE statement allows a program to write to a computer terminal.

FORMAT DECLARATION

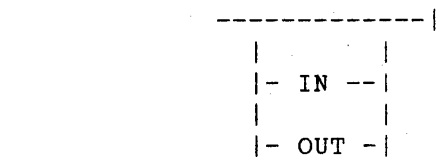
A FORMAT declaration associates a format identifier with a set of editing specifications. These editing specifications can then be used in READ and WRITE statements.

Syntax

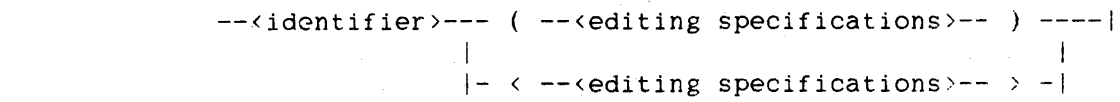
<format declaration>



<in-out part>



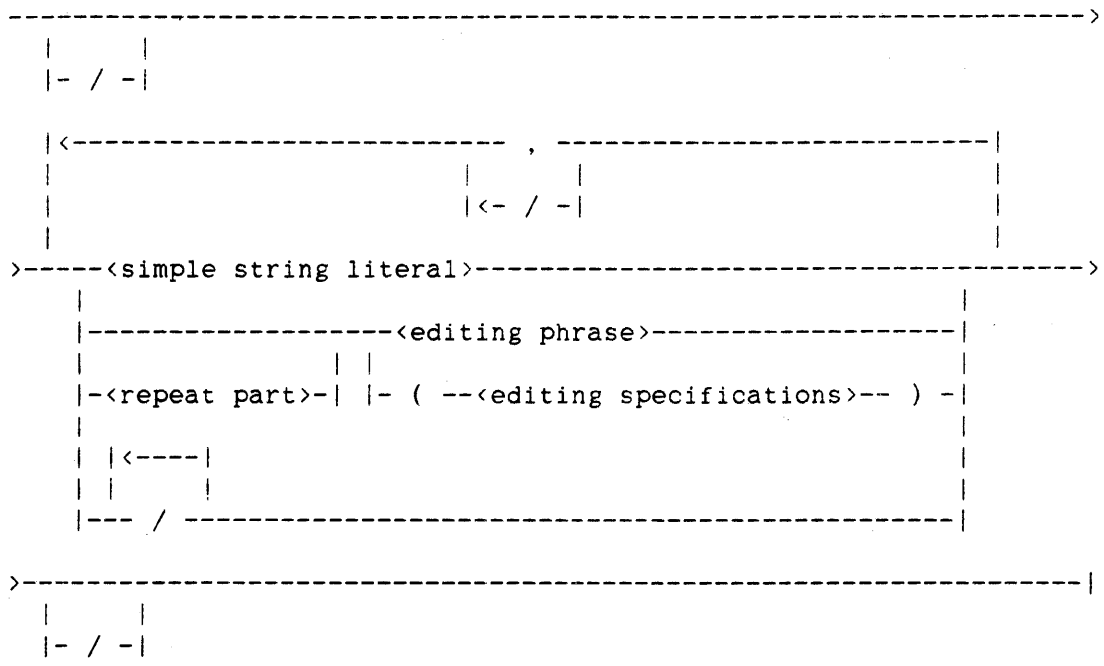
<format part>



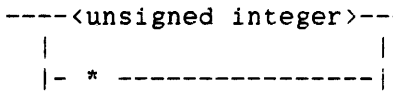
<format identifier>

An <identifier> that is associated with a set of editing specifications in a FORMAT declaration.

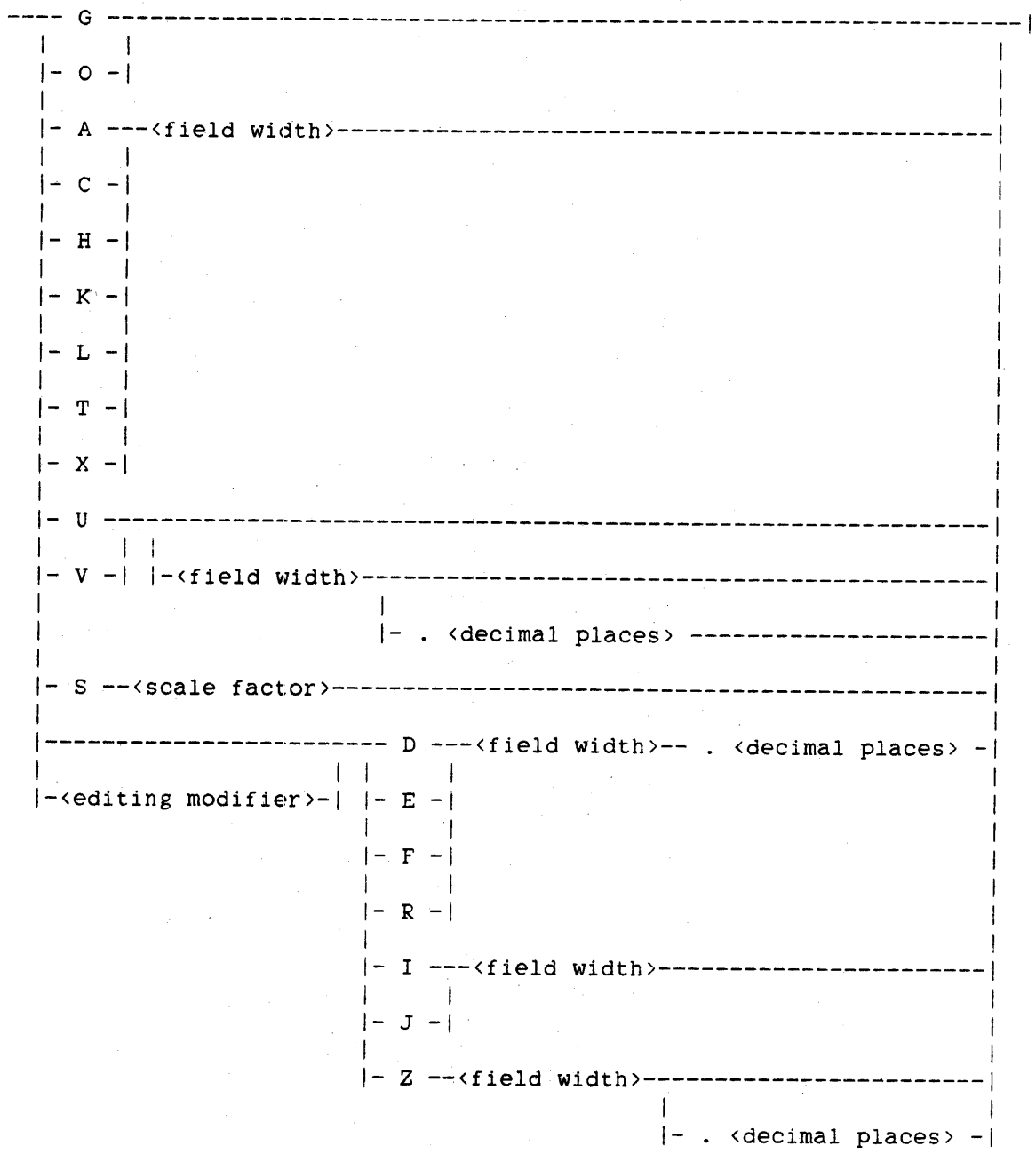
<editing specifications>



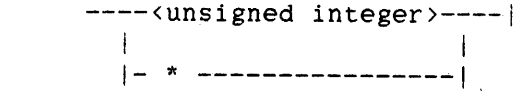
<repeat part>



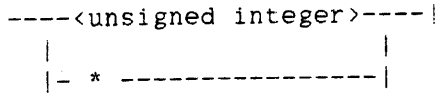
<editing phrase>



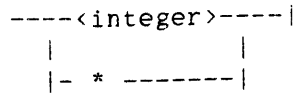
<field width>



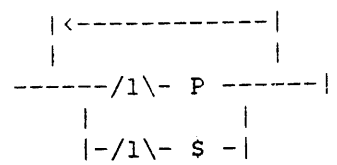
<decimal places>



<scale factor>



<editing modifier>



Semantics

The editing specifications that appear in FORMAT declarations can be used in READ and WRITE statements to format, respectively, the input and output data.

Define identifiers, remarks, and formal symbols of parametric defines cannot be used in formats.

A format identifier can be referenced in a READ statement, WRITE statement, or SWITCH FORMAT declaration. In general, a list is referenced in READ and WRITE statements to indicate a series of data items (specified by the list) along with the formatting action (specified by the format) to be performed on each of the data items.

Examples

The following examples illustrate the FORMAT declaration syntax:

```
FORMAT HDG("THIS REPORT SHOULD BE MAILED TO ROOM W-252")

FORMAT IN EDIT(X4, 2I6, 5E9.2, 3F5.1, X4)

FORMAT IN F1(A6, 5(X3, 2E10.2, 2F6.1)),
          F2(A6, G, A6)
```

```
FORMAT OUT FORM1(X56, "HEADING", X57),
           FORM2(X10, 4A6 / X7, 5A6 / X2, 5A6)

FORMAT FMT1(*I*)

FORMAT FMT2(*V*.*)
```

<in-out part>

The in-out part affects the processing of simple string literals appearing in the editing specifications. If the in-out part of a FORMAT declaration is OUT or unspecified (in which case OUT is assumed), simple string literals appearing in the editing specifications of the format are read-only. If the in-out part is IN, such simple string literals are read-write. (For more information, refer to the discussion of <simple string literal> in this section.)

<format part>

Editing phrases in the editing specifications are separated by a comma (,), a slash (/), or a series of slashes. A slash indicates the end of a record. On input, any remaining characters in the current record are ignored when a slash is encountered in the editing specifications. On output, the construction of the current record is terminated, and any subsequent output is placed in the next output record. Multiple slashes can be used to skip several records of input or to generate several blank records on output. The final right parenthesis or right angle bracket (>) of the editing specifications also indicates the end of the current record.

A carriage control action occurs each time a slash appears in the editing specifications. If a core-to-core part is specified in the file part of a READ statement, a slash is ignored.

Example

```
BEGIN
   FILE READER (KIND=READER),
        LINE   (KIND=PRINTER);
   REAL A,B;
   FORMAT FMT(I2,/,I2);
   READ(READER,FMT,A,B);
   WRITE(LINE,FMT,A,B);
   WRITE(LINE [SKIP 1],FMT,A,B);
END.
```

Given the two input records

```
1234
5678
```

this program produces the following output:

```
12
56
12
 .
 .
 .  [skip to channel 1]

56
```

If all editing specifications have been used before the list of data items is exhausted, a carriage control action occurs, and the editing specifications are reused. If the list of data items is exhausted before all the editing specifications have been used, the I/O operation is complete and the remaining editing specifications are ignored.

<simple string literal>

The presence of a simple string literal in the editing specifications indicates that the characters enclosed in quotation marks (") are to be used as the data. A simple string literal does not require a corresponding list element.

To enable more efficient handling of string literals in formats, 1-, 2-, and 7-bit strings are not allowed. The lengths of 3- and 4-bit strings must be a multiple of 2, to facilitate packing into 6- or 8-bit characters, respectively. BCL string literals are encoded as BCL characters, not as EBCDIC characters.

If no string code appears in a string literal, the default character type is used. The default character type can be designated by the compiler control options ASCII and BCL. If no such compiler control option is used, the default character type is EBCDIC. (For more information, refer to "Default Character Type" in the appendix "Data Representation.")

See also
Default Character Type. 817

Example

The statements

```
    .
    .
WRITE(LINE,<4"C1C2",8"ABC">);
$ SET BCL
WRITE(LINE,<3"646566",6"HIJ">);
    .
    .
```

produce the following output:

```
ABABC
UVWHIJ
```

When a simple string literal appears in editing specifications, only the first digit of the string code is used; if a second or third digit appears, a warning is given at compile time.

Simple string literals appearing in editing specifications can be read-only or read-write, depending on the in-out part specified in the FORMAT declaration. If the in-out part is IN, simple string literals appearing in the editing specifications are read-write, and the format can be used in both READ statements and WRITE statements. When a format used in a READ statement is declared with an in-out part of IN and contains a simple string literal in the editing specifications, then data is read into the memory location of the simple string literal over the original value. The number of characters read always equals the length of the simple string literal as it is defined in the FORMAT declaration. When the format is used in a subsequent WRITE statement, the new data is written to the output record. If the in-out part is OUT or unspecified (in which case OUT is assumed), any simple string literals appearing in the editing specifications are read-only; any attempt to change the value of a read-only simple string literal by using that format in a READ statement results in a run-time error.

<repeat part>

The repeat part specifies the number of times an editing phrase or editing specifications are repeated. If the repeat part is unspecified, a value of 1 is assumed. A repeat part value greater than 4029 results in a syntax error.

Editing specifications and their corresponding repeat parts can be nested. For example, in the WRITE statement

```
WRITE(F,<2(2(2I3))>,INT1,INT2,INT3,INT4,INT5,INT6,INT7,INT8)
```

the first repeat part specifies that the editing specifications "(2(2I3))" are to be repeated twice, the second repeat part specifies that the editing specifications "(2I3)" are to be repeated twice, and the third repeat part specifies that the editing phrase "I3" is to be repeated twice, causing the editing phrase "I3" to be used a total of eight times.

The following examples show the correct syntax of repeat parts:

```
3F10.4

3(A6/)

3(3A6,3(/I12)/)
```

<field width>

In general, the field width specifies, in characters, the width of the field to be read or written. Because the field width specifies the entire length of the field to be used, if <decimal places> is also specified, then the field width value must allow for the number of decimal places requested plus one for the decimal point. Any field width value greater than 4029 results in a syntax error. Field width is covered further in the discussions of the individual editing phrase letters.

<decimal places>

The decimal places value specifies the number of characters following the decimal point in the field that are to be read or written. On input, <decimal places> can be overridden by an explicit decimal point. A decimal places value greater than 4029 results in a syntax error. The decimal places value is covered further in the discussions of the individual editing phrase letters.

<scale factor>

The scale factor is discussed with the S editing phrase letter in this section.

Variable Editing Phrases

A variable editing phrase is one that is not fully specified at compile time. The format is processed from left to right at run time. If "V" is encountered in an editing phrase, the next list element is accessed to provide an editing phrase letter. (For more information, refer to "V Editing Phrase Letter" in this section). If an asterisk (*) is encountered as the repeat part, field width, decimal places, or scale factor, then the next list element is accessed to provide an integer value for that specification. In addition to the list elements to be read or written, the I/O list must contain one list element for each V and asterisk encountered in the editing specifications. The WRITE statements in the following examples use asterisks as both repeat parts and field widths to produce varying I editing phrases.

Examples

```
WRITE(F, <I*>, IWIDTH, A);
WRITE(F, <3I*>, IWIDTH, A, B, C);
WRITE(F, <3(I*)>, IWIDTH1, A, IWIDTH2, B, IWIDTH3, C);

IREPEAT1 := 1;
IREPEAT2 := 2;
WRITE(F, <2(X1,*I*)>, IREPEAT1, IWIDTH1, A,
                      IREPEAT2, IWIDTH2, B, C);
```

When an asterisk is used as the repeat part, the number of repetitions performed depends on the value supplied by the list element. If the value of the list element is greater than zero, that number of repetitions is performed; if the value is equal to zero, an unlimited number of repetitions is performed. If the value is less than zero, no repetitions are performed, and control passes to the next editing phrase.

When an asterisk is used for the field width of an editing phrase, the actual width of the field depends on the value supplied by the list element. If the value of the list element is greater than zero, that value is used as the width of the field; if the value of the list element is less than or equal to zero, no editing is performed, the list elements corresponding to the editing phrase are skipped, and control passes to the next editing phrase.

Editing Phrase Letters

Every valid path through the editing phrase syntax requires an "editing phrase letter" (A, C, D, E, F, G, H, I, J, K, L, O, R, S, T, U, V, X, or Z) that specifies how the data being read or written is to be edited. An editing phrase that contains the editing phrase letter A is called an A editing phrase, an editing phrase that contains the editing phrase letter C is called a C editing phrase, and so on. Descriptions of the editing specified by each editing phrase letter are arranged in alphabetical order in the following paragraphs.

For ease of explanation, lowercase letters are used hereafter to refer to the values for the repeat part, field width, and decimal places as follows:

r = <repeat part>
w = <field width>
d = <decimal places>

A list element of type COMPLEX is always edited as if it were two list elements of type REAL.

In the examples, "b" is used to denote a blank character.

A and C Editing Phrase Letters

The editing phrase letters A and C are used when reading or writing alphanumeric data. Valid list elements are of type INTEGER, REAL, DOUBLE, COMPLEX, BOOLEAN, POINTER, and STRING.

The action specified by the editing phrase letter C is identical to that specified by the editing phrase letter A except for the portion of the word from which characters are read or to which characters are written. Details are given below.

The default character type applies to list elements other than pointers. (For more information, refer to "Default Character Type" in the appendix "Data Representation.") This feature allows BCL data to be read from or written to an EBCDIC file (and vice versa) with translation, when necessary, to preserve character data.

For example, the program

```
BEGIN
   FILE F(KIND=PRINTER, INTMODE=EBCDIC);
   WRITE(F, <A3>, 8"ABC");
   $ SET BCL
   WRITE(F, <A3>, 6"ABC");
END.
```

produces the following output:

```
ABC
ABC
```

In the explanations of the editing phrase letters A and C, Q is used. Q is derived from the following table:

		Default Character Type	
		BCL	EBCDIC
Precision	Single	8	6
	Double	16	12

If the list element is of the form

```
<pointer expression> FOR <arithmetic expression>
```

then the value of the arithmetic expression is used as the value of Q.

See also
Default Character Type. 817

Pointers and String Variables

On input, w characters are transferred from the input record to the pointer-designated location or string variable. On output, w characters are transferred from the pointer-designated location or string variable to the output record. The character size used is that of the pointer or string variable.

Input

On input, the editing phrase letters A and C specify that w characters of data are to be read from the input record and assigned to the corresponding list element.

For the editing phrase letter A, if w is greater than or equal to Q, the rightmost Q characters of the input field are transferred to the list element. If w is less than Q, w characters of the input field are transferred right-justified to the list element. The unused high-order bits of the list element are set to zero.

The action specified by the editing phrase letter C is identical to that specified by the editing phrase letter A except that characters are read to the leftmost portion of the word.

Input Examples

Default Character Type	External String	Editing Phrase	Internal Value
8-bit	ABCDEFGHIJKL	A9	8"DEFGHI"
6-bit	ABCDEFGHIJKL	A9	6"BCDEFGHI"
8-bit	AbCbEbGbIbK	A4	4"0000"8"AbCb"
6-bit	ABCDEFGHIJKL	A4	6"0000ABCD"
(either)	ABCDEFGHIJKL	A12	ABCDEFGHIJKL (pointer as list element)
8-bit	ABCDEFGHIJKL	A12	4"0000"8"ABCDEFGHIJKL" (8-bit pointer FOR 14)
6-bit	ABCDEFGHIJKL	A12	6"JKL" (6-bit pointer FOR 3)
8-bit	ABCDEFGHIJKL	C9	8"DEFGHI"
6-bit	ABCDEFGHIJKL	C9	6"BCDEFGHI"
8-bit	ABCD	C4	8"ABCD"4"0000"
6-bit	ABCDEFGHIJKL	C4	6"ABCD0000"
8-bit	ABCDEFGHIJKL	C12	8"ABCDEFGHIJKL"4"0000" (8-bit pointer FOR 14)
6-bit	ABCDEFGHIJKL	C12	6"JKL" (6-bit pointer FOR 3)

The editing phrase letters A and C do not round values before assigning them to a list element. Therefore, a list element of type INTEGER is not necessarily assigned an integer value. If w is greater than 4, the exponent field of the list element is affected; the result can be a noninteger value. The data representations of real and integer operands are discussed in the appendix "Data Representation."

Output

On output, the editing phrase letters A and C specify that the value of the corresponding list element is to be written as a character string to an output field w characters wide.

For the editing phrase letter A, if w is greater than or equal to Q and the list element is not a pointer expression, the Q characters of the list element are written right-justified with blank fill to the output field. If w is less than Q, the rightmost w characters of the list element are written to the output field. If the character size is eight bits and any of the character fields in the word contain bit patterns that do not correspond to an EBCDIC graphic, question marks (?) are written to those positions.

The action specified by the editing phrase letter C is identical to that specified by the editing phrase letter A except that characters are written from the leftmost portion of the list element.

Output Examples

Default Character Type	Internal Value	Editing Phrase	External String
8-bit	8"DEFGHI"	A9	bbbDEFGHI
6-bit	6"BCDEFGHI"	A9	bBCDEFGHI
8-bit	4"0000000000"8"A"	A4	???A
6-bit	6"0000ABCD"	A4	ABCD
8-bit	8"ABCDEFG" (8-bit pointer FOR 7)	A1	bbbbABCDEFG
6-bit	6"ABCDEFG" (6-bit pointer FOR 7)	A4	DEFG
8-bit	8"DEFGHI"	C9	bbbDEFGHI
6-bit	6"BCDEFGHI"	C9	bBCDEFGHI
8-bit	8"ABCD"4"0000"	C5	ABCD?
6-bit	6"ABCD0000"	C4	ABCD
8-bit	8"ABCDEFG" (8-bit pointer FOR 7)	C1	bbbbABCDEFG
6-bit	6"ABCDEFG" (6-bit pointer FOR 7)	C4	ABCD

C Editing Phrase Letter

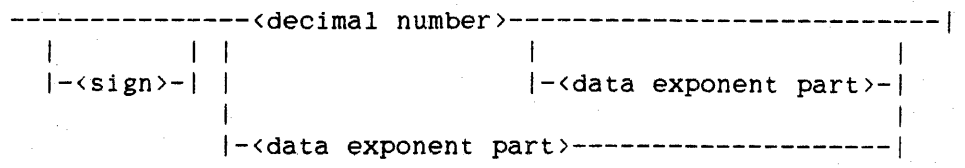
Refer to "A and C Editing Phrase Letters" above.

D Editing Phrase Letter

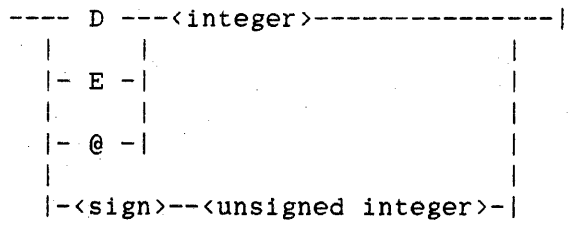
The editing phrase letter D is used for reading or writing floating-point values. Valid list elements are of type INTEGER, REAL, DOUBLE, COMPLEX, and BOOLEAN.

Input

The editing phrase letter D specifies that w characters of input data are to be read, converted to a real value, and assigned to the corresponding list element. The input data must be in the form of a data number; otherwise, a data error is returned. A data number is defined syntactically as follows:



<data exponent part>



The position of the decimal point in the internal value is determined by its position in the input data or by the value of d. If a decimal point appears in the input data, that position is used for the internal value. If no decimal point appears in the input data, one is assumed either d places to the left of the D, E, @, +, or - indicating the beginning of the exponent field or, if an exponent is not present, d places to the left of the right edge of the input field.

For example, if the editing phrase D7.2 is used to read the data number 10005.0, the resulting internal value is 10005.0. However, if the same editing phrase is used to read the data number 10005, the resulting internal value is 100.05.

The value of w must be greater than or equal to the value of d. Blanks are interpreted as zeros.

Input Examples

External String	Editing Phrase	Internal Value
bbbbbb25046	D11.4	+2.5046
bbbbb25.046	D11.4	+25.046
-bb25046E-3	D11.4	-0.0025046
-bbb25046-3	D11.4	-0.0025046
bb250.46D-3	D11.4	+0.25046
bbb250.46-3	D11.4	+0.25046
b-b25.04678	D11.4	-25.04678

Output

On output, the editing phrase letter D specifies that the value of the corresponding list element is to be converted to a string of characters that expresses the value in exponential notation. The string is written right-justified with blank fill to a field w characters wide. The value of the mantissa is rounded to the number of decimal places specified by d before it is written.

The value of w must be greater than or equal to d + 7. This width allows for a four-character exponent part, a decimal point, a digit preceding the decimal point, and a sign. If w is less than d + 7, the field is filled with asterisks (*).

The editing phrase letter D always uses four or seven characters to represent the exponent of the list element being written. The magnitude of the exponent determines which of the following three forms is used:

```
four-character:  D+xx or D-XX (where ABS(XX) <= 99)

four-character:  +XXX or -XXX (where 100 <= ABS(XXX) <= 999)

seven-character:  D+XXXXX or D-XXXXX
                  (where 1000 <= ABS(XXXXX) <= 99999)
```

Output Examples

Internal Value	Editing Phrase	External String
+36.7929	D13.5	bb3.67929D+01
-36.7929	D12.5	-3.67929D+01
-36.7929	D11.5	**********
+36.7929	D10.5	**********
1.234@@-73	D14.5	bbb1.2340OD-73
-789@@1234	D15.3	bb-7.890D+01236
6.54@@321	D9.2	b6.54+321

E Editing Phrase Letter

The action specified by the editing phrase letter E is identical to that specified by the editing phrase letter D except that, when used for output, an E, instead of a D, indicates the beginning of the exponent in the output string.

Output Examples

Internal Value	Editing Phrase	External String
+36.7929	E13.5	bb3.67929Eb01
-36.7929	E12.5	-3.67929Eb01

F Editing Phrase Letter

The editing phrase letter F is used when reading or writing floating-point values. Valid list elements are of type INTEGER, REAL, DOUBLE, COMPLEX, and BOOLEAN.

Input

On input, the action specified by the editing phrase letter F is identical to that specified by the editing phrase letter D.

Output

On output, the editing phrase letter F specifies that the value of the corresponding list element is to be converted to a string of characters that expresses the value in simple decimal notation. The string is written right-justified with blank fill to a field w characters wide. The value of the list element is rounded to the number of decimal places specified by d before it is written.

The value of w must be greater than or equal to d + 1. When writing negative values, w must also allow for the minus sign. The field contains asterisks (*) if the value to be written requires a field wider than w characters.

Output Examples

Internal Value	Editing Phrase	External String
+36.7929	F7.3	b36.793
+36.7934	F9.3	bbb36.793
-0.0316	F6.3	-0.032
0.0	F6.4	0.0000
0.0	F6.2	bb0.00
+579.645	F6.2	579.65
+579.645	F4.2	****
-579.645	F6.2	******

G Editing Phrase Letter

If used to read a BCL file, the editing phrase letter G specifies that eight 6-bit characters of the input data are to be skipped. If used to write to a BCL file, the editing phrase letter G specifies that eight BCL zeros are to be written to the output record.

If used to read an EBCDIC file, the editing phrase letter G specifies that six 8-bit characters of the input data are to be skipped. If used to write to an EBCDIC file, the editing phrase letter G specifies that six EBCDIC zeros are to be written to the output record.

H and K Editing Phrase Letters

The editing phrase letters H and K are used when reading or writing hexadecimal and octal values, respectively. Valid list elements are of type INTEGER, REAL, DOUBLE, COMPLEX, and BOOLEAN.

In the following explanation of the H and K editing phrase letters, Q is used. Q is derived from the following table:

		Editing Phrase Letter	
		H	K
Precision	Single	12	16
	Double	24	32

Input

The editing phrase letter H specifies that w characters of input data are to be read, converted to a hexadecimal value, and assigned to the corresponding list element. The editing phrase letter K specifies that w characters of input data are to be read, converted to an octal value, and assigned to the corresponding list element. The input data must consist only of characters from the set of hexadecimal characters, the blank, or the minus sign when H is specified, or characters from the set of octal characters, the blank, or the minus sign when K is specified; otherwise, a data error is returned. Leading, trailing, and embedded blanks are interpreted as zeros. If a minus sign appears in the input string, 1 is assigned to bit 46 of the list element (bit 46 of the first word of a double-precision list element).

If w is less than or equal to Q, the value is stored right-justified in the storage location (both words of a double-precision variable are included). Unused high-order bits are set to zero. If w is greater than Q, the leftmost w - Q characters must be blanks, zeros, or minus signs; otherwise, a data error is returned.

Input Examples

External String	Editing Phrase	Internal Value
6F	H2	4"00000000006F"
1FFFFFFFFFFF	H12	4"1FFFFFFFFFFF"
-16	H3	4"400000000016"
1234b568	H8	4"000012340568"
FFCb	H4	4"00000000FFC0"
00C1C2C3C4C5C6	H14	4"C1C2C3C4C5C6"
-ABCD	H5	4"4000000000000000000ABCD" (double precision)
123456789ABCDEF	H15	4"00000000123456789ABCDEF" (double precision)
16	K2	3"0000000000000016"
1777777777777777	K16	3"1777777777777777"
-16	K3	3"2000000000000016"
1234b56	K7	3"0000000001234056"
77b	K3	3"0000000000000770"
-567	K4	3"2000000000000000000000000000000567" (double precision)
1234567654321234567	K19	3"0000000000001234567654321234567" (double precision)

Output

On output, the editing phrase letter H specifies that the value of the corresponding list element is to be converted to a string of hexadecimal characters. The editing phrase letter K specifies that the value of the corresponding list element is to be converted to a string of octal characters. The output string is written right-justified with blank fill to a field w characters wide. If w is less than Q, only the contents of the rightmost w * 4 bits (when H is used) or w * 3 bits (when K is used) of the list element are converted. (A double-precision list element is treated as 96 contiguous bits.) The output string does not contain an explicit sign.

Output Examples

Internal Value	Editing Phrase	External Value
4"0000E5551010"	H5	51010
4"0000E5551010"	H12	0000E5551010
4"0000E5551010"	H16	bbbb0000E5551010
8"123456"	H12	F1F2F3F4F5F6
4"00000000000000012345678 (double precision)	H4	5678
8"123456789bbb" (double precision)	H24	F1F2F3F4F5F6F7F8F9404040
3"0005677701234445"	K5	34445
3"0005677701234445"	K16	0005677701234445
3"0005677701234445"	K18	bb0005677701234445
3"00000000000000000000000001234567" (double precision)	K4	4567

I Editing Phrase Letter

The editing phrase letter I is used when reading or writing integer values. Valid list elements are of type INTEGER, REAL, DOUBLE, COMPLEX, and BOOLEAN.

Input

The editing phrase letter I specifies that w characters of input data are to be read, converted to an integer value, and assigned to the corresponding list element. The data must be in the form of an ALGOL integer: otherwise, a data error is returned. Blank characters are interpreted as zeros. The magnitude of the value that can be read depends on the type of the list element.

Input Examples

External String	Editing Phrase	Internal Value
567	I3	+567
bb-329	I6	-329
-bbbb27	I7	-27
27bbb	I5	+27000
b-bb234	I7	-234

Output

On output, the editing phrase letter I specifies that the value of the corresponding list element is to be converted to a character string in the form of an ALGOL integer. The string is written right-justified with blank fill to a field w characters wide. The value of the list element is rounded to an integer before it is written.

Negative values are written with a minus sign; nonnegative values are written without a sign.

If the value of the list element requires a field larger than w, then w asterisks (*) are written.

Output Examples

Internal Value	Editing Phrase	External String
+23	I4	bb23
-79	I4	b-79
+67486	I5	67486
-67486	I5	*****
+978	I1	*
0	I3	bb0
+3.6	I2	b4

J Editing Phrase Letter

The editing phrase letter J is used when reading or writing integer values. Valid list elements are of type INTEGER, REAL, DOUBLE, COMPLEX, and BOOLEAN.

Input

On input, the action specified by the editing phrase letter J is identical to that specified by the editing phrase letter I.

Output

On output, the editing phrase letter J specifies that the value of the corresponding list element is to be converted to a character string in the form of an ALGOL integer. The string is written to a field equal in width to the length of the string. The value of the list element is rounded to an integer before it is written.

Negative values are written with a minus sign; nonnegative values are written without a sign.

If w is less than the number of characters required to express the value of the list element, w asterisks (*) are written.

Output Examples

Internal Value	Editing Phrase	External String
+23	J5	23
-23	J5	-23
+233	J3	233
-233	J3	***
0	J3	0
3.14, -12	2J10	3-12

K Editing Phrase Letter

Refer to "H and K Editing Phrase Letters" above.

L Editing Phrase Letter

The editing phrase letter L is used when reading or writing Boolean values. Valid list elements are of type INTEGER, REAL, DOUBLE, COMPLEX, and BOOLEAN.

Input

The editing phrase letter L specifies that w characters of input data are to be read, converted to one of the Boolean values TRUE or FALSE, and assigned to the corresponding list element. If the first nonblank character of the input data is a "T", bit zero of the list element is assigned the value 1: otherwise, bit zero is assigned the value zero. All other bits in the list element are assigned the value zero. An all-blank field yields the value FALSE. If the list element is a double-precision variable, the first word is assigned a value according to the rules just described, and the second word is set to zero.

Input Examples

External String	Editing Phrase	Internal Value
T	L1	TRUE (4"000000000001")
bbF	L3	FALSE (4"000000000000")
bbbTRU	L6	TRUE (4"000000000001")
b	L1	FALSE (4"000000000000")
T	L1	TRUE (4"00000000000100000000000") (double precision)

Output

On output, the editing phrase letter L specifies that "TRUE" is to be written to the output record if bit zero of the corresponding list element equals 1, and that "FALSE" is to be written if bit zero of the corresponding list element equals zero. If w is less than 5, the first w characters of "TRUE" or "FALSE" are written. If w is greater than 4, "TRUE" or "FALSE" is written right-justified with blank fill.

Output Examples

Internal Value	Editing Phrase	External String
0	L6	bFALSE
1	L5	bTRUE
2	L4	FALS
3	L3	TRU
4	L2	FA

O Editing Phrase Letter

The editing phrase letter O is used when data is to be read or written without editing. Valid list elements are of type INTEGER, REAL, DOUBLE, COMPLEX, BOOLEAN, and POINTER.

In the explanation of the editing phrase letter O below, Q is used. Q is derived from the following table:

		Precision		Pointers		
		Single	Double	4-bit	6-bit	8-bit
Default Character Type	BCL	8	16	12	8	6
	EBCDIC	6	12	12	8	6

Input

The editing phrase letter O specifies that the input data is to be assigned to the corresponding list element without editing. Q characters of input data are read, unless the corresponding list element is of the form

<pointer expression> FOR <arithmetic expression>

When the list element is of this form, the value of Q and the value of the arithmetic expression are compared, and the lesser value is the number of characters read.

Output

On output, the editing phrase letter O writes the value of the list element as an unedited string of characters. Q characters are written to the output record unless the corresponding list element is of the form

<pointer expression> FOR <arithmetic expression>

When the list element is of this form, the value of Q and the value of the arithmetic expression are compared, and the lesser value is the number of characters written.

Example

This example shows the use of the editing phrase letter O. The input and output data are also illustrated.

```
BEGIN
   FILE TD(KIND=REMOTE,MYUSE=IO);
   REAL R;
   READ(TD, <O>, R);
   WRITE(TD, <O>, R);
END.
```

Input	Output
A	A
ABCDEFGH	ABCDEF

R Editing Phrase Letter

The editing phrase letter R is used when reading or writing REAL values and can be used with the editing phrase letter S. Valid list elements are of type INTEGER, REAL, DOUBLE, COMPLEX, and BOOLEAN.

Input

On input, the action specified by the editing phrase letter R is identical to that specified by the editing phrase letter D except when it is immediately preceded by an S editing phrase.

Output

On output, the editing phrase letter R specifies that the value of the corresponding list element is to be converted to a string that expresses the value in either simple decimal or exponential notation. In general, if w is greater than or equal to the number of characters required to express the value of the list element using simple decimal notation, then simple decimal notation is used; if w is less than the number of characters required to express the value using simple decimal notation and greater than or equal to the number of characters required to express the value using exponential notation, then exponential notation is used; if w is less than the number of characters required to express the value using exponential notation, the field is filled with asterisks (*).

Examples

External Input String	List Element Type	Editing Phrase	External Output String
-.333333bb	REAL	R10.4	bbb-0.3333
-.333333bb	DOUBLE	R10.4	bbb-0.3333
-.333333bb	INTEGER	R10.4	bbbb0.0000
3333.333E2	DOUBLE	R10.4	3.3333D+05
3333.333E2	INTEGER	R10.4	3.3333E+05
-.333bbbbb	REAL	R10.9	**********
-.333bbbbb	INTEGER	R10.9	.000000000
333.333E2b	DOUBLE	R10.4	3.3333D+22
bbbbbbbbbbbbb1.23D12	REAL	R20.4	bb1230000000000.0000
bbbbbbbbbb1.23D12345	DOUBLE	R20.4	bbbbbbb1.2300D+12345
bbbb4.3@68	REAL	R10.4	4.3000E+68

S Editing Phrase Letter

The editing phrase letter S is used with an R editing phrase to provide a scale factor.

If the next editing phrase in the editing specifications does not contain the editing phrase letter R, the S editing phrase is ignored. When more than one S editing phrase appears in the editing specifications, each subsequent S editing phrase takes precedence over the preceding one.

Input

On input, the value of the input data corresponding to the subsequent R editing phrase is divided by

10 ** <scale factor>

before it is assigned to the list element.

Input Examples

External String	Editing Specifications	Internal Value
bbbb10000.	S2,R10.2	100.0
bbbbbb5.41	S1,R10.2	0.541
bbbbbb05.5	S1,R10.2	0.55
bbb5.01521	S-1,R10.2	50.1521
bbbbbbb541	S1,R10.2	0.541

Output

On output, the value of the list element corresponding to the subsequent R editing phrase is multiplied by

10 ** <scale factor>

before it is written to the output field.

Output Examples

Internal Value	Editing Specifications	External String
100.0	S2,R10.2	bb10000.00
0.54	S1,R10.2	bbbbbb5.40
0.0056	S1,R10.2	bbbbbb0.06
1.55	S-1,R10.2	bbbbbb0.16

T Editing Phrase Letter

The editing phrase letter T specifies that the buffer pointer is to be moved to character position w of the input or output record. The value of w must be greater than zero; if w is equal to zero, the buffer pointer is moved to the first character position in the record. No list element corresponds to this editing phrase letter.

Input Examples

External String	Editing Specifications	Internal Value
0123456789101112 13	T13,I3	111
0123456789101112 13	T1,I4	123
0123456789101112 13	T15,I4	1213
ABCDEFGHIJKLMNOPQR	T8,A6	HIJKLM

Output Example

```
BEGIN
   FILE DCOM(KIND=REMOTE,MYUSE=IO);
   ARRAY A[0:9];
   WRITE(DCOM, <T11,I3,T1,I3>, 123, 456):    % WRITE STATEMENT 1
   WRITE(DCOM, <T4,A3,T1,A2>, "ABC", "DE"); % WRITE STATEMENT 2
END.
```

The program above produces the following output:

```
WRITE statement 1:  b456bbbbbb123
WRITE statement 2:  DEbABC
```

U Editing Phrase Letter

The editing phrase letter U specifies that output data is to be edited as best suits the type of the corresponding list element. Valid list elements are of type INTEGER, REAL, DOUBLE, COMPLEX, and BOOLEAN.

Input

The editing phrase letter U is not implemented for input.

Output

On output, real, integer, and double-precision list elements are written using a format that combines readability with maximum numerical significance. Boolean values are written as "T" or "F" and occupy one character position in the record. String literals are treated as real values. If the number of characters required to express the list element is greater than the number left in the current record, the output is placed in the next record.

If w is specified and the number of characters required to express the list element is greater than w, the field is filled with asterisks (*).

If d is specified and d is greater than w, then d - w leading blanks are inserted before the field is written. Thus, when using the editing phrase letter U, the number of characters actually written cannot be less than d and can be greater than w.

Output Examples

Internal Value	Editing Phrase	External String
-123.4567	U	-123.4567
789	U	789
1.5@@275	U10	1.5D+275
1234567	U5	1.2+6
1	U10.4	bbb1
123.456	U10.4	123.456
1	U5.8	bbbbbbb1
123.456	U5.8	bbb123.5

V Editing Phrase Letter

The editing phrase letter V allows the type of editing to be specified at run time. The rightmost character of the first word of the next list element (or, if the list element is a pointer, the character pointed at) provides the editing phrase letter to be used to edit the data. Valid list elements are of type INTEGER, REAL, DOUBLE, COMPLEX, BOOLEAN, and POINTER.

The editing phrase letter extracted from the list element is a 6-bit character if the default character type is BCL; otherwise, it is an 8-bit character.

Example

```
      .
      .
      .
   REAL A,B;
   INTEGER I;

   FORMAT FMT1(V8.2),
          FMT2(2V*),
          FMT3(*V*.*);
      .
      .
      .
   READ(KARD,FMT1,"R",A);
   B := 4"C1";
   WRITE(LINE,FMT2,B,6,A,I);
   I := 4"C5";
   READ(KARD,FMT3,2,I,10,4,A,B);
      .
      .
      .
```

In the preceding program, in the first READ statement FMT1 evaluates to R8.2 and corresponds to the list element A; in the WRITE statement FMT2 evaluates to 2A6 and corresponds to the list elements A and I; and in the second READ statement FMT3 evaluates to 2E10.4 and corresponds to the list elements A and B.

For more information, refer to "Variable Editing Phrases" earlier in this section.

X Editing Phrase Letter

On input, the editing phrase letter X specifies that w characters of input are to be skipped. On output, the editing phrase letter X specifies that w blanks are to be written. No list element corresponds to this editing phrase letter.

Z Editing Phrase Letter

The editing phrase letter Z is used when reading or writing REAL values. Valid list elements are of type INTEGER, REAL, DOUBLE, COMPLEX, and BOOLEAN.

Input

On input, the editing phrase letter Z selects one of the editing phrase letters D, I, or L to specify editing action, depending on the type of the corresponding list element, as shown in the following table:

Type	Editing Phrase
REAL or DOUBLE	Dw.d
INTEGER	Iw
BOOLEAN	Lw

Output

The output string has a length of w characters regardless of the value or type of the list element being written. For Boolean list elements, Lw is used. For integer list elements, Iw is used. For real or double-precision list elements, D, E, or F editing is performed depending on the type of the list element and the magnitude of its value.

Output Examples

Internal Value	Editing Phrase	External String
1.23@@250	Z12.6	1.230000+250
1	Z5.1	bbbb1
12345	Z5.1	12345
12	Z8.7	bbbbbb12
12345.678	Z10.4	1.2346E+04
12	Z10.4	bbbbbbbb12
12345678	Z6	******
1234	Z6	bb1234

Editing Modifiers

Editing modifiers can be used to modify the editing performed by the editing phrase letters D, E, F, I, J, R, and Z. Editing modifiers are valid only for output.

P Editing Modifier

The P editing modifier specifies that a comma (,) is to be inserted immediately to the left of every third digit left of the decimal point.

$ Editing Modifier

The $ editing modifier specifies that a dollar sign ($) is to be inserted immediately to the left of the output string.

Examples

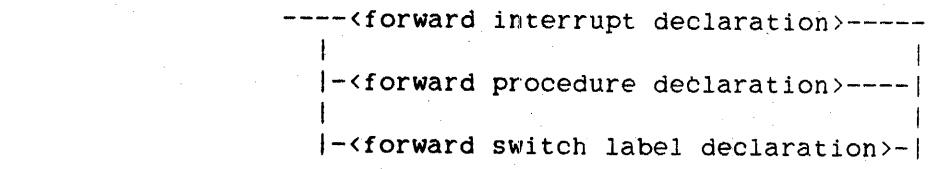
Internal Value	Editing Phrase	External String
17.347	$F10.2	bbbb$17.35
-1234567	PI10	-1,234,567
-1234567	P$Z15.2	bbbb$-1,234,567
1234567.11111	PF15.5	1,234,567.11111
1234567.1234	$PR15.5	bbb$1.23457E+06
1234567.1234	$PR15.0	bbbb$1,234,567.

FORWARD REFERENCE DECLARATION

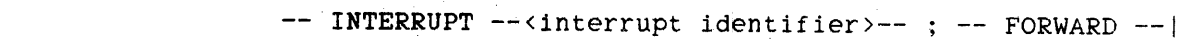
The forward reference declaration enables the ALGOL compiler to handle situations in which two procedures, two interrupts, or two switch labels make references to each other. Normally, a procedure, interrupt, or switch label must be declared before it can be used in a program. However, in the situation described above, regardless of which procedure, interrupt, or switch label is declared first, its body contains a reference to an undeclared entity. The forward reference declaration allows the compiler to recognize such entities before they have been declared in full.

Syntax

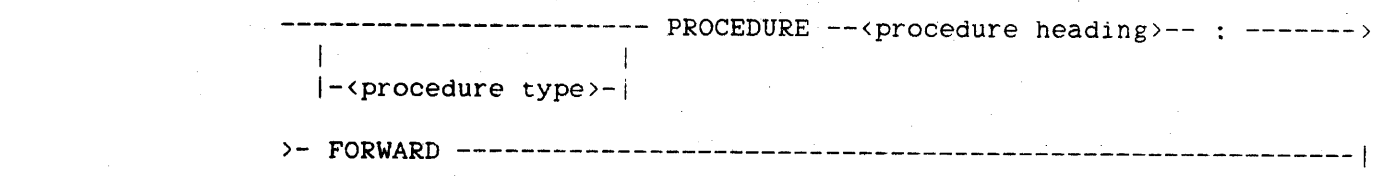
<forward reference declaration>



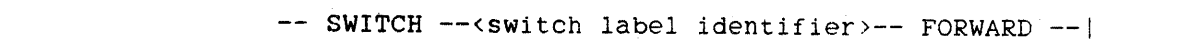
<forward interrupt declaration>



<forward procedure declaration>



<forward switch label declaration>



See also

Semantics

Suppose two procedures, PROC_ONE and PROC_TWO, make references to each other, and PROC_ONE appears before PROC_TWO in the source code. Before PROC_ONE is declared, the following forward reference declaration must appear:

```
PROCEDURE PROC_TWO; FORWARD
```

When PROC_ONE calls PROC_TWO, the compiler recognizes the second procedure. At some later point in the program, the second procedure, PROC_TWO, is declared in full.

Similar methods are used for mutually referencing interrupts and mutually referencing switch labels.

Examples

```
SWITCH SELECT FORWARD
```

Declares a forward reference to a switch label named SELECT. Later in the program, SELECT must be declared in full.

```
INTEGER PROCEDURE SUM(A,B,C);
   VALUE A,B;
   INTEGER A,B;
   REAL C;
   FORWARD
```

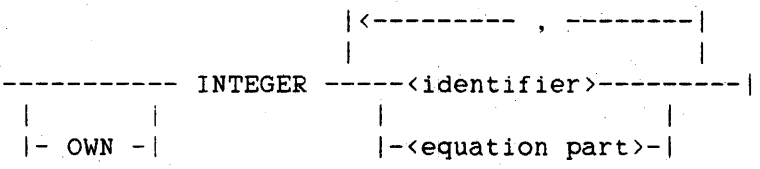
Declares a forward reference to an integer procedure named SUM. Later in the program, SUM must be declared in full, and its parameters must be the same in number and type as in this forward declaration.

INTEGER DECLARATION

An INTEGER declaration declares simple variables that can have integer values (arithmetic values that have exponents of zero and no fractional parts).

Syntax

<integer declaration>



<integer identifier>

An <identifier> that is associated with the INTEGER data type in an INTEGER declaration.

See also
<equation part> 55

Semantics

A simple variable declared to be OWN retains its value when the program exits the block in which the variable is declared, and that value is again available when the program re-enters the block in which the variable is declared.

The equation part causes the simple variable being declared to have the same address as the simple variable associated with the second identifier. This action is called "address equation." An identifier can be address-equated only to a previously declared local identifier or to a global identifier. The first identifier must not have been previously declared within the block of the equation part.

Address equation is allowed only between integer, real, and Boolean variables. Because both identifiers of the equation part have the same address, altering the value of either variable affects the value of both variables.

The following example demonstrates the effects of address-equating Boolean and integer variables:

```
BEGIN
   BOOLEAN B;
   INTEGER I = B;
   I := 4;              % B = FALSE  I = 4
   B := TRUE;           % B = TRUE   I = 1
END.
```

The OWN specification has no effect on an address-equated identifier. The first identifier of an equation part is own only if the second identifier of the equation part is own.

An INTEGER declaration with an equation part is not allowed in the global part of a program unit.

Pragmatics

When an arithmetic value is assigned to an integer simple variable, the value is rounded to an integer, if possible, before it is stored in the simple variable.

When an integer simple variable is allocated, it is initialized to zero (a 48-bit word with all bits equal tc zero). However, to ensure compatibility with ALGOL 60, programmers should explicitly initialize integer simple variables with appropriate assignment statements.

The appendix "Data Representation" contains additional information on the internal structure of an integer operand as implemented on A Series and B 5000/B 6000/B 7000 Series systems.

See also
Integer Operand 821

Examples

INTEGER INDEX

Declares INDEX as an integer simple variable.

```
INTEGER COUNT,VAL,NOEXPONENT
```

Declares COUNT, VAL, and NOEXPONENT as integer simple variables.

```
OWN INTEGER SAVEVALUE,MAX
```

Declares SAVEVALUE and MAX as integer simple variables. Because they are declared to be own, these simple variables retain their values when the program exits the block in which they are declared.

```
INTEGER INT = BOOL, CAL
```

Declares INT and CAL as integer simple variables, and address-equates INT to the previously declared simple variable BOOL. INT and BOOL share the same address.

INTERRUPT DECLARATION

The INTERRUPT declaration declares an interrupt and associates an unlabeled statement with it.

Syntax

<interrupt declaration>

```
-- INTERRUPT --<identifier>-- : --<unlabeled statement>--|
```

<interrupt identifier>

An <identifier> that is associated with an interrupt in an INTERRUPT declaration.

See also
<unlabeled statement> 220

Semantics

An interrupt provides a method of forcing a process to depart from its current point of control and execute the unlabeled statement associated with the interrupt by the INTERRUPT declaration.

After executing the unlabeled statement associated with an interrupt, a program usually returns to its previous point of control. However, the program does not return to this point if a GO TO statement is executed within the unlabeled statement and the specified designational expression references a statement outside of the unlabeled statement.

Once an interrupt is declared, it is enabled until it is explicitly disabled with the DISABLE statement. The DISABLE statement can temporarily render the associated interrupt ineffective. The ENABLE statement is used to re-enable a disabled interrupt.

For an interrupt to be used, the interrupt identifier must be attached to an event through the ATTACH statement. An interrupt can be detached from an event through the DETACH statement.

Pragmatics

An INTERRUPT declaration can be thought of as describing an unlabeled statement (which can be a block) that is automatically entered on the occurrence (CAUSE) of an event. The Master Control Program (MCP) ensures that when a program is executing the unlabeled statement associated with an interrupt, all other interrupts are queued until the program exits the unlabeled statement.

For more information, refer to "ATTACH Statement," "DETACH Statement," "DISABLE Statement," and "ENABLE Statement."

Examples

```
INTERRUPT ERR; GO TO ABORT
```

Declares ERR to be an interrupt and associates the statement "GO TO ABORT" with it.

```
INTERRUPT BLOCK1;
   BEGIN
   DISPLAY("ERROR");
   DISPLAY("INTERRUPT BLOCK1 OCCURRED");
   END
```

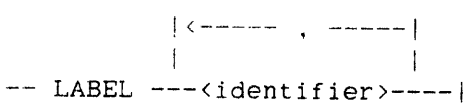
Declares BLOCK1 to be an interrupt. When BLOCK1 is invoked, two messages are displayed. Because no GO TO statement occurs within the declaration, after the interrupt code is executed, the program continues from the point at which the interrupt occurred.

LABEL DECLARATION

A LABEL declaration declares each identifier in the declaration to be a label.

Syntax

<label declaration>



<label identifier>

An <identifier> that is associated with a label in a LABEL declaration.

Semantics

Label identifiers can be used as the targets of GO TO statements and as labels in READ and WRITE statements.

A label identifier must appear in a LABEL declaration within the innermost block in which the label identifier is used to label a statement.

Examples

```
LABEL START
```

Declares START as a label.

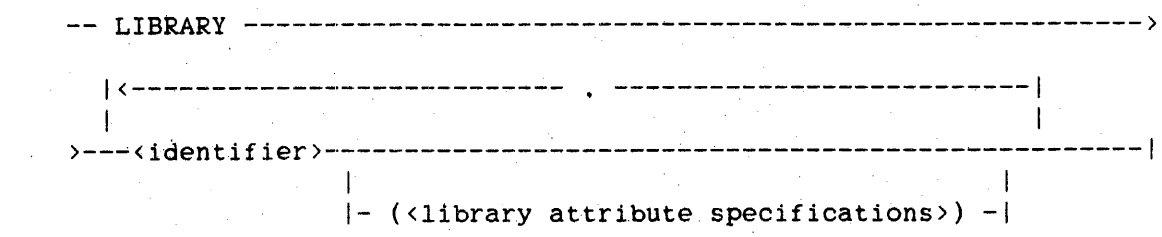
```
LABEL ENTER,EXIT,START,LOOP
```

Declares ENTER, EXIT, START, and LOOP as labels.

LIBRARY DECLARATION

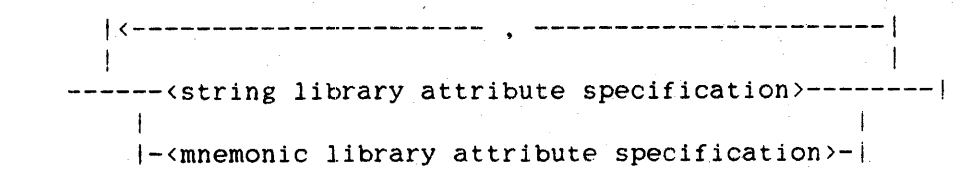
The LIBRARY declaration declares a library identifier and specifies values for the library attributes associated with the library. The library identifier can be used by a program to access entry points in the library.

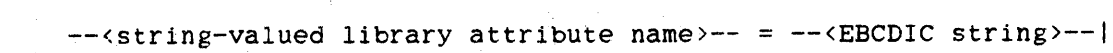
Syntax

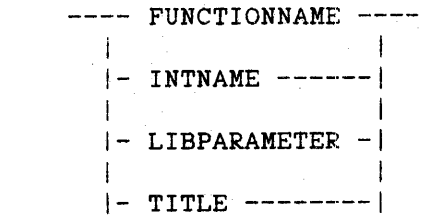


<library identifier>

An <identifier> that is associated with a library in a LIBRARY declaration.







<mnemonic library attribute specification>

```
    --<mnemonic-valued library attribute name>-- = -------------------->

    >-<mnemonic library attribute value>-------------------------------|
```

<mnemonic-valued library attribute name>

```
    -- LIBACCESS --|
```

<mnemonic library attribute value>

```
    ---- BYFUNCTION ----|
      |                 |
      |- BYTITLE ----|
```

Semantics

The LIBRARY declaration appears in a program that accesses a library. The LIBRARY declaration can be used to assign values to the library attributes of a library. In a program that calls a library, the library identifier also appears in the PROCEDURE declarations for the library entry points.

When a value is assigned to the TITLE attribute, the EBCDIC string must be a properly formed file title as defined in the "Work Flow Language (WFL) Reference Manual," and must have a period as its last nonblank character within the quotation marks (").

When a value is assigned to the INTNAME attribute, the EBCDIC string must have a period as its last character and can have leading blanks. The sequence of characters beginning with the first nonblank character up to, but not including, the next blank or period constitutes the INTNAME and must be a "valid identifier," where "valid identifier" is defined to be any sequence of characters beginning with a letter and consisting of letters, digits, hyphens (-), and underscores (_). Blanks can be present between the INTNAME and the period.

Specification of the TITLE and INTNAME attributes is optional; by default, the library identifier being declared is used for the TITLE and INTNAME. If the INTNAME is given and the TITLE is not, the INTNAME is also used for the TITLE.

The EBCDIC string assigned to the LIBPARAMETER attribute is used as a parameter to a selection procedure during dynamic library linkage.

Pragmatics

Libraries can be declared in any block of a user program. The library and its entry points are valid within the scope of the block; when the block is exited. the linkage to the library is broken, and the count of users of the library is decremented.

The chapter "Interface to the Library Facility" contains extended examples of libraries and programs that use libraries, as well as information about library attributes, library linkage, and library usage in general.

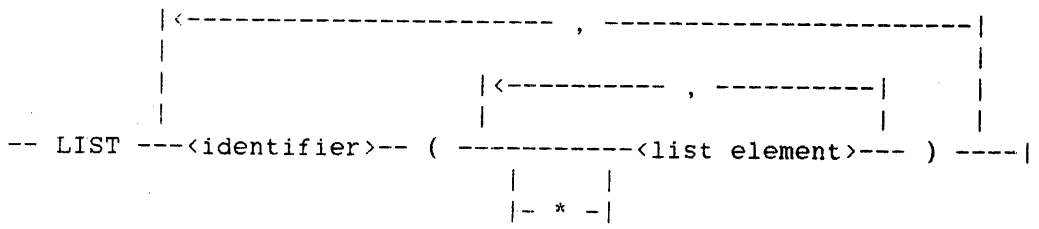
See also
Library Attributes. 665
Library Examples. 671
Linkage Provisions. 659

LIST DECLARATION

A LIST declaration associates an ordered set of list elements with a list identifier. The list identifier is used in a READ statement or WRITE statement to indicate which entities are to be read or written.

Syntax

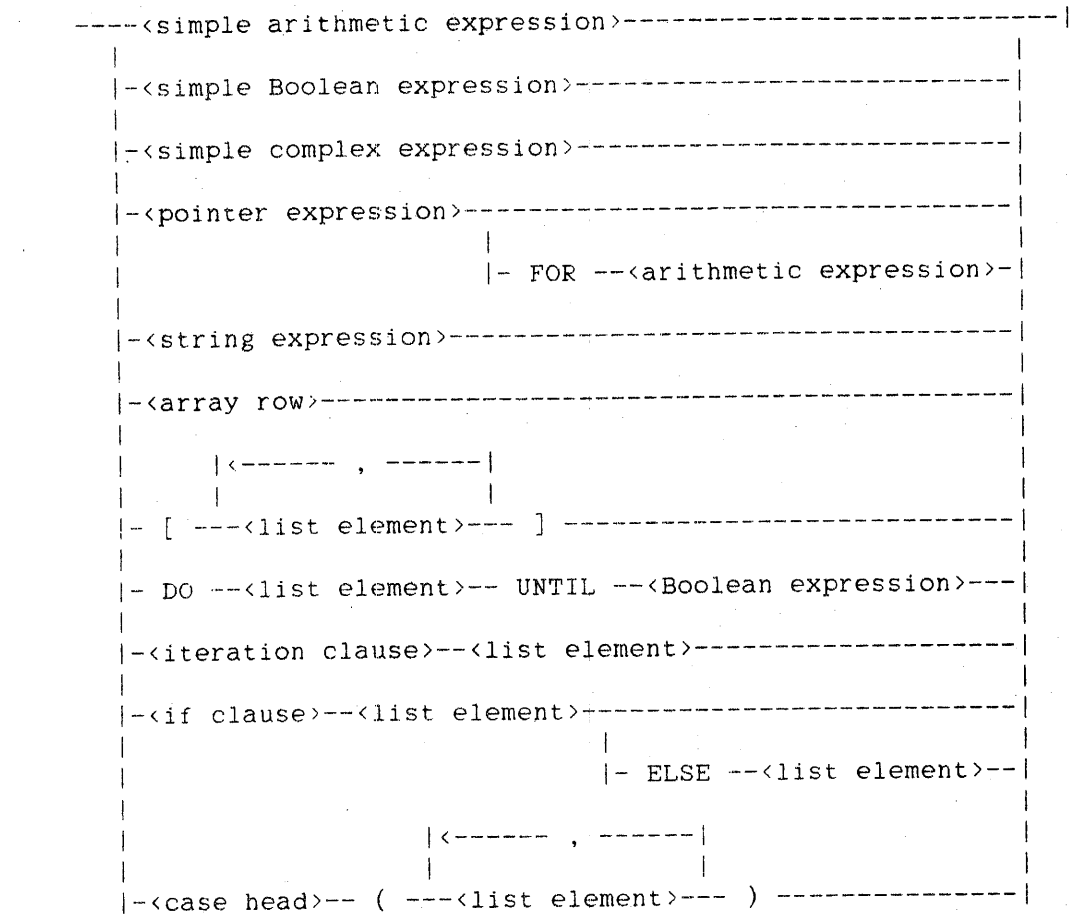
<list declaration>



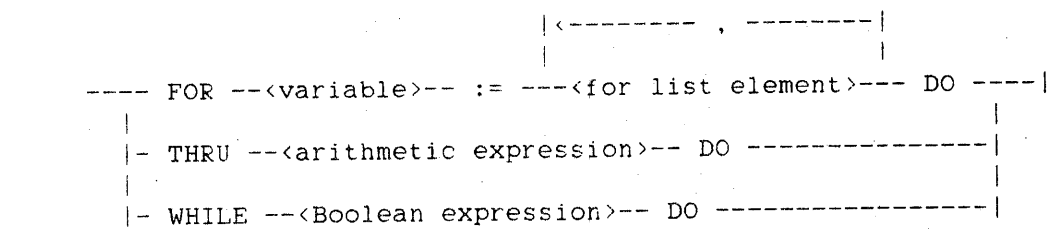
<list identifier>

An <identifier> that is associated with a set of list elements in a LIST declaration.

<list element>



<iteration clause>



See also

Semantics

Although the syntax of the READ statement and WRITE statement allows the list elements to be listed within the statement itself, a LIST declaration provides a way to associate a list identifier with a specific group of list elements.

A simple complex expression or complex value appearing in a list is considered to be a pair of real values: the first value is the real part of the complex value, and the second is the imaginary part.

List elements of the form

    <pointer expression> FOR <arithmetic expression>

allow the user to specify the number of characters to be read to or written from the pointer-specified location.

An array row appearing in a list is interpreted as a sequence of variables of the same type as that of the array. A complex array row is considered to be a real array row containing the real and imaginary parts of the complex values in the following order: the real part of the first element, the imaginary part of the first element, the real part of the second element, the imaginary part of the second element, and so on.

A string variable is a valid list element for editing phrase letters A and C and for free-field formatting. A string variable acts in the same manner as "<pointer expression> FOR <arithmetic expression>" when used with the A and C editing phrases. (For more information about the A and C editing phrases, refer to "FORMAT Declaration.")

Asterisks (*) prefixed to list elements have meaning only for free-field output; they are ignored for other types of I/O. An asterisk prefixed to a list element causes the text of the list element and an equal sign to be written to the left of the edited value of the list element.

Examples

```
LIST L1 (X,Y,A[4,*],FOR I := 2 STEP 1 UNTIL 5 DO B[I])
```

Declares L1 as a list identifier for the list consisting of X, Y, the array row A[4,*], and B[2], B[3], B[4], and B[5]. This list identifier might appear in a WRITE statement such as

```
WRITE (LP_OUT,//,L1);
```

```
LIST ANSWERS (P + Q,Z,SQRT(R)),
     RESULTS (X1,X2,X3,X4/2)
```

Declares ANSWERS and RESULTS as two list identifiers with associated list elements.

```
LIST LIST3 (FOR I := 0 STEP 1 UNTIL 10 DO
              FOR J := 0,3,6 DO
                 A[I,J])
```

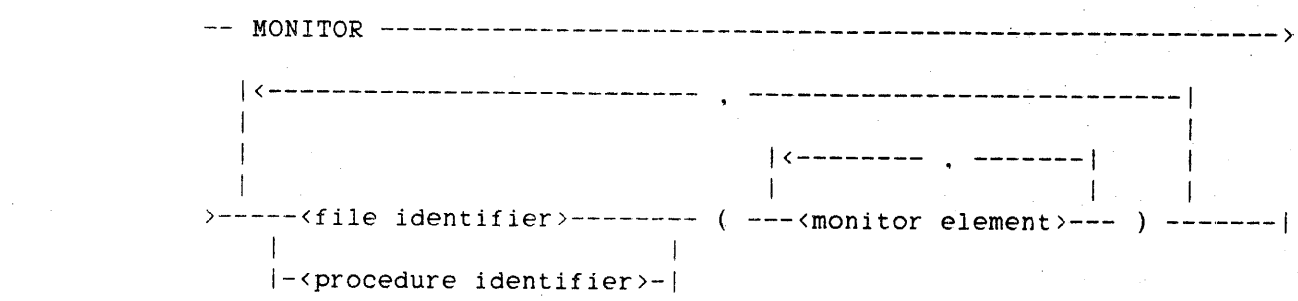
Declares LIST3 as a list identifier with an associated list consisting of nested FOR clauses indexing array A. This list identifier can be used in a READ statement to read the specified elements of array A.

MONITOR DECLARATION

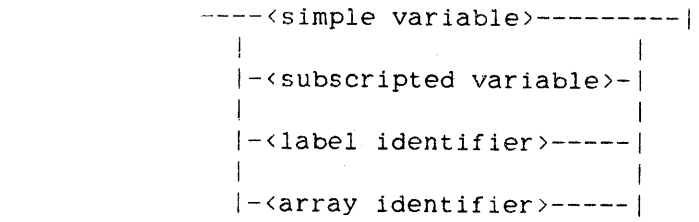
The MONITOR declaration designates items to be monitored during execution of the program and the method by which they are monitored. The monitor declaration is used when diagnostic information is needed.

Syntax

<monitor declaration>



<monitor element>



See also

Semantics

Each time an identifier designated as a monitor element is used in one of the ways described in this section, the identifier and its current value are written to the file or passed as parameters to the procedure specified in the MONITOR declaration.

The monitor action does not occur within procedures that are declared before the MONITOR declaration is encountered, nor does monitoring of a variable in the monitor list occur if this identifier is passed as an actual parameter to a call-by-name formal parameter that is modified within the procedure. The control variable in a FOR statement cannot be monitored. The monitor action does not occur when a value changes as the result of a READ statement or a REPLACE statement.

The diagnostic information produced depends on the forms of the monitor elements. When the LINEINFO compiler control option is TRUE and a file identifier is specified in the MONITOR declaration, a stack number, an at sign (@), a code address, and a sequence number are printed in front of the symbolic name of the monitor element (for example, "0143 @ 003:0003:4 (00007000)"). Diagnostic information is given for the specified monitor elements as follows:

1. When the monitor element is a simple variable or a subscripted variable, the symbolic name and the previous and new values of the variable are printed (for example, "B =0:=13").

2. When the monitor element is a label identifier, the symbolic name of the label is shown (for example, "LABEL L").

3. If the monitor element is an array identifier, the symbolic name of the array, the subscript of the element, and the previous and new values of the changed array element are printed (for example, "A[12] =0:=12").

When a procedure identifier is specified in the MONITOR declaration, printing of the monitor element must be performed by the procedure. Also, the monitoring procedure performs the specified operations depending on the values passed to it.

When the monitor element is a simple variable, the format of the monitoring procedure must be as follows:

```
REAL PROCEDURE MON(NAME,VAL,SPELL);
```

The procedure must be of the same type as the monitor elements. The procedure must have three parameters:

1. The first parameter, NAME, is a call-by-name parameter of the same type as the monitor element. NAME is passed a reference to the monitor element, and it is normally used to store the value of the second parameter, VAL.

2. The second parameter, VAL, is also of the same type as the monitor element, but it is a call-by-value parameter and is passed the new value to be assigned to the monitor element.

3. The third parameter, SPELL, must be a call-by-value real variable that is passed the name of the monitor element as a string of characters. Only the first six characters of the symbolic name are passed to this formal parameter. If the symbolic name is less than six characters long, it is left-justified, and trailing blanks are added, up to six characters.

If the monitor element is to be assigned a value, this assignment must be done by the monitoring procedure. This value can also be assigned to the procedure value to be used, for example, in evaluating the remainder of an expression in which the assignment is embedded. In the example on the next page using the array identifier form, the assignment statement "NAME := MON := VAL;" allows the subsequent use of the value assigned to the monitor element.

When the monitor element is a label identifier, the format of the monitoring procedure must be as follows:

```
PROCEDURE MON(SPELL);
```

The procedure must be untyped and must have only one parameter. This parameter is a call-by-value real variable that is passed the first six characters of the symbolic name. Only the first six characters of the symbolic name are passed to this formal parameter. If the symbolic name is less than six characters long, it is left-justified, and trailing blanks are added, up to six characters. For example, the monitoring procedure could compare this name to the symbolic names in the monitor list in order to identify a particular label.

When the monitor element is an array identifier, the declaration of the monitoring procedure must be as follows:

```
REAL PROCEDURE MON(D1,...,Dn,NAME,VAL,SPELL);
```

The parameters D1,...,Dn of the procedure are index parameters that are passed the subscripts for each dimension of the array element that is modified. There must be as many index parameters as the array has dimensions. Each index parameter is a call-by-value integer. The last three parameters are the same as in the simple variable form, except that NAME and VAL are simple variables of the same type as the array.

The value being assigned to the array element can also be assigned to the procedure value to be used, for example, to evaluate the remainder of an expression containing the array element.

The following procedure could be used to monitor a two-dimensional real array so that the values in the array never become negative.

```
REAL PROCEDURE MON(D1,D2,NAME,VAL,SPELL);
VALUE D1,D2,VAL,SPELL;
REAL NAME,VAL;
REAL SPELL;
INTEGER D1,D2;
   BEGIN
   IF VAL < 0 THEN
      GO TO ERROREXIT; % BAD GO TO
   NAME := MON := VAL; % RETURN VALUE FOR FURTHER USE
   END;
```

The occurrence of the statement

```
B := A[I,J] := 4;
```

where A is monitored by MON, is equivalent to the statement

```
B := MON(I,J,A[I,J],4,"A");
```

Pragmatics

For a debugging feature, refer to the TADS compiler control option in the chapter "Compiling Programs."

See also

‹TADS option› 643

Examples

MONITOR FYLE (A)

Declares the simple variable A to be a monitor element. When A is used in the ways described above, monitoring information on A is written to file FYLE.

```
100 BEGIN
200
300     FILE TERMOUT(KIND=REMOTE);
400     INTEGER I;
500     LABEL FINISH;
600     ARRAY MON1[0:3],
700           MON2[0:3];
800     MONITOR TERMOUT (I,MON1,MON2[1],FINISH);
900
1000     I := 27;
1100     MON1[0] := I;
1200     MON2[0] := 23;
1300     MON2[1] := MON1[0] * 2;
1400     GO TO FINISH;
1500 FINISH:
1600 END.
```

In the above program, simple variable I, array MON1, subscripted variable MON2[1], and label FINISH are monitored. When the program is executed, the following output is written to the terminal:

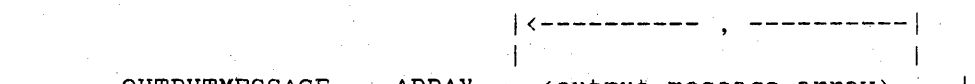
```
0148 @ 003:000E:4 (00001000)  I      =0:=27     (4"00000000001B")
0148 @ 003:0013:4 (00001100)  MON1   [0]=0:=27  (4"00000000001B")
0148 @ 003:0020:4 (00001300)  MON2   [1]=0:=54  (4"000000000036")
0148 @ 003:0024:4 (00001500)  LABEL FINISH
```

OUTPUTMESSAGE ARRAY DECLARATION

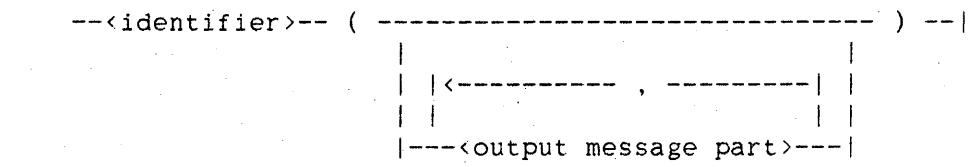
An OUTPUTMESSAGE ARRAY declaration declares output message arrays. An output message array contains output messages to be used by the MultiLingual System (MLS). For a description of how to use these arrays, refer to "MESSAGESEARCHER Statement."

Syntax

<output message array declaration>



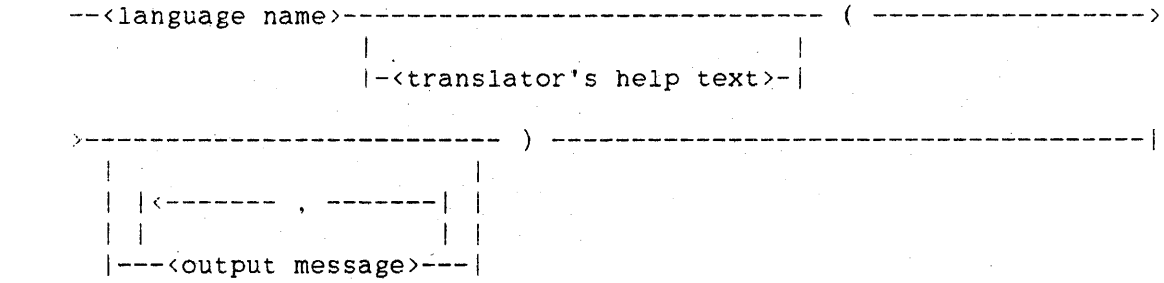
<output message array>

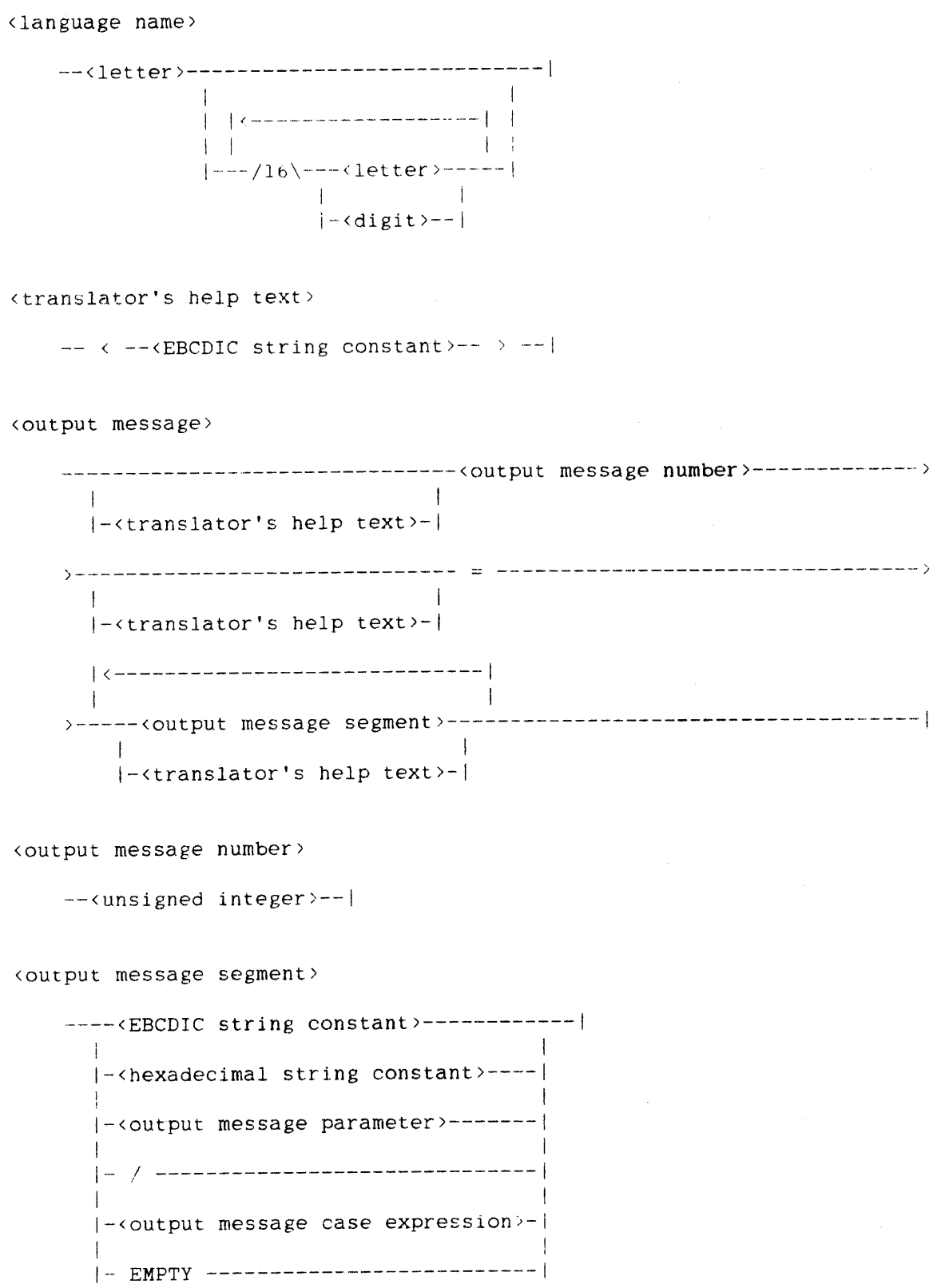


<output message array identifier>

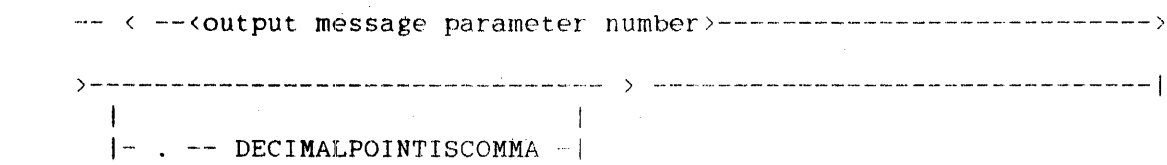
An <identifier> that is associated with an output message array in an OUTPUTMESSAGE ARRAY declaration.

<output message part>



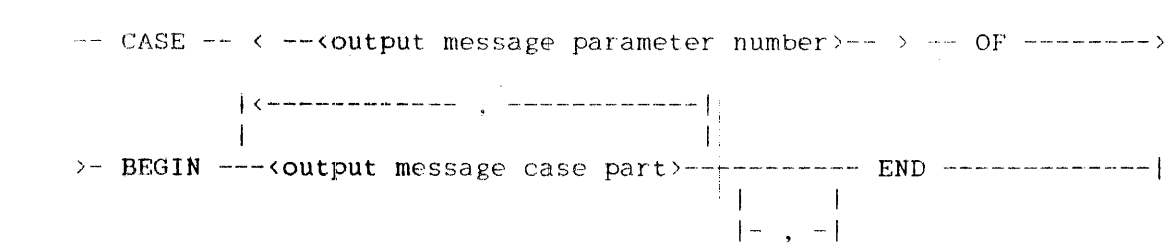

<language name>
<letter>
/16\
<letter>
<digit>
<translator's help text>
<EBCDIC string constant>
<output message>
<output message number>
<translator's help text>
=
<translator's help text>
<output message segment>
<translator's help text>
<output message number>
<unsigned integer>
<output message segment>
<EBCDIC string constant>
<hexadecimal string constant>
<output message parameter>
/
<output message case expression>
EMPTY

<output message parameter>

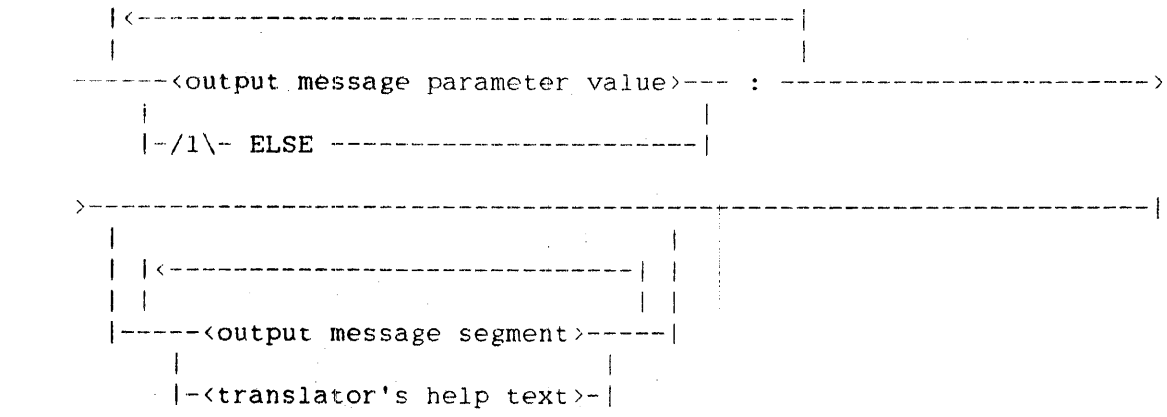


<output message parameter number>

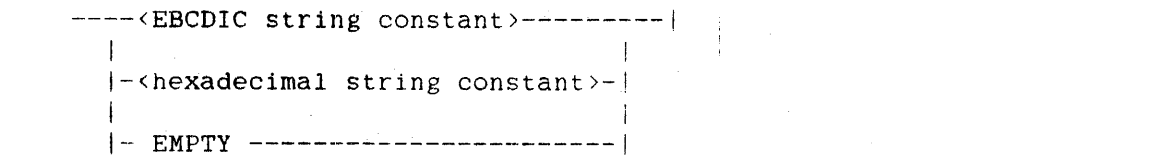
<output message case expression>



<output message case part>



<output message parameter value>



See also
<EBCDIC string constant>. 524
<hexadecimal string constant> 525

Semantics

The OUTPUTMESSAGE ARRAY declaration is part of the ALGOL interface to the MultiLingual System (MLS), which allows the user to word system messages in various human languages.

Each output message array identifier must be unique throughout the entire program. This requirement is an exception to the description of the scope of identifiers given in the chapter "Program Structure."

An output message number must be less than eight digits long. For each output message part, the output message number must uniquely identify an output message. This means that a number is assigned to one and only one output message segment, and each output message segment has only one number assigned to it.

An output message parameter number represents a parameter to be substituted into the message when the MESSAGESEARCHER statement is executed. The number identifies which parameter is to be substituted. The output message parameters are numbered consecutively from 1 through n, where n is the number of parameters in the output message.

DECIMALPOINTISCOMMA indicates that any decimal point (.) appearing in the parameter value corresponding to the preceding output message parameter number is changed to a decimal comma (,), and all commas are changed to decimal points.

A slash (/) causes both a carriage return character (48"0D") and a line feed character (48"25") to be inserted into the completed output message.

If an output message case expression does not contain an ELSE clause and no case exists for the value of the output message parameter, then the result of the output message case expression is a null string and an error result is returned with the completed output message. The program requesting the output message can decide whether or not the partially formed output message should be used.

When multiple output message parts occur within the same output message array, they define the same output messages for different languages. Multiple output message arrays can be used to define different groups of output messages.

Defines are expanded within an OUTPUTMESSAGE ARRAY declaration.

The translator's help text is displayed by the Message Translation Utility when an output message is being translated. (For more information on the Message Translation Utility, refer to the "Message Translation Utility User's Guide.") The translator's help text can occur before or after an output message segment. It can also appear before or after an output message number. If translator's help text needs to appear with all output messages in the language, then the translator's help text is placed after the language name (and before the left parenthesis).

Examples

```
OUTPUTMESSAGE ARRAY ERRORS (
   ENGLISH (
      10 = "POSITIVE INTEGER EXPECTED.",
      20 = "TOO MANY PARAMETERS."
           ),
   FRANCAIS (
      10 = "DEMANDE UN ENTIER POSITIF.",
      20 = "TROP DE PARAMETRES."
           ));
```

The output message array ERRORS shows an OUTPUTMESSAGE ARRAY declaration with the same output messages in two languages. The language of the user and the output message number determine the output message that is selected from this array.

```
OUTPUTMESSAGE ARRAY SUMMARY (
   ENGLISH (
      100 =
         "THIS PROGRAM IS TO BE EXECUTED WITH "
         CASE <1> OF
            BEGIN
            "1": "MAX PROCESSING TIME " <2> " SEC.",
            "2": "MAX I/O TIME " <3> " SEC.",
            "3": "MAX PROCESSING TIME " <2> " SEC., MAX "
                 "I/O TIME " <3> " SEC."
            END
           ),
```

```
FRANCAIS (
   100 =
      "CE PROGRAMME DOIT S'EXECUTER EN MOINS DE "
      CASE <1> OF
         BEGIN
         "1": <2, DECIMALPOINTISCOMMA>
              " SEC. DE CALCUL.".
         "2": <3, DECIMALPOINTISCOMMA> " SEC. D'E/S.",
         "3": <2, DECIMALPOINTISCOMMA>
              " SEC. DE CALCUL OU "
              <3, DECIMALPOINTISCOMMA> " SEC. D'E/S."
         END
         ));
```

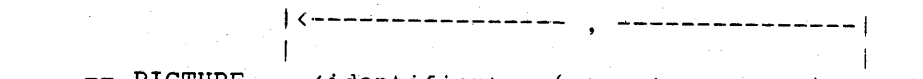
The output message array SUMMARY shows an OUTPUTMESSAGE ARRAY declaration with parameters. The first parameter value is not used as part of the message, but rather to select among case alternatives. The second and third parameters are conditionally inserted into the message, based on the value of the first parameter. Note that the second and third parameters are not necessarily both used. When the message is given in the language FRANCAIS, decimal points in the values of parameters 2 and 3 are changed to decimal commas.

PICTURE DECLARATION

The PICTURE declaration declares pictures that are then used in REPLACE statements to perform general editing of characters.

Syntax

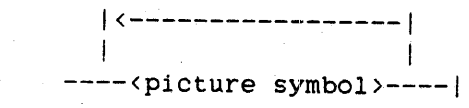
<picture declaration>



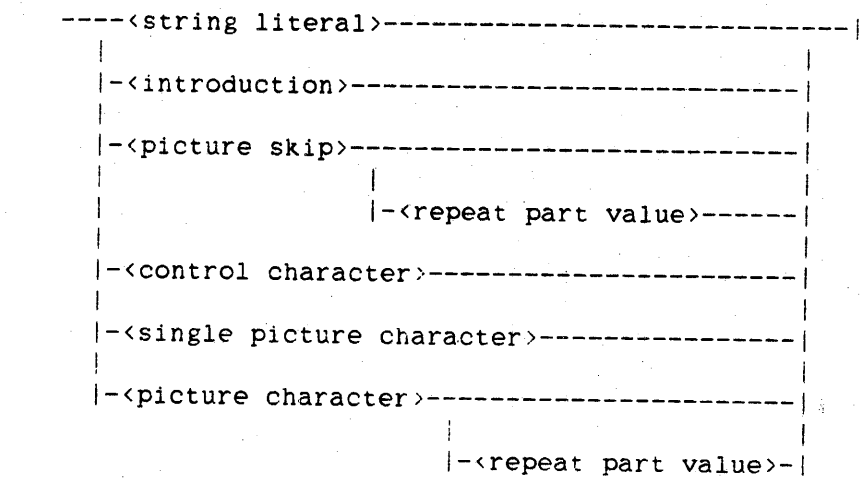
<picture identifier>

An <identifier> that is associated with a picture in a PICTURE declaration.

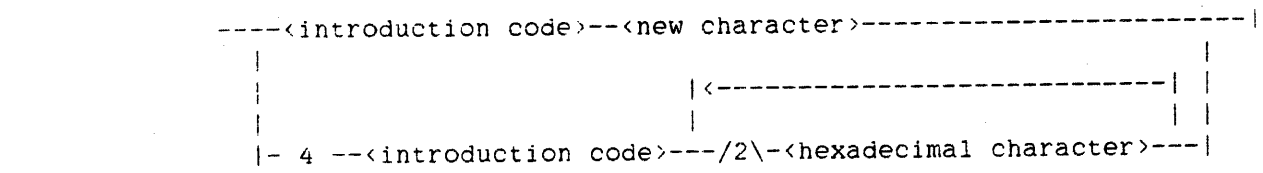
<picture>



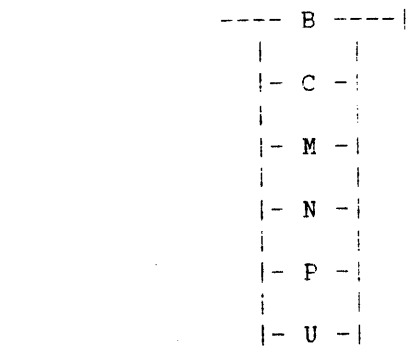
<picture symbol>



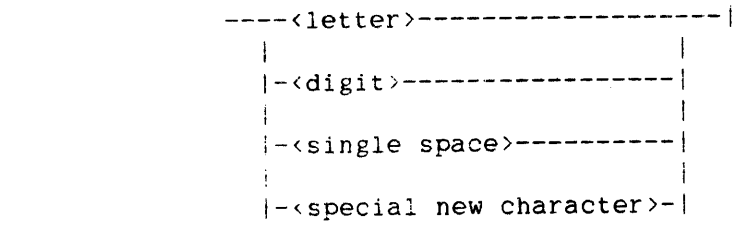
‹introduction›



‹introduction code›



‹new character›



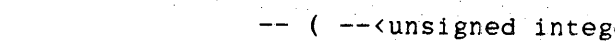
‹special new character›

Any of the following special characters:

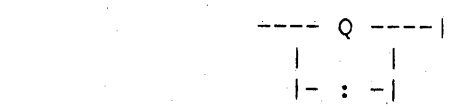
```
.  ,  [  ]  (  )  +  -  /
›  ‹  =  %  &  *  #  @  :
;  $  "
```

‹picture skip›

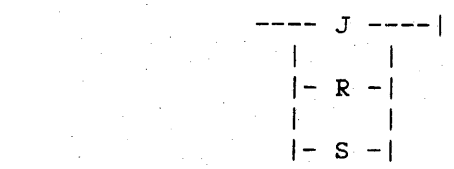
<repeat part value>



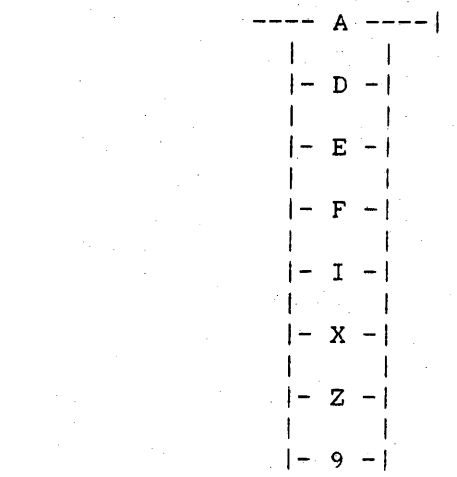
<control character>



<single picture character>



<picture character>



Semantics

A picture is used in a REPLACE statement to perform generalized editing functions as characters are transferred from a source location to a destination. The following editing operations can be performed:

1. Unconditional character moves

2. Moves of characters with leading zero editing

3. Moves of characters with leading zero editing and floating character insertion

4. Moves of characters with conditional character insertion

5. Moves of characters with unconditional character insertion

6. Moves of only the numeric parts of characters

7. Forward and reverse skips of source characters

8. Forward skips of destination characters

9. Insertion of an overpunch sign on the previous character

A picture consists of a named string of picture symbols enclosed in parentheses. The picture symbols specify the editing to be performed and can be combined in any order to perform a wide range of editing functions.

Flip-flops Used by Picture Symbols

Two hardware flip-flops affect the operation of certain picture symbols: the float flip-flop (FLTF) and the external sign flip-flop (EXTF).

The value of FLTF affects the function performed by the picture symbols D, E, F, J, R, and Z. FLTF is set to zero at the beginning of every picture. The picture symbols E, F, and Z may change the value of FLTF to 1, and the picture symbols J, R, and : unconditionally assign zero to FLTF.

The value of EXTF affects the function performed by the picture symbols E, F, J, Q, R, and S. EXTF is not assigned a value by the REPLACE statement that is using the picture; EXTF is in the state in which it was left after the most recent operation that affected it. For example, a REPLACE statement of the form

```
REPLACE <destination> BY <arithmetic expression>
     FOR <arithmetic expression> DIGITS
```

sets EXTF to reflect the sign of the first arithmetic expression: 1 if the arithmetic expression is positive, and zero if it is negative.

Character Fields

Pictures can act on both EBCDIC and hexadecimal characters. In the descriptions of the picture symbols, the term "numeric field" is used to mean either an entire hexadecimal character or the rightmost four bits of an EBCDIC character. The term "zone field" is used to mean the leftmost four bits of an EBCDIC character.

Characters Used by Picture Symbols

Certain picture symbols implicitly define characters to be inserted into the destination. These characters are referred to as the "insert character," "zero character," "nonzero character," "minus character," "plus character," and "dollar character."

The insert character is the character inserted into the destination by the picture symbol I. It is, by default, the period (.), and it can be changed by the introduction code N.

The zero character is used by the picture symbol D, and by the picture symbols E, F, and Z for leading zero replacement. It is, by default, the blank character, and it can be changed by the introduction code B.

The nonzero character is used by the picture symbol D. It is, by default, the comma (,), and it can be changed by the introduction code C.

The minus character is used by the picture symbols E, R, and S. The default minus character is the hyphen (-), and it can be changed by the introduction code M.

The plus character is used by the picture symbols E, R, and S. The default plus character is the plus (+), and it can be changed by the introduction code P.

The dollar character is used by the picture symbols F and J. The default dollar character is the dollar sign ($), and it can be changed by the introduction code U.

String Literals

If a string literal appears in a picture, the string is inserted into the destination. If the destination is EBCDIC, the string is inserted unchanged. If the destination is hexadecimal, only the numeric fields of the characters of the string are inserted into the destination.

Introduction Codes

The introduction codes can be used to change the implicit characters used by some of the picture symbols. The <introduction> construct specifies the new character to be used. If two hexadecimal characters are used to specify the new character, they are assumed to represent a single EBCDIC character.

Introduction Code	Action
B	Specifies the zero character to be used by D, E, F, and Z. The default zero character is the blank character.
C	Specifies the nonzero character to be used by D. The default nonzero character is the comma (,).
M	Specifies the minus character to be used by E, R, and S. The default minus character is the hyphen (-).
N	Specifies the insert character to be used by I. The default insert character is the period (.).
P	Specifies the plus character to be used by E, R, and S. The default plus character is the plus (+).
U	Specifies the dollar character to be used by F and J. The default dollar character is the dollar sign ($).

Picture Skip

The picture skip characters are described in the following table. If a repeat part value is given with the picture symbol, the unsigned integer in the repeat part value specifies how many characters are skipped in the source. If no repeat part value is given, one character is skipped in the source.

Character	Action
>	The source pointer is skipped forward (to the right) the specified number of characters.
<	The source pointer is skipped backward (to the left) the specified number of characters.

Control Characters

The control characters are described in the following table.

Character	Action
Q	If EXTF = 1, a 4"D" character is inserted into the zone field of the preceding destination character. If EXTF = 0, the destination character is not altered. The destination pointer must be EBCDIC, and it is left pointing to the same character that it was pointing to before the Q action was taken.
:	FLTF is unconditionally assigned zero.

Single Picture Characters

The single picture characters are described in the following table.

Character	Action
J	If FLTF = 0, the dollar character is inserted into the destination. If FLTF = 1, no character is inserted, and the destination pointer is not advanced. FLTF is then assigned zero. If the destination is hexadecimal, only the numeric field of the dollar character is inserted.

Character	Action
R	If FLTF = 0 and EXTF = 0, the plus character is inserted into the destination. If FLTF = 0 and EXTF = 1, the minus character is inserted into the destination. If FLTF = 1, no character is inserted, and the destination pointer is not advanced. FLTF is then assigned zero. If the destination is hexadecimal, only the numeric field of the plus or minus character is inserted.
S	If EXTF = 1, the minus character is inserted into the destination; otherwise, the plus character is inserted into the destination. The destination must be EBCDIC.

Picture Characters

The picture characters are described in the following table. If a repeat part value is given with the picture symbol, the unsigned integer in the repeat part value specifies how many characters are skipped, inserted, or transferred from the source to the destination. If no repeat part value is given, one character is skipped, inserted, or transferred from the source to the destination.

Character	Action
A	The specified number of characters are transferred from the source to the destination. If the destination is hexadecimal, only the numeric fields of the characters are transferred.
D	If FLTF = 0, the specified number of zero characters are inserted into the destination. If FLTF = 1, the specified number of nonzero characters are inserted into the destination. If the destination is hexadecimal, only the numeric field of the zero or nonzero character is inserted.

Character	Action
E	For the specified number of source characters, the following action takes place. While FLTF = 0 and the numeric field of the source character is 4"0", the zero character is inserted into the destination. If the destination is hexadecimal, only the numeric field of the zero character is inserted. If FLTF = 0 and the numeric field of the source character is not equal to 4"0", several things happen. If EXTF = 0, the plus character is inserted into the destination. If EXTF = 1, the minus character is inserted into the destination. If the destination is hexadecimal, only the numeric field of the plus or minus character is inserted. The numeric field of the source character is transferred to the destination, with a zone field of 4"F" if the destination is EBCDIC. FLTF is assigned a value of 1. While FLTF = 1, the numeric field of the source character is transferred to the destination, with a zone field of 4"F" if the destination is EBCDIC.
F	For the specified number of source characters, the following action takes place. While FLTF = 0 and the numeric field of the source character is 4"0", the zero character is inserted into the destination. If the destination is hexadecimal, only the numeric field of the zero character is inserted. If FLTF = 0 and the numeric field of the source character is not equal to 4"0", several things happen. The dollar character is inserted into the destination. If the destination is hexadecimal, only the numeric field of the dollar character is inserted. The numeric field of the source character is transferred to the destination, with a zone field of 4"F" if the destination is EBCDIC. FLTF is assigned a value of 1. While FLTF = 1, the numeric field of the source character is transferred to the destination, with a zone field of 4"F" if the destination is EBCDIC.
I	The specified number of insert characters are inserted into the destination. If the destination is hexadecimal, only the numeric field of the insert character is inserted.

Character	Action
X	The destination pointer is skipped forward (to the right) the specified number of characters.
Z	For the specified number of source characters, the following action takes place. While FLTF = 0 and the numeric field of the source character is 4"0", the zero character is inserted into the destination. If the destination is hexadecimal, only the numeric field of the zero character is inserted. If FLTF = 0 and the numeric field of the source character is not equal to 4"0", the numeric field of the source character is transferred to the destination, with a zone field of 4"F" if the destination is EBCDIC. FLTF is assigned a value of 1. While FLTF = 1, the numeric field of the source character is transferred to the destination, with a zone field of 4"F" if the destination is EBCDIC.
9	If the source and destination are both EBCDIC, the numeric fields of the specified number of characters are transferred from the source to the destination with zone fields of 4"F". If the source and destination are both hexadecimal, the specified number of characters are transferred from the source to the destination.

Pragmatics

One value array (also called an "edit table") is generated for each PICTURE declaration; therefore, for run-time efficiency, all pictures should be collected under a single PICTURE declaration.

Examples

PICTURE NUM (ZZZZ9)

This picture transfers five characters from the source to the destination. The first four characters are transferred with leading zero replacement; that is, leading zeros are transferred to the destination as the zero character, which is a blank character by default. The fifth character is not replaced by the zero character. If the source and destination are EBCDIC, digits are transferred as digits, but other characters have their zone field replaced by 4"F", turning them into digits. If the source and destination are hexadecimal, only the numeric field of the zero character is transferred to replace leading zeros. The following table gives some sample results of this picture.

Source	Destination
8"00000"	8" 0"
8"00500"	8" 500"
8"00356"	8" 356"
8"0ABCD"	8" 1234"
4"00000"	4"00000"
4"00500"	4"00500"
4"00356"	4"00356"
4"0ABCD"	4"0ABCD"

PICTURE USECS (ZZZI999999)

This picture transfers nine characters from the source to the destination and inserts one character into the destination, yielding ten characters in the destination. The first three characters from the source are transferred to the destination with leading zero replacement. Then the insert character, which is a period (.) by default, is inserted into the destination. Six characters are then transferred from the source to the destination with no leading zero replacement. The following table gives some sample results of this picture.

Source	Destination
8"000000000"	8" .000000"
8"356000012"	8"356.000012"
8"005123400"	8" 5.123400"
8"150000376"	8"150.000376"

```
PICTURE TIMENOW (N: " " 9(2) I 9(2) I 9(2))
```

This picture transfers six characters from the source. The introduction code N causes the insert character to be the colon (:). The string literal " " causes the blank character to be inserted into the destination. The first and second source characters are transferred to the destination without leading zero replacement, the insert character is inserted into the destination, the third and fourth source characters are transferred to the destination, the insert character is inserted, and the fifth and sixth source characters are transferred to the destination. The destination receives a total of nine characters. The following table gives some sample results of this picture.

Source	Destination
8"000000"	8" 00:00:00"
8"123456"	8" 12:34:56"
8"000523"	8" 00:05:23"
8"150007"	8" 15:00:07"

```
PICTURE TABLE ("1983 = " F(4) X(2) "1984 = " :F(4) X(2)
              "CHANGE = " :E(3) "%")
```

This picture transfers 11 characters from the source to the destination, formatting the information into a table.

First, the string "1983 = " is inserted into the destination. Then four characters are transferred from the source to the destination, with leading zero replacement and a dollar sign ($) inserted in front of the first nonzero character. Then the destination pointer is advanced two characters, and the string "1984 = " is inserted into the destination. The colon (:) control character causes leading zero replacement to be restored. Four characters are transferred from the source to the destination with leading zero replacement and dollar sign insertion. The destination pointer is advanced two characters, and the string "CHANGE = " is inserted into the destination. Again, the colon is used to restore leading zero replacement. Then three characters are transferred from the source to the destination with leading zero replacement and a plus (+) or a minus (-) inserted in front of the first nonzero character, depending on the value of EXTF. Finally, the string "%" is inserted into the destination. A total of 42 destination characters are produced by this picture.

The following table gives some sample results of this picture. In the table, it is assumed that the destination area was filled with blanks before the picture was used, and that EXTF was properly set up to reflect the sign of the change value.

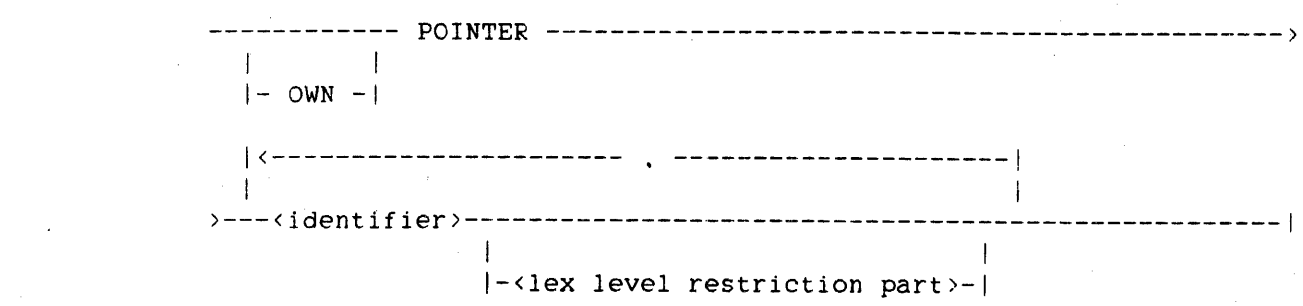
Source	Destination
8"00035000420020"	8"1983 = $35 1984 = $42 CHANGE = +20%"
8"00110003680235"	8"1983 = $110 1984 = $368 CHANGE = +235%"
8"02246021060006"	8"1983 = $2246 1984 = $2106 CHANGE "
8"00089000350061"	8"1983 = $89 1984 = $35 CHANGE = -61%"

POINTER DECLARATION

The POINTER declaration declares a pointer. A pointer can represent the address of a character position in a one-dimensional array or an array row. Because of this, it is said to "point" to a character position.

Syntax

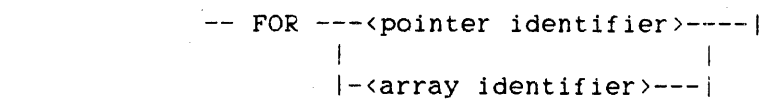
<pointer declaration>



<pointer identifier>

An <identifier> that is associated with a pointer in a POINTER declaration.

<lex level restriction part>



See also
<array identifier>. 42

Semantics

The POINTER declaration establishes each identifier in the list as a pointer identifier.

Pragmatics

Pointers are initialized through the use of a pointer assignment statement or the update pointer construct. Any attempt to use a pointer before it is initialized results in a fault at run time.

Example

```
POINTER PTS,PTD,SOURCE,DEST
```

Declares PTS, PTD, SOURCE, and DEST to be pointers.

Own Pointers

A pointer declared to be OWN retains its value when the program exits the block in which the pointer is declared, and that value is again available when the program re-enters the block in which the pointer is declared.

Own pointers can be assigned only to global arrays or own arrays declared within the scope of the pointer. This restriction applies because the pointer is not deallocated when the block in which it is declared is exited. If an own pointer were assigned to a local array, then when the block in which the pointer is declared is re-entered, the pointer could contain a reference to an array that has been deallocated.

<lex level restriction part>

A global pointer pointing to a local array would access an invalid portion of memory, if the local array is deallocated. To avoid this situation, any construct that could result in a pointer pointing to an array declared at a higher lexical (lex) level than that at which the pointer is declared is disallowed by the compiler. Such an assignment is called an "up-level pointer assignment."

An explicit up-level pointer assignment such as

```
GLOBALPOINTER := POINTER(LOCALARRAY)
```

results in a syntax error, because the locally declared array LOCALARRAY might be deallocated, leaving the global pointer GLOBALPOINTER pointing at an invalid memory location.

A potential up-level pointer assignment such as

```
GLOBALPOINTER := LOCALPOINTER
```

also results in a syntax error, because the local pointer LOCALPOINTER can point to a locally declared array. Of course, LOCALPOINTER can point to an array declared at a lex level equal to or less than that at which GLOBALPOINTER is declared (in which case up-level assignment would not occur), but because there is no way for the compiler to determine where LOCALPOINTER will be pointing when the assignment is executed, such potential up-level pointer assignments are not allowed.

The lex level restriction part causes assignments to the pointer being declared to be restricted so that the pointer can be used to assign values to pointers declared at lower lex levels. The lex level restriction part specifies that, for up-level pointer assignment checking, the compiler is to treat the pointer being declared as if it were declared at the same lex level as the pointer or array whose identifier follows the "FOR." For example, the declaration

```
POINTER LOCALPOINTER FOR GLOBALPOINTER
```

declares a pointer LOCALPOINTER that can point only to arrays declared at lex levels equal to or less than the lex level at which GLOBALPOINTER is declared.

Because assignments to LOCALPOINTER are restricted by the lex level restriction part in the above declaration, an assignment such as

```
GLOBALPOINTER := LOCALPOINTER
```

cannot result in an up-level pointer assignment, and therefore is allowed by the compiler.

The lex level restriction part is not allowed in the formal parameter part of a PROCEDURE declaration or in the global part.

Example 1

```
 100 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
 200 %%%%%%%%%%%%% PROGRAM 1 %%%%%%%%%%%%%%%%
 300 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
 400 BEGIN                                              % LEX LEVEL 2
 500    POINTER GLOBALPOINTER;
 600    ARRAY GLOBALARRAY1,
 700          GLOBALARRAY2[0:9];
 800    GLOBALPOINTER := POINTER(GLOBALARRAY1);
 900    BEGIN                                           % LEX LEVEL 3
1000       POINTER LOCALPOINTER;
1100       ARRAY LOCALARRAY[0:9];
1200       GLOBALPOINTER := LOCALPOINTER;              % SYNTAX ERROR
1300       LOCALPOINTER := POINTER(LOCALARRAY);
1400    END;
1500 END.

 100 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
 200 %%%%%%%%%%%%% PROGRAM 2 %%%%%%%%%%%%%%%%
 300 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
 400 BEGIN                                              % LEX LEVEL 2
 500    POINTER GLOBALPOINTER;
 600    ARRAY GLOBALARRAY1,
 700          GLOBALARRAY2[0:9];
 800    GLOBALPOINTER := POINTER(GLOBALARRAY1);
 900    BEGIN                                           % LEX LEVEL 3
1000       POINTER LOCALPOINTER FOR GLOBALARRAY2;
1100       ARRAY LOCALARRAY[0:9];
1200       GLOBALPOINTER := LOCALPOINTER;
1300       LOCALPOINTER := POINTER(LOCALARRAY);        % SYNTAX ERROR
1400    END;
1500 END.
```

Program 1 and program 2 above are nearly identical. The only difference is found in the POINTER declaration at line 1000. In program 1, LOCALPOINTER is declared without a lex level restriction part, and the potential up-level pointer assignment at line 1200 of program 1 causes a syntax error. In program 2, LOCALPOINTER is declared with the lex level restriction part "FOR GLOBALARRAY2", so the pointer assignment at line 1200 of program 2 cannot be an up-level pointer assignment and does not cause a syntax error. However, the restrictions imposed by the lex level restriction part cause a syntax error at line 1300 of program 2, where no error occurred in program 1.

Example 2

```
BEGIN
   POINTER P1, P2;                                  % LEX LEVEL 2
   ARRAY A[0:9];
   PROCEDURE P(PTRA, PTRB);
   POINTER PTRA, PTRB;
      BEGIN
      PTRA := PTRA + 3;                             % OK
      REPLACE PTRA:PTRA BY PTRB:PTRB FOR 5:         % OK
      PTRA := PTRB;                                 % SYNTAX ERROR
      PTRA := P2;                                   % SYNTAX ERROR
      PTRB := POINTER(A);                           % SYNTAX ERROR
      REPLACE PTRA:PTRB BY "X";                     % SYNTAX ERROR
      END;
   P2 := POINTER(A);
   P(P1, P2);
END.
```

A call-by-name formal pointer parameter cannot be assigned the value of any pointer other than itself, because there is no way for the compiler to determine the lex level of the actual pointer parameter passed to the call-by-name formal pointer parameter.

Example 3

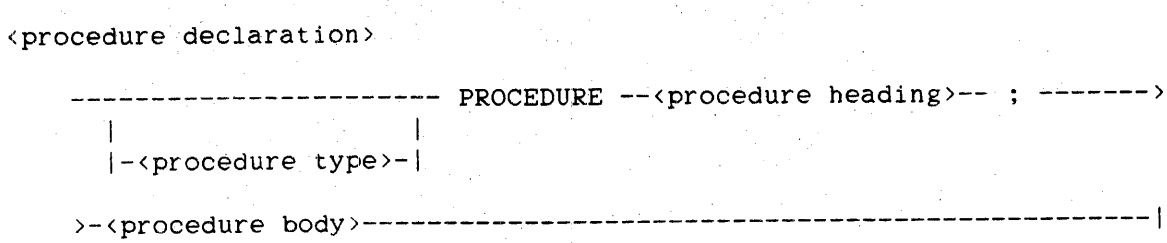
```
[POINTER PTRA,PTRB;]
PROCEDURE P;
   BEGIN
   ARRAY A[0:9];
   PTRA := PTRA + 2;      % OK
   PTRA := POINTER(A);    % SYNTAX ERROR -- THIS IS AN
                          %   UP-LEVEL POINTER ASSIGNMENT.
   PTRA := PTRB;          % SYNTAX ERROR -- THE LEX LEVELS
                          %   OF PTRA AND PTRB ARE NOT KNOWN,
                          %   SO THIS IS A POTENTIAL UP-LEVEL
                          %   POINTER ASSIGNMENT.
   END.
```

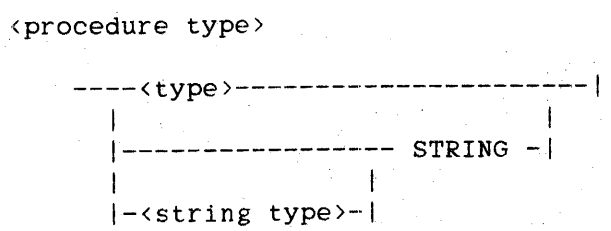
To prevent up-level pointer assignments that can result from separate compilation of procedures with global parts, a pointer declared in the global part cannot be assigned the value of any pointer other than itself.

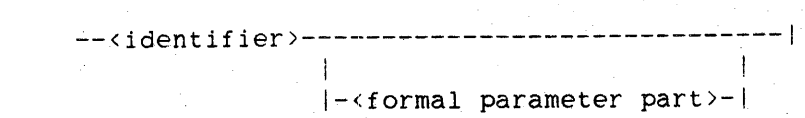
PROCEDURE DECLARATION

A PROCEDURE declaration defines a procedure and associates a procedure identifier with it. The procedure can then be invoked by using the procedure identifier.

Syntax







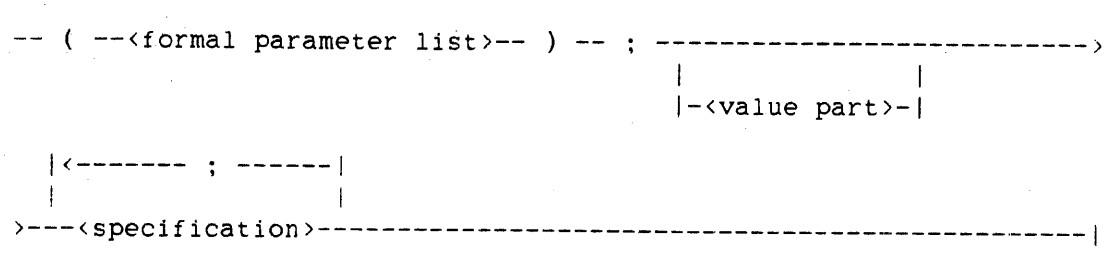
<procedure identifier>

An <identifier> that is associated with a procedure in a PROCEDURE declaration.

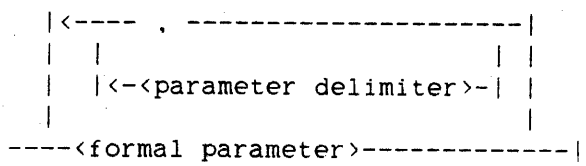
<string procedure identifier>

An <identifier> that is associated with a procedure that is declared a STRING procedure in a PROCEDURE declaration.

<formal parameter part>

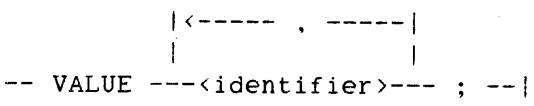


<formal parameter list>

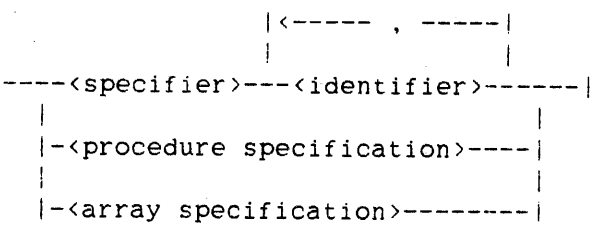


<formal parameter>

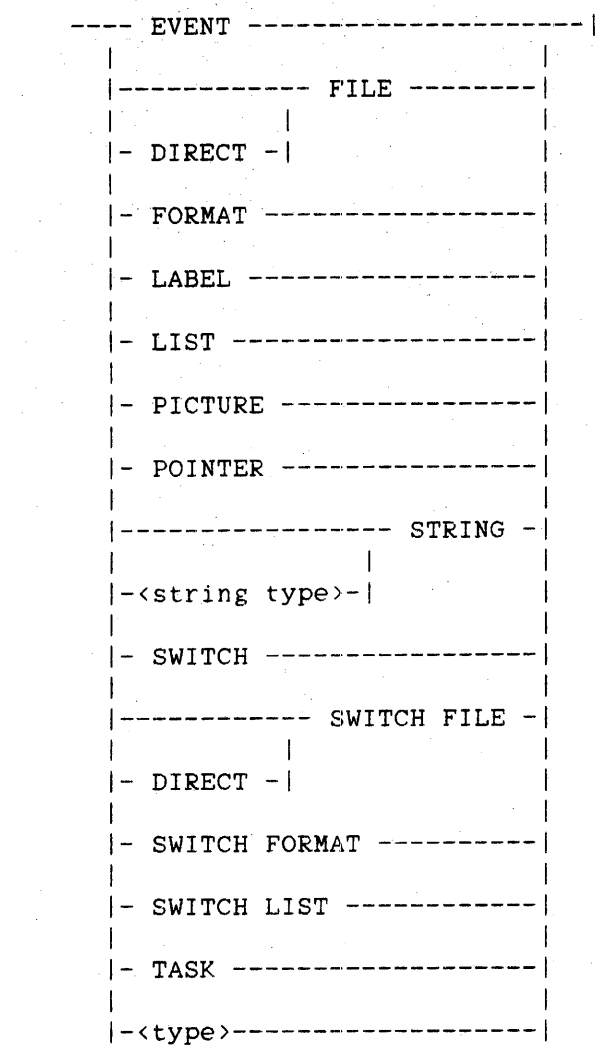
<value part>



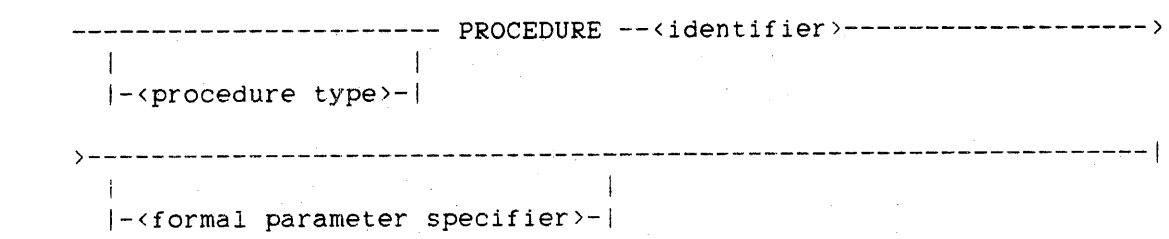
<specification>



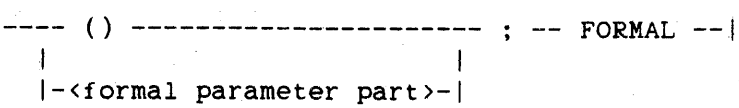
<specifier>



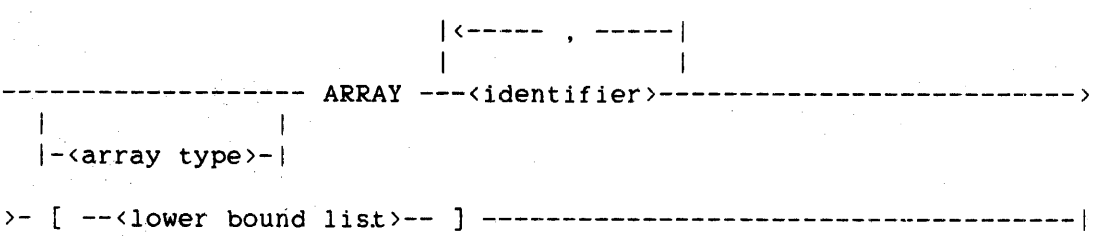
<procedure specification>



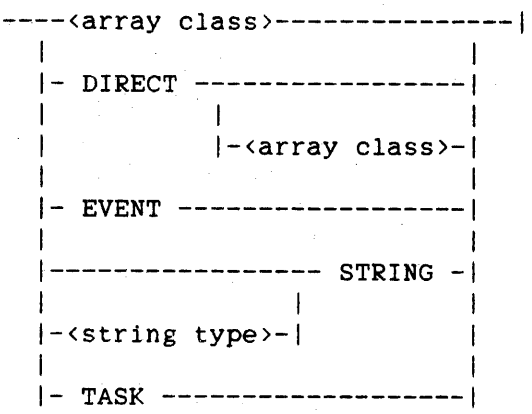
<formal parameter specifier>



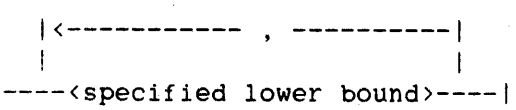
<array specification>



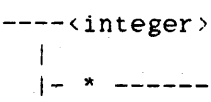
<array type>

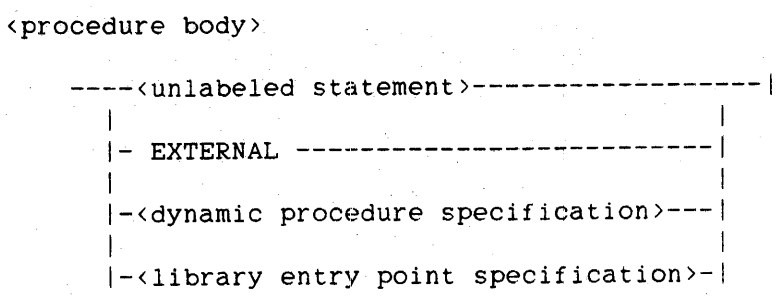


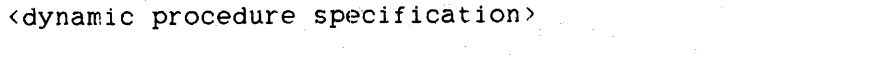
<lower bound list>

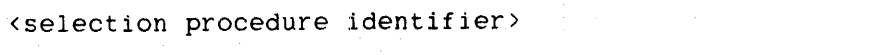


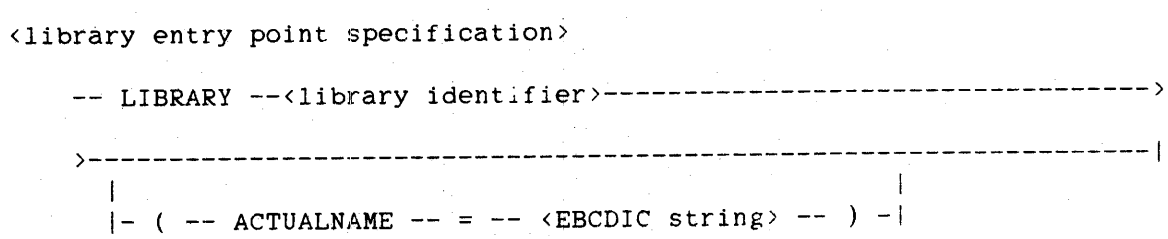
<specified lower bound>











See also

<array class> 41
<library identifier> 129
<string type> 185
<type> 41
<unlabeled statement> 220

Semantics

A procedure becomes a "function" by preceding the word PROCEDURE with a procedure type and by assigning a value (the result to be returned by the procedure) to the procedure identifier somewhere within the procedure body. This kind of procedure is referred to in ALGOL as a "typed procedure." (For examples of typed procedures, see procedures RESULT, HEXPROC, MATCH, and MUCHO under Examples.) A typed procedure can be used either as a statement or as a function. When used as a statement, the returned result is automatically discarded.

If <string type> is not specified in the <procedure type> construct in the declaration of a string procedure, then the procedure is a string procedure of the default character type. The default character type can be designated by the compiler control options ASCII and BCL. If no such compiler control option is used, the default character type is EBCDIC. (For more information, refer to "Default Character Type" in the appendix "Data Representation.")

See also
Default Character Type. 817

The formal parameter part lists the items to be passed in as parameters when the procedure is invoked. A formal parameter part is optional. Every formal parameter for a procedure must appear in a specification.

The value part specifies which formal parameters are to be "call-by-value." When a formal parameter is call-by-value, the formal parameter is assigned the value of the corresponding actual parameter when the procedure is invoked. Thereafter, the formal parameter is handled as a variable that is local to the procedure body. That is, any change made to the value of a call-by-value formal parameter has no effect outside the procedure body.

Only arithmetic, Boolean, complex, designational, pointer, and string expressions can be passed as actual parameters to call-by-value formal parameters. These expressions are evaluated once before entry into the procedure body.

Formal parameters not listed in the value part are "call-by-name," except for string parameters (described below) and file parameters. Wherever a call-by-name formal parameter appears in the procedure body, the formal parameter is, in effect, replaced by the actual parameter itself and not by the value of the actual parameter. A call-by-name formal parameter is effectively global to the procedure body, because any change made to its value within the procedure body also changes the value of the corresponding actual parameter outside the procedure body. If the formal parameter is a complex call-by-name parameter and the actual parameter is not of type COMPLEX, an assignment within the procedure body to the formal parameter causes the program to discontinue with a fault.

An expression can be passed as an actual parameter to a call-by-name formal parameter. This situation results in a "thunk," or "accidental entry." A thunk is a compiler-generated typed procedure that calculates and returns the value of the expression each time the formal parameter is used. This situation can be time-consuming if the formal parameter is repeatedly referenced. In addition, a fault occurs if an attempt is made to store into that parameter.

The default mode of passing a string is "call-by-reference" instead of call-by-name. Any string expression can be passed to a call-by-reference string formal parameter. When a string variable or a subscripted string variable is passed as an actual parameter to a call-by-reference string formal parameter, a reference to the actual string is passed. If the value of the formal parameter is changed within the procedure body, the actual string is also changed. If any other form of string expression is passed as an actual parameter to a call-by-reference string formal parameter, the string expression is evaluated once at the time the expression is passed, and a reference to the value of the expression is passed to the called procedure. This value can be altered by the called procedure. However, any change in the value of the formal parameter within the procedure body has no effect outside the procedure body.

An array specification must be provided for every formal array. The array specification specifies the number of dimensions in the formal array and indicates the lower bound for each dimension.

A specified lower bound that is an integer indicates that the corresponding dimension of the formal array has a lower bound given by this integer. An asterisk (*) used as a specified lower bound indicates that the corresponding dimension of the formal array has a lower bound that is passed to the procedure with the actual array.

Array rows that are passed as actual parameters to procedures have their subscripts evaluated at the time of the procedure call, rather than at the time the corresponding formal array is referenced.

If a program is a procedure, parameters can be passed to it. If the procedure is initiated through CANDE (which passes only one parameter, a quoted string), then the formal parameter must be declared as a real array with an asterisk lower bound. If the procedure is initiated through Work Flow Language (WFL), a formal parameter for a string actual parameter must be declared as a real array with an asterisk lower bound. Both CANDE and WFL pass strings as arrays. (For more information, refer to the EXECUTE command in the "CANDE Reference Manual" and the RUN statement in the "Work Flow Language (WFL) Reference Manual.") When the program is initiated, the array is allocated the minimum number of words

needed to contain the string plus at least one null character (48"00"), which is appended to the end of the string.

The procedure body EXTERNAL is used to declare a procedure that is to be "bound in" to the program (as opposed to actually appearing within the program) or that is an external code file to be invoked. An attempt to invoke a procedure that is declared external but has not been bound in nor associated with an external code file results in a run-time error.

A dynamic procedure specification is used in a library program to declare a procedure that is to be exported dynamically. Such a procedure is also called a "by-calling procedure." For more information on by-calling procedures, refer to the chapter "Interface to the Library Facility." The by-calling procedure cannot be declared FORWARD and cannot be a separately compiled procedure. Also, the by-calling procedure cannot be referenced directly in the library program that declares it.

A selection procedure identifier must specify an untyped procedure with two parameters. The first parameter must be a call-by-value EBCDIC string. The second parameter must be a fully specified untyped procedure with one parameter that is a task. When the Master Control Program (MCP) invokes this selection procedure, the task variable passed to its procedure parameter must already be associated with a library that has been processed using this task variable.

A library entry point specification declares a procedure to be an entry point in the library known to this program by the library identifier. The procedure cannot be declared FORWARD or EXTERNAL.

If a program declares a library and entry points in that library, the object code file for the program contains a structure called a library template, which describes the library and its declared entry points. Each library declared has one template. The template's description of an entry point includes the entry point's name, a description of the procedure's type, and descriptions of its parameters.

When a library entry point is called, the description of the entry point in the library template of the calling program is compared to the description of the entry point of the same name in the library directory associated with the referenced library. (Refer to "EXPORT Declaration" for a discussion of library directories.) If the entry point does not exist in the library or if the two entry point descriptions are not compatible, then a run-time error is given and the program is terminated.

The name given for an entry point in a library template is the procedure identifier in the declaration of the entry point, unless an ACTUALNAME clause appears, in which case it is given by the EBCDIC string. The EBCDIC string in the ACTUALNAME clause must not contain any leading, trailing, or embedded blanks and must be a "valid identifier"; that is, any sequence of characters beginning with a letter and consisting of letters, digits, hyphens (-), and underscores (_).

Procedures can be called recursively; that is, inside the procedure body, a procedure can invoke itself.

Pragmatics

For maximum efficiency, as many formal parameters as possible should be call-by-value, and each specified lower bound should have a value of zero.

The formal parameter specifier causes the compiler to generate more efficient code for passing procedures as parameters. When a procedure is declared FORMAL, the compiler checks the parameters of the actual procedure passed to it at compile time; otherwise, the parameters are checked at run time. If FORMAL is specified, the formal procedure is called a "fully specified" formal procedure.

Allowed Formal and Actual Parameters

All parameters can be declared to be call-by-name (or, in the case of strings, call-by-reference). The following types of parameters can also be declared to be call-by-value:

ASCII string	integer simple variable
Boolean simple variable	label
complex simple variable	pointer
double simple variable	real simple variable
EBCDIC string	string
hexadecimal string	

Array Parameters

If a formal parameter is an array, the actual parameter passed to that formal array must be an array designator that has the same number of dimensions as the formal array.

The types of actual arrays that can be passed to formal arrays are given by the following table.

Formal Parameter	Allowed Actual Parameters
ASCII array	ASCII array ASCII value array
ASCII string array	ASCII string array
BCL array	BCL array BCL value array
Boolean array	Boolean array direct Boolean array Boolean value array
complex array	complex array complex value array
direct ASCII array	direct ASCII array
direct BCL array	direct BCL array
direct Boolean array	direct Boolean array
direct double array	direct double array
direct EBCDIC array	direct EBCDIC array
direct hexadecimal array	direct hexadecimal array
direct integer array direct real array	direct integer array direct real array
double array	double array direct double array double value array
EBCDIC array	EBCDIC array EBCDIC value array

Formal Parameter	Allowed Actual Parameters
EBCDIC string array	EBCDIC string array
event array	event array
hexadecimal array	hexadecimal array hexadecimal value array
hexadecimal string array	hexadecimal string array
integer array real array	integer array real array direct integer array direct real array integer value array real value array
task array	task array

Procedure Parameters

If a formal parameter is a procedure, the actual parameter passed to that formal procedure must be the identifier of a procedure for which the following is true:

1. The actual procedure has the same number of parameters as the formal procedure.

2. Each parameter of the actual procedure must have the same type as the corresponding parameter in the formal procedure.

3. Each parameter of the actual procedure must be passed in the same manner (call-by-name or call-by-value) as the corresponding parameter in the formal procedure.

The types of the procedures that can be passed to formal procedures are given in the following table.

Formal Parameter	Allowed Actual Parameters
ASCII string procedure	ASCII string procedure
Boolean procedure	Boolean procedure
complex procedure	complex procedure
double procedure	double procedure
EBCDIC string procedure	EBCDIC string procedure
hexadecimal string procedure	hexadecimal string procedure
integer procedure real procedure	integer procedure real procedure
untyped procedure	untyped procedure

Simple Variable Parameters

The types of actual parameters that can be passed to formal parameters that are simple variables are given in the following table.

Formal Parameter	Allowed Actual Parameters
Boolean simple variable (call-by-name or call-by-value)	Boolean identifier Boolean procedure identifier Boolean expression
complex simple variable (call-by-name or call-by-value)	complex identifier double identifier integer identifier real identifier complex procedure identifier double procedure identifier integer procedure identifier real procedure identifier arithmetic expression (single or double precision) complex expression

Formal Parameter	Allowed Actual Parameters
double simple variable (call-by-name)	double identifier double procedure identifier arithmetic expression (double precision only)
double simple variable (call-by-value)	double identifier integer identifier real identifier double procedure identifier integer procedure identifier real procedure identifier arithmetic expression (single or double precision)
integer simple variable real simple variable (call-by-name)	integer identifier real identifier integer procedure identifier real procedure identifier arithmetic expression (single precision only)
integer simple variable real simple variable (call-by-value)	double identifier integer identifier real identifier double procedure identifier integer procedure identifier real procedure identifier arithmetic expression (single or double precision)

String Parameters

The types of actual parameters that can be passed to formal parameters that are strings are given in the following table.

Formal Parameter	Allowed Actual Parameters
ASCII string (call-by-reference or call-by-value)	ASCII string identifier ASCII string procedure identifier ASCII string expression
EBCDIC string (call-by-reference or call-by-value)	EBCDIC string identifier EBCDIC string procedure identifier EBCDIC string expression
hexadecimal string (call-by-reference or call-by-value)	hexadecimal string identifier hexadecimal string procedure identifier hexadecimal string expression

File Parameters

The types of actual parameters that can be passed to formal parameters that are files are given in the following table.

Formal Parameter	Allowed Actual Parameters
direct file	direct file identifier subscripted direct switch file identifier
direct switch file	direct switch file identifier
file	file identifier subscripted switch file identifier
switch file	switch file identifier

Other Types of Parameters

The types of actual parameters that can be passed to formal parameters that are not arrays, procedures, simple variables, strings, or files are given in the following table.

Formal Parameter	Allowed Actual Parameters
event	event identifier an element of an event array file identifier. event-valued file attribute name subscripted switch file identifier. event-valued file attribute name
format	format identifier subscripted switch format identifier
label (call-by-name or call-by-value)	label identifier subscripted switch identifier designational expression
list	list identifier subscripted switch list identifier
picture	picture identifier
pointer (call-by-name)	pointer identifier
pointer (call-by-value)	pointer identifier pointer expression
switch label	switch label identifier
switch format	switch format identifier
switch list	switch list identifier
task	any task designator

Examples

The examples below show how the procedure body of a procedure can vary in complexity from a simple unlabeled statement to a block.

```
PROCEDURE SIMPL;
   X := X + 1
```

Declares SIMPL to be an untyped procedure with no parameters. The body of SIMPL is a single statement.

```
PROCEDURE TUFFER(PARAM);
VALUE PARAM;
REAL PARAM;
   X := X + PARAM
```

Declares TUFFER to be an untyped procedure with one parameter, PARAM, which is a call-by-value real variable. The body of TUFFER consists of a single statement.

```
REAL PROCEDURE RESULT(PARAM,FYLEIN);
REAL PARAM;
FILE FYLEIN;
   BEGIN
      .
      .
      .
   RESULT := X + PARAM;
      .
      .
      .
   END
```

Procedure RESULT is a typed procedure that returns a real value. The value to be returned is assigned to the procedure identifier by the assignment "RESULT := X + PARAM;". RESULT has two parameters, a call-by-name real variable and a file.

```
HEX STRING PROCEDURE HEXPROC;
   HEXPROC := 4"123"
```

Declares HEXPROC to be a typed procedure that returns a hexadecimal string value. The value to be returned is assigned to the procedure identifier in the assignment that makes up the body of HEXPROC.

```
BOOLEAN PROCEDURE MATCH(A,B,C);
VALUE A,B,C;
INTEGER A,B,C;
   MATCH := A=B OR A=C OR B=C
```

Declares MATCH to be a typed procedure that returns a Boolean value. MATCH has three parameters that are all call-by-value, integer variables.

```
PROCEDURE FURTHERON;
   FORWARD
```

This is a forward procedure declaration for the procedure FURTHERON. For more information, refer to "Forward Reference Declaration."

```
DOUBLE PROCEDURE MUCHO(DBL1,DBL2,BOOL);
VALUE DBL2,BOOL;
DOUBLE DBL1,DBL2;
BOOLEAN BOOL;
   BEGIN
   REAL LOCALX,LOCALY;
     .
     .
     .
   MUCHO := DOUBLE(LOCALX,LOCALY);
   END OF MUCHO
```

Declares MUCHO to be a double-precision procedure with three parameters. DBL1 is a call-by-name double-precision variable, DBL2 is a call-by-value double-precision variable, and BOOL is a call-by-value Boolean variable. The body of MUCHO is a block.

```
PROCEDURE GETDATA(A);                  % BY-CALLING PROCEDURE
ARRAY A[*];
   BY CALLING SELECTDATASOURCE
```

Declares GETDATA to be a by-calling procedure. The selection procedure is SELECTDATASOURCE. GETDATA has one parameter, a one-dimensional real array, A, with an asterisk lower bound, meaning that the lower bound will be passed as a parameter.

```
INTEGER PROCEDURE NUMRECORDS(TYPE);   % LIBRARY ENTRY POINT
VALUE TYPE;
INTEGER TYPE;
   LIBRARY DATAHANDLER (ACTUALNAME="COUNTRECS")
```

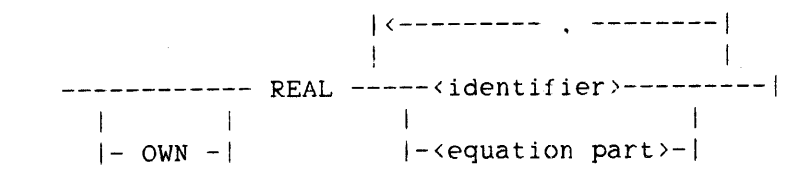
Declares NUMRECORDS to be an entry point in the library DATAHANDLER. The entry point is exported from DATAHANDLER with the name COUNTRECS, but will be called NUMRECORDS in this program.

REAL DECLARATION

A REAL declaration declares simple variables that can have real values (arithmetic values that have exponents and fractional parts).

Syntax

‹real declaration›



‹real identifier›

An ‹identifier› that is associated with the REAL data type in a REAL declaration.

See also
‹equation part› 55

Semantics

A simple variable declared to be OWN retains its value when the program exits the block in which the variable is declared, and that value is again available when the program re-enters the block in which the variable is declared.

The equation part causes the simple variable being declared to have the same address as the simple variable associated with the second identifier. This action is called "address equation." An identifier can be address-equated only to a previously declared local identifier or to a global identifier. The first identifier must not have been previously declared within the block of the equation part.

Address equation is allowed only between integer, real, and Boolean variables. Because both identifiers of the equation part have the same address, altering the value of either variable affects the value of both variables.

The following example demonstrates the effects of address-equating real and Boolean variables.

```
BEGIN
   BOOLEAN B;
   REAL R = B;
   R := 4;              % B = FALSE   R = 4.0
   B := TRUE;           % B = TRUE    R = 1.0
END.
```

The OWN specification has no effect on an address-equated identifier. The first identifier of an equation part is own only if the second identifier of the equation part is own.

A REAL declaration with an equation part is not allowed in the global part of a program unit.

Pragmatics

If a real or integer value is assigned to a real variable, it is stored "as is" into the variable. If a double-precision value is assigned to a real variable, it is rounded to single precision before it is stored in the variable.

When a real simple variable is allocated, it is initialized to zero (a 48-bit word with all bits equal to zero). However, to ensure compatibility with ALGOL 60, programmers should explicitly initialize real simple variables with appropriate assignment statements.

The appendix "Data Representation" contains additional information on the internal structure of a real operand as implemented on A Series and B 5000/B 6000/B 7000 Series systems.

See also
Real Operand. 820

Examples

```
REAL INDX,X,Y,TOTAL
```

Declares INDX, X, Y, and TOTAL as real variables.

```
REAL CALC = BOOL, INDEX, VALU = INTR
```

Declares CALC, INDEX, and VALU as real variables. CALC is address-equated to the simple variable BOOL, and VALU is address-equated to the simple variable INTR. This means CALC and BOOL share the same address, and VALU and INTR share the same address.

```
OWN REAL DISTANCE, REALINDEX
```

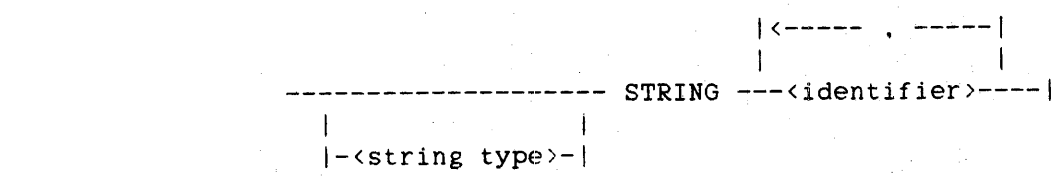
Declares DISTANCE and REALINDEX as real variables. Because they are declared to be own, these variables retain their values when the program exits the block in which they are declared.

STRING DECLARATION

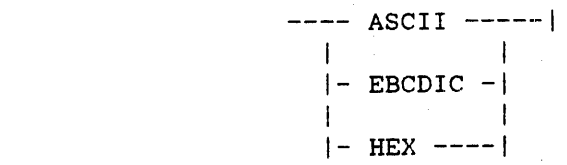
A STRING declaration declares simple variables to be strings. Strings allow storage and manipulation of character strings in a program.

Syntax

<string declaration>



<string type>



<string identifier>

An <identifier> that is associated with the STRING data type in a STRING declaration.

Semantics

The type STRING is a structured data type that contains characters of only one character type. A string has two components: contents and length. No trailing blanks or null characters are added to a string, so the length of a string is exactly the number of characters stored in the string. The maximum string length allowed is 2**16-2 characters.

All strings declared in a STRING declaration are of the same string type. If no string type is specified in the STRING declaration, then the default character type is used. If the default character type in this case is BCL, a syntax error is given. The default character type can be designated by the compiler control options ASCII and BCL. If no such compiler control option is designated, the default character type is EBCDIC. (For more information, refer to "Default Character Type" in the appendix "Data Representation.")

See also
Default Character Type. 817

Pragmatics

The number of strings that can be declared in a program is limited by the Master Control Program (MCP) to 500. If this limit is exceeded, the message "STRING POOL EXCEEDED" is given.

Examples

```
ASCII STRING S1,S2,S3
```

Declares S1, S2, and S3 as string simple variables of string type ASCII. S1, S2, and S3 will contain ASCII characters.

```
EBCDIC STRING S5,S6,S7,S8
```

Declares S5, S6, S7, and S8 as string simple variables of string type EBCDIC. These strings will contain EBCDIC characters.

```
STRING S9
```

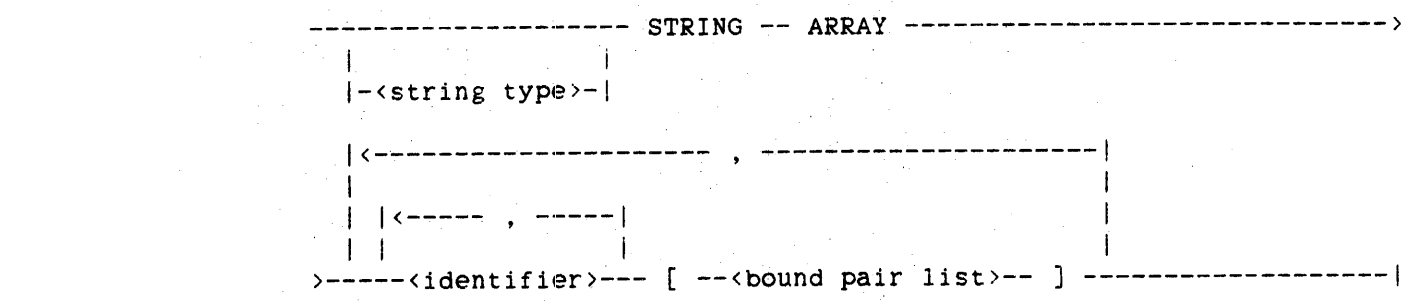
Declares S9 as a string simple variable. Because no string type is specified, the default character type is used. This character type is EBCDIC unless the compiler control option ASCII is TRUE, in which case the string type is ASCII, or the compiler control option BCL is TRUE, in which case the string type is BCL. If the default character type is BCL, this declaration is given a syntax error.

STRING ARRAY DECLARATION

A STRING ARRAY declaration declares string arrays. A string array is an array that has string elements.

Syntax

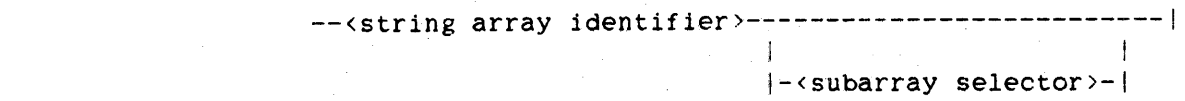
<string array declaration>



<string array identifier>

An <identifier> that is associated with a string array in a STRING ARRAY declaration.

<string array designator>



See also

<bound pair list> 42
<string type> 185
<subarray selector> 44

Semantics

All string arrays declared in a STRING ARRAY declaration are of the same string type. If no string type is specified, the default character type is used. If the default character type in this case is BCL, a syntax error is given. The default character type can be designated by the compiler control options ASCII and BCL. If no such compiler control option is used, the default character type is EBCDIC. (For more information, refer to "Default Character Type" in the appendix "Data Representation.")

The restrictions that apply to arrays also apply to string arrays. (For more information, refer to "ARRAY Declaration.")

See also
Default Character Type. 817

Examples

```
STRING ARRAY SA,SB,SC[0:10]
```

Declares SA, SB, and SC as one-dimensional arrays of strings, each with a lower bound of zero and an upper bound of 10. Because no string type is specified, the default character type is used. This character type is EBCDIC unless the compiler control option ASCII is TRUE, in which case the string type is ASCII, or the compiler control option BCL is TRUE, in which case the string type is BCL. If the default character type is BCL, this declaration is given a syntax error.

```
EBCDIC STRING ARRAY ESA[1:15], ESB, ESC[0:10, 0:10]
```

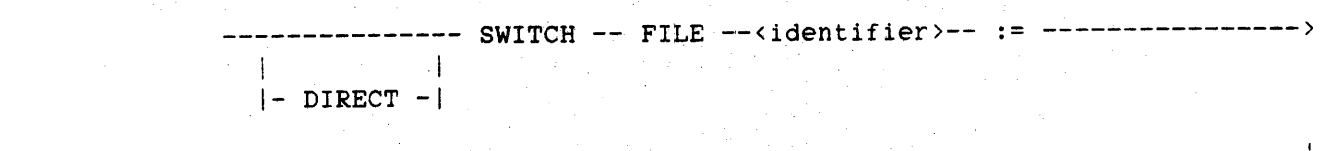
Declares ESA, ESB, and ESC as arrays of strings. The string type is EBCDIC, so each is an array of EBCDIC strings. ESA is one-dimensional and has a lower bound of 1 and an upper bound of 15. Arrays ESB and ESC are two-dimensional arrays with lower bounds of zero and upper bounds of 10 for both dimensions.

SWITCH FILE DECLARATION

A SWITCH FILE declaration associates an identifier with a list of file designators. Any of these file designators can later be referenced by using the identifier and a number corresponding to the position of the file designator in the list.

Syntax

<switch file declaration>



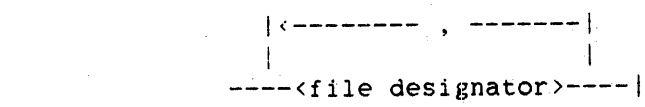
<switch file identifier>

An <identifier> that is associated with a switch file list in a SWITCH FILE declaration.

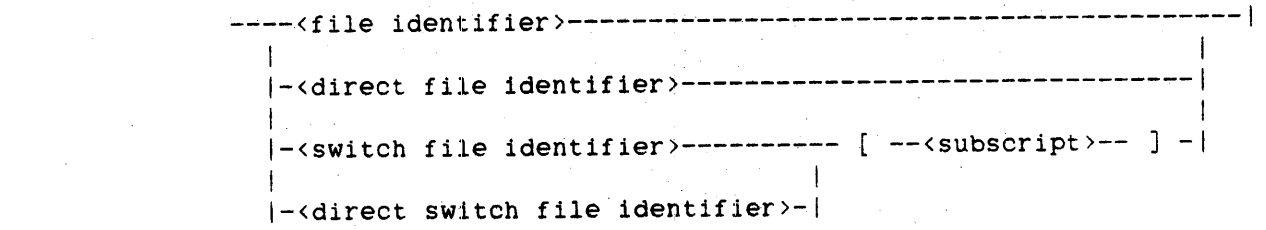
<direct switch file identifier>

An <identifier> that is associated with a switch file list in a DIRECT SWITCH FILE declaration.

<switch file list>



<file designator>



See also

<direct file identifier> 85
<file identifier> 85
<subscript> 43

Semantics

An integer index is associated with each file designator in the switch file list. The indexes are 0, 1, 2, and so on through N-1, where N is the number of file designators in the list. These indexes are obtained by counting the file designators in order of their appearance in the list. A file designator in the list can be referenced by subscripting the switch file identifier with a subscript whose value is equal to the index of the file designator.

If a subscript to a switch file identifier yields a value outside the range of the switch file list (that is, less than zero or greater than N-1), a fault occurs at run time.

Any subscripts in the switch file list are evaluated at the time of the switch file declaration.

A switch file can reference itself in the switch file list, in which case a stack overflow might occur when the program is executed. For example, if a switch file is declared as

```
SWITCH FILE SF := F1, F2, SF[N]
```

then if N equals 2, the subscripted switch file identifier SF[N] references itself indefinitely.

The switch file list of a switch file that is not DIRECT can contain only file designators that are not DIRECT, and the switch file list of a switch file that is DIRECT can contain only file designators that are DIRECT.

Examples

```
SWITCH FILE CHOOSEUNIT :=
   CARDOUT,
   TAPEOUT,
   PRINTOUT;
  .
  .
  .
WRITE(CHOOSEUNIT[0], 14, A[*]);    % WRITES TO CARDOUT
WRITE(CHOOSEUNIT[1], 14, A[*]);    % WRITES TO TAPEOUT
WRITE(CHOOSEUNIT[2], 14, A[*]);    % WRITES TO PRINTOUT
```

SWITCH FORMAT DECLARATION

A SWITCH FORMAT declaration associates an identifier with a list of items representing editing specifications. Any of these items and the associated editing specifications can later be referenced by using the identifier and a number corresponding to the position of the item in the list.

Syntax

<switch format declaration>

```
-- SWITCH -- FORMAT --<identifier>-- := --<switch format list>--|
```

<switch format identifier>

An <identifier> that is associated with a switch format list in a SWITCH FORMAT declaration.

<switch format list>

```
    |<----------- , ----------|
    |                         |
----<switch format segment>----|
```

<switch format segment>

```
----<format designator>-------------------|
    |                                     |
    |- ( --<editing specifications>-- ) -|
    |                                     |
    |- < --<editing specifications>-- > -|
```

<format designator>

```
----<format identifier>-----------------------------------|
    |                                                     |
    |-<switch format identifier>-- [ --<subscript>-- ] -|
```

See also

Semantics

An integer index is associated with each switch format segment in the switch format list. The indexes are 0, 1, 2, and so on through N-1, where N is the number of switch format segments in the list. These indexes are obtained by counting the switch format segments in order of their appearance in the list. A switch format segment in the list can be referenced by subscripting the switch format identifier with a subscript whose value is equal to the index of the switch format segment.

If a subscript to a switch format identifier yields a value outside the range of the switch format list (that is, less than zero or greater than N-1), a fault occurs at run time.

Any subscripts in the switch format list are evaluated at the time the subscripted switch format identifier is encountered.

A switch format can reference itself in the switch format list, in which case a stack overflow might occur when the program is executed. For example, if a switch format is declared as

```
SWITCH FORMAT SF := FMT1, FMT2, SF[N]
```

then if N equals 2, the subscripted switch format identifier SF[N] references itself indefinitely.

A simple string literal in a SWITCH FORMAT declaration is always read-only if the switch format segment in which it appears consists of editing specifications rather than a format designator.

Examples

```
SWITCH FORMAT SF := (A6, 3I4, I2, X60),   % 0
                    (I4, X2, 2I4, 3I2),   % 1
                    (X78, I2),            % 2
                    (X2)                  % 3
```

Declares SF to be a switch format identifier with a switch format list of four sets of editing specifications. The editing specifications (X78, I2), for example, can be referenced as SF[2].

```
SWITCH FORMAT SWHFT := FMT1,FMT2,FMT3
```

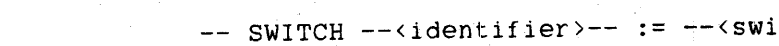
Declares SWHFT to be a switch format identifier with a switch format list of three format designators. SWHFT[0] evaluates to format FMT1, SWHFT[1] to FMT2, and SWHFT[2] to FMT3.

SWITCH LABEL DECLARATION

A SWITCH LABEL declaration associates an identifier with a list of designational expressions, which are expressions that evaluate to labels. Any of these designational expressions can later be referenced by using the identifier and a number corresponding to the position of the designational expression in the list.

Syntax

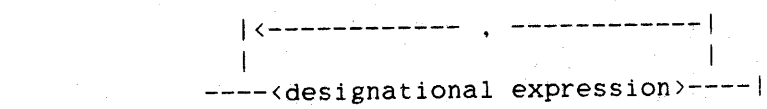
<switch label declaration>



<switch label identifier>

An <identifier> that is associated with a switch label list in a SWITCH LABEL declaration.

<switch label list>



Semantics

An integer index is associated with each designational expression in the switch label list. The indexes are 1, 2, 3, and so on through N, where N is the number of designational expressions in the list. These indexes are obtained by counting the designational expressions in order of their appearance in the list. A designational expression in the list can be referenced by subscripting the switch label identifier with a subscript whose value is equal to the index of the designational expression.

Note that the indexing of a switch label list begins at 1.

If a subscript to a switch label identifier yields a value outside the range of the switch label list (that is, less than 1 or greater than N), the statement using the switch label is not executed, and control proceeds to the next statement. Typically, the next statement is a specification of some form of error handling.

The designational expressions in a switch label list are evaluated at the time the subscripted switch label identifier is encountered.

A switch label can reference itself in the switch label list, in which case a stack overflow might occur when the program is executed. For example, if a switch label is declared as

```
SWITCH SW := L1, L2, L3, SW[N]
```

then if N equals 4, the designational expression SW[N] references itself indefinitely.

Examples

```
SWITCH CHOOSEPATH := L1,L2,L3,L4
```

Declares CHOOSEPATH to be a switch label identifier with labels L1, L2, L3, and L4 in the switch label list. CHOOSEPATH[1] evaluates to label L1, CHOOSEPATH[2] to L2, and so on.

```
SWITCH SELECT := START,          % 1
                 ERROR1,         % 2
                 CHOOSEPATH[2]   % 3
```

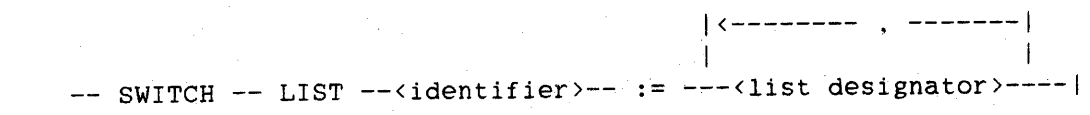
Declares SELECT to be a switch label identifier with labels START and ERROR1 and designational expression CHOOSEPATH[2] in the switch label list. Note that from the previous SWITCH LABEL declaration, CHOOSEPATH[2] evaluates to L2, so SELECT[3] evaluates to L2.

SWITCH LIST DECLARATION

A SWITCH LIST declaration associates an identifier with a list of list designators. Any of these list designators can later be referenced by using the identifier and a number corresponding to the position of the list designator in the list.

Syntax

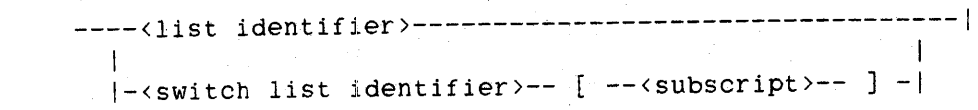
<switch list declaration>



<switch list identifier>

An <identifier> that is associated with a list of list designators in a SWITCH LIST declaration.

<list designator>



See also
<list identifier> 132
<subscript> 43

Semantics

An integer index is associated with each list designator in the declaration. The indexes are 0, 1, 2, and so on through N-1, where N is the number of list designators in the declaration. These indexes are obtained by counting the list designators in order of their appearance in the declaration. Any of these list designators can be referenced by subscripting the switch list identifier with a subscript whose value is equal to the index of the list designator.

If a subscript to a switch list identifier yields a value outside the range of the list of list designators (that is, less than zero or greater than N-1), a fault occurs at run time.

Any subscripts in the list of list designators are evaluated at the time the subscripted switch list identifier is encountered.

A switch list can reference itself in the list of list designators, in which case a stack overflow might occur when the program is executed. For example, if a switch list is declared as

```
SWITCH LIST SL := L1, L2, SL[N]
```

then if N equals 2, the subscripted switch list identifier SL[N] references itself indefinitely.

Example

```
SWITCH LIST NUMVARIABLES := NOVARS,    % 0
                            ONEVAR,    % 1
                            TWOVARS,   % 2
                            THREEVARS  % 3
```

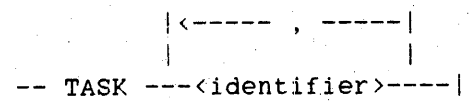
Declares NUMVARIABLES to be a switch list identifier and associates four list designators with it. NUMVARIABLES[0] evaluates to the list NOVARS, NUMVARIABLES[1] evaluates to ONEVAR, and so on.

TASK AND TASK ARRAY DECLARATIONS

The TASK and TASK ARRAY declarations are used to declare tasks and task arrays, respectively, which can then be associated with a process or coroutine. Task attributes can be used to control or to contain information about the process or coroutine.

Syntax

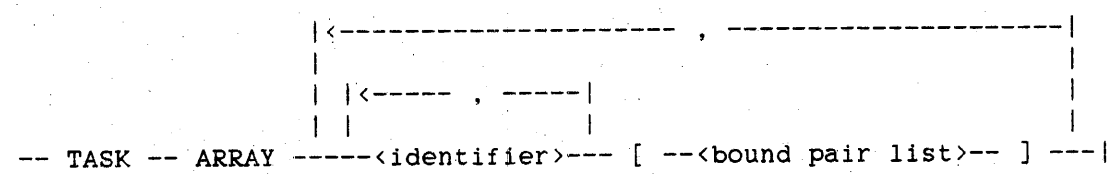
<task declaration>



<task identifier>

An <identifier> that is associated with a task in a TASK declaration.

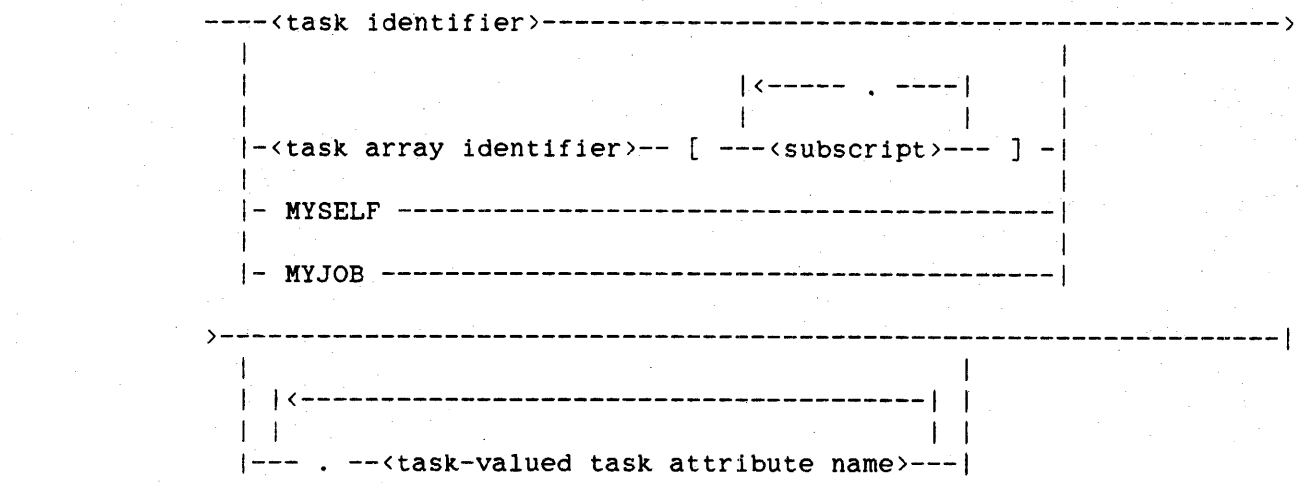
<task array declaration>



<task array identifier>

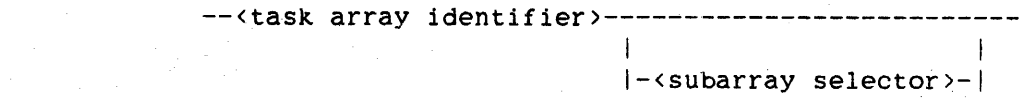
An <identifier> that is associated with a task array in a TASK ARRAY declaration.

<task designator>



<task-valued task attribute name>

<task array designator>



See also

Semantics

A task array is an array whose elements are tasks. A task array can have no more than 15 dimensions.

A task designator represents a single task. A task array designator represents an array of tasks. MYSELF is the task designator for the currently running program. MYJOB is the task designator for the currently running job.

When a process or coroutine is invoked, a task can be associated with it. For example, a task designator can appear in a CALL statement, PROCESS statement, or RUN statement. Attributes of the task can be assigned values by the program to control the process or coroutine, and the program can interrogate the values of attributes of the task as the process or coroutine executes.

Attributes associated with a task designator can be assigned values or interrogated in a program by specifying the task designator and the appropriate task attribute names in assignment statements.

For information on processes and coroutines, refer to "CALL Statement," "PROCESS Statement," and "RUN Statement." For more information on assigning and interrogating task attributes, refer to the ‹arithmetic task attribute› construct under "Arithmetic Assignment," the ‹Boolean task attribute› construct under "Boolean Assignment," and "Task Assignment."

See also
‹arithmetic task attribute› 227
‹Boolean task attribute›. 235
Task Assignment 246

Examples

```
TASK PROCESSTASK
```

Declares PROCESSTASK to be a task identifier.

```
TASK ARRAY CHILDREN[0:LIM]
```

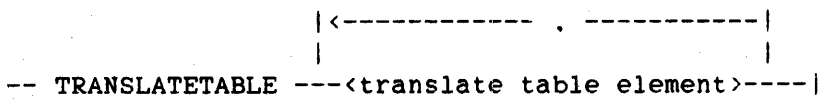
Declares CHILDREN as a one-dimensional task array with a lower bound of zero and an upper bound of LIM. CHILDREN might be used to store the tasks associated with a group of processes and coroutines initiated by a program.

TRANSLATETABLE DECLARATION

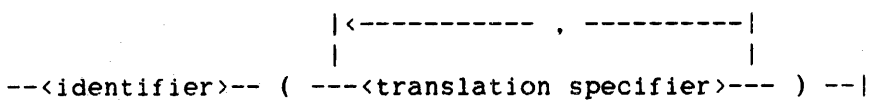
The TRANSLATETABLE declaration defines one or more translate tables. Used in a REPLACE statement, a translate table indicates translations to be performed from one group of characters to another group of characters.

Syntax

<translate table declaration>



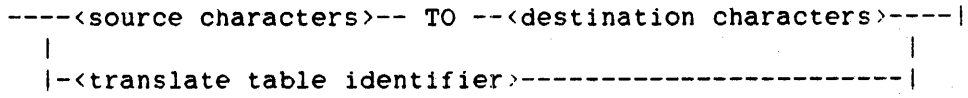
<translate table element>



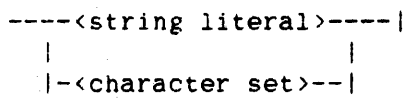
<translate table identifier>

An <identifier> that is associated with a group of one or more translation specifiers in a TRANSLATETABLE declaration.

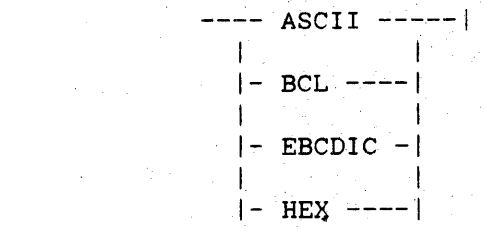
<translation specifier>



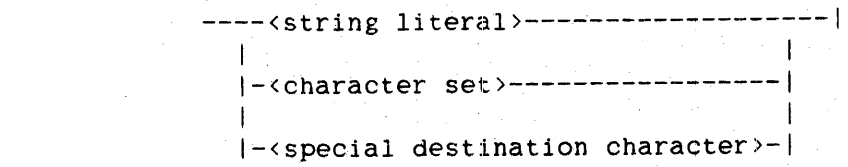
<source characters>



<character set>



<destination characters>



<special destination character>

A <string literal> that is one character long.

Semantics

Specifying a character set is equivalent to specifying all the characters in that set, in ascending binary sequence. The length of a character set is equal to the total number of characters in the set.

A string literal specifies all the characters in the string literal. The length of a string literal is equal to the number of characters in the string literal in terms of the largest character size specified by the string literal.

Within a pair of parenthesis, each succeeding translation specifier overrides the previous translation specifiers.

Within a single translate table, all source character sizes and all destination character sizes must be the same, although the character sizes of the source and destination parts need not be the same.

The number of destination characters must equal the number of source characters, unless the special destination character is used or unless a character set is used for both the source characters and the destination characters. If the special destination character is used, all the source characters are translated to the special destination character.

Every translate table has a default base in which all source characters are translated to characters with all bits equal to zero. This means that all source characters that do not appear in the TRANSLATETABLE declaration are translated to the character whose binary representation had all bits equal to zero.

The use of a character set for both the source and destination parts invokes a standard table from the Master Control Program (MCP) and provides a way of obtaining a legitimate base on which additional translation specifiers can be used, if desired, to override certain parts of the standard table. The use of a translate table identifier as a translation specifier can also be used to provide a base.

When string literals of equal length are used for the source and destination parts, translation is based on the corresponding positions of the source and destination characters, from left to right.

See also
Short and long string literals. 387

Examples

```
TRANSLATETABLE ALCHEMY ("LEAD" TO "GOLD")
```

Translates the letters "L" to "G", "E" to "O", "A" to "L", and "D" to "D". All other characters are translated to the character whose binary representation has all bits equal to zero. Both the source and the destination characters are of the default character type.

```
TRANSLATETABLE UPCASE (EBCDIC TO EBCDIC,
                       "abcdefghijklmnopqrstuvwxyz" TO
                       "ABCDEFGHIJKLMNOPQRSTUVWXYZ")
```

Translates all EBCDIC characters to themselves except for the lowercase letters, which it translates to uppercase letters.

```
TRANSLATETABLE PAREN_TO_BRACKET (EBCDIC TO EBCDIC, 8"(" TO 8"[")
```

Translates all EBCDIC characters to themselves except the left parenthesis, "(", which is translated to the left square bracket, "[".

```
TRANSLATETABLE NUMBERS_TO_PERIODS (EBCDIC TO EBCDIC,
                                   "0123456789" TO ".")
```

Translates all EBCDIC characters to themselves except for the digits, which it translates to periods (.).

Translate Table Indexing

The size of a translate table is determined by the size of the source characters (the characters to be translated): 4-bit characters require a 4-word table; 6-bit characters require a 16-word table; 7-bit and 8-bit characters require a 64-word table. A translate table is a one-dimensional read-only array.

Each word in a translate table (Figure 4-1) has its low-order 32 bits divided into four 8-bit fields, numbered 0 to 3 from left to right. The high-order 16 bits are all zeros.

When a character is to be translated, the binary representation of the character is divided into two parts: a "word index" and a "field index." The field index consists of the two low-order bits; the word index consists of the remaining high-order bits. The word index designates the word in the translate table in which the field index designates the character into which the source character is to be translated.

The diagram below shows indexing for the translation of "a" to "A" that would result from the declaration

```
TRANSLATETABLE UPCASE (EBCDIC TO EBCDIC,
                       "abcdefghijklmnopqrstuvwxyz" to
                       "ABCDEFGHIJKLMNOPQRSTUVWXYZ")
```

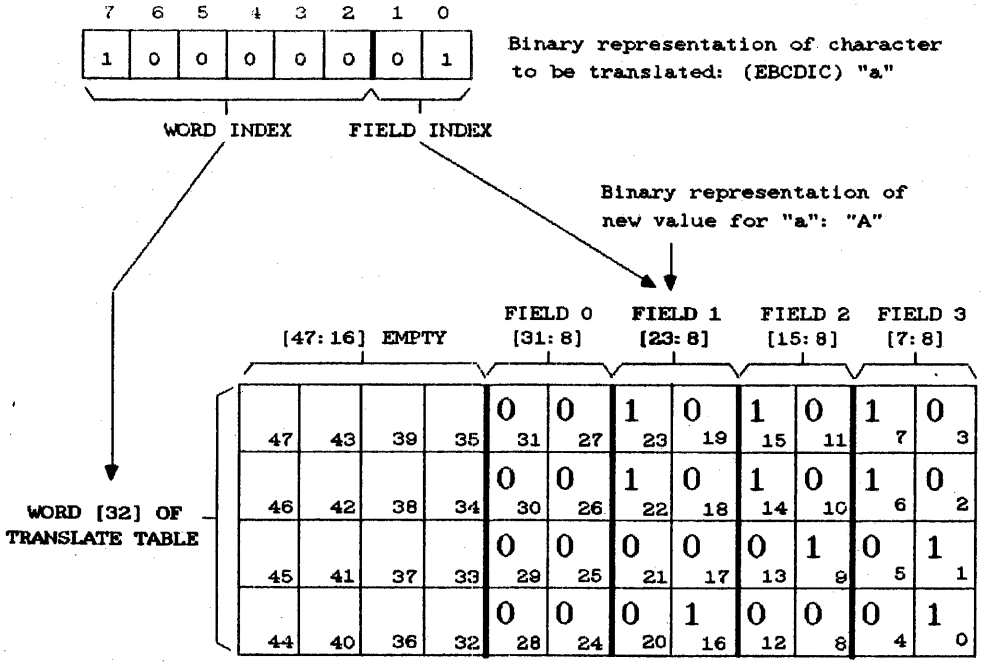


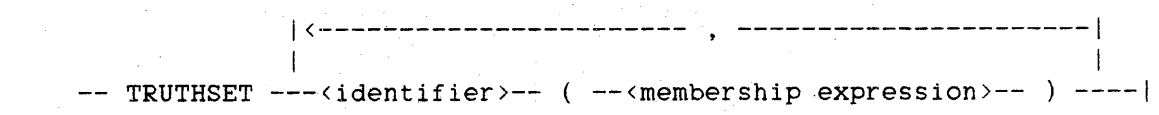
Figure 4-1. Translate Table Indexing

TRUTHSET DECLARATION

The TRUTHSET declaration associates an identifier with a set of characters. The identifier can then be used in a SCAN statement to scan while or until any character in the truth set occurs. The identifier can also appear as a condition in a REPLACE statement, so that replacement takes place while or until any character in the truth set occurs.

Syntax

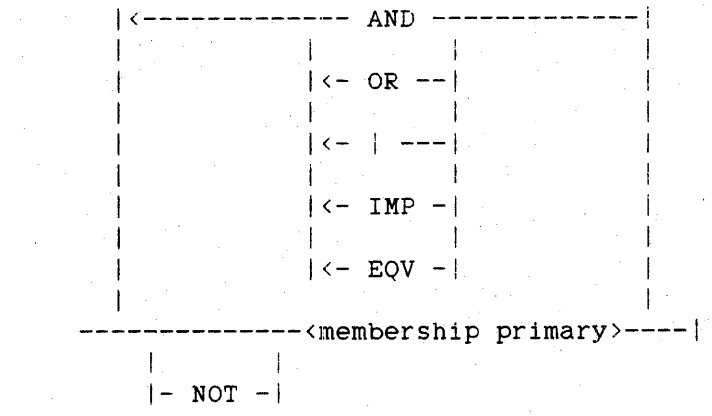
<truth set declaration>



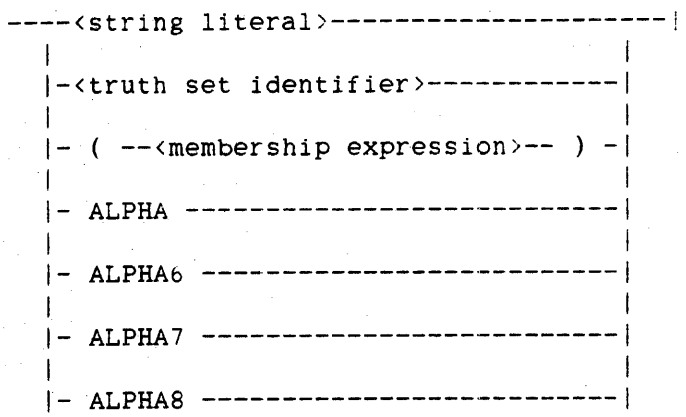
<truth set identifier>

An <identifier> that is associated with a membership expression in a TRUTHSET declaration.

<membership expression>



<membership primary>



Semantics

All membership primaries of a membership expression must be of the same character size (4-bit, 6-bit, 7-bit, or 8-bit); this character size determines the type of the truth set. The character size of a string literal is determined by the maximum character size indicated by its component string codes. (For more information, refer to "String Literal" in the chapter "Language Components.")

A membership expression is evaluated according to the normal rules of precedence for Boolean operators. This precedence is described under "Boolean Expression."

ALPHA, ALPHA6, ALPHA7, and ALPHA8 are intrinsic truth sets defined as follows:

ALPHA6	A truth set that contains the BCL digits and uppercase letters
ALPHA7	A truth set that contains the ASCII digits and uppercase letters
ALPHA8	A truth set that contains the EBCDIC digits and uppercase letters
ALPHA	A truth set that contains the digits and uppercase letters of the default character type

If a default character type is not explicitly specified by the compiler control options ASCII or BCL, then the default character type is EBCDIC, and ALPHA is the same as ALPHA8. If the ASCII compiler control option is TRUE, then ALPHA is the same as ALPHA7. If the BCL compiler control option is TRUE, then ALPHA is the same as ALPHA6.

NOTE

The BCL data type is not supported on all A Series and B 5000/B 6000/B 7000 Series systems. The appearance of a BCL construct that may cause the creation of a BCL descriptor, such as the ALPHA6 intrinsic truth set, will cause the program to get a compile-time warning message.

Pragmatics

From the characters in a TRUTHSET declaration, the compiler builds a truth set table, which is used in a truth set test to determine whether a given character is a member of that group of characters.

All truth sets declared by a single TRUTHSET declaration are stored in a single read-only array. Separate TRUTHSET declarations produce separate read-only arrays.

A truth set test references a bit in the read-only array containing the truth set by dividing the binary representation of the character being tested into two parts: the low-order five bits are used as a bit index, and the three high-order bits are used as a word index. If the size of the source character is smaller than eight bits, high-order zero bits are inserted to make an 8-bit character before the indexing algorithm is used.

The word index selects a particular word in the truth set table. The bit index is then subtracted from 31, and the result is used to reference one of the low-order 32 bits in the selected word. If the bit selected by the following expression is equal to 1, the character is a member of the truth set:

```
TABLE[CHAR.[7:3]].[(31-CHAR.[4:5]):1]
```

Figure 4-2 shows an example of a truth set test. In this example, the referenced bit (13) is equal to 1; therefore, the test character is a member of the truth set.

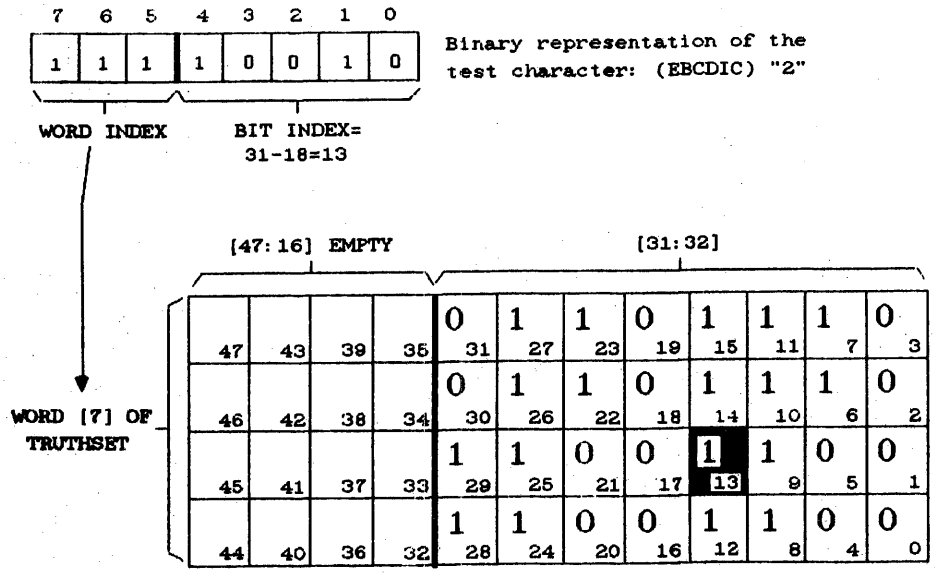


Figure 4-2. Truth Set Test

Examples

TRUTHSET T(ALPHA)

Declares T to be a truth set with membership equal to that of ALPHA. ALPHA consists of all uppercase letters and the digits 0 through 9, in the default character set.

TRUTHSET Z(ALPHA8 OR "-")

Declares Z to be a truth set with membership of ALPHA8 and the hyphen (-).

TRUTHSET NUMBERS("0123478956")

Declares NUMBERS to be a truth set with a membership of the digits 0 through 9 in the default character set.

```
TRUTHSET LETTERS(ALPHA AND NOT NUMBERS)
```

Declares LETTERS to be a truth set with a membership of ALPHA but not the digits 0 through 9; that is, consisting of the uppercase letters in the default character set.

```
TRUTHSET HEXN(4"123"), BCLN(6"123"), ASCN(7"123")
```

Declares three truth sets:

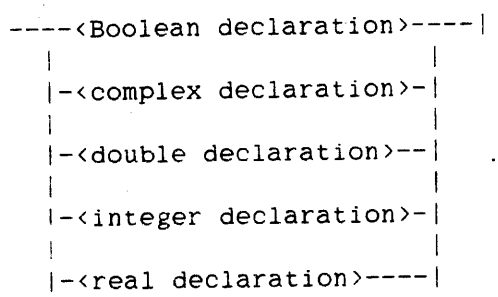
1. HEXN, with a membership of the hexadecimal characters 1, 2, and 3
2. BCLN, with a membership of the BCL characters 1, 2, and 3
3. ASCN, with a membership of the ASCII characters 1, 2, and 3

TYPE DECLARATION

A type declaration declares simple variables that can be used in a manner appropriate to the specified type.

Syntax

<type declaration>



Semantics

Type-transfer functions can be used (as can the equation part feature) to perform operations on a variable other than those that are valid for the type of the variable.

Each type of simple variable is used as follows:

Type	Meaning/Description
BOOLEAN	Boolean values. A Boolean variable is a one-word variable in which the Boolean value (TRUE or FALSE) depends on the low-order bit (bit zero) of the word; use of partial word parts and concatenation allows all 48 bits to be tested or manipulated as needed.
COMPLEX	Complex values. A complex variable is a two-word variable in which the first word contains the real part and the second word contains the imaginary part.
DOUBLE	Double-precision arithmetic values. A double-precision variable is a two-word variable.
INTEGER	Integer arithmetic values. An integer value is one that has an exponent of zero and no fractional part. Integer variables are one-word variables.

Type	Meaning/Description
REAL	Real arithmetic values. A real value is one that can have an exponent and a fractional part. Real variables are one-word variables.

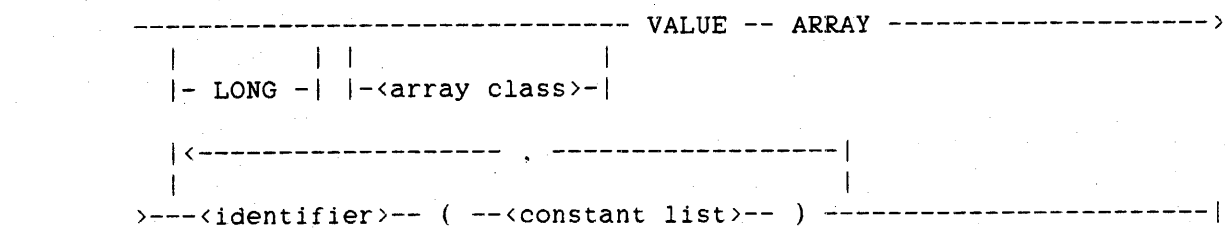
The appendix "Data Representation" contains more information regarding the internal structure of each type of simple variable as implemented on A Series and B 5000/B 6000/B 7000 Series systems.

VALUE ARRAY DECLARATION

A VALUE ARRAY declaration declares a read-only, one-dimensional array of constants.

Syntax

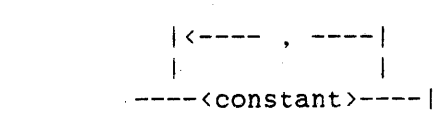
<value array declaration>



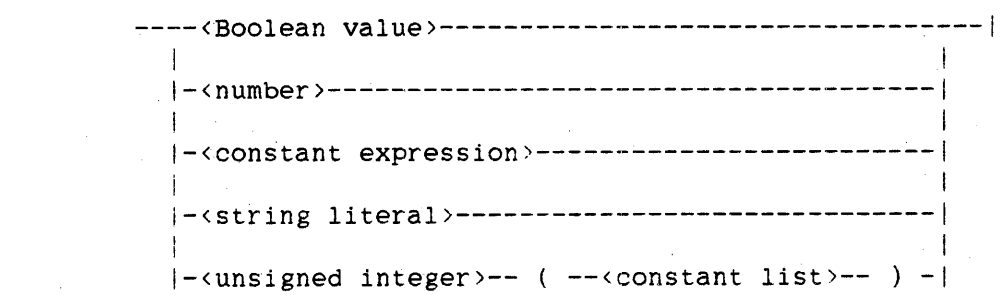
<value array identifier>

An <identifier> that is associated with a value array in a VALUE ARRAY declaration.

<constant list>



<constant>



<constant expression>

An arithmetic, Boolean, or complex expression that can be entirely evaluated at compile time.

See also

Semantics

A value array is a one-dimensional, read-only array. An element of a value array is referenced in the same manner as for any other array; that is, through a subscripted variable or by using a pointer. However, an attempt to store a value into a value array is flagged with a compile-time or run-time error.

The lower bound of a value array is zero.

Normally, a value array longer than 1024 words is automatically paged (segmented) at run time into segments 256 words long. LONG specifies that the value array is not to be paged, regardless of its length.

If no array class appears in a VALUE ARRAY declaration, REAL is assumed.

NOTE

The BCL data type is not supported on all A Series and B 5000/B 6000/B 7000 Series systems. The appearance of a BCL construct that may cause the creation of a BCL descriptor, such as a BCL value array, will cause the program to get a compile-time warning message.

Each constant initializes an integral number of words. The number of words initialized depends on the type of the array and the kind of constant.

Single-precision numbers, single-precision expressions, Boolean values, and Boolean expressions initialize one word in value arrays other than double or complex value arrays. In double value arrays, this word is extended with a second word of zero. In complex value arrays, this word is normalized and then extended with an imaginary part of zero.

Double-precision numbers and expressions are stored unchanged in two words in double value arrays. In complex value arrays, the value is rounded and normalized to single precision and then extended with an imaginary part of zero. For other types of value arrays, the second

word of the double-precision value is dropped and the first word initializes one word of the array.

Complex expressions can appear only in complex value arrays, and they initialize two words of the array.

String literals more than 48 bits long initialize as many words as are needed to contain the string and are left-justified with trailing zeros inserted in the last word, if necessary. In complex and double value arrays, long string literals can initialize an odd number of words, causing the following constant to start in the middle of a two-word element of the array.

String literals less than or equal to 48 bits long are right-justified within one word with leading zeros, if necessary. This word initializes one word in value arrays other than double or complex value arrays. In double value arrays, this word is extended with a second word of zero. In complex value arrays, this word is normalized and then extended with an imaginary part of zero.

The "<unsigned integer> (<constant list>)" form of constant causes the values within the parentheses to be repeated the number of times specified by the unsigned integer.

Pragmatics

The Master Control Program (MCP) overlays value arrays more efficiently than other arrays because value arrays need not be written to disk when their space in memory is relinquished.

The maximum size of an unpaged (unsegmented) value array is 4095 words; the maximum size of a paged value array is 32,767 words.

Example

```
VALUE ARRAY DAYS ("MONDAY      ", "TUESDAY     ",
                  "WEDNESDAY   ", "THURSDAY    ",
                  "FRIDAY      ", "SATURDAY    ",
                  "SUNDAY      ")
```

Declares DAYS to be a value array of real elements. DAYS stores the names of the days of the week, one day name in each two words. The string "FRIDAY ", for example, is stored in DAY[8] and DAY[9], and can be retrieved by assigning a pointer to DAY[8] and using the pointer.

5 STATEMENTS

Statements are the active elements of an ALGOL program. They indicate an operation to be performed. Statements are normally executed in the order in which they appear in the program. This sequential flow of execution can be altered by a statement that transfers control to another program location.

Syntax

<statement>

Note that <statement> can be null or empty. Each of the following metatokens represents a valid ALGOL statement.

<accept statement>
<assignment statement>
<attach statement>
<block>
<breakpoint statement>
<call statement>
<cancel statement>
<case statement>
<cause statement>
<causeandreset statement>
<changefile statement>
<checkpoint statement>
<close statement>
<compound statement>
<continue statement>
<deallocate statement>
<detach statement>
<disable statement>
<display statement>
<do statement>
<enable statement>
<exchange statement>
<fill statement>
<fix statement>
<for statement>
<free statement>
<freeze statement>
<go to statement>
<if statement>
<liberate statement>
<lock statement>
<merge statement>
<messagesearcher statement>
<multiple attribute assignment statement>
<on statement>
<open statement>
<procedure invocation statement>
<process statement>
<procure statement>
<programdump statement>
<read statement>
<removefile statement>
<replace family-change statement>
<replace pointer-valued attribute statement>
<replace statement>
<reset statement>
<resize statement>
<rewind statement>
<run statement>
<scan statement>
<seek statement>
<set statement>
<sort statement>
<space statement>
<swap statement>
<thru statement>
<wait statement>
<waitandreset statement>
<when statement>
<while statement>
<write statement>
<zip statement>

Semantics

The syntax for <statement> is recursive--a statement can be a block or a compound statement, each of which, in turn, can include statements. For a description of the syntax of <block> and <compound statement>, refer to the chapter "Program Structure."

Statements can be labeled or unlabeled. A <labeled statement> is of the following form:

--<label identifier>-- : --<statement>--|

An <unlabeled statement> is any statement that is NOT a labeled statement.

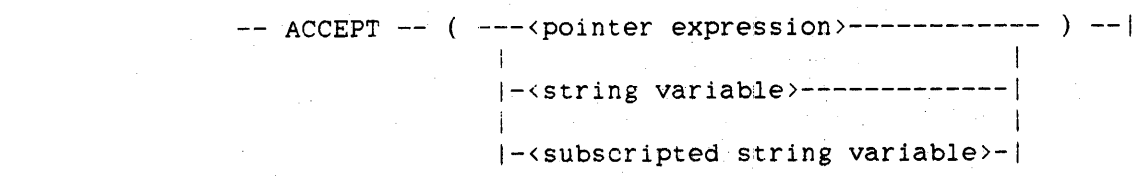
With the exceptions of <block> and <compound statement>, which are described in the chapter "Program Structure." each of the above statements is described in this chapter.

ACCEPT STATEMENT

The ACCEPT statement causes the display of a specified message on the Operator Display Terminal (ODT).

Syntax

<accept statement>



See also
<string variable> 525
<subscripted string variable> 525

Semantics

The message displayed on the ODT is designated by the parameter to the ACCEPT statement. If the parameter is a pointer expression, then execution of the ACCEPT statement causes the characters to which the pointer expression points to be displayed on the ODT. The pointer expression must point to EBCDIC characters, and the message to be displayed must be terminated by the EBCDIC null character (48"00"). Following display of the characters, the program is suspended until a response is entered at an ODT. The response is placed, left-justified, with leading blanks discarded and with an EBCDIC null character added at the end, into the location to which the pointer expression points, and the program continues execution with the statement following the ACCEPT statement.

If the parameter to the ACCEPT statement is a string variable or subscripted string variable, then execution of the ACCEPT statement causes the contents of the specified string to be displayed on the ODT. The string variable or subscripted string variable must be of type EBCDIC. Following the display of the characters, the program is suspended until a response is entered at an ODT. The response is placed in the string variable or subscripted string variable, and the program continues execution with the statement following the ACCEPT statement.

The ACCEPT statement can be used as a Boolean function. If a response is not available, the value of the ACCEPT statement is FALSE. If a response is available, the value of the ACCEPT statement is TRUE, and the response is placed in the specified location. The program continues execution regardless of the value returned by the ACCEPT statement.

Pragmatics

No more than 430 characters can be displayed by the ACCEPT statement. No more than 960 characters can be accepted as a response.

The response to the ACCEPT statement can be entered before the actual execution of that statement. The response can be entered using the AX (Accept) ODT command. For more information, refer to the "Operator Display Terminal (ODT) Reference Manual."

Examples

```
ACCEPT(POINTER(Z,8))
```

Displays the string of EBCDIC characters in the array Z, from the beginning of the array to the EBCDIC null character (48"00).

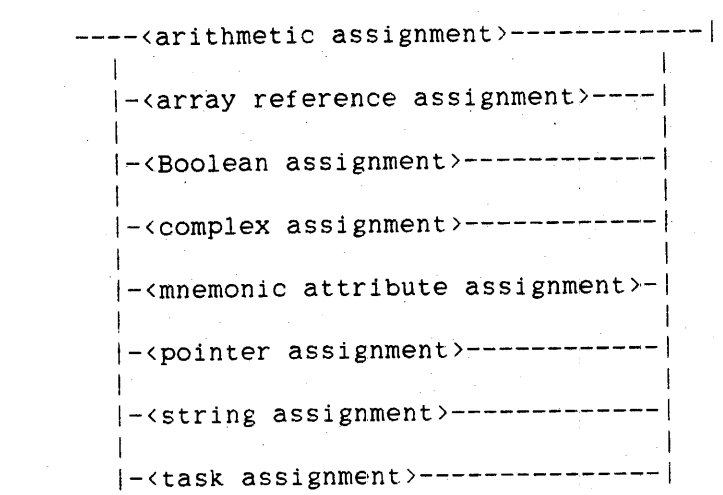
```
IF ACCEPT(STR) THEN
   DISPLAY("THANK YOU.")
ELSE
   DISPLAY("PLEASE RE-ENTER.")
```

Displays the contents of string STR on the ODT. If a response is available, the string "THANK YOU." is displayed. If no response is available, the string "PLEASE RE-ENTER." is displayed.

ASSIGNMENT STATEMENT

The assignment statement causes the item on the right of the assignment operator (:=) to be evaluated and the resulting value to be assigned to the item on the left of the assignment operator.

Syntax



Semantics

The action of an assignment statement is as follows:

1. The location of the target is determined.
2. The item following the assignment operator (:=) is evaluated.
3. The resulting value is assigned to the target.

The syntax, semantics, pragmatics, and examples for each form of the assignment statement are discussed in the following pages.

The various forms of the assignment statement are called "assignments" instead of "statements" because they can appear both as statements and in expressions. For example,

```
A := A + 1
```

is a statement when it stands alone. However, the same construct can be used in an expression, such as in

```
IF (A := A + 1) > 100 THEN <statement>
```

Pragmatics

Too many arithmetic, Boolean, complex, pointer, or string assignments in one statement can cause a stack overflow fault in the compiler. The fault can be avoided by breaking the statement into several separate statements, each containing fewer assignments, or by increasing the maximum stack size for the program by using the task attribute STACKLIMIT.

Examples

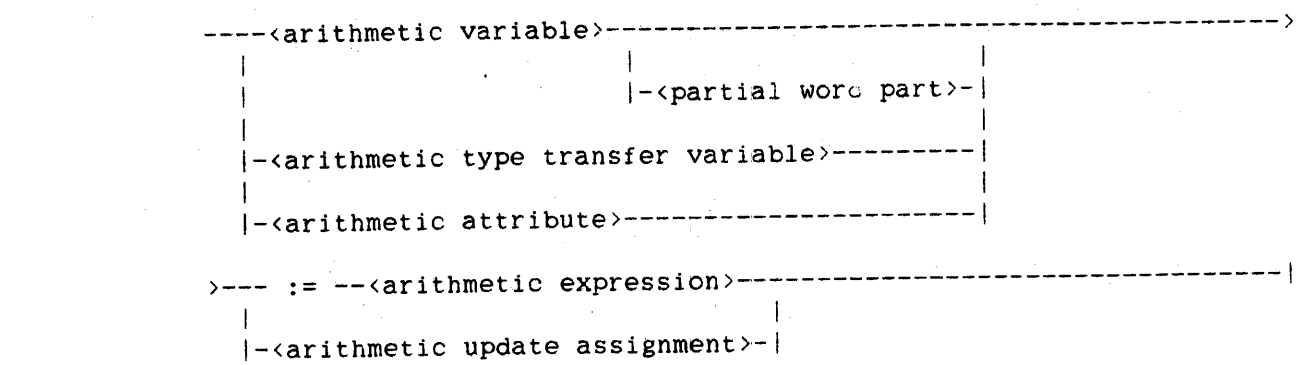
```
A := A + 1

XRAY := A[3,*]

BOOL := FALSE

CMP := COMPLEX(R1,I1)

L.LIBACCESS := VALUE(BYTITLE)

PTR := POINTER(INARAY,8)

STR := "LONG MESSAGE"

TSK.EXCEPTIONTASK := T1
```

Arithmetic Assignment

An arithmetic assignment assigns the value of the arithmetic expression on the right side of the assignment operator (:=) to the arithmetic target on the left side.

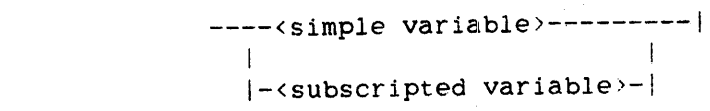
Syntax

<arithmetic assignment>



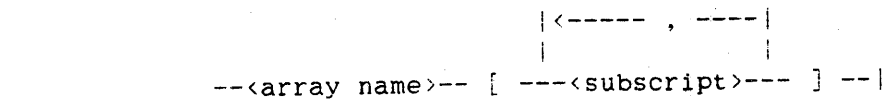
<arithmetic variable>

<variable>

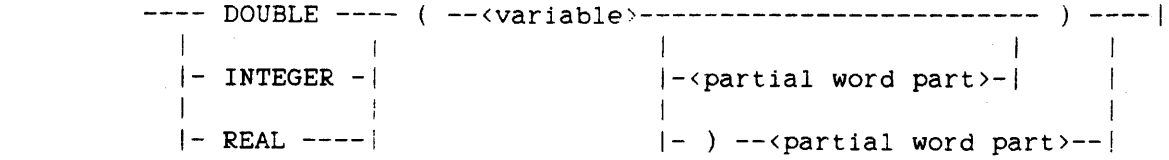


<simple variable>

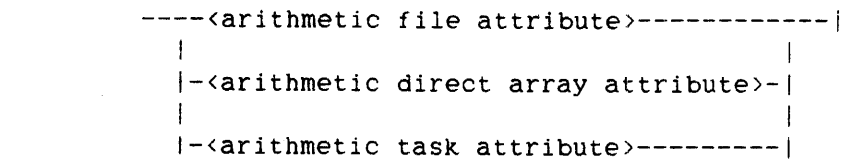
<subscripted variable>



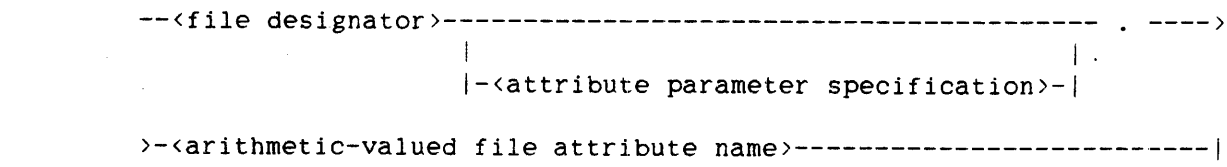
<arithmetic type transfer variable>



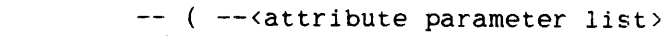
<arithmetic attribute>



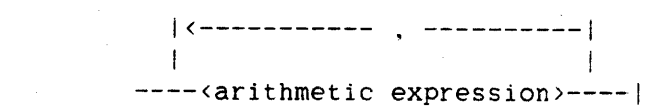
<arithmetic file attribute>



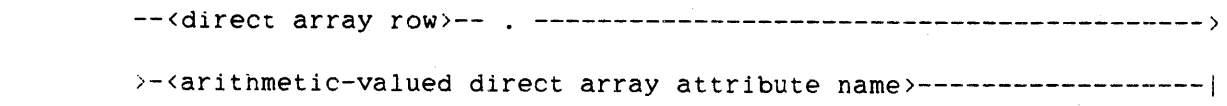
<attribute parameter specification>



<attribute parameter list>



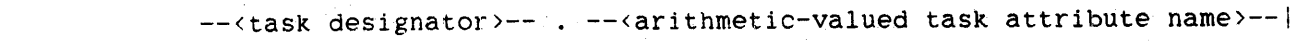
<arithmetic direct array attribute>



‹arithmetic-valued direct array attribute name›

ALGOL supports all direct array attributes and direct array attribute values described in the "I/O Subsystem Reference Manual."

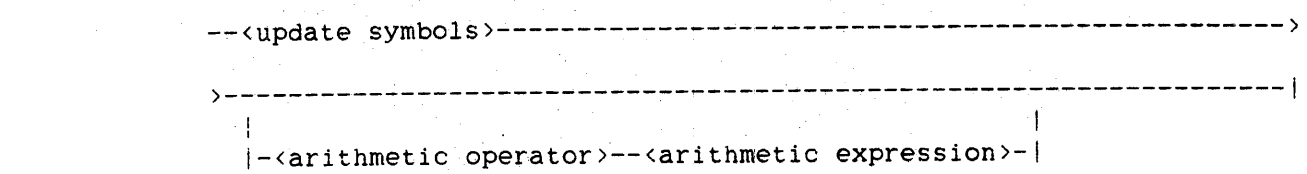
‹arithmetic task attribute›



‹arithmetic-valued task attribute name›

CLASS	MAXLINES	STARTTIME
COMPILETYPE	MAXPROCTIME	STATION
COREESTIMATE	OPTION	STATUS
DECLAREDPRIORITY	ORGUNIT	STOPPOINT
ELAPSEDTIME	PROCESSIOTIME	SUBSPACES
HISTORY	PROCESSTIME	TARGETTIME
INITIATOR	RESTART	TASKATTERR
JOBNUMBER	STACKNO	TASKVALUE
MAXCARDS	STACKSIZE	TYPE
MAXIOTIME		

‹arithmetic update assignment›



‹update symbols›

See also

Semantics

If the declared type of the target item to the left of the assignment operator (:=) and the type of the value to be assigned to it are different, then the appropriate implicit type conversion is performed according to the following rules:

1. If the left side is of type INTEGER and the expression value is of type REAL, then the value is rounded to an integer before it is stored.

2. If the left side is of type INTEGER and the expression value is of type DOUBLE, then the value is rounded to a single-precision integer before it is stored.

3. If the left side is of type REAL and the expression value is of type INTEGER, then the value is stored unchanged.

4. If the left side is of type REAL and the expression value is of type DOUBLE, then the value is rounded to single precision before it is stored.

5. If the left side is of type DOUBLE and the expression value is of type INTEGER or REAL, then the value is converted to double precision by appending a second word of zero (all bits equal to zero) before it is stored.

If more than one assignment operator appears in a single assignment (for example, A := B := C := 1.414), assignment of values is executed from right to left. If, during this process, a value is converted to another type so that it can be assigned, then it remains in that converted form following that assignment; that is, the value does not resume its original form. For example, if the following program is executed:

```
BEGIN
   DOUBLE DBL1, DBL2;
   REAL REL1, REL2;
   INTEGER INT1;
   DBL2 := REL2 := INT := REL1 := DBL1 := 1.414213562373095048801;
END.
```

the variables are assigned the following values:

```
DBL1 = 1.414213562373095048801
REL1 = 1.41421356237
INT  = 1
REL2 = 1.0
DBL2 = 1.0
```

The arithmetic update assignment is a shorthand form of assignment that can be used when the arithmetic target on the left side of the assignment operator also appears in the arithmetic expression on the right side of the operator. The arithmetic update assignment form can be specified only following an arithmetic target that does not contain a partial word part. The asterisk (*) represents a duplication of the item to the left of the assignment operator. For example,

```
A := * + 1
```

produces the same results as

```
A := A + 1
```

The target item is not re-evaluated at the appearance of the asterisk. Hence, if I equals zero initially, then

```
B[I := I + 1] := * + 1
```

is equivalent to

```
B[1] := B[1] + 1
```

but it is not equivalent to

```
B[1] := B[2] + 1
```

If the item to the left of the assignment operator is a subscripted variable, it cannot reference a value array.

Pragmatics

If the "<arithmetic variable> <partial word part>" syntax or the "<arithmetic attribute>" syntax appears in a statement with multiple assignments, then it must appear as the leftmost target in the statement. The following examples illustrate this rule.

Allowed	Not Allowed
`X.[7:8] := Y := 1`	`Y := X.[7:8] := 1`
`F1.MAXRECSIZE := RECLNGTH := 30`	`RECLNGTH := F1.MAXRECSIZE := 30`

Examples

```
VAL := 7

A[4,5].[30:4] := X

FYLE.AREAS := 50

FYLE(5).AREAS := 10

DIRARAY.IOCW := 4"1030"

TSK.COREESTIMATE := 10000

NEWARRAY[I] := * + OLDARRAY[I]

ONE := SIN(X := 3)**2 + COS(X)**2

DISTANCE := SQRT(X**2 + Y**2 + 2**2)
```

Array Reference Assignment

An array reference assignment associates a variable, called an array reference variable, with an array or a portion of an array. The array reference variable can then be used to reference the array or array portion.

Syntax

<array reference assignment>

--<array reference variable>-- := --<array designator>--|

<array reference variable>

--<array reference identifier>--|

See also
<array designator>. 43
<array reference identifier>. 52

Semantics

The array designator indicates the array or array portion to be associated with the array reference variable. Following an array reference assignment, the array reference variable becomes a referred array, describing the same data as the array designator, which can itself be an original array or another referred array.

The array reference variable cannot be global to the array designator.

If the array reference variable is declared as DIRECT, then only an array designator for a direct array can be assigned to it. However, a non-direct array reference variable can be assigned an array designator for either a direct or a non-direct array.

The dimensionality of the array reference variable and the array designator must be the same. If both are multidimensional, then the array classes must be compatible. INTEGER, REAL, and BOOLEAN types are compatible with each other. Other array classes are compatible only with themselves. If the array reference identifier and the array designator are both one-dimensional, then they can have any array class.

The size of each dimension of a multidimensional array reference variable is the same as the size of the corresponding dimension of the array designator. The size of a one-dimensional array reference variable is determined by the size and element width of the array designator and the element width for the array class with which the array reference variable was declared. Let Sa and Wa be the size and element width, respectively, of the array designator, and let Wr be the element width for the array reference variable. The size of the array reference variable, Sr, is then

```
Sr := (Sa * Wa) DIV Wr
```

Because of the truncation implicit in the DIV operation, Sr * Wr may be less than Sa * Wa. In this case, indexing the array reference variable by S + LB, where LB is the lower bound in the ARRAY REFERENCE declaration, causes an invalid index fault. Nevertheless, pointer operations using the array reference variable can access the entire area of memory allocated to the original array to which the array designator ultimately refers; the memory area may hold more than Sr elements of width Wr.

If the array designator is an uninitialized array reference variable, the array reference assignment causes the target array reference variable to become uninitialized.

Pragmatics

An array reference assignment generates a "copy descriptor" of an array or array row.

Typical uses of an array reference assignment include the following:

- To perform more efficiently arithmetic operations on multidimensional arrays (for example, by extracting a particular row to avoid repeated indexing to the same row)
- For concurrent, but different, uses of the same array (for example, for storing values of type REAL into an array that is originally declared as Boolean)

Examples

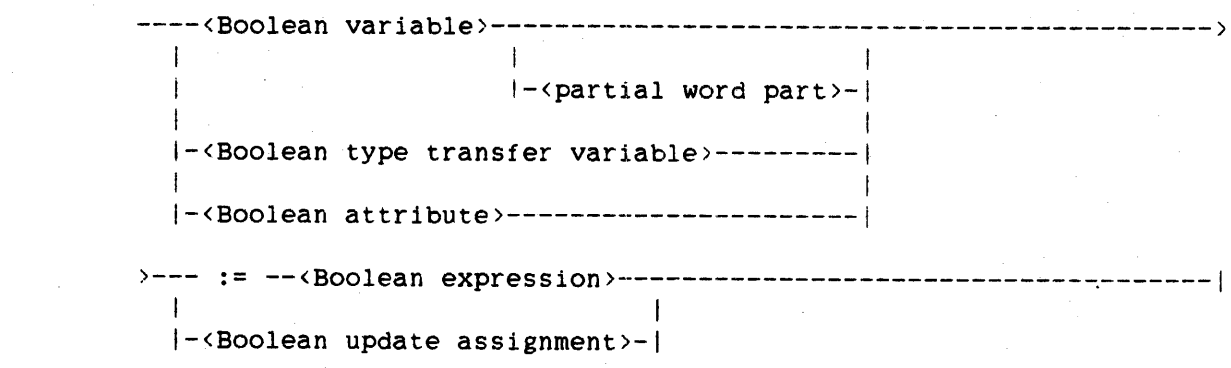
```
BOOLARRAY := REELARRAY

EBCDICARAY := INPUTARAY[*]

SUBARRAY := BIGARRAY[N,*,*]

ARAYROW := MULTIDIMARAY[I,J,K,*]
```

Boolean Assignment

A Boolean assignment assigns the value of the Boolean expression on the right side of the assignment operator (:=) to the Boolean target on the left side.

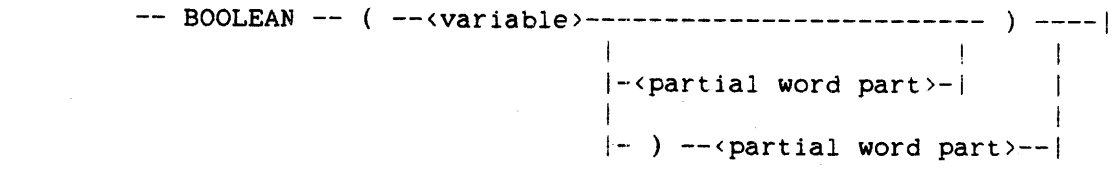
Syntax

<Boolean assignment>

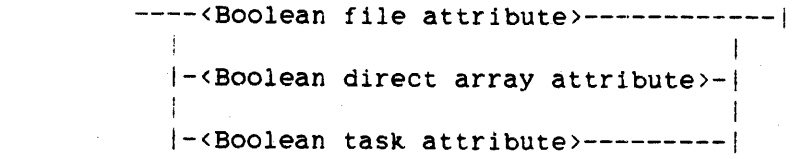


<Boolean variable>

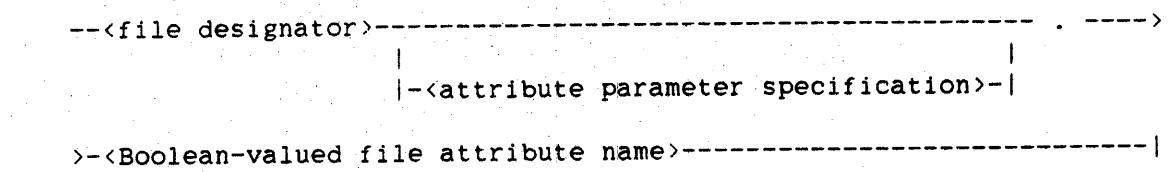
<Boolean type transfer variable>



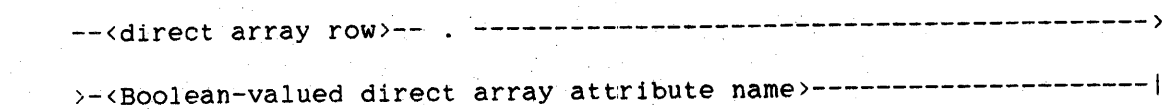
<Boolean attribute>



<Boolean file attribute>



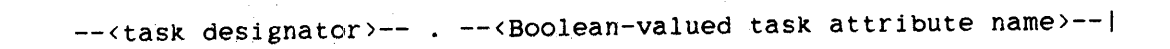
<Boolean direct array attribute>



<Boolean-valued direct array attribute name>

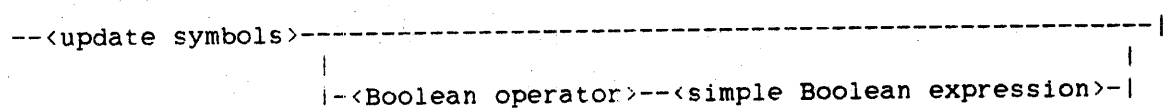
ALGOL supports all direct array attributes and direct array attribute values described in the "I/O Subsystem Reference Manual."

<Boolean task attribute>



<Boolean-valued task attribute name>

<Boolean update assignment>



See also

Semantics

The Boolean update assignment is a shorthand form of assignment that can be used when the Boolean target on the left side of the assignment operator (:=) also appears in the Boolean expression on the right side of the operator. The Boolean update assignment form can be specified only following a Boolean target that does not contain a partial word part. The asterisk (*) represents a duplication of the item to the left of the assignment operator. For example,

```
B := * AND BOOL
```

produces the same results as

```
B := B AND BOOL
```

The target item is not re-evaluated at the appearance of the asterisk.

If the item to the left of the assignment operator is a subscripted variable, it cannot reference a value array.

Examples

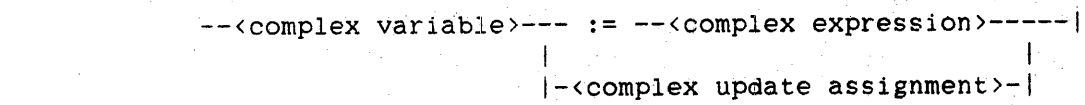
```
BOOL := TRUE

BOOLARRAY[N].[30:1] := Q < VAL

HIGHER := PTR > PTS FOR 6

TAUTOLOGY := * OR TRUE
```

Complex Assignment

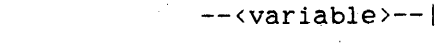
A complex assignment assigns the value of the complex expression on the right side of the assignment operator (:=) to the complex variable on the left side.

Syntax

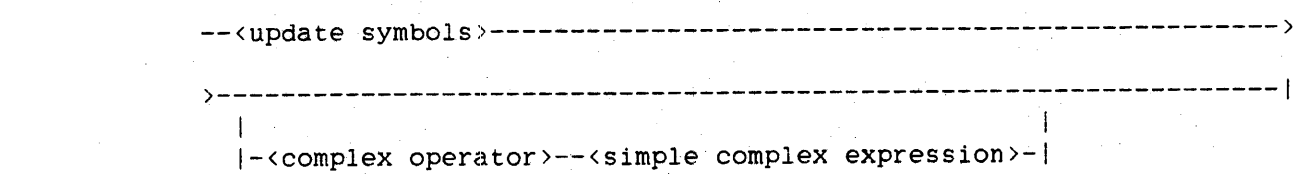
<complex assignment>



<complex variable>



<complex update assignment>



See also

Semantics

The complex update assignment is a shorthand form of assignment that can be used when the complex variable on the left side of the assignment operator (:=) also appears in the complex expression on the right side of the operator. The asterisk (*) represents a duplication of the variable to the left of the assignment operator. For example,

```
C := * + COMPLEX(3,4)
```

produces the same results as

```
C := C + COMPLEX(3,4)
```

The target variable is not re-evaluated at the appearance of the asterisk.

If the item to the left of the assignment operator is a subscripted variable, it cannot reference a value array.

Examples

```
C1 := COMPLEX(8,1.5)

C2 := * + C1/2
```

Mnemonic Attribute Assignment

A mnemonic attribute assignment assigns a value to the mnemonic-valued library attribute LIBACCESS.

Syntax

<mnemonic attribute assignment>

```
--<mnemonic attribute>-- := -- VALUE -- ( ------------------------>

>-<mnemonic attribute value>-- ) ---------------------------------|
```

<mnemonic attribute>

```
--<mnemonic library attribute>--|
```

<mnemonic library attribute>

```
--<library identifier>-- . ---------------------------------------->

>-<mnemonic-valued library attribute name>------------------------|
```

<mnemonic attribute value>

```
--<mnemonic library attribute value>--|
```

See also
<library identifier>. 129
<mnemonic library attribute value>. 130
<mnemonic-valued library attribute name>. 130

Semantics

Refer to "Library Attributes" in the "Interface to the Library Facility" chapter for a description of the library attribute LIBACCESS.

See also
Library Attributes. 665

Examples

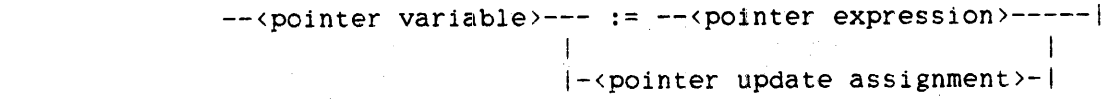
```
L.LIBACCESS := VALUE(BYTITLE)

L.LIBACCESS := VALUE(BYFUNCTION)
```

Pointer Assignment

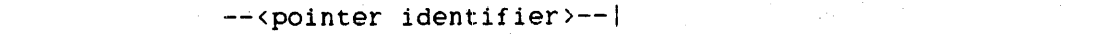
A pointer assignment assigns the pointer on the left side of the assignment operator (:=) to point to the location in an array indicated by the expression on the right side of the assignment operator. Such a pointer is then considered "initialized" and can be used in the REPLACE and SCAN statements for character manipulation.

Syntax

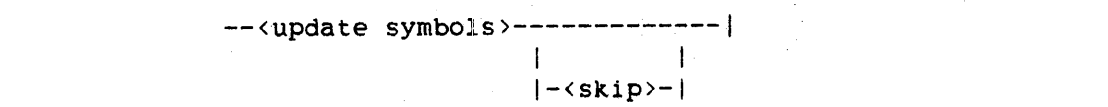
<pointer assignment>



<pointer variable>



<pointer update assignment>



See also
<pointer identifier>. 160
<skip>. 519
<update symbols>. 227

Pragmatics

A pointer assignment causes the creation of a copy descriptor to an array. The pointer variable (copy descriptor) can be set up with the needed character size by using the POINTER function syntax.

See also
<pointer function>. 565

Examples

```
PTS := EBCDICARAY[5]
```

Assigns a pointer named PTS to point to the EBCDIC character in the EBCDIC array EBCDICARAY identified by the subscripted variable EBCDICARAY[5].

```
PTR := POINTER(REALARAY)
```

Assigns a pointer named PTR to point to the leftmost character position in the first element of the real array REALARAY.

```
PINFO := PTR + 17
```

Assigns the pointer PINFO to point to the 17th character position after the character position pointed to by the pointer PTR.

```
POUT := POINTER(INSTUFF[N],4)
```

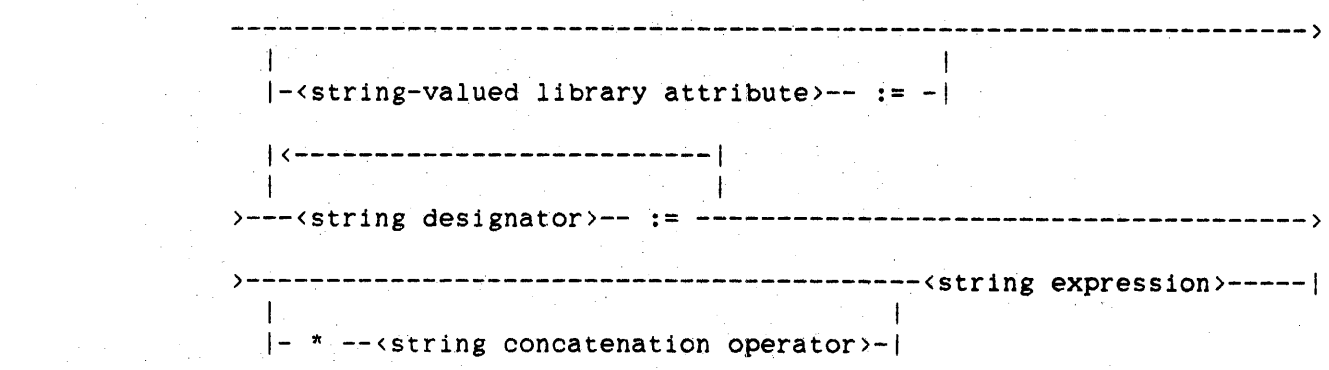
Assigns the pointer POUT to point to the leftmost character position in the array element identified by INSTUFF[N]. The "4" following the comma indicates that POUT is a hexadecimal pointer and thus points to hexadecimal characters.

String Assignment

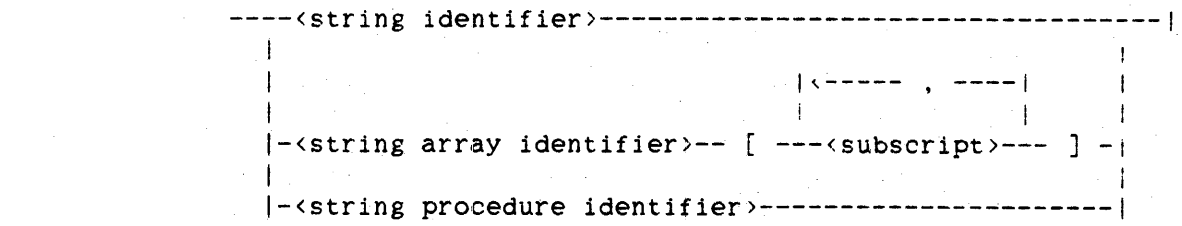
A string assignment assigns the string that results from evaluation of the string expression on the right side of the assignment operator (:=) to the string target on the left side.

Syntax

<string assignment>



<string designator>



See also

Semantics

The result of the expression on the right side of the assignment operator (:=) must be a string of the same character type as the declared type of the string designator on the left side.

Embedded assignment is not allowed. For example.

```
S1 := DROP(S2 := "ABC", 2)
```

is not allowed.

Assignment can be made to a string procedure identifier only within the body of that string procedure.

The "* <string concatenation operator>" form is a shorthand form of assignment that can be used when the string designator on the left side of the assignment operator also appears in the expression on the right side of the operator. The asterisk (*) represents a duplication of the item to the left of the assignment operator. For example,

```
S := * CAT "ABC"
```

produces the same results as

```
S := S CAT "ABC"
```

Examples

```
STRI := 8"ABCD123"
```

Assigns the EBCDIC string ABCD123 to the string variable STRI.

```
S1 := S2 := "1234"
```

Assigns the string 1234 (of the default character type) to both of the string variables S2 and S1.

```
SOUT := SOUT1 := * CAT "INPUT"
```

Concatenates the string INPUT onto the end of the string stored in SOUT1, and then assigns the result to both of the string variables SOUT1 and SOUT.

```
SOUT := * || SOUT || "ABC"
```

Concatenates the string ABC onto the end of the string stored in SOUT, and this string is then concatenated onto the end of the string stored in SOUT. The resulting string is assigned to the string variable SOUT.

Task Assignment

A task assignment associates the task designator on the right side of the assignment operator (:=) with the task indicated by the expression on the left side.

Syntax

<task assignment>

```
--<task designator>-- . --<task-valued task attribute name>-- := -->

>-<task designator>-----------------------------------------------|
```

See also
<task designator> 200
<task-valued task attribute name> 200

Semantics

The EXCEPTIONEVENT attribute of the EXCEPTIONTASK of a program is "caused" whenever the status of that program changes (for example, if the program is suspended or terminated).

The PARTNER task attribute is used in conjunction with the CONTINUE statement.

Examples

```
TISKIT.EXCEPTIONTASK := TASKIT
```

Assigns the task TASKIT to the EXCEPTIONTASK attribute of TISKIT.

```
TSK.EXCEPTIONTASK := TASKARAY[N]
```

Assigns the task identified by the task array element TASKARAY[N] to the EXCEPTIONTASK attribute of TSK.

```
TASKVARB.PARTNER := COHORT
```

Assigns the task COHORT to the PARTNER attribute of TASKVARB.

```
MYSELF.PARTNER := COWORKERS[INDX]
```

Assigns the task identified by the task array element COWORKERS[INDX] to the PARTNER attribute of MYSELF.

```
MYSELF.PARTNER.EXCEPTIONTASK := MYSELF.PARTNER.PARTNER
```

Assigns the task that is the PARTNER attribute of the task MYSELF.PARTNER to the task that is the EXCEPTIONTASK attribute of the task MYSELF.PARTNER.

ATTACH STATEMENT

The ATTACH statement associates an interrupt with an event so that when the event is caused, the program is interrupted, and the interrupt code is placed in execution (provided that the interrupt is enabled).

Syntax

<attach statement>

```
-- ATTACH --<interrupt identifier>-- TO --<event designator>--|
```

See also
<event designator> 78
<interrupt identifier> 126

Pragmatics

Although different interrupts can be simultaneously attached to the same event, a particular interrupt can be attached to only a single event at any one time. For this reason, if, at attach time, the interrupt is found to be already attached to an event, then it is automatically detached from the old event and attached to the new event. Any pending invocations of the interrupt are lost.

An interrupt can be attached to an event that is declared in a different block. For example, a local interrupt can be attached to a formal event. Such an attachment can cause compile-time or run-time up-level attach errors if the block containing the event can be exited before the block that contains the interrupt is exited.

Examples

```
ATTACH THEPHONE TO THEBELL
```

Attaches the interrupt THEPHONE to the event THEBELL. When THEBELL is caused, the code associated with THEPHONE begins executing.

```
ATTACH ANSWERHI TO MYSELF.EXCEPTIONEVENT
```

Attaches the interrupt ANSWERHI to the event MYSELF.EXCEPTIONEVENT. Whenever the task MYSELF undergoes a change in status, the EXCEPTIONEVENT attribute is caused, and the code associated with ANSWERHI begins executing.

BREAKPOINT STATEMENT

The BREAKPOINT statement invokes the breakpoint intrinsic, which allows the programmer to examine or change the values of variables during the execution of a program.

Syntax

<breakpoint statement>

Semantics

The commands accepted by the breakpoint intrinsic are described below under "Interaction with the Breakpoint Intrinsic."

To establish the required environment for running the breakpoint intrinsic, the BREAKHOST compiler control option must be used. The breakpoint intrinsic can also be called implicitly by using the BREAKPOINT compiler control option. These options are described under "Compiler Control Options" in the chapter "Compiling Programs."

NOTE

> The BREAKHOST and BREAKPOINT compiler control options and the BREAKPOINT statement will be deimplemented on the Mark 3.7 release. For a debugging facility. refer to the TADS option under "Compiler Control Options."

See also

<breakhost option> 608
<breakpoint option> 609
<TADS option> 643

Interaction with the Breakpoint Intrinsic

When the breakpoint intrinsic is called, it suspends execution of the program and waits for input from a terminal in the form of breakpoint commands. Breakpoint commands are user instructions to the breakpoint intrinsic given at run time. They are not part of ALGOL.

Breakpoint commands are of three types:

1. Display commands. These are used to display the values of variables.

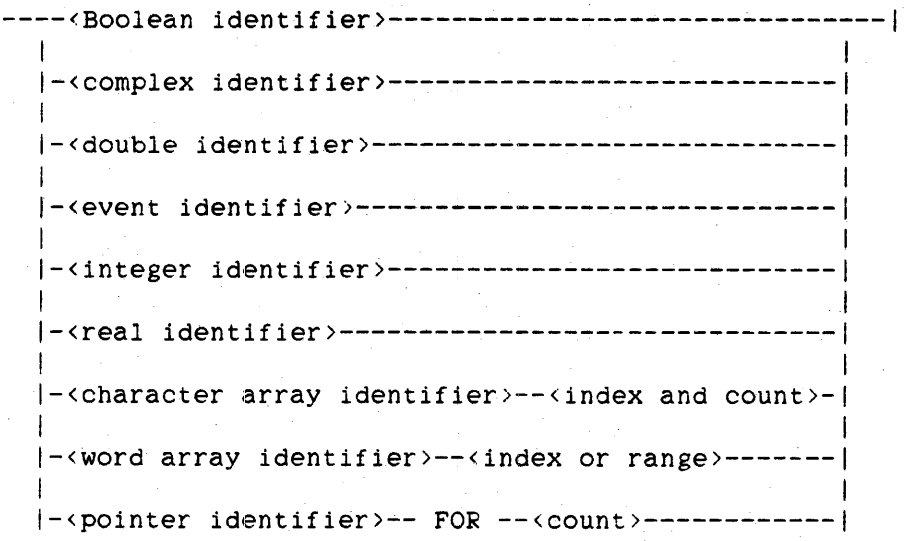
2. Control commands. These are used to change the format used for display and to continue execution of the program.

3. Reformat commands. These are used to redisplay the most recently requested variable in a new format, display its address couple, or alter its value.

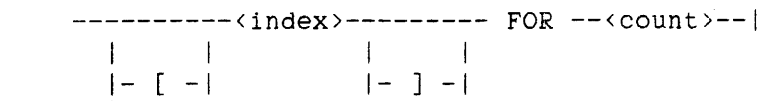
Display Commands

Display commands are used to display the values of variables.

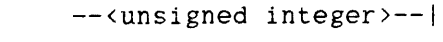
Syntax



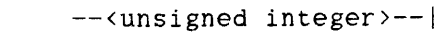
<index and count>



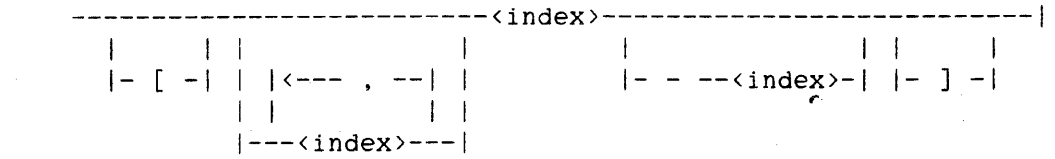
<index>



<count>



<index or range>



See also

Semantics

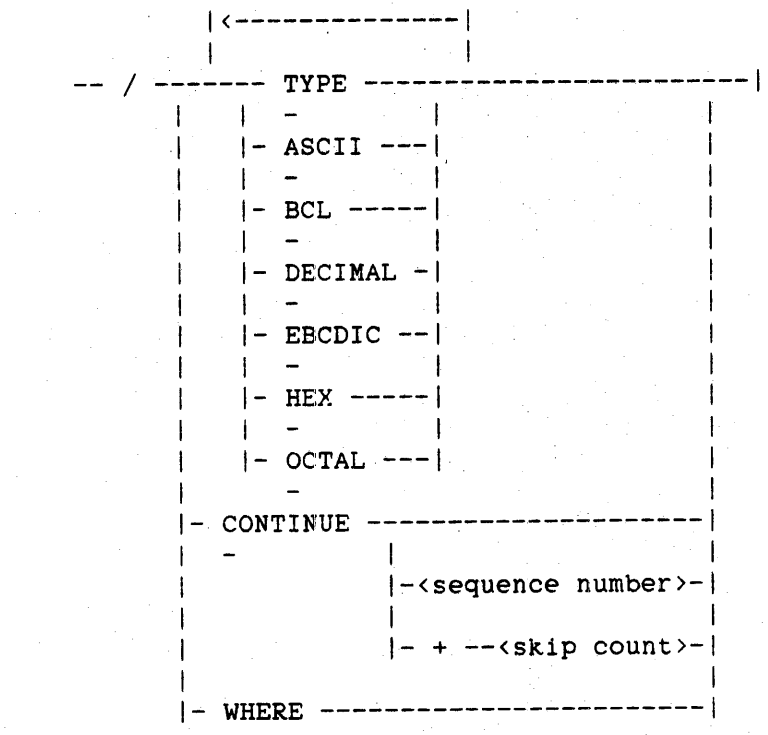
The value of a simple variable can be displayed by entering its name. The contents of a character array can be displayed by entering its name, a starting index, and a count giving the number of characters to be displayed. Only the contents of one-dimensional character arrays can be displayed. The contents of word arrays can be displayed in two ways: (1) a single element can be displayed by entering the array name and a fully specified subscript list; and (2) a range of elements within an array row can be displayed by indicating an index range in the specification for the final subscript; for example, A[2,0-3]. Arrays of type DOUBLE, COMPLEX, and EVENT cannot be displayed.

All array elements must be referenced as if the lower bound of the array is zero. For example, if an array has been declared as ARRAY A[5:10], the third element is referenced as A[7] within the program but is requested as A[2] in breakpoint display commands.

Control Commands

Control commands are used to change the format used for display and to continue execution of the program.

Syntax



<skip count>

See also
<sequence number> 613

Semantics

Control commands begin with a slash (/). The TYPE, ASCII, BCL, DECIMAL, EBCDIC, HEX, and OCTAL control commands establish the format or formats in which variables requested in subsequent display commands are displayed. The TYPE control command specifies that the format of subsequent displays is determined by the declared type of the variable as follows:

Variable Type	Display Format
Boolean	TRUE or FALSE
Complex	Floating point (real part only)
Double	Floating point (first word only)
Event	Floating point (first word only)
Integer	Integer
Real	Floating point
Word array elements	Same as simple variable of same type
Pointer	According to the character size of the pointer
Character array	According to the character type of the array

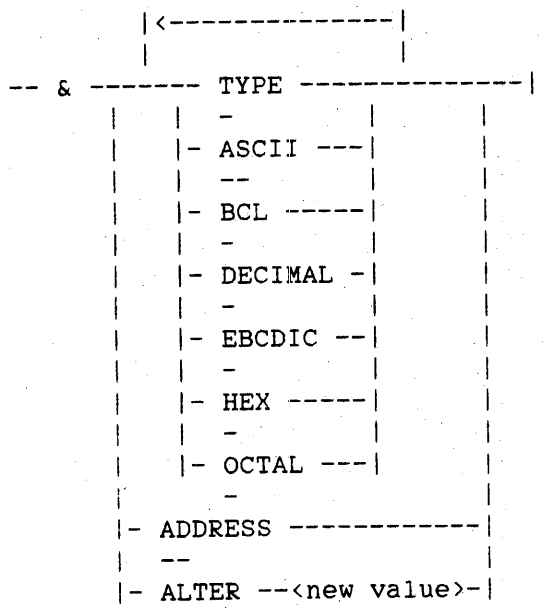
The CONTINUE control command causes program execution to continue with the ALGOL statement following the current breakpoint call. If a sequence number is specified, execution continues directly to the breakpoint call at the specified sequence number, and all intervening breakpoint calls are ignored. If no breakpoint call occurs at the specified sequence number, program execution continues to the end of the program. If a skip count is specified, program execution continues, without stopping for breakpoint input, until the breakpoint intrinsic has been called ‹skip count› times. That is, ‹skip count› - 1 breakpoint calls are ignored. "CONTINUE" and "CONTINUE + 1" are equivalent.

The WHERE control command displays the sequence number of the statement at which program execution is currently suspended.

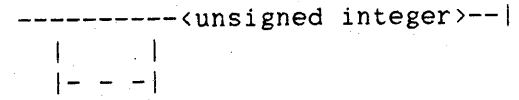
Reformat Commands

Reformat commands are used to redisplay in a new format the most recently displayed variable, display its address couple, or alter its value.

Syntax



<new value>



Semantics

Reformat commands begin with an ampersand (&). The TYPE, ASCII, BCL, DECIMAL, EBCDIC, HEX, and OCTAL reformat commands cause the variable requested by the last display command to be redisplayed in the specified format or formats. The TYPE reformat command specifies that the variable be redisplayed in the format appropriate for its declared type. The formats chosen for each type of variable are the same as those chosen when the TYPE control command is used.

The ADDRESS reformat command displays the address couple of the item requested by the last display command. The ALTER reformat command assigns the specified new value to the item requested by the last display command. Arrays and pointers cannot be altered using the ALTER command.

Example

```
1000 BEGIN
1100 $ SET BREAKHOST
1200
1300     BOOLEAN B;
1400     COMPLEX C;
1500     DOUBLE D;
1600     EVENT E;
1700     INTEGER I;
1800     REAL A, R;
1900     POINTER P;
2000     ARRAY RARRAY[0:10];
2100     BOOLEAN ARRAY BARRAY[0:5,0:8];
2200     HEX ARRAY HA[0:100];
2300
2400     PROCEDURE PROC(PARAM);
2500     VALUE PARAM;
2600     INTEGER PARAM;
2700        BEGIN
2800        REAL L;
2900        L := PARAM * 2;
3000        BREAKPOINT;
3100        END;
3200
3300     A := "ABCABC";
3400     B := TRUE;
3500 $ SET BREAKPOINT
3600
3700     C := COMPLEX(1,2);
3800     D := DOUBLE(1,2);
3900     CAUSE(E);
4000     I := 25;
4100     R := 3.14159265;
4200 $ POP BREAKPOINT
4300
4400     P := POINTER(RARRAY);
4500     REPLACE P BY "ABCK" FOR 44;
4600     BARRAY[3,3] := TRUE;
4700     REPLACE HA BY 4"ABCABC123123";
4800     PROC(3);
4900     BREAKPOINT;
5000 END.
```

In the example program above, execution proceeds normally until statement 3700, where the breakpoint intrinsic is called. Thereafter, the breakpoint intrinsic is called after each statement through statement 4100, at statement 3000 in procedure PROC, and at statement 4900.

The following is a sample execution of the program shown above. The arrows (-->) indicate breakpoint commands.

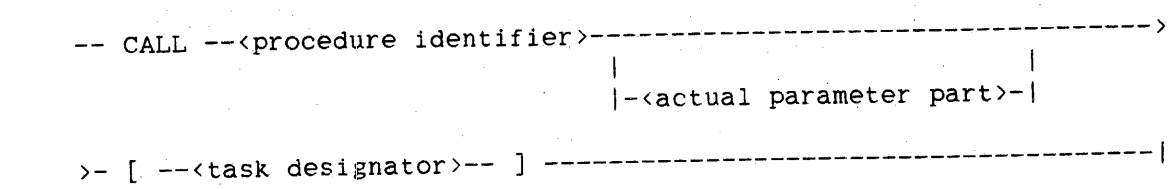
```
     BREAK @ 3700
--> /HEX DECIMAL
     BLOCK#1
--> A
     A = HEX C1C2C3C1C2C3
      -1.46817073645E+14
--> &EBCDIC
     EBC ABCABC
--> B
     B = HEX 000000000001
      1.0
--> &TYPE
     TRUE
--> /CONTINUE 3900

     BREAK @ 3900
--> C
     FIRSTWORD C = HEX 000000000001
      1.0
--> D
     FIRSTWORD D = HEX 000000000001
      1.0
--> E
     FIRSTWORD E = HEX 000000000001
      1.0
--> &ADDRESS
     E IS  (2 , 10)
--> I
     I = HEX 000000000000
      0
--> &ALTER -1
     I =
     HEX 400000000001
      -1
--> /WHERE
     BREAK @ 3900
--> /CONTINUE + 2
```

```
      BREAK @ 3000
  --> PARAM
      PROC OF BLOCK#1
      PARAM = HEX 000000000003
       3
  --> L
      L = HEX 000000000006
       6.0
  --> /CONTINUE

      BREAK @ 4900
  --> R
      BLOCK#1
      R = HEX 263243F6A792
       3.14159265
  --> P FOR 12
      P FOR 12
      ABCKABABCKAB
  --> /TYPE
  --> P FOR 12
      P FOR 12
      ABCKABABCKAB
  --> BARRAY[3,0-3]
      BARRAY[3,0-3] =
      FALSE FALSE FALSE TRUE
  --> HA[3] FOR 3
      HA[3] FOR 3
      ABC
  --> /CONTINUE
      [program finished]
```

CALL STATEMENT

The CALL statement initiates a procedure as a coroutine.

Syntax



See also
<actual parameter part> 346
<procedure identifier>. 165
<task designator> 200

Semantics

Initiation of a coroutine consists of setting up a separate stack, passing any parameters (call-by-name or call-by-value), and beginning the execution of the procedure.

Processing of the initiating program, called the "initiator" or the "primary coroutine," is suspended.

The called procedure, referred to as the "secondary coroutine," cannot be a typed procedure. The actual parameter part must agree in number and type with the formal parameter part in the declaration of the procedure; otherwise, a run-time error occurs.

The task designator associates a task with the coroutine at initiation; the values of the task attributes of that task, such as COREESTIMATE, STACKSIZE, and DECLAREDPRIORITY, can be used to control the execution of the coroutine. For more information about assigning values to task attributes, refer to <arithmetic task attribute> under "Arithmetic Assignment," <Boolean task attribute> under "Boolean Assignment," and "Task Assignment" in this chapter.

Every coroutine has a "partner" task to which control can be passed by using the CONTINUE statement. The partner task of the secondary coroutine is the initiator by default but can be changed by assignment to the task-valued task attribute PARTNER of the task designator. Local variables and call-by-value parameters of the secondary coroutine retain their values as control is passed to or from the coroutine.

The "critical block" (described under "PROCESS Statement") in the initiator cannot be exited until the secondary coroutine is terminated. Any attempt by the initiator to exit that block before the secondary coroutine is terminated causes the initiator (and all tasks it has initiated through CALL or PROCESS statements) to be terminated.

A secondary coroutine is terminated by exiting its own outermost block or by execution in the initiator of the following statement:

```
<task designator>.STATUS := VALUE(TERMINATED)
```

where the task designator specifies the task associated with the secondary coroutine to be terminated.

Pragmatics

The CALL statement causes the initiation of a separate stack as a coroutine. Because of the overhead involved, a coroutine should be established once and then used through CONTINUE statements. If a CALL statement is used to invoke a procedure, overall system efficiency is severely degraded.

See also

<arithmetic task attribute> 227
<Boolean task attribute>. 235
Task Assignment 246

Example

```
CALL COROOTEEN(X, Y, 7, X + Y + Z) [T]
```

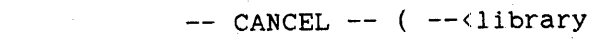
Initiates as a coroutine the procedure COROOTEEN, and passes the parameters X, Y, 7, and X + Y + Z. COROOTEEN has the task designator T associated with it.

CANCEL STATEMENT

The CANCEL statement can be used to delink a library from a program and cause the library program to "unfreeze" and resume running as a regular program.

Syntax

<cancel statement>



See also
<library identifier>. 129

Semantics

Normally, a library is linked to a program when the program calls one of the library's entry points, and the library is delinked from the program when the block in which the library is declared is exited. The CANCEL statement can be used to delink a library before it would normally be delinked.

When a library is canceled, all users of the library are delinked from the library, and the library unfreezes (thaws) and resumes running as a regular program regardless of whether it is temporary or permanent. (Refer to "FREEZE Statement" for a discussion of temporary and permanent libraries.)

After a program has canceled a library, the program can again link to the library as if for the first time.

Pragmatics

Only libraries whose SHARING compiler control option is specified as PRIVATE or SHAREDBYRUNUNIT can be canceled. If an attempt is made to cancel a library that is not PRIVATE or SHAREDBYRUNUNIT, a run-time message is given and the library remains linked to the program.

When an asynchronous process links to a library whose template is owned by the initiator program, no record of that linkage is made. If an attempt is made to cancel a library that is accessible to a task initiated by a CALL or PROCESS statement, the cancel operation is not performed, and a run-time message is given explaining why the library was not canceled.

To delink a program from a library without affecting any other users of the library, use the DELINKLIBRARY function.

For more information on libraries, refer to the chapter "Interface to the Library Facility."

See also
<delinklibrary function>. 543

Example

```
CANCEL(LIB)
```

Delinks the library LIB from the program.

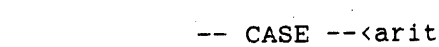
CASE STATEMENT

The CASE statement provides a means of dynamically selecting one of many alternative statements.

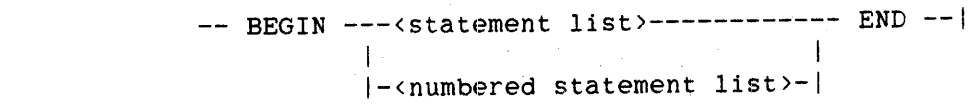
Syntax

<case statement>

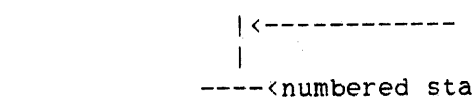
<case head>



<case body>

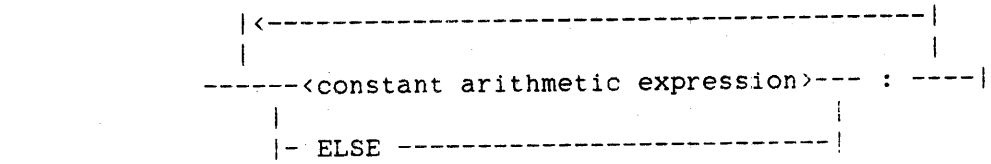


<numbered statement list>



<numbered statement group>

<number list>



See also

<constant arithmetic expression>. 476
<statement list>. 9

Semantics

Unnumbered Statement List

If the case body contains an unnumbered statement list, then the statement to be executed is selected in the following manner:

1. The arithmetic expression in the case head is evaluated. If the resulting value is not an integer, it is integerized by rounding.

2. The integer value is used as an index into the list of statements in the case body. The N statements in the case body are numbered 0 to N-1. The statement corresponding to the index value is the statement executed.

If the index value is less than zero or greater than N-1, the program is discontinued with a fault.

Numbered Statement List

If the case body contains a numbered statement list, then the statement list to be executed is selected in the following manner:

1. The arithmetic expression in the case head is evaluated. If the resulting value is not an integer, it is integerized by rounding.

2. If the integer value is equal to one of the statement numbers, the statement list associated with the number is executed.

   If the integer value is not equal to any of the statement numbers, then an invalid index fault occurs unless "ELSE" appears in a number list in the CASE statement, in which case control is transferred to the statement list following "ELSE".

The statement numbers given by the constant arithmetic expressions in the number list must lie in the range 0 to 1023, inclusive. The word "ELSE" can appear only once in a CASE statement.

Examples

```
CASE I OF
   BEGIN
      J := 1;    % STATEMENT 0
      J := 20;   % STATEMENT 1
      BEGIN      % STATEMENT 2
         J := 3;
         K := 0;
      END;
      J := 4;    % STATEMENT 3
   END;


CASE I OF
   BEGIN
      1:
      2:
      5:
      7:
         J := 3;
         Q := J-1;
      3:
      4:
      20:
         J := 4;
      ELSE:
         GO TO BADCASEVALUE;
   END;
```

CAUSE STATEMENT

The CAUSE statement activates all tasks that are waiting on the specified event.

Syntax

<cause statement>

```
-- CAUSE -- ( --<event designator>-- ) --|
```

See also
<event designator>. 78

Semantics

Normally, the CAUSE statement also sets the happened state of the event to TRUE (happened). Refer to "WAITANDRESET Statement" for exceptions.

If an enabled interrupt is attached to the event, each cause of the event results in one execution of the interrupt code.

Pragmatics

Activating a task does not necessarily place the task into immediate execution. Activating a task consists of delinking the task from an event queue (each event has its own queue) and linking that task in priority order into a system queue called the ready queue.

The ready queue is à queue of all tasks that are capable of running. Tasks are taken out of the ready queue either when a processor is assigned to the task or when the task must wait for an operation (such as an I/O operation) to complete or for an event to be caused. A task is placed in actual execution only when it is the top item in the ready queue and a processor is available.

A cause of a happened event is essentially a "no-operation"; the system does not "remember" every cause unless an interrupt is attached to the event. For more information on events, refer to "Event Statement."

Examples

CAUSE(EVNT)

Activates the tasks waiting for the event EVNT.

CAUSE(EVNTARAY[INDX])

Activates the tasks waiting for the event identified by EVNTARAY[INDX].

CAUSE(TSK.EXCEPTIONEVENT)

Activates the tasks waiting for a change in the status of the task TSK.

CAUSEANDRESET STATEMENT

The CAUSEANDRESET statement activates all tasks that are waiting on the specified event and sets the happened state of the event to FALSE (not happened).

Syntax

<causeandreset statement>

```
-- CAUSEANDRESET -- ( --<event designator>-- ) --|
```

See also
<event designator>. 78

Semantics

This statement differs from the CAUSE statement in that the happened state of the event is set to FALSE (not happened).

Pragmatics

The pragmatics of the CAUSEANDRESET statement are the same as those of the CAUSE statement.

Examples

```
CAUSEANDRESET(EVNT)
```

Activates the tasks waiting for the event EVNT, and sets the happened state of EVNT to FALSE (not happened).

```
CAUSEANDRESET(EVNTARAY[INDX])
```

Activates the tasks waiting for the event identified by EVNTARAY[INDX], and sets the happened state of that event to FALSE (not happened).

CAUSEANDRESET(TSK.EXCEPTIONEVENT)

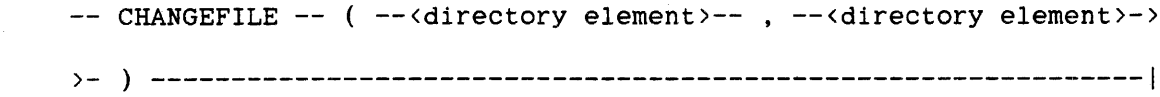
Activates the tasks waiting for a change in the status of the task TSK, and sets the happened state of TSK.EXCEPTIONEVENT to FALSE (not happened).

CHANGEFILE STATEMENT

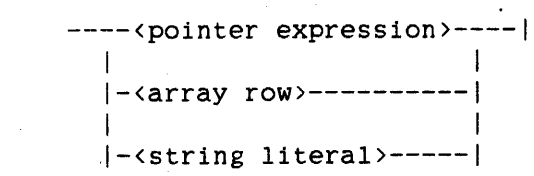
The CHANGEFILE statement changes the names of files without opening them.

Syntax

<changefile statement>



<directory element>



See also
<array row> 43

Semantics

A directory element is a file name, a directory name, or both a file name and a directory name. A directory name references a group of files. For example, the following files are all in the directory named "(JAMES)". The first six files are in the directory named "(JAMES)OBJECT", and the first five files are in the directory named "(JAMES)OBJECT/TEST". Note that "(JAMES)OBJECT/TEST/PRIMES" is both a file name and a directory name.

```
(JAMES)OBJECT/TEST/COMM
(JAMES)OBJECT/TEST/SORT
(JAMES)OBJECT/TEST/PRIMES
(JAMES)OBJECT/TEST/PRIMES/1
(JAMES)OBJECT/TEST/PRIMES/2
(JAMES)OBJECT/LIBRARY1
(JAMES)MEMO
```

In the CHANGEFILE statement, the second directory element (the "target") designates the name to which the first directory element (the "source") is to be changed. If the change applies to files on pack, and a family substitution specification is not in effect (either by default through the USERDATA file or by specification in either CANDE or WFL), the second directory element must include "ON <family name>", and the first directory element must not include a family name. If a family substitution specification is in effect, "ON <family name>" is not required; if "ON <family name>" does not appear, the family substitution specification is used to determine the family on which the files reside.

The CHANGEFILE statement returns a value of TRUE if an error occurs. Error numbers, stored in field [39:20] of the result, correspond to the causes of failure as follows:

10 The first directory element is in error.

20 The second directory element is in error.

30 File names have not been changed.

Pragmatics

File names and directory names must be specified in EBCDIC and must be followed by a period. All errors in the names are detected at run time.

If a family substitution specification is in effect, the CHANGEFILE statement affects only the substitute family, not the alternate family.

If a directory name is specified as the source, the names of the files in that directory are changed according to the following rules:

1. If the specified target directory is a new directory, then the names of all the files in the source directory are changed.

2. If the specified target directory is not a new directory, then only files that do not have corresponding names in the target directory are changed.

For example, the first column below shows file names that exist before the statement

```
CHANGEFILE("A.","B.")
```

is executed, and the second column shows the file names resulting from execution of the statement.

Existing files	Resulting files
A/B/C	B/B/C
A/B/D	A/B/D
A/C/C	B/C/C
B/B/D	B/B/D
B/C/D	B/C/D

Note that because the file name B/B/D already exists, the file name A/B/D is not changed.

3. A directory element of the form "<file name>/=" affects only files in that directory. It does not affect a file named "<file name>".

Example

The following program changes A/B to C/D and then removes C/D.

```
BEGIN
   ARRAY OLD, NEW[0:44];
   BOOLEAN B;
   REPLACE POINTER(OLD) BY 8"A/B.";
   REPLACE POINTER(NEW) BY 8"C/D.";
   IF B := CHANGEFILE(OLD,NEW) THEN
      DISPLAY("CHANGEFILE ERROR");
   IF B := REMOVEFILE(8"C/D.") THEN
      DISPLAY("REMOVEFILE ERROR");
END.
```

CHECKPOINT STATEMENT

The CHECKPOINT statement writes to a disk file the complete state of the job at a specified point. Using the disk file, the job can later be restarted from this point.

Syntax

<checkpoint statement>

```
-- CHECKPOINT -- ( --<device>-- , --<disposition>-- ) --|
```

<device>

```
---- DISK --------|
  |              |
  |- DISKPACK -|
  |              |
  |- PACK -----|
```

<disposition>

```
---- LOCK -----|
  |           |
  |- PURGE -|
```

Semantics

The checkpoint/restart facility can protect a program against the disruptive effects of unexpected interruptions during the program's execution. If a Halt/Load or other system interruption occurs, a job is restarted either before the initiation of the task that was interrupted or, if the operator permits, at the last checkpoint, whichever is more recent. Checkpoint information can also be retained after successful runs to permit restarting jobs to correct bad data situations.

The device options determine the medium to be used for the checkpoint files.

The disposition option PURGE causes all checkpoint files to be removed at successful termination of the job and protects the job against system failures. The LOCK option causes all checkpoint files to be saved indefinitely and can be used to restart a job even if it has terminated normally.

The CHECKPOINT statement can be used as a Boolean function. An attempted checkpoint returns a value with the following information:

```
[ 0: 1] = Exception bit
[10:10] = Completion code (refer to "Checkpoint/Restart
          Messages" below)
[25:12] = Checkpoint number
[46: 1] = Restart flag (1 = restart)
```

When a checkpoint is invoked, the following files are created:

1. The checkpoint file, CP/<JN>/<CPN>, where <JN> is a four-digit job number and <CPN> is a three-digit checkpoint number. If the PURGE option has been specified, the checkpoint number is always zero, and each succeeding checkpoint with PURGE removes the previous file. If the LOCK option is used, the checkpoint number starts with a value of 1 for the first checkpoint and is incremented by 1 for each succeeding checkpoint with LOCK. If the two types are mixed within a job, the LOCKed checkpoints use the ascending numbers and the PURGEd checkpoints use zero, leaving files 0 through N at the completion of the job.

2. Temporary files, CP/<JN>/T<FN>, where <FN> is a three-digit file number beginning with 1 and incremented by 1 for each temporary disk or system resource pack file.

3. The job file, CP/<JN>/JOBFILE. This file is created under the LOCK option only.

The LOCK and PURGE options are also effective when the task terminates. If the task terminates abnormally and the last checkpoint has used the PURGE option, then the checkpoint file (numbered zero) is changed to have the next sequential checkpoint number, and the job file is created (if necessary). If the job terminates normally and only PURGE checkpoints have been taken, the CP/<JN> directory is removed.

A job can be restarted in two ways:

1. After a Halt/Load. The system automatically attempts to restart any job that was active at the time of a Halt/Load. If a checkpoint has been invoked during the execution of the interrupted task, then the operator is given a message requiring a response to determine whether the job should be restarted. The operator can respond with the Operator Display Terminal (ODT) command OK (to restart at the last checkpoint), DS (to prevent a restart), or QT (to prevent a restart but save the files for later restart if the job was a checkpoint with PURGE).

2. By a Work Flow Language (WFL) RERUN statement. A WFL job can be restarted programmatically by use of the WFL "RERUN" statement.

Pragmatics

The following conditions can inhibit a successful restart:

1. An invalid usercode
2. An OLAYROW value after the checkpoint that is different from the OLAYROW value before the checkpoint
3. Recompilation of the program since the checkpoint
4. An MCP level after the checkpoint that is different from the MCP level before the checkpoint
5. Intrinsics after the checkpoint that are different from the intrinsics before the checkpoint

The following can inhibit a successful checkpoint/restart:

1. Direct I/O (direct arrays or files)
2. Datacomm I/O (open datacomm files)
3. Open Data Management System II (DMSII) sets
4. The task being checkpointed must have no tasks initiated through CALL or PROCESS statements, it must have been initiated by a WFL job, and this WFL job must not have initiated other tasks that are also running.
5. Paper tape I/O
6. ODT files
7. Duplicated files
8. Output directly to a printer or card punch (backup files are acceptable)
9. Running tasks in swap space
10. Checkpoints taken inside sort input or output procedures. The sort intrinsic provides its own restart capability; refer to "SORT Statement."
11. Checkpoints taken in a compile-and-go program

If a job that produces printer backup files is restarted, the backup files can already have been printed and removed, and on restart, the job requests the missing backup files. In this situation, when the backup files are requested, the operator must respond with the ODT command OF (Optional File). A new backup file is created. Output preceding the checkpoint is not re-created.

For jobs that take a large number of checkpoints with LOCK, the checkpoint number counts up to 999 and then recycles to 1 (leaving zero undisturbed). When this recycling occurs, previous checkpoint files are lost as new ones using the same numbers are created.

If a temporary disk file is open at a checkpoint, it is locked under the CP directory. If it is subsequently locked by the program, the name is changed to the current file title. At restart time, the file is sought only under the CP directory, resulting in a no-file condition. To avoid this condition, all files that are to be locked eventually should be opened with the file attribute PROTECTION assigned the value SAVE. (To remove the file, it must be closed with PURGE.) True temporary files, which are never locked, do not have this problem. All data files must be on the same medium as at the checkpoint, but need not be on the same units or the same locations on disk or disk pack. They must retain the same characteristics (blocking, and so forth). The checkpoint/restart system makes no attempt to restore the contents of a file to their state at the time of the checkpoint; the file is merely repositioned. At this time, volume numbers are not verified.

CANDE and Remote Job Entry (RJE) cannot be used to run a program with checkpoints. The checkpoints are ignored if used.

If a rerun is initiated and the job number is in use by another job, a new job number is supplied, and the CP/<JN> directory node is changed to reflect the new job number.

If a rerun is initiated and the PROCESSID function is used, the value returned by the function can be different for the restarted job. Refer to the description of the PROCESSID function in the chapter "Expressions" for more information.

When a job is restarted at some checkpoint before the last, subsequent checkpoints taken from the restarted job continue in numerical sequence from the checkpoint used for the restart. Previous higher numbered checkpoints are lost.

See also
<processid function>. 568

Example

```
BOOL := CHECKPOINT(DISK,PURGE)
```

Checkpoint/Restart Messages

The messages in the following list can appear as the result of a checkpoint/restart.

Checkpoint Message	Completion Code
CHECKPOINT#nn	0
INVALID AREA IN STACK	1
SYSTEM ERROR	2
BAD IPC ENVIRONMENT	3
NO USER DISK FOR CP FILE	4
IO ERROR DURING CHECKPOINT	5
# ROWS IN CP FILE > 1024	6
DIRECT FILE NOT ALLOWED	7
TOO MANY TEMPORARY DISK FILES	8
PAPER TAPE FILE NOT ALLOWED	9
DUPLICATED FILE NOT ALLOWED	10
CON FILE NOT ALLOWED	11
CARD PUNCH FILE NOT ALLOWED	12
OPEN REVERSED TAPE FILE NOT ALLOWED	13
DISKHEADER IN STACK	14
DMS AREA IN STACK	15
DIRECT ARRAY IN STACK	16
DIRECT DOPE VECTOR IN STACK	17
SUBSPACE IN STACK	18

Checkpoint Message	Completion Code
STACKMARK	19
SORT AREA IN STACK	20
REMOTE FILE NOT ALLOWED	21
ILLEGAL CONSTRUCT	22
BDBASE ILLEGAL	23
TEMP FILE ON NAMED PACK	24

The messages in the following list can appear as the result of an attempt to restart.

Restart Messages

RESTART PENDING (RSVP)

MISSING CHECKPOINT FILE

IO ERROR DURING RESTART

USERCODE NO LONGER VALID

OPERATOR DSED RESTART

OPERATOR QTED RESTART

MISSING CODE FILE

NOT ABLE TO RESTART

INVALID JOB FILE

RESTART AS CP/nnnn

MISSING JOB FILE

FILE POSITIONING ERROR

WRONG JOB FILE

WRONG CODE FILE

BAD CHECKPOINT FILE

BAD STACK NUMBER

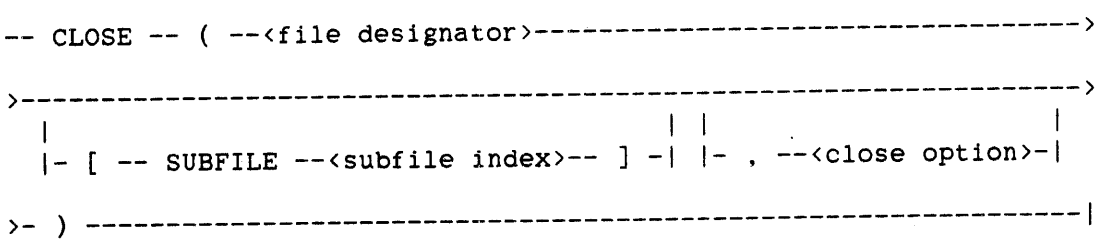
WRONG MCP

CLOSE STATEMENT

The CLOSE statement breaks the link between a logical file declared in the program and its associated physical file, which is the actual file data is sent to or from.

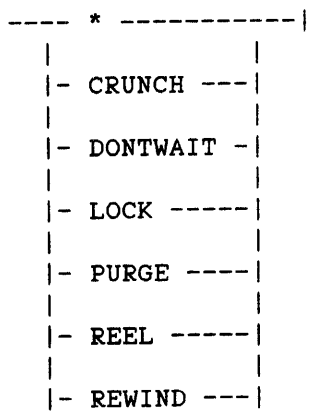
Syntax

<close statement>



<subfile index>

<close option>



See also

<file designator> 189

Semantics

The <subfile index> syntax is used to specify the subfile to be closed.

The CLOSE statement can be used as an arithmetic function, in which case it returns the same values as the file attribute AVAILABLE. The AVAILABLE attribute is described in the "I/O Subsystem Reference Manual."

When no close option is specified, the CLOSE statement closes the file, depending on the kind of file, as follows:

Card Output File

A card containing an ending label is punched. The file must be labeled.

Line Printer File

The printer is skipped to channel 1, an ending label is printed, and the printer is again skipped to channel 1. The file must be labeled.

Unlabeled Tape Output File

A double tape mark is written after the last block on the tape, and the tape is rewound.

Labeled Tape Output File

A tape mark is written after the last block on the tape; then an ending label is written followed by a double tape mark, and the tape is rewound.

Disk File

If the file is a temporary file, the disk space is returned to the system.

For all types of files, the I/O unit and the buffer areas are released to the system.

<close option>

If the asterisk (*) is used and the file is a tape file, the I/O unit remains under program control, and the tape is not rewound. This construct is used to create multifile reels.

When the asterisk is used on multifile input tapes and the value of the LABEL file attribute is STANDARD, the CLOSE statement closes the file as follows:

1. If the value of the DIRECTION file attribute is FORWARD, the tape is positioned forward to a point just following the ending label of the file.

2. If the value of the DIRECTION file attribute is REVERSE, the tape is positioned to a point just in front of the beginning label for the file.

3. If the end-of-file branch of a READ statement or WRITE statement has been taken, the CLOSE statement does not position the file.

The close action performed on a single-file reel is the same as that performed on a multifile reel. The next I/O operation performed on the file must be done in the direction opposite to that of the prior I/O operations; otherwise, an end-of-file error is returned.

When the asterisk is used and the LABEL file attribute does not have the value STANDARD, the tape is spaced beyond the tape mark (on input), or a tape mark is written going forward (on output). The essential difference is that if LABEL is OMITTEDEOF, labels are not spaced over, but if LABEL is STANDARD, labels are spaced over.

The CRUNCH option is meaningful only for disk files. It causes the unused portion of the last row of disk space (beyond the end-of-file indicator) to be returned to the system. The file cannot be expanded but can be written inside of the end-of-file limit.

The DONTWAIT option is meaningful only for files for which the KIND file attribute has the value PORT. Refer to the "I/O Subsystem Reference Manual" for a description of the DONTWAIT option.

If the LOCK option is used, the file is closed. If the file is a tape file, it is rewound, and a system message is printed that notifies the operator that the reel must be saved. The tape unit is made inaccessible to the system until the operator readies it manually. If

the file is a disk file, it is retained as a permanent file on disk. The file buffer areas are returned to the system.

If the PURGE option is used, the file is closed, purged, and released to the system. If the file is a permanent disk file, it is removed from the disk directory, and the disk space is returned to the system.

If the REEL option is used, the file must be a multireel tape file. The current reel is closed, and a subsequent reference to the file implicitly opens the next reel. This option is provided primarily for use with direct tape files, for which the system does not automatically perform reel switching.

If the REWIND option is specified, the file is closed. If the file is a paper tape or magnetic tape file, it is rewound. For disk files, the record pointer is reset to the first record of the file. The file buffer areas are returned to the system, and the I/O unit remains under program control. For paper tape files, the REWIND option can be used only on input.

All forms of the CLOSE statement that are not appropriate for the type of unit assigned to the file are equivalent to using the REWIND option.

Examples

CLOSE(FILEID)

If FILEID is a temporary disk file, this statement closes the file and returns the disk space to the system.

CLOSE(FILEID,*)

Closes FILEID and, assuming FILEID is a tape file, positions the tape according to the description under the heading "<close option>."

CLOSE(FILEID,PURGE)

Closes, purges, and releases FILEID to the system. If FILEID is a permanent disk file, it is removed from the disk directory and the disk space is returned to the system.

CLOSE(FILEID,REEL)

Closes the current reel of FILEID. Assuming FILEID is a multireel tape file, any subsequent reference to FILEID implicitly opens the next reel.

CLOSE(FILEID,CRUNCH)

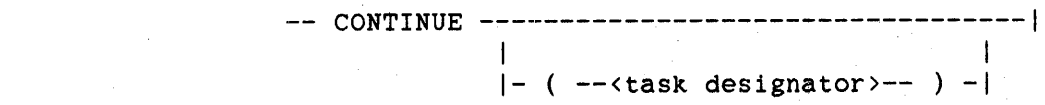
Closes FILEID and, assuming FILEID is a disk file, returns to the system the unused portion of the last row of FILEID.

CONTINUE STATEMENT

The CONTINUE statement causes control to pass from the program in which the statement appears to a coroutine.

Syntax

<continue statement>



See also
<task designator> 200

Semantics

A coroutine is a procedure that is initiated as a separate task by using a CALL statement. The "caller" is referred to as the "primary" coroutine and the called procedure as the "secondary" coroutine.

Because the execution of CONTINUE statements causes control to alternate between primary and secondary coroutines, processing always continues at the point where it last terminated.

The secondary coroutine uses the CONTINUE statement form without the task designator to pass control back to its "partner" task, which is the primary coroutine by default. The task designator is used by the primary coroutine to pass control to the secondary coroutine associated with that task designator by the CALL statement. For more information, refer to "CALL Statement."

Examples

CONTINUE

Passes control from this program, a secondary coroutine, to its "partner" task, which is, by default, the primary coroutine.

CONTINUE(TSK)

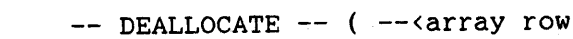
Passes control to the coroutine associated with the task TSK.

DEALLOCATE STATEMENT

The DEALLOCATE statement causes the contents of the specified array row to be discarded and the memory area to be returned to the system.

Syntax

<deallocate statement>



See also
<array row> 43

Pragmatics

When an array row is deallocated, it is made not present (all data is lost). When the array row is used again, it is made present, and each element is re-initialized to zero.

Array rows of paged (segmented) arrays and event arrays cannot be deallocated by using the DEALLOCATE statement.

Examples

DEALLOCATE(ARAY)

Discards the contents of ARAY and returns the memory area to the system. Note that ARAY must be a one-dimensional array or a syntax error results.

DEALLOCATE(MATRIXARY[INDX,*])

Discards the contents of the row MATRIXARY[INDX,*] and returns the memory area to the system.

DETACH STATEMENT

The DETACH statement severs the association of an interrupt with an event.

Syntax

<detach statement>

```
-- DETACH --<interrupt identifier>--|
```

See also
<interrupt identifier>. 126

Semantics

Any pending invocations of a detached interrupt are discarded. Detaching an interrupt that is not attached to an event is essentially a "no-operation"; no error occurs.

The enabled/disabled condition of an interrupt is not changed by a DETACH statement. When an interrupt is attached after it has been detached, the enabled/disabled condition of the interrupt is the same as it was before it was detached. (For more information, refer to "ATTACH Statement," "DISABLE Statement," "ENABLE Statement," and "INTERRUPT Declaration.")

Example

```
DETACH THEPHONE
```

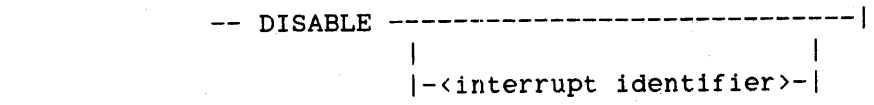
Severs the association between the interrupt THEPHONE and the event it is attached to.

DISABLE STATEMENT

The DISABLE statement prevents interrupt code from being executed.

Syntax

<disable statement>



See also
<interrupt identifier>. 126

Semantics

A DISABLE statement that does not specify an interrupt identifier is referred to as a "general disable." A general disable has the effect of disabling all the interrupts for the task. The interrupts whose associated events are caused are placed in an interrupt queue for the task.

If the DISABLE statement specifies an interrupt identifier, only that interrupt is disabled. The system queues these interrupts until the interrupt is enabled.

Interrupts are queued to ensure that none are lost during the time they are attached. Queuing continues until the appropriate ENABLE statement is executed.

Disabling or enabling an interrupt is not affected by whether or not the interrupt is attached to an event.

For more information, refer to "ATTACH Statement," "DETACH Statement," "ENABLE Statement," and "INTERRUPT Declaration."

Examples

```
DISABLE
```

General disable--disables all interrupts.

```
DISABLE THEPHONE
```

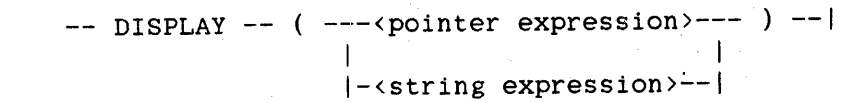
Disables the interrupt named THEPHONE.

DISPLAY STATEMENT

The DISPLAY statement causes the specified message to be displayed on the Operator Display Terminal (ODT) and to be printed in the job summary of the program.

Syntax

<display statement>



Semantics

The message to be displayed is specified by the pointer expression or the string expression. If the parameter to the DISPLAY statement is a pointer expression, execution of the DISPLAY statement causes the characters to which the pointer expression points to be displayed on the ODT. The pointer expression must point to EBCDIC characters, and the message to be displayed must be terminated by a null character (48"00").

If the parameter to the DISPLAY statement is a string expression, execution of the DISPLAY statement causes the contents of the string specified by the string expression to be displayed on the ODT. The string expression must be of type EBCDIC.

Display messages from programs run in CANDE appear on the user's terminal if the MESSAGES option of the CANDE "SO" command has been specified.

A maximum of 430 characters can be displayed.

Examples

```
DISPLAY(POINTER(Q,8))
```

Displays the EBCDIC characters stored in array Q, from the beginning of the array to the EBCDIC null character (48"00").

```
DISPLAY("VALUE IS " CAT STR)
```

Displays the string created by concatenating "VALUE IS " and the string STR.

```
DISPLAY(MESSAGESTRING)
```

Displays the string stored in the string variable MESSAGESTRING.

DO STATEMENT

The DO statement causes a statement to be executed until a specified condition is met.

Syntax

<do statement>

```
-- DO --<statement>-- UNTIL --<Boolean expression>--|
```

See also
<statement> 219

Semantics

The statement following "DO" is executed. The Boolean expression is evaluated, and if it is FALSE, the statement is executed again and the Boolean expression is re-evaluated. This sequence of operations continues until the value of the Boolean expression is TRUE. At that time, control passes to the statement following the DO statement.

Note that both <block> and <compound statement> are statements and can be substituted for <statement>.

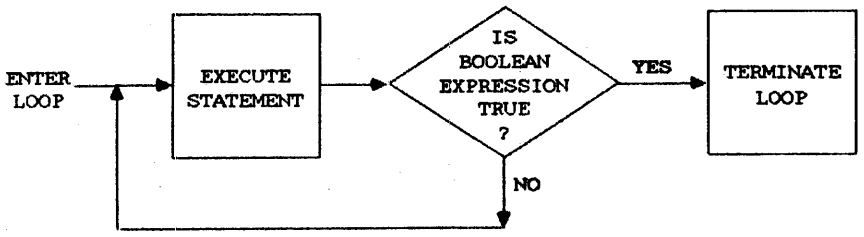


Figure 5-1. DO-UNTIL Loop

Examples

```
DO
   BEGIN
   PTR := *-4;
   CTR := *+4;
   END
UNTIL PTR IN LOOKEDFOR
```

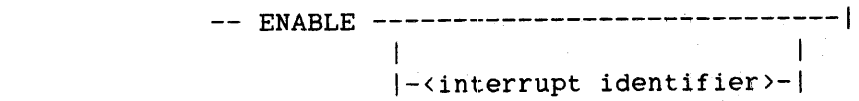
```
DO
   J := J/2
UNTIL BUF[J] < JOB
```

ENABLE STATEMENT

The ENABLE statement allows interrupt code to be executed.

Syntax

<enable statement>



See also
<interrupt identifier>. 126

Semantics

Previously disabled interrupts can be enabled with the ENABLE statement. If the event associated with the interrupt is caused after an interrupt has been enabled, then the interrupt code is executed.

An ENABLE statement that does not specify an interrupt identifier is referred to as a "general enable" and causes the system to look for, and place in execution, all interrupts that are in the interrupt queue of the task.

If the ENABLE statement specifies an interrupt identifier, only that interrupt is enabled. All occurrences of the interrupt in the interrupt queue are placed in execution.

Disabling or enabling an interrupt is not affected by whether or not the interrupt is attached to an event.

For more information, refer to "ATTACH Statement," "DETACH Statement," "DISABLE Statement," and "INTERRUPT Declaration."

Examples

ENABLE

General enable--enables all previously disabled interrupts.

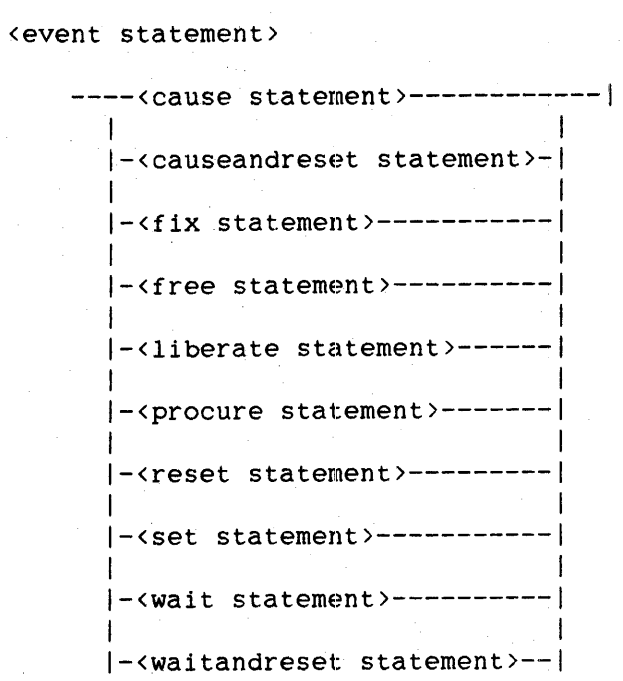
ENABLE THEPHONE

Enables the interrupt named THEPHONE.

EVENT STATEMENT

Events have two Boolean characteristics, happened and available. Each characteristic can be in one of two states: TRUE or FALSE. These states can be changed using event statements.

Syntax



Semantics

The happened and available states of an event can be interrogated using the HAPPENED function and the AVAILABLE function.

See also
<available function>. 535
<happened function> 555

EXCHANGE STATEMENT

The EXCHANGE statement is used to exchange rows between two disk files.

Syntax

<exchange statement>

```
-- EXCHANGE -- ( --<file designator>-- [ --<row/copy numbers>-- ] ->
>- , --<file designator>-- [ --<row/copy numbers>-- ] -- ) --------|
```

<row/copy numbers>

```
--<row number>--------------------|
               |                  |
               |-<copy number>-|
```

<row number>

```
--<arithmetic expression>--|
```

<copy number>

```
--<arithmetic expression>--|
```

See also
<file designator> 189

Semantics

Row numbers begin with zero, and copy numbers begin with 1. If there are copies of the file and a copy number is specified, then only the rows of that copy are exchanged.

Pragmatics

The two files must be closed when the EXCHANGE statement is executed, the two rows must be the same size, the specified row numbers and the specified copy numbers must be valid, and the two files cannot be code files of any kind.

For the exchange to take place, the referenced files must be closed with retention. For more information, refer to "CLOSE Statement."

If the system detects an error, the exchange is not performed, and the program resumes execution with the next statement. After using the EXCHANGE statement, the row addresses should be checked by using file attributes to ensure that the exchange was successfully completed.

Examples

```
EXCHANGE(FILE1[ROW6],FILE2[ROW0])
```

Exchanges the contents of row ROW6 of FILE1 with the contents of row ROW0 of FILE2.

```
EXCHANGE(MASTERFYLE[I],REBUILTFYLE[J])
```

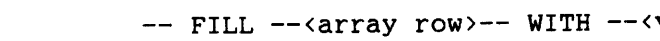
Exchanges row I of MASTERFYLE with row J of REBUILTFYLE.

FILL STATEMENT

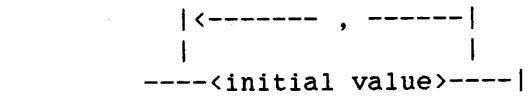
The FILL statement fills an array row with specified values.

Syntax

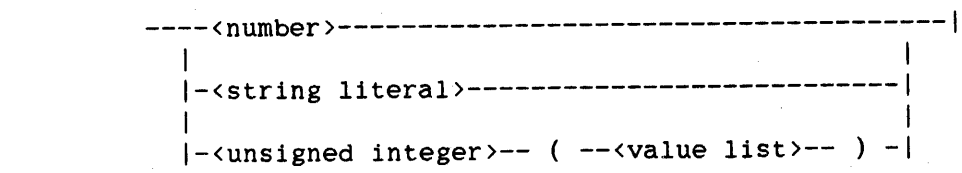
<fill statement>



<value list>



<initial value>



See also
<array row> 43

Semantics

The FILL statement cannot be used with character arrays.

Each initial value initializes an integral number of words. The number of words initialized depends on the type of the array and the kind of initial value.

Single-precision numbers initialize one word in arrays other than double or complex arrays. In double arrays, this word is extended with a second word of zero. In complex arrays, this word is normalized and then extended with an imaginary part of zero.

Double-precision numbers are stored unchanged in two words in double arrays. In complex arrays, the value is rounded and normalized to single precision and then extended with an imaginary part of zero. For other types of arrays, the second word of the double-precision value is dropped and the first word initializes one word of the array.

String literals more than 48 bits long initialize as many words as are needed to contain the string and are left-justified with trailing zeros inserted in the last word, if necessary. In complex and double arrays, long string literals can initialize an odd number of words, causing the following initial value to start in the middle of a two-word element of the array.

String literals less than or equal to 48 bits long are right-justified within one word with leading zeros, if necessary. This word initializes one word in arrays other than double or complex arrays. In double arrays, this word is extended with a second word of zero. In complex arrays, this word is normalized and then extended with an imaginary part of zero.

An initial value of the form "<unsigned integer> (<value list>)" causes the values in the value list to be repeated the number of times specified by the unsigned integer.

If the value list contains more values than will fit in the array row, filling stops when the array row is full.

If the value list contains fewer values than the array row can hold, the remainder of the array row is left unchanged.

The length of the value list cannot exceed 4095 48-bit words.

Examples

```
FILL MATRIX[*] WITH 250(0)
```

Fills the first 250 words of the one-dimensional array MATRIX with zeros.

```
FILL GROUP[1,*] WITH .25, "ALGOL", """", "LONGER STRING"
```

Fills the designated row of array GROUP with the value .25, the string ALGOL right-justified with leading zeroes, the character " right-justified with leading zeros, and with the string LONGER STRING, which fills two words and part of a third word. Trailing zeros fill the rest of the third word.

FIX STATEMENT

The FIX statement examines the available state of an event. After the FIX statement executes, the available state of the designated event is always FALSE (not available).

Syntax

<fix statement>

```
-- FIX -- ( --<event designator>-- ) --|
```

See also
<event designator>. 78

Semantics

The FIX statement can be used as a Boolean function. If the available state of the specified event is TRUE (available), the event is procured, the state is set to FALSE (not available), and FALSE is returned as the value of the function. If the available state of the specified event is FALSE (not available), the FIX statement returns TRUE, and the available state is left unchanged.

The FIX statement is sometimes referred to as the "conditional procure function."

When the FIX statement has finished execution, the available state of the event is FALSE (not available).

Examples

FIX(EVNT)

Examines the available state of the event EVNT.

FIX(EVENTARRAY[INDEX])

Examines the available state of the event designated by EVENTARRAY[INDEX].

```
IF GOTIT := FIX(FILELOCK) THEN...
```

Examines the available state of event FILELOCK and stores in GOTIT a value indicating this state.

```
FIX(MYSELF.EXCEPTIONEVENT)
```

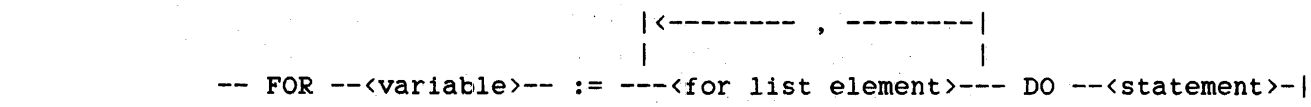
Examines the available state of the task's EXCEPTIONEVENT.

FOR STATEMENT

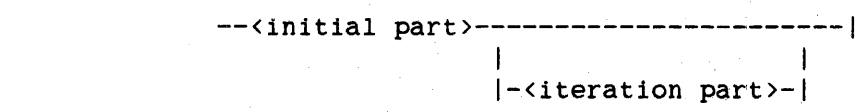
The FOR statement constructs a loop consisting of one or more statements that are executed a specified number of times.

Syntax

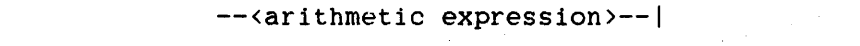
<for statement>



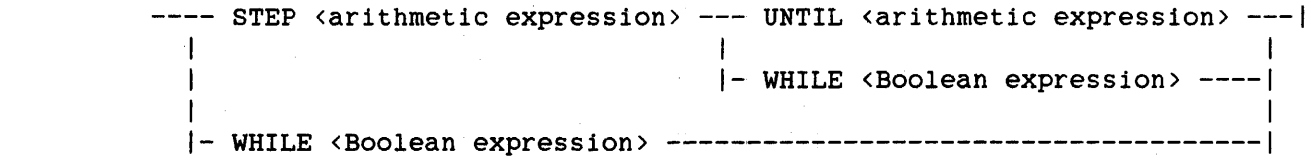
<for list element>



<initial part>



<iteration part>



See also

<statement> 219
<variable>. 225

Semantics

The number of times a FOR loop is traversed is determined by a variable, called the "control variable," which is initialized when the FOR statement is first entered, and which can be updated during each iteration of the loop.

The action of a FOR statement can be described by isolating the following three distinct steps:

1. Assignment of a value to the control variable
2. Test of the limiting condition
3. Execution of the statement following "DO"

Each type of ‹for list element› syntax specifies a different process. However, all of these processes have one property in common: the initial value assigned to the control variable is that of the arithmetic expression in the ‹initial part› construct.

The ‹for list element› construct establishes which values are assigned to the control variable and which test to make of the control variable to determine whether or not the statement following "DO" is executed. When a for-list element is exhausted, the next for-list element, if any, is evaluated, progressing from left to right. When all for-list elements have been used, the FOR statement is considered completed, and execution continues with the statement following the FOR statement. The statement following "DO" can transfer control outside the FOR statement, in which case some for-list elements may not have been exhausted before the FOR statement is exited.

In the discussion below of the various forms of the FOR statement, V is the control variable; AEXP1, AEXP2,... are arithmetic expressions; BEXP is a Boolean expression; and S1 is a statement.

FOR-DO Loop

If a for-list element consists of only an initial part, such as

```
FOR V := AEXP1, AEXP2, ... DO
```

then that for-list element designates only one value to be assigned to the control variable. Because no limiting condition is present, no test is made. After assignment of the arithmetic expression to the control variable, the statement following "DO" is executed, and the for-list element is considered exhausted.

Figure 5-2 illustrates the FOR-DO loop.

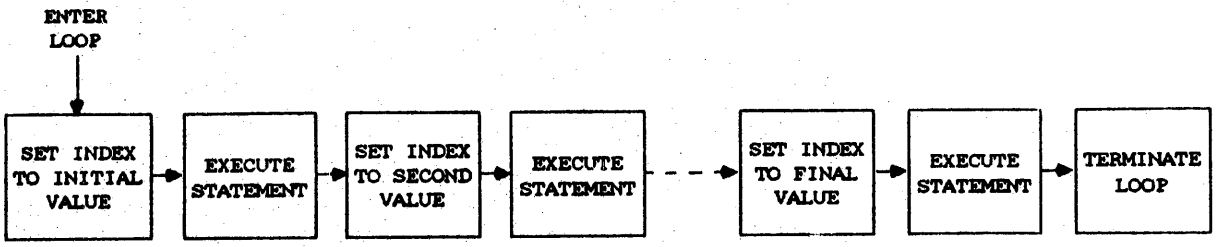


Figure 5-2. FOR-DO Loop

FOR-STEP-UNTIL Loop

If a for-list element is of the form "<initial part> STEP <arithmetic expression> UNTIL <arithmetic expression>", such as

```
FOR V := AEXP1 STEP AEXP2 UNTIL AEXP3 DO
```

then a new value is assigned to the control variable V before each execution of the statement following "DO". First, the initial value, that of AEXP1, is assigned to the control variable. After each execution of the statement following "DO", the assignment "V := V + AEXP2" is performed. Both AEXP2 and AEXP3 are re-evaluated each time through the loop.

A test is made immediately after each assignment of a value to V (including the first) to determine whether or not the value of V has "passed" the value of AEXP3. Whether AEXP3 is an upper or a lower limit depends on the sign of AEXP2: AEXP3 is an upper limit if AEXP2 is positive and a lower limit if AEXP2 is negative. If AEXP3 is an upper limit, then V has passed AEXP3 when the expression "V LEQ AEXP3" is no longer TRUE. If AEXP3 is a lower limit, then V has passed AEXP3 when the expression "V GEQ AEXP3" is no longer TRUE. If V has not passed AEXP3, the statement following "DO" is executed; otherwise, the for-list element is exhausted. Figure 5-3 illustrates the FOR-STEP-UNTIL loop.

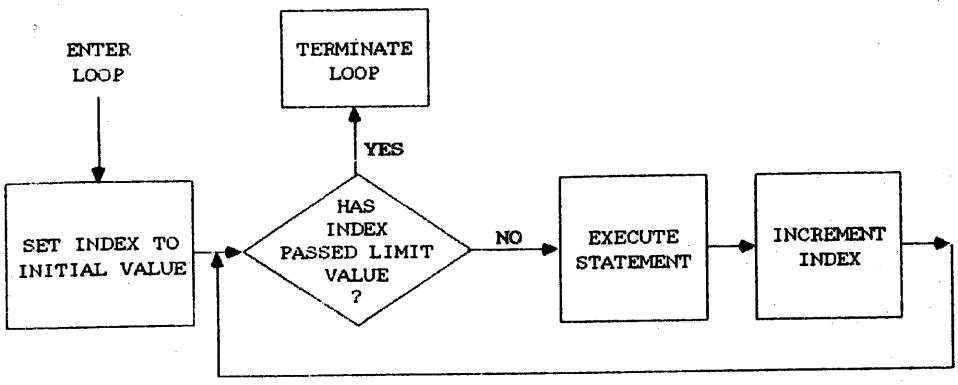


Figure 5-3. FOR-STEP-UNTIL Loop

FOR-STEP-WHILE Loop

If a for-list element is of the form "<initial part> STEP <arithmetic expression> WHILE <Boolean expression>", such as

```
FOR V := AEXP1 STEP AEXP2 WHILE BEXP DO
```

then a new value is assigned to the control variable V before each execution of the statement following "DO". First, the initial value, that of AEXP1, is assigned to the control variable. After each execution of the statement following "DO", the assignment "V := V + AEXP2" is performed. AEXP2 is re-evaluated each time through the loop. After each assignment to V, the Boolean expression BEXP is evaluated and, if BEXP is TRUE, the statement following "DO" is executed. If BEXP is FALSE, this for-list element is exhausted. Figure 5-4 illustrates the FOR-STEP-WHILE loop.

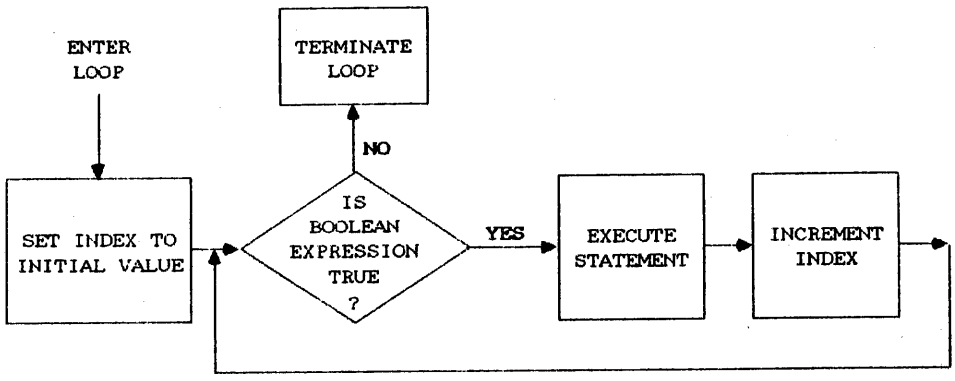


Figure 5-4. FOR-STEP-WHILE Loop

FOR-WHILE Loop

If the for-list element is of the form "<initial part> WHILE <Boolean expression>", such as

```
FOR V := AEXP1 WHILE BEXP DO
```

then the control variable V is assigned the value of AEXP1 before each execution of the statement following "DO". AEXP1 is re-evaluated for each assignment to V. After each assignment to V, the Boolean expression BEXP is evaluated. If the value of BEXP is TRUE, the statement following "DO" is executed. If the value of BEXP is FALSE, this for-list element is exhausted. For example, in the FOR statement

```
FOR V := V + 1 WHILE V LEQ 5 DO
   S1;
```

if V had the value zero before execution of this statement, S1 would be executed five times.

Figure 5-5 illustrates the FOR-WHILE loop.

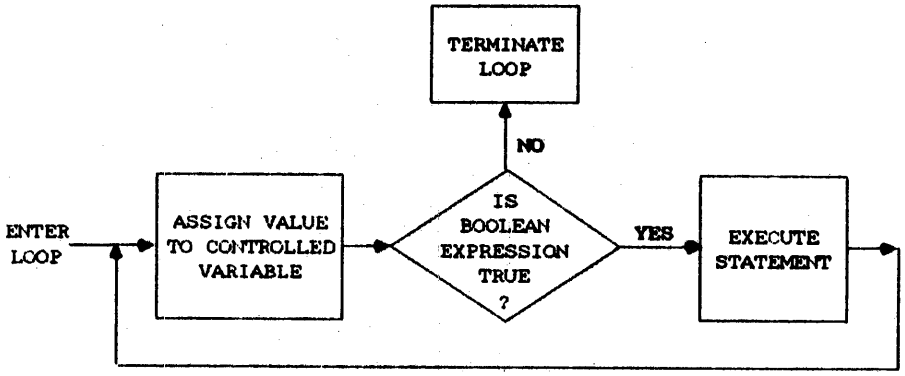


Figure 5-5. FOR-WHILE Loop

Examples

```
FOR I := 0 DO
```

Executes the statement following "DO" just once, with I assigned zero.

```
FOR J := 0 STEP 1 UNTIL 255 DO
   LOOKEDFOR[J] := 1
```

Assigns 1 to elements 0 through 255 of array LOOKEDFOR.

```
FOR INDEX := 0, 1, 2, 10, 15, 37, 5, 16 DO
   BUF[INDEX] := ITEM
```

Assigns ITEM to elements 0, 1, 2, 5, 10, 15, 16, and 37 of array BUF.

```
FOR X := 0 STEP 1 UNTIL 5, 29, 47 STEP 3 UNTIL LIM DO
   FETCH(X)
```

Calls FETCH repeatedly, passing the values 0, 1, 2, 3, 4, 5, 29, and the values of (47 + 3 * X) where X = 0, 1, 2, and so on, as long as (47 + 3 * X) is less than LIM.

```
FOR NEXT := BEG STEP AMT WHILE NOT DONE DO
   PANHANDLE
```

Calls PANHANDLE and assigns to NEXT values equal to BEG, BEG + AMT, BEG + 2*AMT, and so on, as long as DONE is FALSE.

```
FOR N := IX + 7 WHILE TARGET LEQ RANGE DO
   TARGET := * + N
```

Increments TARGET by the value IX + 7 as long as TARGET is less than or equal to RANGE.

FREE STATEMENT

The FREE statement sets the available state of the specified event to TRUE (available).

Syntax

<free statement>

```
-- FREE -- ( --<event designator>-- ) --|
```

See also
<event designator>. 78

Semantics

The FREE statement can be used as a Boolean function that returns FALSE if the available state of the event is already TRUE (available) and TRUE if the available state of the event is FALSE (not available). In either case, the available state of the event is unconditionally set to TRUE (available).

The FREE statement does not activate any task attempting to procure the event, nor does it activate any task waiting on the event.

Examples

FREE(EVNT)

Sets the available state of the event EVNT to TRUE (available).

FREE(EVNTARAY[INDX])

Sets the available state of the event designated by EVNTARAY[INDX] to TRUE (available).

IF WASPROCURED := FREE(FYLELOCK) THEN ...

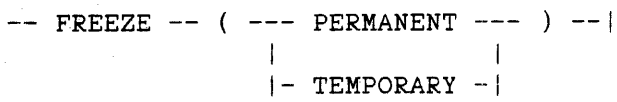
Assigns to WASPROCURED a value indicating the available state of the event FYLELOCK, and sets the available state of FYLELOCK to TRUE (available).

FREEZE STATEMENT

The FREEZE statement changes the running program into a library.

Syntax

<freeze statement>



Semantics

At least one EXPORT declaration must appear in the same block as the FREEZE statement. The procedures affected by a FREEZE statement are the procedures that appear in EXPORT declarations in the same block as the FREEZE statement. After the FREEZE statement is executed, these procedures are library entry points.

The PERMANENT and TEMPORARY specifications of the FREEZE statement control the permanence of the library. A permanent library remains available until it is discontinued. A temporary library remains available as long as there are users of the library. A temporary library that is no longer in use unfreezes (thaws) and resumes running as a regular program. When a library unfreezes, it cannot execute another FREEZE statement in an attempt to become a library again.

Because a library program initially runs as a regular program, the flow of execution can be such that the execution of a FREEZE statement is conditional and can occur anywhere in the outer block of the program.

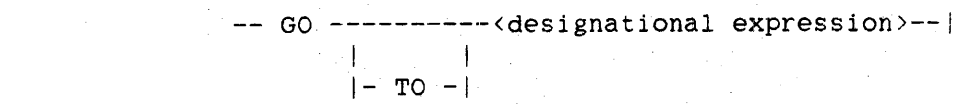
If a calling program causes a library to be initiated and this library does not execute a FREEZE statement (if, for example, it was not a library program and thus had no FREEZE statement), then the attempted linkage to the library entry points cannot be made, and the calling program is discontinued. For more information on libraries, refer to the "Interface to the Library Facility" chapter.

GO TO STATEMENT

The GO TO statement transfers control to the statement in the program with the specified label.

Syntax

<go to statement>



Semantics

The value of the designational expression specifies the label to which control is transferred.

Because labels must be declared in the innermost block in which they occur as statement labels, a GO TO statement cannot lead from outside a block to a point inside that block. Each block must be entered at the BEGIN so that the declarations associated with that block are invoked. For more information on labels, refer to "LABEL Declaration."

Bad Go To

A "bad go to" occurs when a GO TO statement in an inner block transfers control to a label that is global to that block. A necessary side effect of a "bad go to" is that the block in which it occurs is exited abruptly and local variables are deallocated immediately.

A "bad go to" requires cutting back the lexical (lex) level to a more global block. To perform a "bad go to," the Master Control Program (MCP) is invoked to cut back the stack and discard any locally declared items that occupy memory space outside of the stack (sometimes referred to as "nonstack items"), such as files, arrays, and interrupts.

Examples

```
GO TO LABEL1
```

Control is transferred to the statement with the label LABEL1.

```
GO LABEL2
```

Control is transferred to the statement with the label LABEL2.

```
GO TO SELECTIT[INDX]
```

Control is transferred to the statement with the label designated by the subscripted switch label identifier SELECTIT[INDX].

```
GO TO IF K=1 THEN SELECT[2] ELSE START
```

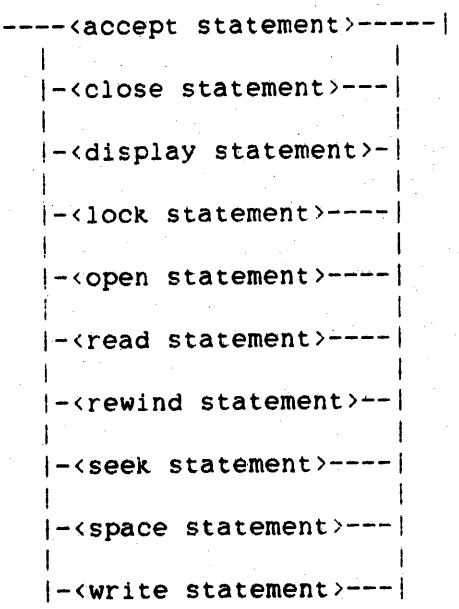
If K is equal to 1, control is transferred to the statement with the label designated by the subscripted switch label identifier SELECT[2]. Otherwise control is transferred to the statement with the label START.

I/O STATEMENT

An I/O statement causes information to be exchanged between a program and a peripheral device, and allows the programmer to perform certain control functions.

Syntax

<I/O statement>



Semantics

ALGOL input/output (I/O) is handled by a part of the Master Control Program (MCP) called the I/O subsystem, a thorough description of which is beyond the scope of this manual. Refer to the "I/O Subsystem Reference Manual" for specific information.

The ACCEPT statement and DISPLAY statement are unique in that the file to or from which data is transferred need not be specified. For more information, refer to "ACCEPT Statement" and "DISPLAY Statement."

The remaining I/O statements reference a file that must be declared by the programmer. For more information, refer to "FILE Declaration."

Two distinct methods of I/O are available. The first and typical method is referred to as "normal I/O"; the second method is called "direct I/O." The major differences between normal I/O and direct I/O have to do with "buffering," the overlap of program execution, and the overlap of I/O operations. These two I/O methods are described in general below. Their effect on a particular I/O statement is presented in the description of the statement.

Normal I/O

Normal I/O is indicated when direct files and direct arrays are not used. Normal I/O includes many automatic facilities provided by the MCP, such as the following:

1. Buffering: the automatic overlap of program processing and I/O traffic to and from the peripheral units

2. Blocking: more than one logical record per physical block

3. Translation as needed between the character set of the unit and that required by the program

The amount of buffering between the I/O statements and program execution depends on the number of buffers allocated for the file. Refer to "FILE Declaration" for information on how to specify the number of buffers.

In normal I/O, a READ statement causes the automatic testing of the availability of the needed record. The program is suspended in the READ statement until the record is actually available for use.

In normal I/O, a WRITE statement transfers the specified data to a buffer; the program is immediately released to begin execution of the next statement. If all the buffers are full when the WRITE statement is executed, the program is suspended until a buffer is available.

Direct I/O

Direct I/O is indicated when direct files and direct arrays are used.

Direct I/O allows more direct control of the actual I/O operations. In certain situations, avoiding suspension of the program is desirable. In other situations, nonstandard I/O operations (and masking of certain types of error conditions which could arise) is desirable.

When direct I/O is used, buffering, blocking, and translation are the responsibility of the programmer.

The syntax for a direct read or direct write operation employs the "<arithmetic expression>, <array row>" form of <format and list part>. An event designator is the only allowable form of <action labels or finished event> for direct I/O. The value of the arithmetic expression has the following meaning: field [16:17] contains the number of words to be transferred, and field [19:3] contains the number of trailing characters to be transferred. The array row is called the I/O area of the user. A direct array identifier must be used for the <array name> part in the array row construct. Thus, the statement

```
READ(FID, 10, A[*]) [EVT]
```

could be used to perform a direct read of 10 words from file FID into direct array A using the event EVT as the finished event.

The MCP establishes a relationship between the I/O area and the finished event, if one is specified. Before any subsequent use of the I/O area can be made in the program, either for calculations or for further I/O, the direct I/O operation must be finished. The finished event can be inspected (1) by using the HAPPENED function, (2) by obtaining the value of the STATE file attribute using the WAIT statement as a Boolean function and specifying a direct array row as a parameter, or (3) by using the WAIT statement on the event to deactivate the process until the event is caused. Once the operation has been completed, the happened state of the event should be set to FALSE (not happened) before reusing it. Refer to "WAIT Statement" for more information.

The finished event can be associated with a direct array row that is declared in a different block. For example, a formal event can be associated with a local array. Such an association can cause compile-time or run-time up-level event errors if the block containing the finished event can be exited before the block that contains the direct array is exited.

In direct I/O, the I/O operations analogous to the SPACE and REWIND statements are performed as if they were read or write operations, except that the IOCW direct array attribute is specifically assigned the proper hardware instructions for the operation.

When performing direct I/O with the SPACE operation, the device's spacing limitation overrides any user-specified spacing. In the case of a line printer, this limitation is two.

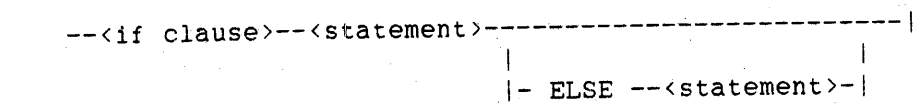
See also
<happened function> 555

IF STATEMENT

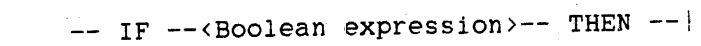
The IF statement causes a statement to be executed or not executed based on the value of a Boolean expression.

Syntax

<if statement>



<if clause>



See also
<statement> 219

Semantics

In the descriptions that follow, BEXP represents any Boolean expression, and S1, S2, and S3 represent statements.

<if clause> <statement>

When the IF statement is of this form, such as

```
IF BEXP THEN S1
```

then if the value of the Boolean expression BEXP is TRUE, the statement S1 is executed. If BEXP is FALSE, then S1 is not executed. In either case, execution continues with the statement following the IF statement.

<if clause> <statement> ELSE <statement>

When the IF statement is of this form, such as

```
IF BEXP THEN S1 ELSE S2
```

then if the value of BEXP is TRUE, the statement S1 is executed and the statement S2 is ignored. If the value of BEXP is FALSE, then the statement S2 is executed, and S1 is ignored. In either case, execution continues with the statement following the IF statement.

Further Information

Note that both <block> and <compound statement> are statements and can be substituted for <statement>.

IF statements can be nested; that is, the statements following the reserved words THEN or ELSE (or both) can also be IF statements.

When IF statements are nested, the correct correspondence between the reserved words THEN and ELSE must be maintained. The compiler matches the innermost THEN to the first ELSE that follows it and that yields a syntactically correct IF statement. Consider the following IF statement:

```
IF BEXP1 THEN IF BEXP2 THEN S2 ELSE S1
```

The ELSE is paired with the innermost THEN, which is the THEN following BEXP2, as illustrated below.

```
IF BEXP1 THEN
   IF BEXP2 THEN
      S2
   ELSE
      S1
```

If it is desired to pair the ELSE with the THEN following BEXP1, the inner IF statement must be made a compound statement by using BEGIN and END as follows:

```
IF BEXP1 THEN
   BEGIN
   IF BEXP2 THEN
      S2
   END
ELSE
   S1
```

A GO TO statement can lead to a labeled statement within an IF statement. The subsequent action is equivalent to the action that would result if the IF statement was entered at the beginning and evaluation of the Boolean expression caused execution of the labeled statement.

Examples

```
IF ALLDONE THEN
   GO AWAY
```

If ALLDONE is TRUE, control is transferred to the statement with the label AWAY. If ALLDONE is FALSE, the statement following the IF statement is executed.

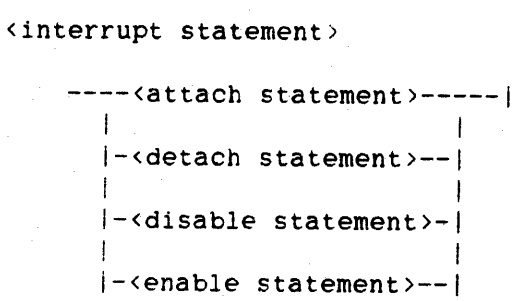
```
IF X > LIMIT THEN
   ERROR
ELSE
   X := * + 1
```

If the value of X is greater than the value of LIMIT, procedure ERROR is called. If the value of X is less than or equal to the value of LIMIT, the value of X is incremented by 1. In either case, execution continues with the statement following the IF statement.

INTERRUPT STATEMENT

Interrupts provide a way to interrupt a process when a specific event occurs. Interrupt statements allow interrupts to be attached to and detached from events, and allow interrupts to be enabled and disabled.

Syntax



Semantics

The ATTACH statement is used to associate an interrupt with an event.

The DETACH statement is used to sever the association between an interrupt and the event to which it is attached.

The ENABLE statement and DISABLE statement are used to explicitly enable and disable, respectively, an interrupt.

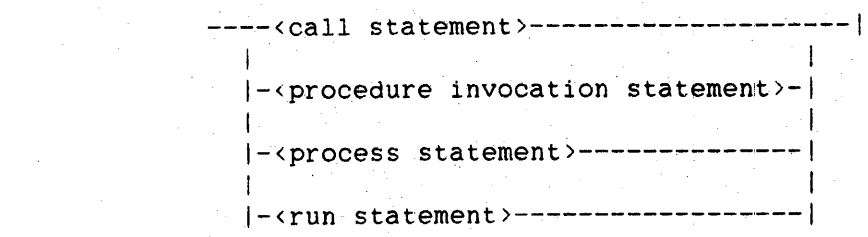
For more information on interrupts, refer to "INTERRUPT Declaration."

INVOCATION STATEMENT

An invocation statement causes a previously declared procedure to be executed as a subroutine, an asynchronous process, a coroutine, or an independent program.

Syntax

<invocation statement>



Semantics

The CALL statement invokes a procedure to execute as a coroutine. The procedure invocation statement invokes a procedure to execute as a subroutine. The PROCESS statement invokes a procedure to run as an asynchronous process. The RUN statement invokes a procedure to run as an independent program.

With the exception of the procedure invocation statement, a separate stack is initiated, and the specified procedure cannot be a typed procedure.

With the exception of the RUN statement, parameters can be call-by-name or call-by-value. All parameters passed in the RUN statement must be call-by-value.

LIBERATE STATEMENT

The LIBERATE statement activates all tasks waiting on the specified event. It can also change the happened state of the event to TRUE (happened).

Syntax

<liberate statement>

```
-- LIBERATE -- ( --<event designator>-- ) --|
```

See also
<event designator>. 78

Semantics

The LIBERATE statement causes the execution of an implicit CAUSE statement for the specified event. This implicit CAUSE statement can result in a change to the happened state of the event, if the waiting tasks have used the WAITANDRESET statement. (For more information, refer to "CAUSE Statement" and "WAITANDRESET Statement.") The available state of the event is set to TRUE (available).

Pragmatics

Although all waiting tasks are activated, they are linked into the ready queue in priority order. At that point, all tasks that were waiting to procure the event are in the ready queue in priority order. (For more information about procuring events, refer to "PROCURE Statement.")

Examples

```
LIBERATE(ANEVENT)
```

Causes the event ANEVENT and sets its available state to TRUE (available).

```
LIBERATE(EVENTARRAY[INDEX])
```

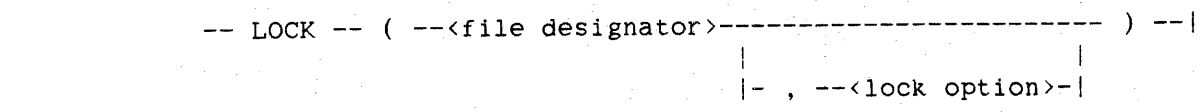
Causes the event designated by EVENTARRAY[INDEX] and sets its available state to TRUE (available).

LOCK STATEMENT

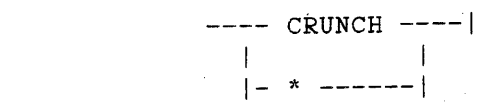
The LOCK statement causes the specified file to be closed.

Syntax

<lock statement>



<lock option>



See also
<file designator> 189

Semantics

If the specified file is a tape file, it is rewound, and the tape unit is made inaccessible to the system until the operator readies it again. If the file is a disk file, it is retained as a permanent file on disk. The file buffer areas are returned to the system.

A LOCK statement with a lock option performs the same action as a CLOSE statement that specifies CRUNCH. Whether CRUNCH or an asterisk (*) appears as the lock option, the action of the LOCK statement is the same. The file must be a disk file. The unused portion of the last row of disk space (beyond the end-of-file indicator) is returned to the system. The disk file can no longer be expanded without being copied into a new file; however, data can be written to existing records.

Examples

```
LOCK(FILEA)
```

If FILEA is a disk file, it is retained as a permanent file.

```
LOCK(FYLE,CRUNCH)
```

The unused portion of the last row of disk file FYLE is returned to the system.

```
LOCK(FYLE,*)
```

The unused portion of the last row of disk file FYLE is returned to the system.

MERGE STATEMENT

The MERGE statement causes data in the specified files to be combined and returned.

Syntax

<merge statement>

```
-- MERGE -- ( --<output option>-- , --<compare procedure>-- , ----->

>-<record length>-- , --<merging option list>-- ) -----------------|
```

<merging option list>

```
  |<------- , -------|
  |                   |
----<merging option>----|
```

<merging option>

```
--<input option>--|
```

See also

Semantics

The compare procedure determines the manner in which the data is combined. The output option specifies how the data is to be returned from the merge operation.

The merging option list must contain between two and eight input options, inclusive, which must be files or Boolean procedures.

Example

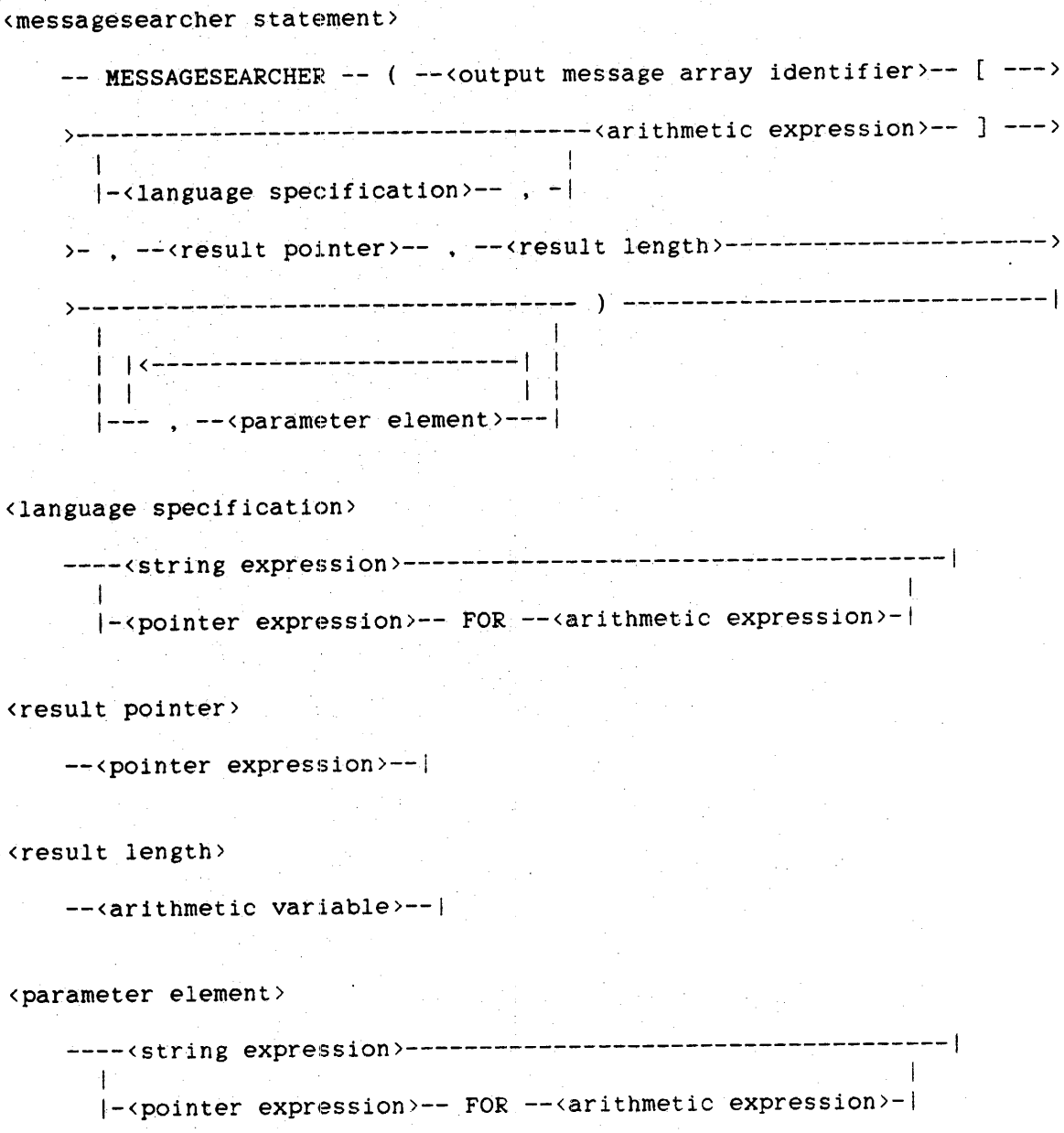
```
MERGE(LINEOUT,COMP,14,IN1,IN2)
```

Merges records from files IN1 and IN2 according to a scheme given in compare procedure COMP. The merged result is written to file LINEOUT. The records of IN1 and IN2 have a maximum record size of 14.

MESSAGESEARCHER STATEMENT

The MESSAGESEARCHER statement returns a completed output message based on the information passed to it.

Syntax



See also

Semantics

The output message array identifier indicates the output message array from which the output message is to be obtained.

The language specification indicates the preferred language for the requested output message.

The arithmetic expression within the square brackets ([]) indicates the output message number of the message that is to be completed and cannot be a double-precision value.

The result pointer is a call-by-value EBCDIC pointer that points to where the completed output message is to be stored. An EBCDIC null character (48"00") is placed after the last character of the message. The null character is not included in the returned message length.

The result length is an integer or real variable that is assigned the length of the returned output message, not counting the null character that is appended at the end.

Each parameter element contains the actual value of a parameter that was specified in the declaration of the requested output message. The first parameter element refers to parameter <1>, the second to parameter <2>, and so on.

The following method is used to find the requested message so that it can be completed.

First, an initial language in which to search for the message must be selected. If a language specification is given as a parameter to the MESSAGESEARCHER statement, that language is selected; otherwise, the language in the language specification of the task requesting the message is used. If the task does not have a language specification, the system default language is used.

If the requested message cannot be found in the initial language and the initial language is not the system default language, the message is searched for in the system default language. If the message still

cannot be found, then the message is searched for in the languages that exist in the specified output message array, beginning with the first language, the second language, and so on. If none of the languages in the output message array contains the message, an error message that specifies the message number is produced in place of the message.

The MESSAGESEARCHER statement can be used as an arithmetic function that returns an integer result indicating whether or not the message was successfully found and formatted. The possible values for this result are as follows:

1	The message is not in the requested language; it is in MYSELF.LANGUAGE or SYSTEMLANGUAGE.
0	The message was found and formatted as requested.
-1	Too few parameters were specified.
-2	No matching <output message case part> was found.
-3	The message is in the first available language.
-4	The array row referenced by the result pointer is too small.
-5	The message was not found.
-6	The version of the output message array is incompatible with the version of the Master Control Program (MCP).
-7	The output message array is in error.
-8	A fault occurred while obtaining the output message.
-9	The length passed with a parameter is too long.

For more information, refer to "OUTPUTMESSAGE ARRAY Declaration."

MULTIPLE ATTRIBUTE ASSIGNMENT STATEMENT

The multiple attribute assignment statement is used to assign values at run time to one or more attributes of a specified file.

Syntax

<multiple attribute assignment statement>

```
--<file identifier>-- ( --<attribute specifications>-- ) --|
```

See also
<attribute specifications> 85
<file identifier> 85

Semantics

If the name of a Boolean file attribute in the attribute specifications is not followed by an equal sign (=) and a value, it is assigned a value of TRUE; that is, the attribute specifications

```
DEPENDENTSPECS,KIND = DISK
```

have the same effect as the attribute specifications

```
DEPENDENTSPECS = TRUE,KIND = DISK
```

An assignment specified in a multiple attribute assignment statement occurs at run time and overrides any assignment made to the attribute in a FILE declaration or through file equation.

Pragmatics

One intrinsic call is generated to assign all attributes, except when a pointer-valued file attribute name is assigned a pointer expression. In this case, the compiler generates a separate intrinsic call for the pointer-valued attribute assignment.

Examples

```
AFILE(BUFFERS = 3,INTMODE = EBCDIC,KIND = DISK)
```

At run time, the BUFFERS attribute of file AFILE is assigned the value 3, the INTMODE attribute is assigned EBCDIC, and the KIND attribute is set to DISK.

```
LINE(TITLE = P,INTNAME = Q)
```

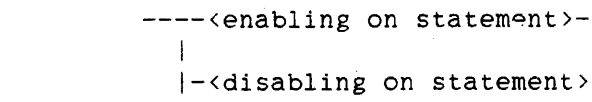
At run time, the TITLE attribute of file LINE is assigned the value pointed to by pointer P, and the INTNAME attribute is assigned the value pointed to by pointer Q.

ON STATEMENT

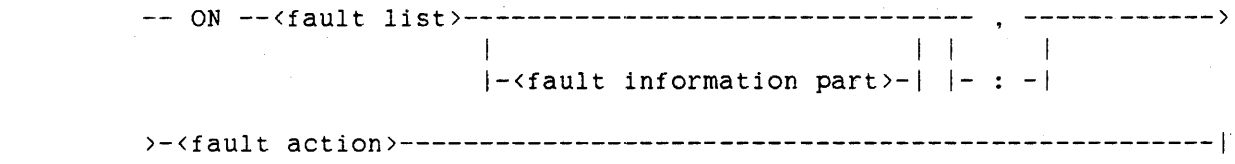
The ON statement is used to enable or disable an interrupt for one or more fault conditions.

Syntax

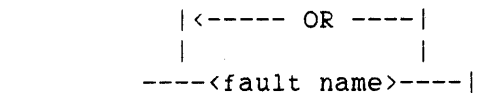
<on statement>



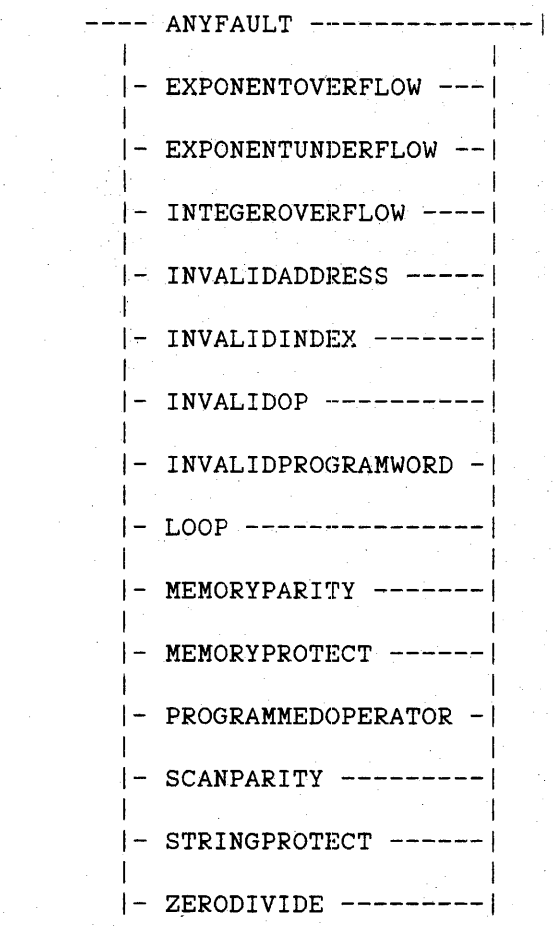
<enabling on statement>



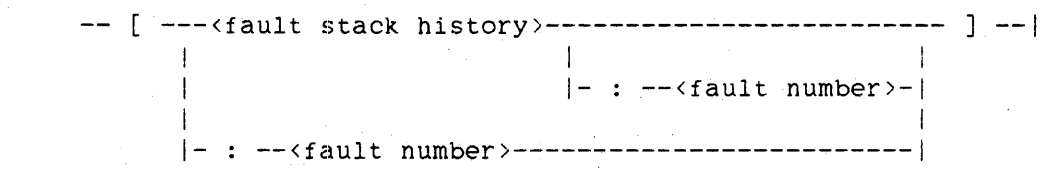
<fault list>



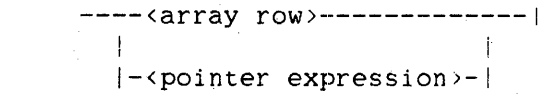
<fault name>



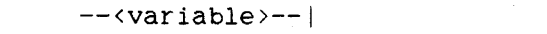
<fault information part>



<fault stack history>



<fault number>



<fault action>

--<statement>--|

<disabling on statement>

-- ON --<fault list>--|

See also

Semantics

The two forms of enabling ON statements are the "implicit call" form and the "implicit branch" form.

Once an interrupt is enabled, it remains enabled until one of the following conditions occurs:

1. The procedure or block that contains the ON statement is exited.

2. The interrupt is explicitly disabled.

3. A new interrupt is enabled for the same fault condition.

Whenever the block that contains an ON statement is exited, the interrupt status for that fault condition reverts to the interrupt status it had when the block was entered.

The fault list allows the user to enable or "arm" several faults with the same fault action or to disable or "disarm" one or more faults at the same time. An example of the use of fault lists appears below. The occurrence of any one of the faults in the fault list is sufficient to cause transfer of control to the fault action. The fault name ANYFAULT is used to arm or disarm all faults.

The fault information part provides access to the stack history at the time of the occurrence of the fault and to the number corresponding to the fault kind. The fault number, when it is used, is assigned one of the following values when the corresponding fault occurs:

Value	Fault
1	ZERODIVIDE
2	EXPONENTOVERFLOW
3	EXPONENTUNDERFLOW
4	INVALIDINDEX
5	INTEGEROVERFLOW
7	MEMORYPROTECT
8	INVALIDOP
9	LOOP
10	MEMORYPARITY
11	SCANPARITY
12	INVALIDADDRESS
14	STRINGPROTECT
15	PROGRAMMEDOPERATOR
18	INVALIDPROGRAMWORD

If the fault stack history option is used, a string of EBCDIC characters representing the stack history is stored into the array row or the array specified by the pointer expression. The stack history information is always stored as EBCDIC characters regardless of the character type of the array row or pointer expression.

The format of the stack history is

SSS:AAAA:Y,#SSS:AAAA:Y,#...,#SSS:AAAA:Y.

or

SSS:AAAA:Y#(DDDDDDDD),#...,#SSS:AAAA:Y#(DDDDDDDD).

where SSS is a code segment number, AAAA is a code word address, Y is a code syllable number, # is a blank space, and DDDDDDDD is a sequence number (present only if the compiler control option LINEINFO was TRUE during program compilation).

One of these entries is generated for each active lexical level in the stack in effect when the fault is encountered. Each entry is followed by a comma (,), and the last complete entry is terminated by a period (.). If the user-specified array is sufficiently long, the entire stack history is stored. If it is not long enough, then only a portion of the stack history is stored, with the last complete entry in the array terminated by a period. The code segment number field, SSS, is expanded to four characters, SSSS, for segment numbers greater than 4095; that is, for segment numbers whose hexadecimal representation requires four characters.

The array row or pointer expression that makes up the fault stack history and the variable that makes up the fault number are evaluated once when the ON statement is executed, and not at the time the fault occurs. Thus, in the following ON statement, array row A[I,*] is determined by the value of I at the execution of the ON statement and not when a ZERODIVIDE fault actually occurs. This determination is also true for the variables B[J] and J.

```
ON ZERODIVIDE[A[I,*]:B[J]]:  GO TO ERROR_HANDLING
```

The form of the ON statement that includes a comma, instead of a colon (:), before the fault action is the implicit call form. With this form of ON statement, when a specified fault occurs, the program calls the fault action statement as a procedure. If the fault action statement does a "bad go to" to a label outside the block in which the fault occurred, the fault condition is discarded, and the program continues running. If the fault action statement exits without doing a "bad go to" around the fault, the fault condition for which the fault action statement was called still exists. If an ON statement is enabled for that condition in a more global block, then control is passed to that ON statement; otherwise, the program is discontinued as a result of that fault.

A GO TO statement cannot be executed from outside the fault action statement to a label inside the fault action statement. Undefined results occur when a GO TO statement specifies a label passed as a parameter (a formal label).

The form of the ON statement that includes a colon, instead of a comma, before the fault action is the implicit branch form of the ON statement. With this form of ON statement, the program branches to the statement given as the fault action when a specified fault occurs. The fault condition is discarded, and the program continues execution from that point.

The disabling ON statement disables or disarms the interrupts corresponding to the fault names in the fault list.

No call on the block exit intrinsic is required to deactivate the armed faults for a block.

Examples

```
ON ZERODIVIDE OR INVALIDINDEX [FAULTARRAY:FAULTNO]:
   BEGIN
   REPLACE FAULTARRAY[8] BY FAULTNO FOR * DIGITS;
   WRITE(LINE, 22, FAULTARRAY);
   REPLACE FAULTARRAY BY " " FOR 22 WORDS;
   CASE FAULTNO OF
      BEGIN
      1: DIVISOR := 1;
      4: INDEX := 100;
      END;
   GO BACK;
   END
```

If either a divide-by-zero fault or an invalid index fault occurs at run time, the fault condition is discarded and control transfers to the compound statement in this ON statement. The stack history information is written to the array row FAULTARRAY, and the fault number of the fault that occurred is stored in FAULTNO.

```
ON MEMORYPROTECT OR LOOP: Q := 2
```

If either of the specified faults occurs at run time, the fault condition is discarded and control is transferred to the assignment statement in the ON statement. After execution of the assignment statement, execution continues with the statement following the ON statement.

```
ON EXPONENTUNDERFLOW    % DISABLING ON STATEMENT
```

Disables the interrupt associated with the exponent underflow fault.

```
ON ANYFAULT [POINTR + 2:Z], HANDLEFAULTS(Z)
```

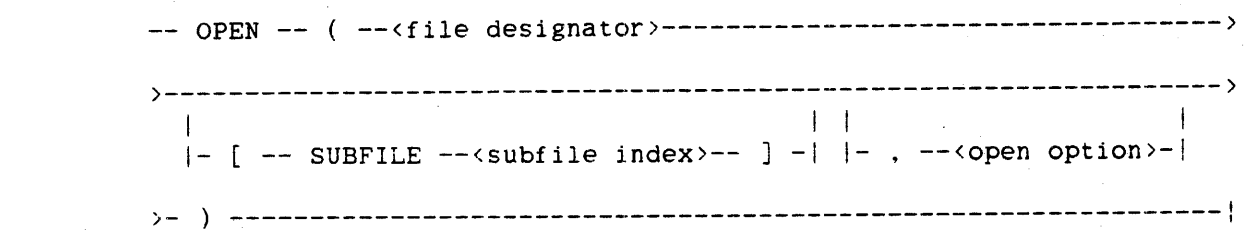
If any fault occurs, the statement HANDLEFAULTS(Z) is called as a procedure. The stack history information is written to the location indicated by the pointer expression POINTR + 2, and the fault number of the fault that occurred is stored in Z.

OPEN STATEMENT

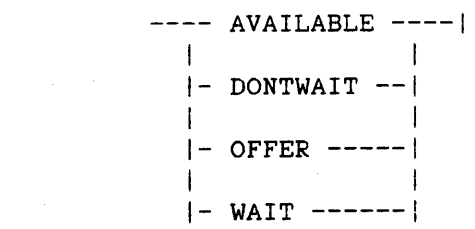
The OPEN statement causes the referenced file or subfile to be opened.

Syntax

<open statement>



<open option>



See also

Semantics

The subfile index specifies the subfile to be opened.

The OPEN statement can be used as an arithmetic function and returns the same values as the file attribute AVAILABLE returns. For a description of these values, see the "I/O Subsystem Reference Manual."

The open options AVAILABLE, DONTWAIT, OFFER, and WAIT are described in the "I/O Subsystem Reference Manual."

If no open option is specified, the WAIT option is assumed. Any open option can be used with any type of file. However, DONTWAIT and OFFER are meaningful only for port files. For other kinds of files, DONTWAIT and OFFER are ignored and an open with WAIT is performed.

Examples

```
OPEN(FILEID)
```

Opens file FILEID. Execution of the program is suspended until FILEID is open.

```
OPEN(FILEID[SUBFILE I],OFFER)
```

Opens subfile I of port file FILEID and offers it for matching. The program does not wait for a matching subfile to be found.

```
OPEN(FILEID[SUBFILE I],WAIT)
```

Opens subfile I of port file FILEID and offers it for matching. The program is suspended until a matching subfile is found.

```
IF OPEN(FILEID[SUBFILE I],AVAILABLE) = 1 THEN
   PROCESSOPEN
```

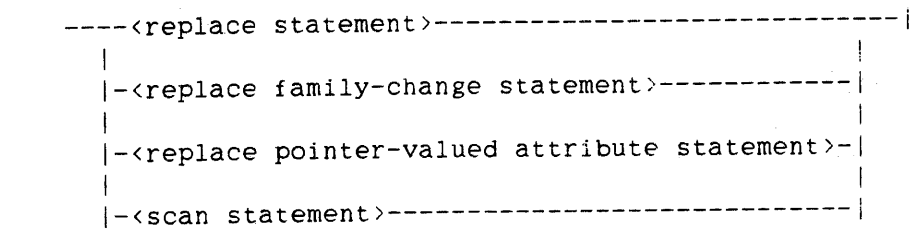
Opens subfile I of port file FILEID and offers it for matching. A search is made for a matching subfile that has been offered. If one is found, the subfile is opened, the result returned by the OPEN statement is 1, and PROCESSOPEN is called; otherwise, an error result is returned and PROCESSOPEN is not called.

POINTER STATEMENT

Pointer statements are used to examine, transfer, and edit character data stored in arrays.

Syntax

<pointer statement>



Semantics

The REPLACE statement can be used to move character data into an array row. Within a single REPLACE statement, the character data to be moved can be taken from several sources. Each of these sources can be one of several different types. A source can be another array row, a string literal, the value of an arithmetic expression, the value of a string expression, or the value of a pointer-valued attribute. Furthermore, as the character data is moved from a source to the destination, the characters can be translated or edited. Also, an arithmetic expression source can be treated as a binary value and converted into the equivalent decimal number expressed as a string of numeric characters.

The replace family-change statement is the language construct provided to add datacomm stations to or remove datacomm stations from a family of stations.

The replace pointer-valued attribute statement is the language construct provided to assign character data to pointer-valued file and task attributes.

The SCAN statement can be used to examine character data located in an array row.

Pointer statements process character data from left to right.

Pragmatics

Many of the operations performed by pointer statements require the use of temporary storage for intermediate results. In describing the actions of a pointer statement, a discussion of how this temporary storage is initialized, changed, and disposed of is necessary. These discussions use the following names for these temporary storage locations:

1. Stack-source-pointer
2. Stack-destination-pointer
3. Stack-auxiliary-pointer
4. Stack-integer-counter
5. Stack-test-character
6. Stack-source-operand

The prefix "stack" denotes that none of these parameters correspond to any program variables. They exist only until execution of the pointer statement is completed.

The stack-source-pointer, the stack-destination-pointer, and the stack-auxiliary-pointer have the same internal structure as a pointer variable that can be declared in a program. These temporary storage locations are initialized either from pointer expressions in the pointer statement or from previous corresponding temporary storage locations.

The initial value of the stack-source-pointer points to the first source character to be used by the associated operation. As the execution of the instruction progresses, the stack-source-pointer is modified to point to each successive source character. When the operation is complete, the stack-source-pointer points to the first "unprocessed" character in the source data (the "process" is determined by the particular form of the pointer statement). This final value can be stored into a pointer variable, or it can be discarded.

The initial value of the stack-destination-pointer points to the first destination character position to be used by the associated operation. As the execution of the operation progresses, the stack-destination-pointer is modified to point to each successive destination character position. When the operation is complete, the stack-destination-pointer points to the first unfilled character position in the destination. If more than one source is to be processed, the stack-destination-pointer value corresponding to the

completed processing of one element in the source list is used as the initial value for the subsequent source. If no more sources are to be processed, this final value can be stored into a pointer variable, or it can be discarded.

The initial value of the stack-auxiliary-pointer points to the first entry in a table of data to be used by the operation in its execution. This table can be a translate table if the operation to be performed is extracting characters from the source data, translating the characters to different characters (possibly containing a different number of bits per character), and storing the translated characters in the destination. This table can be a truth set describing a particular set of characters if the operation to be performed requires a membership test. Finally, this table can be a "picture"--a table that contains instructions of a special type describing how the source data is to be edited before being stored in the destination.

The stack-integer-counter, when required by a pointer statement, is initialized by an arithmetic expression supplied in the pointer statement. The value of this arithmetic expression is integerized before it is used. The stack-integer-counter has different meanings depending on the type of pointer statement involved. In some cases, the number of characters in a source string to be processed is dictated solely by this parameter. The number of numeric characters to be placed in the destination while converting the value of an arithmetic expression to character form is also dictated by the stack-integer-counter.

In some forms of the pointer statement, two controlling factors exist that dictate how many characters are to be processed from a source string. One factor depends on the source data and is called a condition. The other factor is a maximum count contained in the stack-integer-counter and is provided by an arithmetic expression in the pointer statement. For example, with such a pointer statement, the following instructions could be written: "translate characters from the source string to the destination until either 14 characters have been transferred or a period is encountered in the source string, whichever comes first." The final value of the stack-integer-counter is available for storage, or it is discarded.

The stack-test-character is initialized by an arithmetic expression (usually, but not necessarily, of the form of a single-character string, such as "B"). Although the stack-test-character parameter is one entire word of memory that contains the single-precision value of the arithmetic expression, only the rightmost character position of the word is used. When a condition employing a relational operator is used in a pointer statement, the stack-test-character must contain the character against which the individual characters in the source string are to be compared.

The stack-source-operand is used when the source data is given by the value of an arithmetic expression rather than a value located in an array row into which the stack-source-pointer points. The stack-source-operand is initialized by the arithmetic expression.

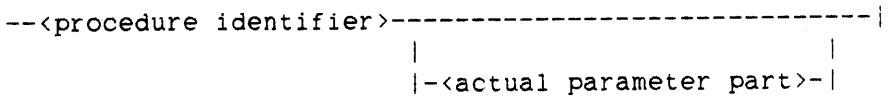
Refer to the discussions of the specific pointer statements for more detailed information.

PROCEDURE INVOCATION STATEMENT

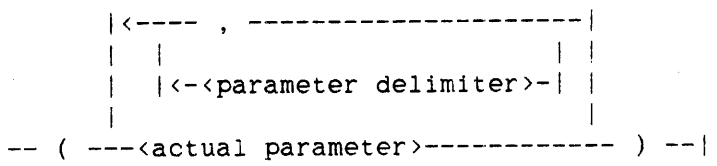
A procedure invocation statement causes a previously declared procedure to be executed as a subroutine.

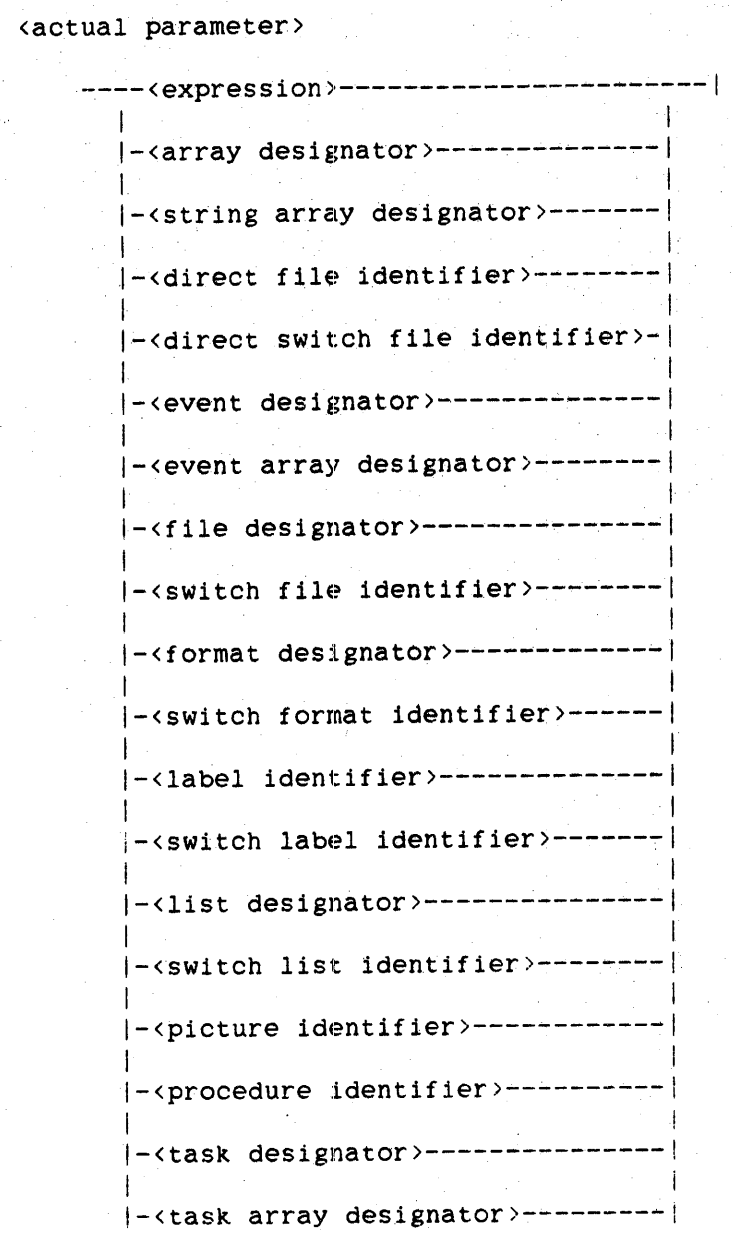
Syntax

<procedure invocation statement>



<actual parameter part>




<actual parameter>
<expression>
<array designator>
<string array designator>
<direct file identifier>
<direct switch file identifier>
<event designator>
<event array designator>
<file designator>
<switch file identifier>
<format designator>
<switch format identifier>
<label identifier>
<switch label identifier>
<list designator>
<switch list identifier>
<picture identifier>
<procedure identifier>
<task designator>
<task array designator>

See also

Semantics

When a procedure is invoked, program control is transferred from the point of the procedure invocation statement to the referenced procedure. When the procedure is completed, program control is transferred back to the statement following the procedure invocation statement, unless a "bad go to" is executed in the referenced procedure. Bad GO TO statements are described under "GO TO Statement."

A typed procedure returns a value. However, when a typed procedure is used in a procedure invocation statement, this value is discarded.

Calling Procedures with Parameters

The actual parameter part of a procedure invocation statement must have the same number of entries as the formal parameter list in the declaration of the procedure. Correspondence between the actual parameters and formal parameters is obtained by matching the parameters that occur in the same relative position in the two lists. Corresponding formal and actual parameters must be of compatible types. Parameters can be call-by-name or call-by-value.

If a formal parameter is of type INTEGER, REAL, or DOUBLE, then the actual parameter must also be INTEGER, REAL, or DOUBLE, but not necessarily the same type as its formal counterpart. If a mismatch among these types occurs, then the action that takes place depends on whether the formal parameter is call-by-name or call-by-value. If it is call-by-value, the type of the actual parameter is converted to the type of the formal parameter before the formal parameter is assigned the value of the actual parameter. If the formal parameter is call-by-name, the appropriate conversion takes place each time the formal parameter is referenced.

If the formal parameter of a non-formal procedure is a simple variable of type COMPLEX, then the corresponding actual parameter can be of type INTEGER, REAL, DOUBLE, or COMPLEX. However, if the COMPLEX formal parameter is call-by-name and the corresponding actual parameter is not of type COMPLEX, an assignment to that formal parameter within the procedure body causes the program to be discontinued with a fault.

Actual parameters of all types other than INTEGER, REAL, and DOUBLE must match the type of the formal parameter exactly.

For more information on procedures, refer to "PROCEDURE Declaration."

Examples

```
SIMPL
```

Invokes the procedure SIMPL, which has no parameters.

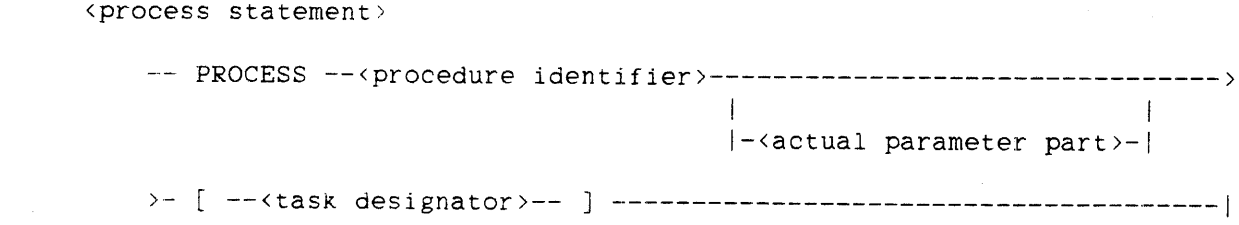
```
HEAVY(X,Y,A[*],SQRT(BINGO+BASE))
```

Invokes the procedure HEAVY and passes it four parameters: X, Y, the array row A[*], and the expression SQRT(BINGO+BASE).

PROCESS STATEMENT

The PROCESS statement initiates a procedure as an asynchronous process.

Syntax



See also

‹actual parameter part› 346
‹procedure identifier›. 165
‹task designator› 200

Semantics

Initiation of an asynchronous process consists of setting up a separate stack for the process, passing any parameters (call-by-name or call-by-value), and beginning the execution of the procedure. The initiating program continues execution, and both the initiating program and the initiated procedure run in parallel.

The specified procedure cannot be a typed procedure.

The actual parameter part must agree in number and type with the formal parameter part in the declaration of the procedure; otherwise, a run-time error occurs.

The task designator associates a task with the process at initiation; the values of the task attributes of that task, such as COREESTIMATE, STACKSIZE, and DECLAREDPRIORITY, can be used to control execution of the process. For information about assigning values to task attributes, refer to ‹arithmetic task attribute› under "Arithmetic Assignment," ‹Boolean task attribute› under "Boolean Assignment," and "Task Assignment." Many task attributes can be interrogated while the process is running.

See also
‹arithmetic task attribute› 227
‹Boolean task attribute›. 235
Task Assignment 246

An asynchronous process depends on its initiator for global variables and call-by-name actual parameters. Thus, for each process, a "critical block" is present in the initiator that cannot be exited until the process is terminated. The critical block is the block of highest lexical level that contains one or more of the following items:

- The declaration of the procedure itself
- The declarations of the actual parameters passed to the call-by-name formal parameters
- The declaration of the task designator
- Any compiler-generated code for evaluating arithmetic expressions passed to call-by-name parameters

The critical block can be the block that contains the PROCESS statement, the outer block of the program, or some block in between. An attempt by the initiator to exit the critical block before the process is terminated causes the initiator (and all tasks it has initiated through CALL or PROCESS statements) to be terminated.

A process is terminated by exiting its own outermost block or by execution in the initiator of the following statement:

‹task designator›.STATUS := VALUE(TERMINATED)

where the task designator specifies the task associated with the process to be terminated.

Pragmatics

The processed procedure must not declare any own arrays. An attempt to do so results in a run-time error.

Examples

```
PROCESS AGENT [TSK]
```

The procedure AGENT, which has no parameters, is invoked as an asynchronous process. The task TSK is associated with the process.

```
PROCESS ACHILD(OUTARRAY,YOUREVENT[INDX],COUNT) [TSKARAY[INDX]]
```

The procedure ACHILD is invoked as an asynchronous process and passed the three parameters OUTARRAY, YOUREVENT[INDX], and COUNT. The task designated by TSKARAY[INDX] is associated with the process.

PROCURE STATEMENT

The PROCURE statement tests the available state of an event.

Syntax

<procure statement>

```
-- PROCURE -- ( --<event designator>-- ) --|
```

See also
<event designator>. 78

Semantics

If the available state of the event is FALSE (not available), the program is suspended and put in the procure list until some other task executes the LIBERATE statement for that event. If the available state of the event is TRUE (available), the available state is set to FALSE (not available), and the program continues execution with the statement following the PROCURE statement.

Pragmatics

The PROCURE statement provides a means for different programs to share resources. For example, a convention could be established that a certain shared resource that is available for use by more than one program is not to be used by a program unless that program has procured the event that is used as the interlock. When the program has completed its use of the resource, it should execute a LIBERATE statement on the event.

Examples

PROCURE(EVNT)

If the available state of EVNT is TRUE (available), EVNT is procured by setting its available state to FALSE (not available). Otherwise, the program is suspended until EVNT is made available.

PROCURE(EVNTARAY[INDX])

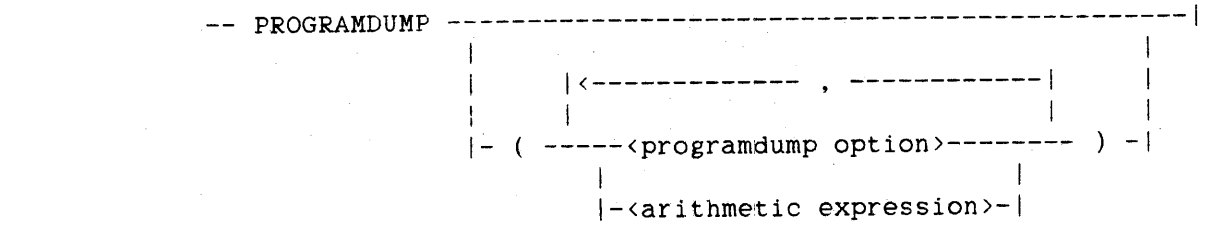
If the available state of the event designated by EVNTARAY[INDX] is TRUE (available), then that event is procured by setting its available state to FALSE (not available). Otherwise, the program is suspended until the event designated by EVNTARAY[INDX] is made available.

PROGRAMDUMP STATEMENT

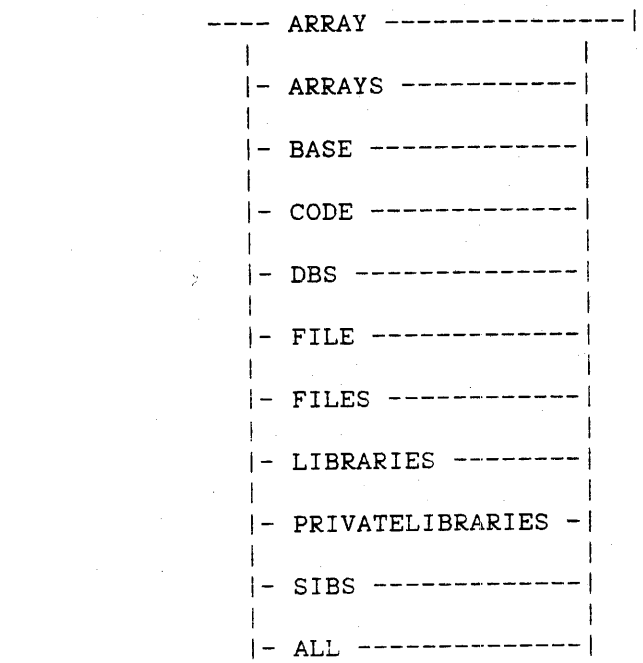
The PROGRAMDUMP statement causes the Master Control Program (MCP) to print out the contents of the stack of the program. Several options are available to specify which items of the stack are to analyzed and printed.

Syntax

<programdump statement>



<programdump option>



Semantics

The information produced by the PROGRAMDUMP statement is written to the file specified by the TASKFILE task attribute of the program.

If no programdump options are specified, the stack is analyzed and printed according to the specifications in the task attribute OPTION of the program.

If the contents of the arrays of the program are to be printed, the option ARRAY or ARRAYS must be specified.

The bottom (or "base") of the stack of the program is printed if the BASE option is specified. The MCP uses a portion of each stack to contain various words needed to control, identify, and log the program.

The segment dictionary of the program is printed out if the CODE option is specified. The actual code is printed only for segments that are referenced by the program at the time of the PROGRAMDUMP statement. Value arrays in the segment dictionary are printed when both the CODE and either the ARRAY or ARRAYS options are specified.

The DBS option causes the output of database stacks and, prior to the Mark 3.5 release, structure information blocks (SIBs). The SIBS option causes the output of SIBs on Mark 3.4 and earlier releases; on Mark 3.5 and later releases, it has no effect.

If program files are to be printed and analyzed, the FILE or FILES option must be specified. As each file is encountered, each word of the file information block (FIB) is separately named and, in some cases, analyzed.

The LIBRARIES option causes the stacks of all libraries that are being used by the program to be printed. The PRIVATELIBRARIES option causes the stacks of all private libraries that are being used by the program to be printed.

Specifying the ALL option is equivalent to specifying all the other options.

If the arithmetic expression option is used, the individual bits of the value of the expression are interpreted as follows:

[7:1] = 1	The base of the user stack is to be printed.
[8:1] = 1	Array contents are to be printed.
[9:1] = 1	The segment dictionary is to be printed.
[10:1] = 1	Files are to be analyzed and printed.
[15:1] = 1	Database stacks and SIBs are to be printed.
[18:1] = 1	SIBs are to be printed.
[19:1] = 1	Stacks for libraries are to be printed.
[20:1] = 1	Stacks for private libraries are to be printed.

When the MCP has completed analyzing and printing the specified items, control passes to the next statement.

Pragmatics

Diagnostic and debugging information can also be written to the TASKFILE so that the program dump and the information are coordinated.

Examples

```
PROGRAMDUMP
```

Analyzes and prints the program stack according to the value of the OPTION task attribute of the program.

```
PROGRAMDUMP(ARRAYS)
```

Analyzes and prints the basic information plus the contents of all arrays.

```
PROGRAMDUMP(ARRAYS,BASE,CODE,FILE)
```

Analyzes and prints the contents of arrays, value arrays, the base of the stack, the segment dictionary, referenced code segments, and files.

PROGRAMDUMP(ALL)

Analyzes and prints the maximum amount of information about the program stack.

PROGRAMDUMP(DUMPPARAM)

Analyzes and prints the program stack according to the value of DUMPPARAM.

PROGRAMDUMP(0 & 1 [10:1])

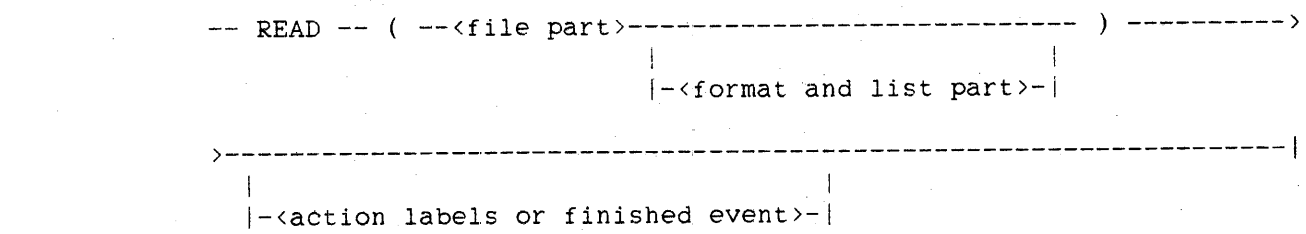
Equivalent to the statement PROGRAMDUMP(FILES). This statement analyzes and prints the contents of files of the program.

READ STATEMENT

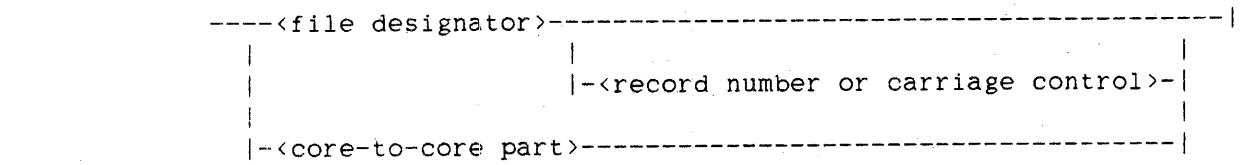
The READ statement allows data to be read from files and assigned to program variables.

Syntax

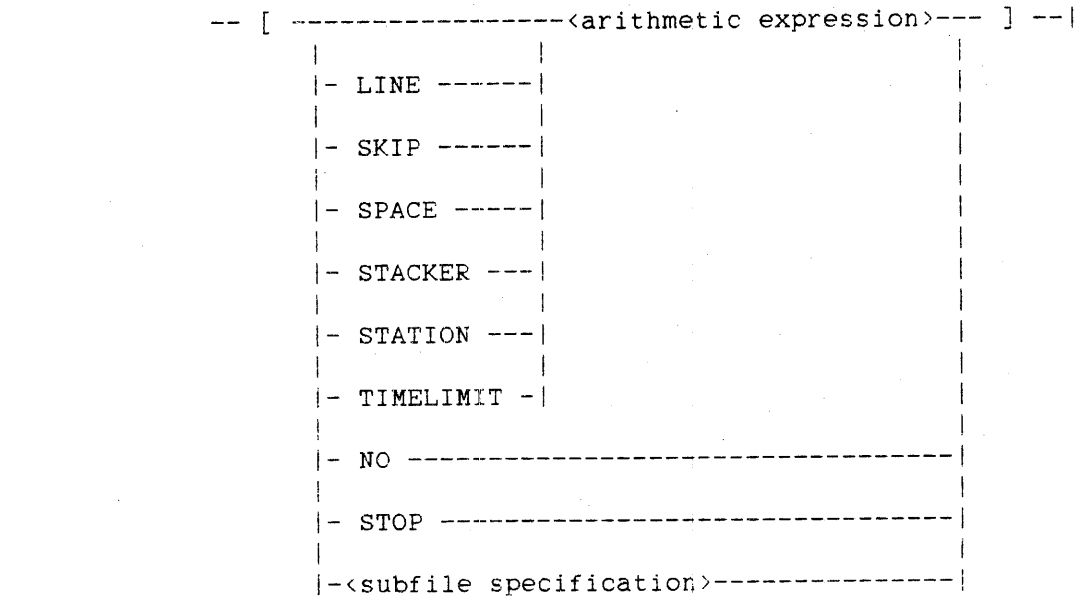
<read statement>

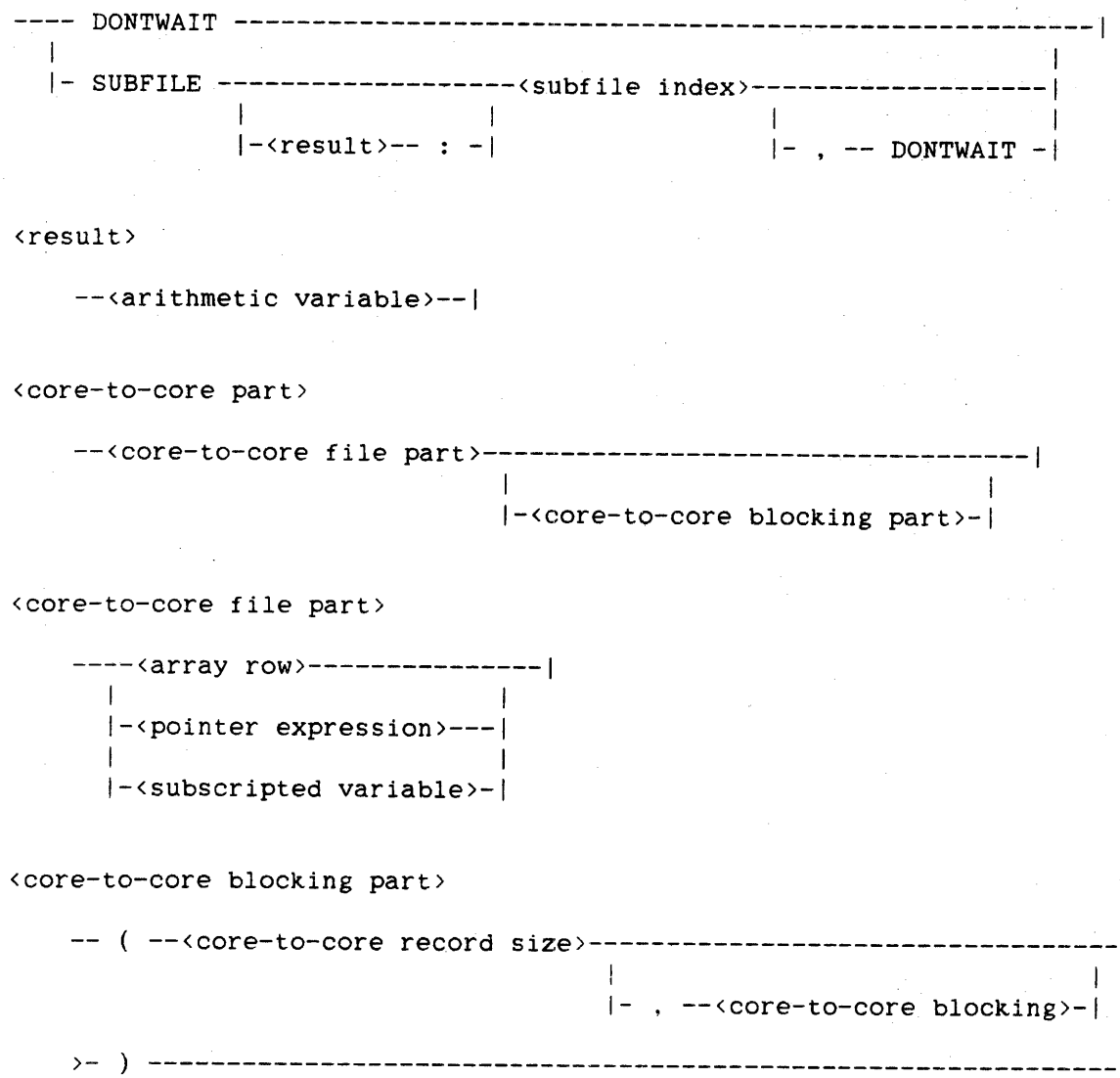


<file part>

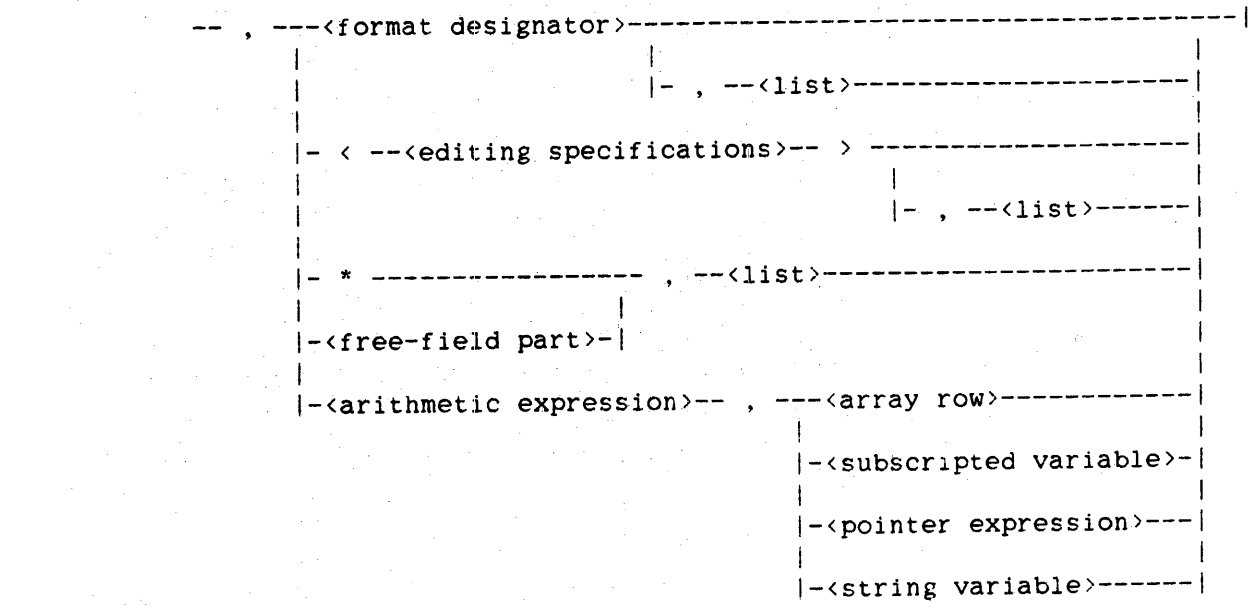


<record number or carriage control>

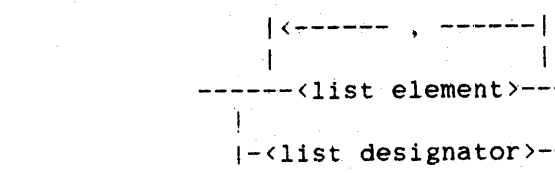



<subfile specification>
DONTWAIT
SUBFILE
<subfile index>
<result>
: -
, -- DONTWAIT
<result>
<arithmetic variable>
<core-to-core part>
<core-to-core file part>
<core-to-core blocking part>
<core-to-core file part>
<array row>
<pointer expression>
<subscripted variable>
<core-to-core blocking part>
(--<core-to-core record size>
, --<core-to-core blocking>
) -
<core-to-core record size>
<arithmetic expression>
<core-to-core blocking>
<arithmetic expression>

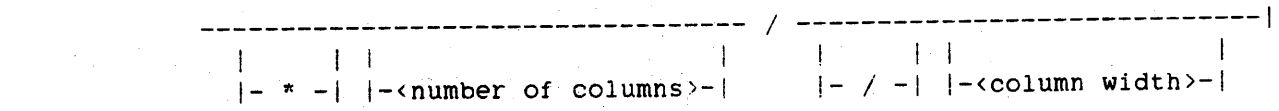
<format and list part>



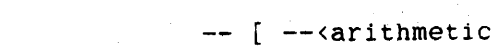
<list>



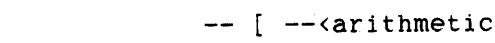
<free-field part>

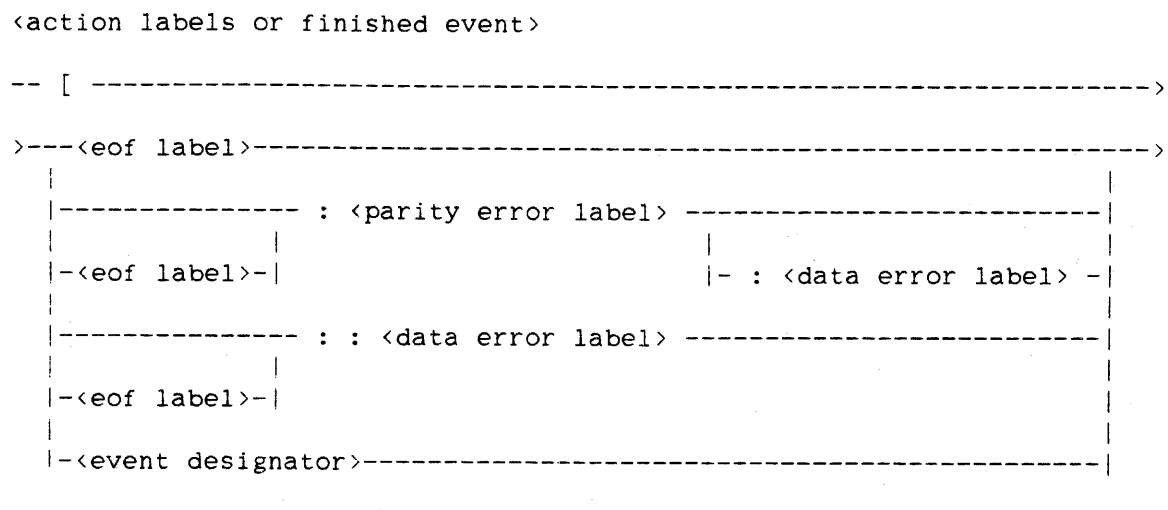


<number of columns>



<column width>





<eof label>

--<designational expression>--|

<parity error label>

--<designational expression>--|

<data error label>

--<designational expression>--|

See also

NOTE

The syntax of the READ statement and the syntax of the WRITE statement are nearly identical. Differences in the semantics are discussed separately in the semantics for each statement.

Semantics

The action of the READ statement depends on the form of the <file part> element and on the form of the <format and list part> element.

The READ statement can be used as a Boolean function. When the read operation fails, the value TRUE is returned. When the read operation succeeds, the value FALSE is returned. Specifically, the READ statement returns a value identical to that returned by the file attribute STATE. For more information, refer to the discussion of the STATE attribute in the "I/O Subsystem Reference Manual."

Examples of READ Statement Syntax

```
READ(FILEID)

READ(FILEID,FMT)

READ(FILEID,FMT,LISTID)

READ(FILEID,*,LISTID)

READ(SPOFILE,FMT,A,B,C)

READ(SPOFILE,/,SIZE,LENGTH,MASS)

READ(FILEID,FMT,7,2,A,B,C,ARAY[A],B+C,F)

READ(FILEID,/,J,FOR I := 0 STEP 1 UNTIL J DO ARRY[I])

READ(FILEID,*,A,B,C,FOR A := B*A STEP C UNTIL J DO ARY[I])

READ(SWFILEID[IF X > N THEN X+N ELSE 0],25,ARRY[2,*])

READ(FILEID,/,SWLISTID[I])

READ(FILEID,FMT,SWLISTID[I])

READ(SPOFILE,SWFMT[16],A,B,C)
```

```
READ(FILEID,50,STR)

READ(FILEID,/,L,M,N,ARRY[2]) [EOFL]

READ(FILEID[3][NO]) [:PARL]

READ(SWFILEID[14][NO],FMT,A+EXP(B),ARRY[I,J,*]) [:PARSWL[M]]

READ(FILEID[NO],SWFMT[6+J],LISTID) [EOFSWL[Q*3]::DATAERRORL]

READ(SWFILEID[A+B],*,SWLISTID[2+H/K]) [EOFL:PARL]

READ(FILEID[NO]) [EOFSWL[I]:PARSWL[J]]

READ(FYLE) [EOFL:PARL:DATAERRL]

READ(DIRFYLE) [EVNT]

READ(DIRFYLE,30,DIRARAY) [EVNT]
```

<file part>

The file part specifies the location of the data to be read.

<file designator>

The file designator specifies the file to be read. For more information, refer to "SWITCH FILE Declaration."

<record number or carriage control>

If the <record number or carriage control> element is not specified, the record currently addressed by the record pointer is read, and the record pointer is adjusted to point to the next record in the file.

If the <record number or carriage control> element is invalid for the physical file associated with the file designator, it is ignored.

If the <record number or carriage control> element is an arithmetic expression, its value indicates the zero-relative record number of the record in the file that is to be read. The record pointer is adjusted to point to the specified record before the read is performed, and the record pointer is adjusted after the read operation to point to the next record.

If the <record number or carriage control> element is "NO", then the record pointer is not adjusted following the read operation. That is, the record can be read again. This <record number or carriage control> element has no effect if the KIND attribute of the file being read is equal to REMOTE.

If the <record number or carriage control> element is of the form "[SPACE <arithmetic expression>]", then the number of records specified by the value of the arithmetic expression are skipped. Spacing is forward if the arithmetic expression has a positive value and backward if the arithmetic expression has a negative value.

The "[TIMELIMIT <arithmetic expression>]" construct, which is meaningful only for remote files, assigns the value of the arithmetic expression to the TIMELIMIT attribute of the file. Refer to the "I/O Subsystem Reference Manual" for information on the TIMELIMIT attribute. The value of this attribute applies to all subsequent READ and WRITE statements on that file. If the value of the TIMELIMIT attribute is greater than zero and if no input is received within that number of seconds (the value can be fractional), then a time-out error is reported.

The "[STATION <arithmetic expression>]" construct is meaningful only for remote files. The value of the arithmetic expression is assigned to the LASTSUBFILE attribute of the file. Refer to the "I/O Subsystem Reference Manual" for information on the LASTSUBFILE attribute.

<subfile specification>

If the file to be read is a port file (a file for which the KIND attribute is equal to PORT), an array row read containing a subfile specification must be used. Refer to "Array Row Read" in this section.

The subfile specification is meaningful only for port files. It is used to specify the subfile to be used for the read operation and the type of read operation to be performed.

If the subfile index is used, the value of the subfile index is assigned to the LASTSUBFILE attribute of the file. It specifies the subfile to be used for the read operation. If the subfile index is zero, a nonselective read is performed. If the subfile index is nonzero, then a read from the specified subfile is performed. The result variable, if any, is assigned the resultant value of the LASTSUBFILE attribute. For more information on the LASTSUBFILE attribute, refer to the "I/O Subsystem Reference Manual."

If DONTWAIT is specified in a READ statement, and if no input is available, no data is returned and the program is not suspended.

<core-to-core part>

If the <file part> consists of a <core-to-core part>, then a core-to-core read is performed. A core-to-core read operation reads from a location in memory, not from a physical device; therefore, it is much faster than a physical read. Editing is performed exactly as it is performed when reading from a physical device.

<core-to-core file part>

If the core-to-core file part is a hexadecimal, BCL, or EBCDIC array row or pointer, then the default record size (the number of characters considered to be in the record) depends on the character size of the array row or pointer and is determined by the actual length of the designated string.

The maximum size of the core-to-core file part for BCL and hexadecimal arrays is 65,535 words. Core-to-core I/O on BCL and hexadecimal arrays longer than 65,535 words is permitted only if the core-to-core file part is indexed far enough into the array such that the length between that point and the end of the array does not exceed 65,535 words. If an attempt is made to use an array or array segment more than 65,535 words long, a run-time error occurs.

For single- and double-precision array rows or subscripted variables, the default record size is computed by multiplying the length of the array row (or remaining length of the array row, when a subscripted variable is used) by the number of characters per word, where characters per word is derived from the following table:

	Default Character Type	
	BCL	EBCDIC
Precision: single	8	6
Precision: double	16	12

<core-to-core blocking part>

To specify a record size smaller than the default size, a value can be provided for <core-to-core record size>. This value is in terms of characters. By supplying a value for <core-to-core blocking>, the "file" can be blocked into more records than the default number, which is one.

With formatted I/O, if the format requires more records than indicated by the core-to-core blocking value, a run-time error is given. Also, the format can require more characters than the core-to-core file part contains; this situation also results in a run-time error. In such cases, the number of characters indicated in the core-to-core blocking part (this number is computed by multiplying the core-to-core record size by the core-to-core blocking) can appear to be large enough to satisfy the format, but the core-to-core blocking part can indicate more characters than the core-to-core file part actually contains. The core-to-core file part, the core-to-core blocking part, and the format must be compatible or run-time errors will occur.

Examples

```
BEGIN
   ARRAY A[0:9];
   REAL B,C;
   READ (A(80),<T50,A6,I10>,B,C);  % EXAMPLE 1
   WRITE(A(15,3),<X5,I15>,1,2,3);  % EXAMPLE 2
   WRITE(A(20,2),<X5,I15>,1,2,3);  % EXAMPLE 3
   B := "  ITEM";
   WRITE(A(15,4),<".",X2,A6,I2,X4>,B,1,B,2,B,3,B,4); % EXAMPLE 4
END.
```

The statement labeled "EXAMPLE 1" in the program above results in a run-time error (format error 217), because the format requires 65 characters, but the file part (array A) contains only 60 characters.

The statement labeled "EXAMPLE 2" results in a run-time error (format error 117), because the format requires 20-character records, but 15-character records were specified in the blocking part.

The statement labeled "EXAMPLE 3" results in a run-time error (format error 120), because the three list elements require three repetitions of the format. Thus, three records are required, but only two records were specified in the blocking part.

The statement labeled "EXAMPLE 4" fills array A with the following EBCDIC data ("|" denotes the end of the data):

```
.   ITEM 1   .   ITEM 2   .   ITEM 3   .   ITEM 4   |
```

<format and list part>

The <format and list part> element indicates the interpretation of the data in the file and the variables to which the data is assigned.

If the <format and list part> element does not appear, the input record is skipped.

Formatted Read

A READ statement that contains a format designator, editing specifications, or a free-field part is called a "formatted read."

A format designator without a list indicates that the referenced format contains a string literal into which corresponding characters of the input data are to be placed. The string literal in the FORMAT declaration is replaced by the string literal in the input data.

A format designator with a list indicates that the input data is to be edited according to the specifications of the format and assigned to the variables of the list.

Editing specifications can appear in place of a format designator and have the same effect as if they had been declared in a FORMAT declaration and had been referenced through a format designator. For more information, refer to "FORMAT Declaration."

On any formatted I/O statement (excluding core-to-core I/O), the number of characters allowed in the record is determined solely by the value of the file attribute MAXRECSIZE of the file. If the format requires more characters than are contained in the record, a format error occurs at run time.

The free-field part is discussed under "Data Format for Free-field Input" in this section.

Binary Read

A READ statement of the form

```
READ(<file part>,*,<list>)
```

is called a "binary read."

An asterisk (*) followed by a list specifies that the input data is to be processed as full words and assigned to the elements of the list without being edited. The number of words read is determined by the number of elements in the list or the maximum record size, whichever is smaller.

When data is read into character arrays, only full words are read. If there is a partial word left at the end of the data, it is ignored. For example, if A is an EBCDIC array and FILEID contains the string "12345678", the statement

```
READ(FILEID,*,A)
```

reads only the characters "123456".

When a string is read into a string variable using a binary READ statement, the first word read from the record is assumed to specify the length of the string. This word is evaluated, and the resulting value is the number of characters read beginning with the next word of the record. The binary WRITE statement automatically writes a word of length information before the text of each string variable; therefore,

```
WRITE(F,*,STR,STRARRAY[5],STR || "ABC")
```

can later be read by

```
READ(F,*,STR1,STR2,STRARRAY[0])
```

For more information, see "Binary Write" under "WRITE Statement."

The results are undefined for binary READ statements that attempt to read data not containing length information into string variables.

Array Row Read

A READ statement of any of the forms

```
READ(<file part>,<arithmetic expression>,<array row>)
READ(<file part>,<arithmetic expression>,<subscripted variable>)
READ(<file part>,<arithmetic expression>,<pointer expression>)
READ(<file part>,<arithmetic expression>,<string variable>)
```

is called an "array row read."

The first three forms of the array row read specify that input data is to be read without editing and assigned left-justified to the array specified by the array row, subscripted variable, or pointer expression. The arithmetic expression specifies the number of words or the number of characters, depending on the value of the FRAMESIZE attribute for the file, to be read. Refer to the "I/O Subsystem Reference Manual" for information on the FRAMESIZE attribute. The number of words or characters actually read equals whichever of the following values is smallest:

- the MAXRECSIZE attribute of the file being read
- the length of the array row (or portion of the array to the right of where the pointer expression points or to the right of the element specified by the subscripted variable)
- the absolute value of the arithmetic expression

A READ statement of the form

```
READ (<file part>,<arithmetic expression>,<string variable>)
```

specifies that input data is to be read without editing and assigned to the string variable. The number of characters read is the smaller of the value of the MAXRECSIZE attribute of the file being read or of the absolute value of the arithmetic expression. The value of the arithmetic expression always specifies the number of characters (not words) to be read.

Example

```
BEGIN
   FILE IN(TITLE="TEST.", UNITS=CHARACTERS, MAXRECSIZE=20);
   STRING S1,S2;
   READ(IN,15,S1); % READS 15 CHARACTERS INTO S1
   READ(IN,50,S2); % READS 20 CHARACTERS INTO S2
END.
```

<action labels or finished event>

The <action labels or finished event> element provides a means of transferring control from a READ statement, WRITE statement, or SPACE statement when exception conditions occur. A branch to the eof label takes place when an end-of-file condition occurs. A branch to the parity error label takes place if an irrecoverable parity error is encountered. A branch to the data error label takes place if a conflict exists between the format and the data. If the appropriate label is not provided when an exception condition occurs, the program is terminated.

The "[<event designator>]" syntax can be used only for direct I/O. The event is caused when the I/O operation is finished. For more information, refer to "Direct I/O" under "I/O Statement."

Exception conditions occurring during a READ statement can also be handled without the use of the <action labels or finished event> syntax. The READ statement can be used as a Boolean function, and the value returned can be tested to determine if any exception conditions exist. (For more information, refer to the discussion of the STATE attribute in the "I/O Subsystem Reference Manual.") When exception conditions are handled in this manner, the <action labels or finished event> syntax cannot be used. The user assumes all responsibility for handling exception conditions. Core-to-core I/O statements of the forms

```
READ(<array row>,<arithmetic expression>,<array row>)
WRITE(<array row>,<arithmetic expression>,<array row>)
```

cannot be used with the <action labels or finished event> syntax and cannot be used as Boolean functions. Attempting to do either results in a syntax error.

See also
Direct I/O. 316

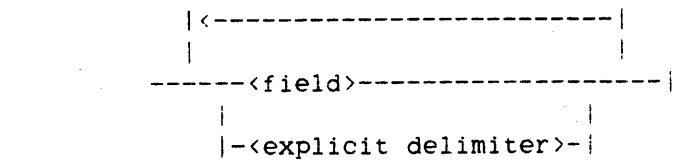
Data Format for Free-field Input

The use of a <free-field part> element in a READ statement allows input to be performed with editing but without using editing specifications. The appropriate format is selected automatically.

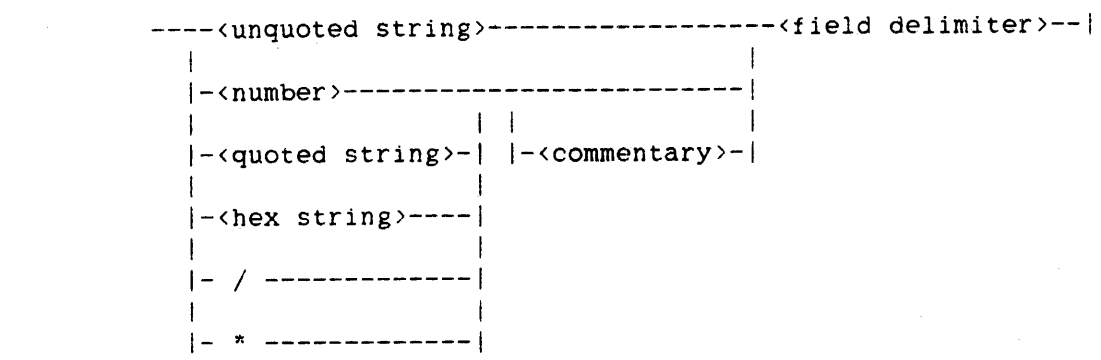
On input, only the simplest forms of the free-field part, a single slash (/) or double slash (//), can be used. These formats allow input from records in the form of free-field data records. A single slash indicates that data items are delimited by a comma; a double slash indicates that data items are delimited by one or more blanks.

Free-field Data Format

The format of a free-field input data record is as follows:



<field>



<unquoted string>

Any string not containing an <explicit delimiter>.

<quoted string>

-- <EBCDIC string> --|

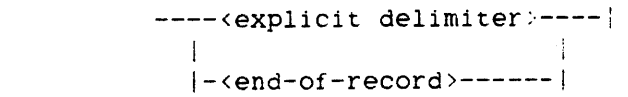
<hex string>

-- 4" --<hexadecimal string>-- " --|

<commentary>

Any string not containing an <explicit delimiter>.

<field delimiter>



<explicit delimiter>

Comma (,) for the single-slash form or one or more blanks for the double-slash form. An empty record is not considered an explicit delimiter.

<end-of-record>

The end of the input record.

Semantics

Each record of free-field input data must be in the form described above.

Empty records are ignored. <commentary> is ignored.

Each field except the slash is associated with the list element to which it corresponds by position.

Fields

The single-slash format interprets a field that contains only a comma or a comma preceded by blanks as a null field. Such a field is skipped along with its associated list element, which is left unaltered.

The different types of fields are described in the following paragraphs.

<unquoted string>

If an unquoted string is read into a list element of type string or pointer, all characters preceding the explicit delimiter (including quotation marks if present) are transferred to the list element. The <end-of-record> is not recognized as a delimiter.

If an unquoted string is read into a list element of type string, characters are read until an explicit delimiter is detected or until the maximum string length (2**5 - 2) is reached.

If an unquoted string is read into a list element of type pointer, characters are read until an explicit delimiter is detected or until the end of the array is reached.

If an unquoted string is read into a list element of type Boolean, the value TRUE is assigned to the list element if the first character of the string is "T". If the first character is not the letter "T", the value FALSE is assigned to the list element. The unquoted string is read until a field delimiter is detected.

If an unquoted string is read into a list element of any type other than string, pointer, or Boolean, it is treated as commentary.

<number>

A number that is represented as an integer is treated as type INTEGER unless it is larger than the largest allowable integer, in which case it is treated as type REAL. Numbers that contain a decimal fraction are treated as type REAL. However, when the list element is double precision, results are treated as type DOUBLE. When the field delimiter is a comma, blanks within numbers are ignored.

Complex values are divided into real and imaginary values. When a complex variable or complex subscripted variable appears in the list of a free-field READ statement, two fields are necessary to complete the read operation. The value in the first field is assigned to the real part, and the value in the second field is assigned to the imaginary part.

<quoted string>

A quoted string of any length can be read into single- or double-precision list elements. Each single-precision EBCDIC or BCL list element receives six characters or eight characters, respectively (12 or 16 characters, respectively, for double-precision list elements), until either the list or the string is exhausted. If the number of characters in the string is not a multiple of six (for EBCDIC) or eight (for BCL), then the last list element receives the remaining characters of the string. The string characters are stored, right-justified, in the list elements.

<hex string>

A hexadecimal string can be read into a single- or double-precision list element. If fewer than 12 hexadecimal digits are read into a single-precision variable (or fewer than 24 hexadecimal digits into a double-precision variable), the string is stored right-justified in the variable. If a minus sign precedes the string (for example, -4"A"), bit 46 of the resulting value is complemented.

slash (/)

The slash field causes the remainder of the current buffer to be ignored. The buffer following the slash is considered the beginning of a new field. The slash is a field by itself and must not be placed within another field or between a field and its explicit delimiter.

asterisk (*)

The asterisk field terminates the READ statement. The program continues with the statement following the READ statement. The list element corresponding to the asterisk remains unchanged, as do any subsequent elements in the list.

Examples

```
1,

2.5,     / anything to the right of a slash is ignored

2.48 @ -20,  / blanks are ignored if using single-slash editing

3 4      / two data elements if the delimiter is a blank

3,4,     / two data elements if the delimiter is a comma

"THIS IS A QUOTED STRING"

THIS IS AN UNQUOTED STRING AND THE DELIMITER IS A COMMA, 123

THIS-IS-AN-UNQUOTED-STRING-AND-THE-DELIMITER-IS-A-BLANK 456

2.5  ANY COMMENT OR NOTE NOT CONTAINING A COMMA.

4"AB" / A HEX STRING

-4"40000000000A" / BIT 46 IS COMPLEMENTED, THE RESULT = +10

,,,  / null fields; the three corresponding list elements are
     / skipped with no alteration to their contents.

4,   ,5 / null field is ignored

* THIS DATA RECORD TERMINATES THE READ STATEMENT
```

NOTE

Additional information about I/O operations can be found under "I/O Statement" and "WRITE Statement."

REMOVEFILE STATEMENT

The REMOVEFILE statement removes files without opening them.

Syntax

<removefile statement>

```
-- REMOVEFILE -- ( --<directory element>-- ) --|
```

See also
<directory element> 270

Semantics

The syntax and semantics of <directory element> appear under "CHANGEFILE Statement."

If the directory element is a directory name, all files in that directory are removed. If the directory element is both a file name and a directory name, that file and all files in the directory are removed.

A directory element of the form "<file name>/=" removes only files in that directory. It does not remove a file named "<file name>".

The REMOVEFILE statement can be used as a Boolean function, in which case it returns a value of TRUE if an error occurs. The value in field [39:20] of the result defines the failure as follows:

10	File name or directory name is in error.
30	Files have not been removed.

If a pointer expression is used as a directory element, it must point to an array that contains the name of the file or directory to be removed.

Pragmatics

Family substitution is used if the task has an active family specification and the family name involved in the REMOVEFILE statement is the target family name that the FAMILY specification substitutes.

If a family substitution specification is in effect, the REMOVEFILE statement affects only the substitute family, not the alternate family.

Example

```
BOOL := REMOVEFILE("MYTEST ON PACKFOUR.")
```

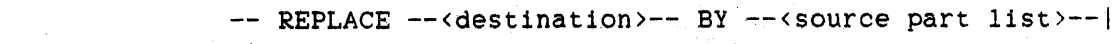
Removes the file MYTEST and, if the remove is successful, assigns FALSE to the variable BOOL.

REPLACE STATEMENT

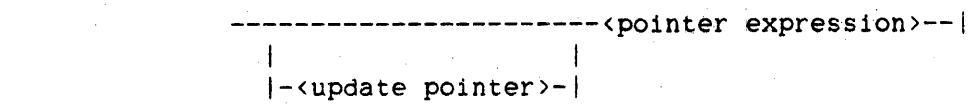
The REPLACE statement causes character data from one or more sources to be stored in a designated portion of an array row.

Syntax

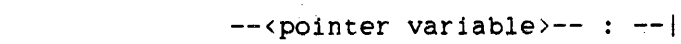
<replace statement>



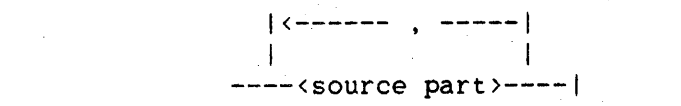
<destination>



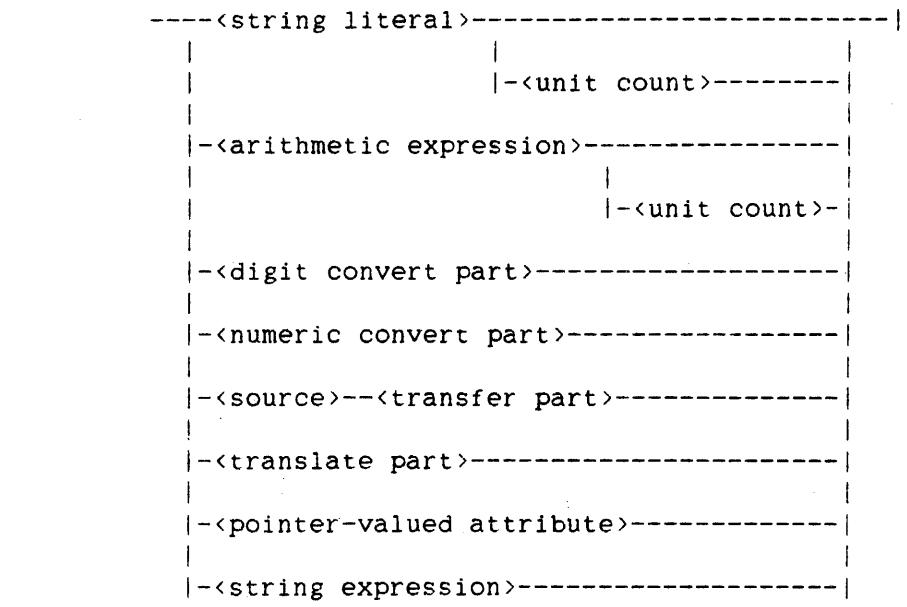
<update pointer>



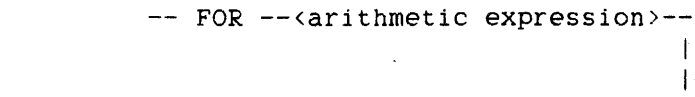
<source part list>



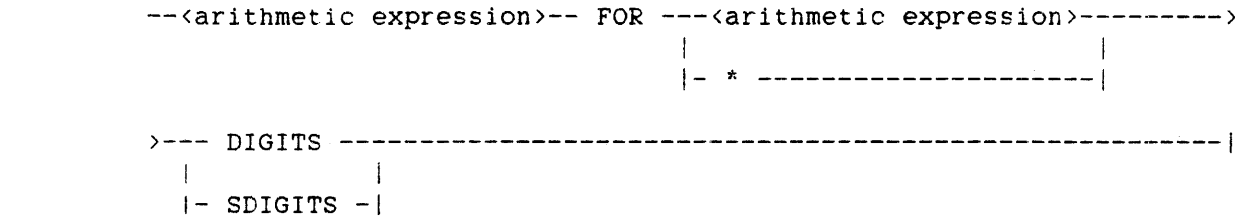
<source part>



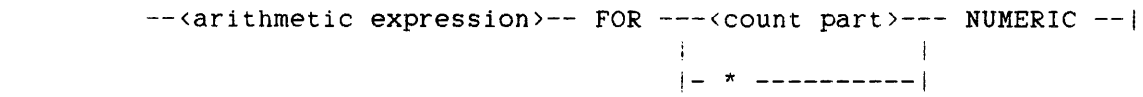
<unit count>

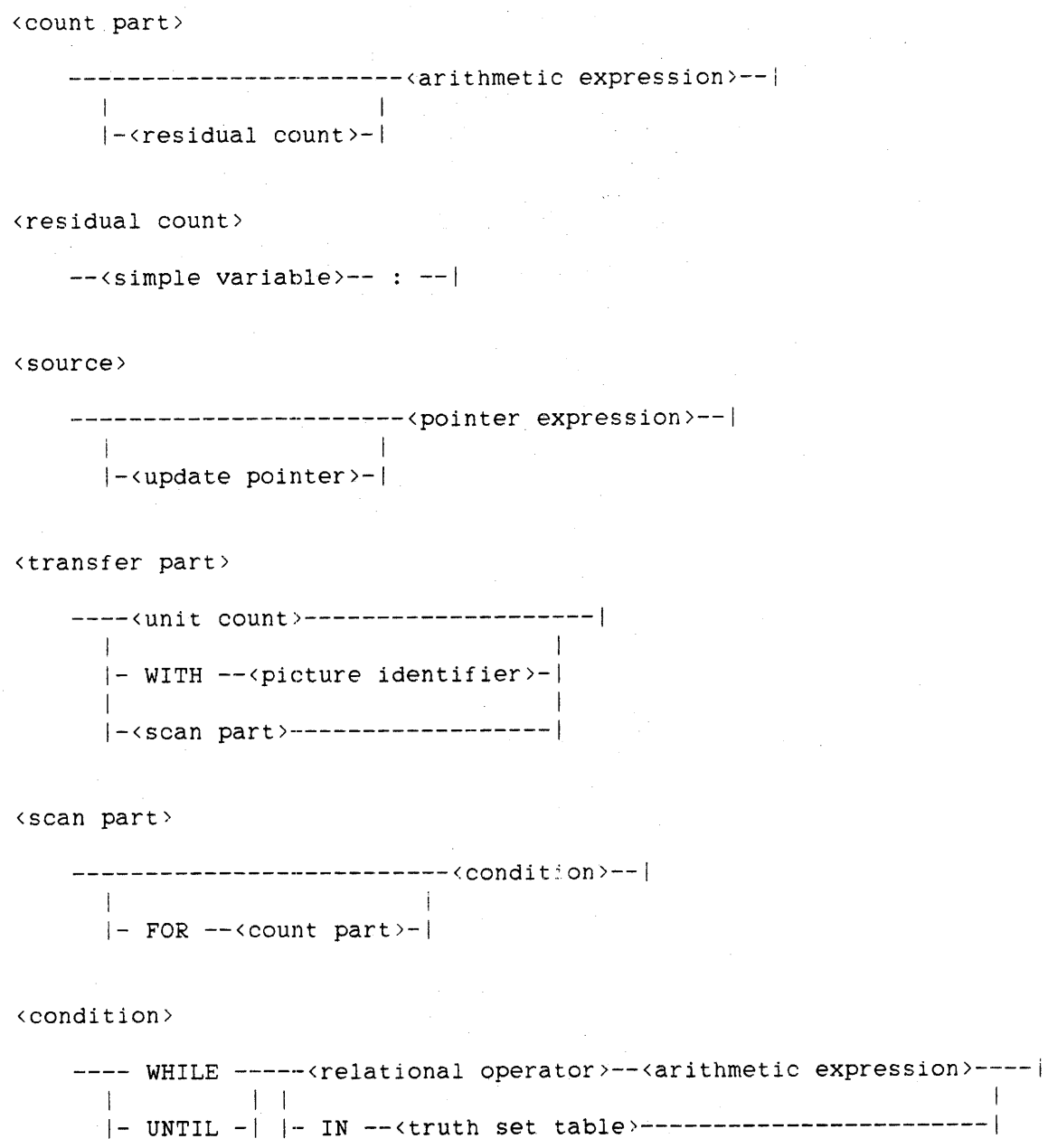


<digit convert part>

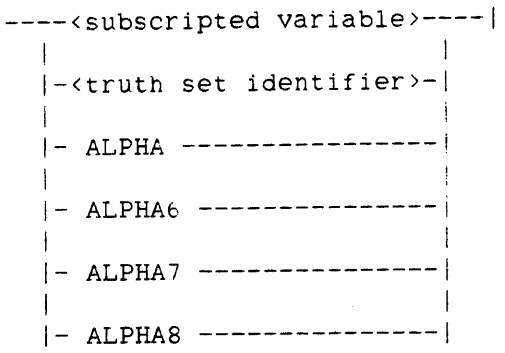


<numeric convert part>

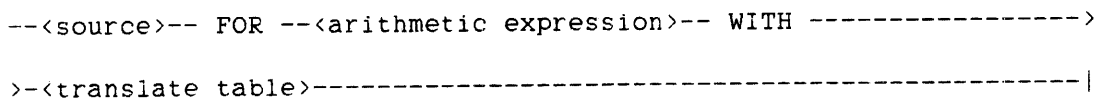



<count part>
<arithmetic expression>
<residual count>
<residual count>
<simple variable> : <source>
<pointer expression>
<update pointer>
<transfer part>
<unit count>
WITH <picture identifier>
<scan part>
<scan part>
<condition>
FOR <count part>
<condition>
WHILE
<relational operator>
<arithmetic expression>
UNTIL
IN <truth set table>

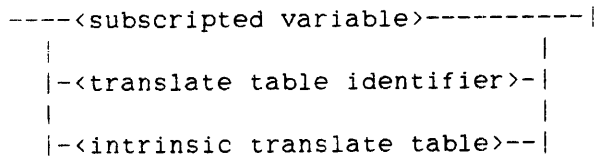
<truth set table>

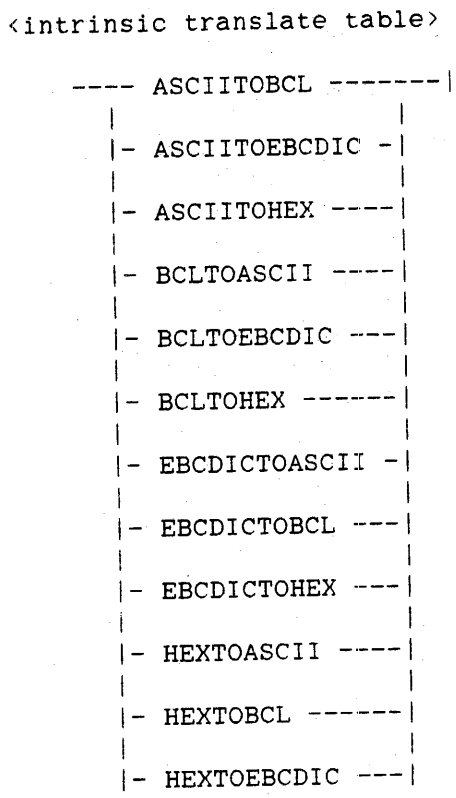


<translate part>



<translate table>





See also
‹picture identifier› 147
‹pointer variable› 241
‹pointer-valued attribute› 411
‹relational operator› 493
‹simple variable› 225
‹subscripted variable› 225
‹translate table identifier› 202
‹truth set identifier› 207

Semantics

The description of the REPLACE statement, which makes up the remainder of this section, makes frequent reference to the temporary storage locations defined under "Pointer Statement" in this chapter.

The REPLACE statement stores character data from one or more data sources into a designated portion of an array row. The array row and the starting character position within the array row are both determined by the pointer expression part of ‹destination›. The value of this pointer expression initializes the stack-destination-pointer. As each character is moved into the destination array row, the stack-destination-pointer is correspondingly incremented one character

position. When the last character has been stored in the destination array row, the corresponding final value of the stack-destination-pointer is stored in the pointer variable of the update pointer, if specified; otherwise, it is discarded.

The source part list consists of one or more source parts. Each source part specifies source data and the processing to be performed on the data. All the data specified by a single source part is processed by a single method, but the various source parts of the source part list can specify a variety of processing methods.

With certain forms of the source part, provisions are made to store the final value of the stack-source-pointer. With several source parts in a single REPLACE statement, several "final values" for the stack-source-pointer arise. Corresponding to these final values are values of the stack-destination-pointer. These latter values are not accessible to the programmer but serve as the initial values of the stack-destination-pointer for the processing of the next source part.

<source> is the same syntactic construct encountered in the SCAN statement. <source> contains a pointer expression that initializes the stack-source-pointer to a particular character position in an array row. The character size associated with this pointer expression must be the same as that associated with the pointer expression that initialized the stack-destination-pointer. If the update pointer option for <source> is present, the pointer variable specified by the update pointer is assigned the final value of the stack-source-pointer for this source part.

Pragmatics

The stack-source-pointer and the stack-destination-pointer can both reference the same array during a REPLACE statement. However, if the stack-source-pointer references a character position between the initial position of the stack-destination-pointer and its current position, the result is undefined. For example,

```
REPLACE POINTER(A)+6 BY POINTER(A) FOR 12
```

produces an undefined result. On the other hand,

```
REPLACE POINTER(A) BY POINTER(A) FOR 12
```

produces a well-defined result.

Examples of REPLACE Statement Syntax

```
REPLACE PTR BY "A"

REPLACE PTR:PTR BY "*" FOR 75

REPLACE PTR BY ITEM

REPLACE PTR BY (4"03").[7:48] FOR 1

REPLACE PTR BY " " FOR N WORDS

REPLACE PTR:PTR BY PST FOR 18

REPLACE PTR BY PST:PST FOR NUM WORDS

REPLACE PTR BY PINFO WITH PIC

REPLACE PTR:PTR BY PST WHILE NEQ " "

REPLACE PTR BY PST WHILE IN ALPHA

REPLACE P BY X FOR * DIGITS

REPLACE P BY X FOR 50 NUMERIC

REPLACE P BY X FOR * NUMERIC

REPLACE PTR BY PST WHILE IN MYTRUTHTABLE

REPLACE PTR BY PST UNTIL = ","

REPLACE PTR:PTR BY PST:PST UNTIL IN ALPHA8

REPLACE PTR BY PST FOR THELENGTH WHILE > "0"

REPLACE PTR BY PST FOR LEFT:25 WHILE IN ACCEPTABLE

REPLACE PTR BY PST FOR 120 UNTIL NEQ " "

REPLACE PTR BY PST FOR M:N UNTIL IN ALPHA

REPLACE PTR:PTR BY SUMTOTAL FOR 6 DIGITS, "."

REPLACE PTR BY FYLE.TITLE

REPLACE PTR BY PST:PST FOR L WITH XLATTABLE

REPLACE PTR BY STR

REPLACE PTR BY SARRAY[4,J]
```

```
REPLACE P BY S1 || S2

REPLACE P BY PTR:PTR FOR 10

REPLACE P BY TAKE(S,2) || SA[4]

REPLACE P BY HEAD(S,ALPHA)
```

<source part> Combinations

The formal syntax of the <source part> can be reduced to the following combinations:

```
<string literal>
<string literal> FOR <arithmetic expression>
                 FOR <arithmetic expression> WORDS

<arithmetic expression>
<arithmetic expression> FOR <arithmetic expression>
                        FOR <arithmetic expression> WORDS
                        FOR <arithmetic expression> DIGITS
                        FOR * DIGITS
                        FOR <arithmetic expression> SDIGITS
                        FOR * SDIGITS
                        FOR <count part> NUMERIC
                        FOR * NUMERIC

<source> FOR <arithmetic expression>
         FOR <arithmetic expression> WORDS
         FOR <arithmetic expression> WITH <translate table>

<source> WITH <picture identifier>

<source> WHILE <relational operator> <arithmetic expression>
         UNTIL <relational operator> <arithmetic expression>
         WHILE IN <truth set table>
         UNTIL IN <truth set table>

<source> FOR <count part> WHILE <relational operator>
                                 <arithmetic expression>
<source> FOR <count part> UNTIL <relational operator>
                                 <arithmetic expression>

<source> FOR <count part> WHILE IN <truth set table>
         FOR <count part> UNTIL IN <truth set table>

<pointer-valued attribute>

<string expression>
```

The remainder of the information about the REPLACE statement is organized according to the combinations listed above.

In all examples, P and Q are 8-bit pointers and the default character type is EBCDIC.

String Literal Source Parts

Short and Long String Literals

A string literal of 96 bits or less is a short string literal. A short string literal is evaluated at compile time and stored, left-justified, in a one- or two-word operand. Character size information is discarded.

A string literal of more than 96 bits is a long string literal. A long string literal is evaluated at compile time and stored in a portion of an array called a "pool array." The character size and address of the string literal are stored in a pointer called a "pool array pointer."

The compiler calculates the number of characters in a string literal in terms of the largest character size specified by the string literal. For example,

```
4"C1"      is two characters long.
8"AB"      is two characters long.
48"01"     is one character long.
4"01""A"   is two characters long (if the default character
              type is EBCDIC).
```

<string literal>

If the source part is a short string literal, it is processed as follows:

1. At compile time, the number of characters in the string is calculated.

2. At run time, the string literal is stored, left-justified with zero fill, in a one- or two-word stack-source-operand.

3. The stack-integer-counter is assigned the value for the string length calculated at compile time (see step 1).

4. Characters are copied from the stack-source-operand to the destination specified by the stack-destination-pointer. The

stack-integer-counter specifies the number of characters copied, and the stack-destination-pointer specifies the character size. If the destination is specified by a non-character array row or array element, the character size is eight bits.

If the source part is a long string literal, it is processed as follows:

1. At compile time, the number of characters in the string is calculated.

2. At run time, the stack-source-pointer is assigned the value of the pool array pointer to the long string literal, which includes the character size and address.

3. The character sizes of the stack-source-pointer and the stack-destination-pointer are compared. If they are not equal, the program is discontinued with a fault.

4. The stack-integer-counter is assigned the value for the string length calculated at compile time (see step 1).

5. The number of characters specified by the stack-integer-counter are copied from the pool array to the destination specified by the stack-destination-pointer.

Examples

REPLACE P BY "ABC"

The three EBCDIC characters ABC are copied to the destination pointed to by P.

REPLACE P:P BY "A MUCH LONGER STRING"

The 20-character EBCDIC string is copied to the destination pointed to by P. At the end of the statement, P is left pointing to the first character position after the last character copied.

REPLACE P BY 4"1234"

Because the string literal is four characters long and P is an 8-bit pointer, four 8-bit characters are copied to the destination. That is, the leftmost 32 bits of the stack-source-operand 4"123400000000", or 4"12340000", are copied to the destination.

<string literal> FOR <arithmetic expression>

If the source part is a short string literal, it is processed as follows:

1. At compile time, the string literal is stored in a one- or two-word operand. If the string literal is less than or equal to 48 bits long, it is stored, left-justified, and repeated for fill in a one-word operand. If the string literal is more than 48 bits long, it is stored, left-justified with zero fill, in a two-word operand.

2. At run time, this operand is assigned to the stack-source-operand.

3. If the arithmetic expression yields a positive value, this value is rounded to an integer, if necessary, and assigned to the stack-integer-counter; otherwise, zero is assigned to the stack-integer-counter.

4. The number of characters specified by the stack-integer-counter are copied to the destination specified by the stack-destination-pointer. If the stack-source-operand contains fewer than the specified number of characters, it is reused as many times as necessary. The character size is specified by the stack-destination-pointer. If the destination is specified by a non-character array row or array element, the character size is eight bits.

In the following examples, the first column shows a source part, and the second column shows the resulting string. A question mark (?) represents a null character.

Source Part	Result
"A" FOR 20	AAAAAAAAAAAAAAAAAAAA
"AB" FOR 20	ABABABABABABABABABAB
"ABC" FOR 20	ABCABCABCABCABCABCAB
"ABCD" FOR 20	ABCDABABCDABABCDABAB
"ABCDEF" FOR 20	ABCDEFABCDEFABCDEFAB
"ABCDEFGH" FOR 20	ABCDEFGH????ABCDEFGH

If the source part is a long string literal, it is processed as follows:

1. The stack-source-pointer is assigned the value of the pool array pointer to the long string literal, which includes the character size and address.

2. The character sizes of the stack-source-pointer and the stack-destination-pointer are compared. If they are not equal, the program is discontinued with a fault.

3. If the arithmetic expression yields a positive value, this value is rounded to an integer, if necessary, and assigned to the stack-integer-counter: otherwise, zero is assigned to the stack-integer-counter.

4. The number of characters specified by the stack-integer-counter are copied to the destination specified by the stack-destination-pointer.

The "<string literal> FOR <arithmetic expression>" syntax is undefined for a long string literal if the integerized value of the arithmetic expression is greater than the length of the string literal in characters. For example, the result of

```
REPLACE POINTER(A) BY "ABCDEFGHIJKLMNO" FOR 30
```

is undefined.

<string literal> FOR <arithmetic expression> WORDS

If the source part is a short string literal, it is processed as follows:

1. At compile time, the string literal is stored in a one- or two-word operand. If the string literal is less than or equal to 48 bits long, is stored, left-justified and repeated for fill, in a one-word operand. If the string literal is more than 48 bits long, it is stored, left-justified with zero fill, in a two-word operand.

2. At run time, this operand is assigned to the stack-source-operand.

3. If the arithmetic expression yields a positive value, this value is rounded to an integer, if necessary, and assigned to the stack-integer-counter; otherwise, zero is assigned to the stack-integer-counter.

4. The stack-destination-pointer is moved forward, if necessary, to the nearest word boundary.

5. The number of words specified by the stack-integer-counter are copied from the stack-source-operand to the destination specified by the stack-destination-pointer. If the stack-source-operand contains fewer than the specified number of words, it is reused as often as necessary.

In the following examples, the first column shows a source part, and the second column shows the resulting string. A question mark (?) represents a null character.

Source Part	Result
"ABCD" FOR 2 WORDS	ABCDABABCDAB
"ABCDEFGH" FOR 2 WORDS	ABCDEFGH????
"ABCDEFGH" FOR 3 WORDS	ABCDEFGH????ABCDEF

If the source part is a long string literal, it is processed as follows:

1. The stack-source-pointer is assigned the value of the pool array pointer to the long string literal, which includes the character size and address.

2. The character sizes of the stack-source-pointer and the stack-destination-pointer are compared. If they are not equal, the program is discontinued with a fault.

3. If the arithmetic expression yields zero or a positive value, this value is rounded to an integer, if necessary, and assigned to the stack-integer-counter; otherwise, zero is assigned to the stack-integer-counter.

4. The stack-destination-pointer is moved forward, if necessary, to the nearest word boundary.

5. The number of words specified by the stack-integer-counter are copied from the stack-source-operand to the destination indicated by the stack-destination-pointer.

The "<string literal> FOR <arithmetic expression> WORDS" syntax is undefined for a long string literal if the integerized value of the arithmetic expression is greater than the length of the string literal in 48-bit words. For example, the result of

```
REPLACE POINTER(A) BY "ABCDEFGHIJKLMNO" FOR 6 WORDS
```

is undefined.

Arithmetic Expression Source Parts

String Literals as Arithmetic Expressions

When a string literal is to be interpreted as an arithmetic expression, it must be enclosed in parentheses. Without the parentheses, the compiler interprets it as a string literal and generates code or issues syntax errors accordingly. For example,

```
REPLACE POINTER(A) BY "A" FOR 3 DIGITS
```

is an invalid statement and results in a syntax error. However,

```
REPLACE POINTER(A) BY ("A") FOR 3 DIGITS
```

is valid.

<arithmetic expression>

A source part of this form is processed as follows:

1. The arithmetic expression is evaluated, rounded to single precision, if necessary, and stored in the stack-source-operand.

2. The stack-source-operand is copied once to the destination specified by the stack-destination-pointer.

The character size of the stack-destination-pointer is irrelevant.

Examples

In the following examples, the first column shows a REPLACE statement, and the second column shows, in hexadecimal format, the resulting string.

Statement	Result
REPLACE P BY 7.5	267800000000
REPLACE P BY 3	000000000003
REPLACE P BY 1.68@@2	248540000000
REPLACE P BY ("A")	0000000000C1

<arithmetic expression> FOR <arithmetic expression>

A source part of this form is processed as follows:

1. The first arithmetic expression is evaluated, rounded to single precision, if necessary, and stored in the stack-source-operand.

2. If evaluation of the second arithmetic expression yields a positive value, this value is rounded to an integer, if necessary, and assigned to the stack-integer-counter; otherwise, zero is assigned to the stack-integer-counter.

3. The number of characters specified by the stack-integer-counter are copied from the stack-source-operand to the destination specified by the stack-destination-pointer. If the stack-source-operand contains fewer than the specified number of characters, it is reused as often as necessary. The character size is specified by the stack-destination-pointer. If the destination is specified by a non-character array row or array element, the character size is eight bits.

Examples

```
REPLACE P BY 3 FOR 1
```

Copies the character 48"00" to P. The stack-source-operand is 4"000000000003", and the leftmost character of this operand is copied to P.

```
REPLACE P BY (3).[7:48] FOR 1
```

Copies the character 48"03" to P.

```
REPLACE P BY ("A").[7:48] FOR 1
```

Copies the EBCDIC character A to P.

<arithmetic expression> FOR <arithmetic expression> WORDS

A source part of this form is processed as follows:

1. If the stack-destination-pointer points to an array of type DOUBLE and evaluation of the first arithmetic expression yields a double-precision value, this double-precision value is assigned to a two-word stack-source-operand. Otherwise, the value of the first arithmetic expression is rounded to a single-precision value, if necessary, and assigned to a one-word stack-source-operand.

2. If evaluation of the second arithmetic expression yields a positive value, this value is rounded to an integer, if necessary, and assigned to the stack-integer-counter; otherwise, zero is assigned to the stack-integer-counter.

3. The stack-destination-pointer is moved forward, if necessary, to the nearest word boundary.

4. The number of words specified by the stack-integer-counter are copied from the stack-source-operand to the destination specified by the stack-destination-pointer. If the stack-integer-counter specifies more than one word (when the stack-source-operand is single precision) or more than two words (when the stack-source-operand is double precision), then the stack-source-operand is reused as often as necessary.

Example

```
REPLACE POINTER(A) BY 0 FOR SIZE(A) WORDS
```

Copies a single-precision zero into every element of array A.

<arithmetic expression> FOR <arithmetic expression> DIGITS

A source part of this form is processed as follows:

1. The absolute value of the first arithmetic expression is rounded to an integer value, if necessary, and assigned to the stack-source-operand.

2. If evaluation of the second arithmetic expression yields a positive value, this value is rounded to an integer, if necessary, and assigned to the stack-integer-counter; otherwise, zero is assigned to the stack-integer-counter.

3. A string of 12 hexadecimal characters that represents the decimal value of the stack-source-operand is generated. If the value of stack-source-operand can be expressed in fewer than 12 digits, the string is filled on the left with zeros.

4. The N rightmost hexadecimal characters, where N is the number specified by the stack-integer-counter, are copied from this hexadecimal string to the destination. If the character size of the stack-destination-pointer is four bits, the characters are copied without change; if it is six or eight bits, the appropriate zone field is supplied.

The sign of the first arithmetic expression is placed in the external sign flip-flop (EXTF). For an explanation of the external sign flip-flop, refer to "PICTURE Declaration."

If the value of the stack-integer-counter is greater than 12, the program is discontinued with a fault. If the value is not large enough to include all nonzero decimal characters, the overflow flip-flop (OFFF) is assigned the value TRUE.

Examples

In the following examples, the first column shows the source part, the second column shows the resulting string when the destination is an 8-bit pointer, and the third column shows the resulting string when the destination is a 4-bit pointer.

Source part	8-bit destination	4-bit destination
1234 FOR 6 DIGITS	8"001234"	4"001234"
7.5 FOR 3 DIGITS	8"008"	4"008"
-10 FOR 3 DIGITS	8"010"	4"010"
1234 FOR 3 DIGITS	8"234"	4"234"

‹arithmetic expression› FOR * DIGITS

This source part functions similarly to a source part of the form

‹arithmetic expression› FOR ‹arithmetic expression› DIGITS

except that the stack-integer-counter is assigned a value equal to the minimum number of characters required to express accurately the value of the stack-source-operand.

Examples

In the following examples, the first column shows the source part, the second column shows the resulting string when the destination is an 8-bit pointer, and the third column shows the resulting string when the destination is a 4-bit pointer.

Source part	8-bit destination	4-bit destination
1234 FOR * DIGITS	8"1234"	4"1234"
7.5 FOR * DIGITS	8"8"	4"8"
-10 FOR * DIGITS	8"10"	4"10"

‹arithmetic expression› FOR ‹arithmetic expression› SDIGITS

This source part functions similarly to a source part of the form

‹arithmetic expression› FOR ‹arithmetic expression› DIGITS

except that the sign of the first arithmetic expression is also recorded. If the character size of the stack-destination-pointer is four bits, then a 4"D" (1"1101") character, indicating a negative value, or a 4"C" (1"1100") character, indicating a positive value, is copied before the first digit. If the character size is eight bits, the zone field of the rightmost digit is changed to 1"1101" for negative values or 1"1100" for positive values.

When the character size of the stack-destination-pointer is four bits, the 4"C" or 4"D" character, indicating the sign of the value, is not counted as a digit.

For example, the statement

```
REPLACE POINTER(A,4) BY -123 FOR 3 SDIGITS
```

yields

```
D123
```

Four, not three, characters are copied to the destination.

Strings produced by this form of source part can later be converted to an integer value with the correct sign using the INTEGER function. For example, the statement in the above example could be followed by the statement

```
I := INTEGER(POINTER(A,4),3)
```

after which I would contain the value -123.

Examples

In the following examples, the first column shows the source part, the second column shows the resulting string when the destination is an 8-bit pointer, and the third column shows the resulting string when the destination is a 4-bit pointer.

Source part	8-bit destination	4-bit destination
1234 FOR 6 SDIGITS	4"F0F0F1F2F3C4"	4"C001234"
-1234 FOR 6 SDIGITS	4"F0F0F1F2F3D4"	4"D001234"

‹arithmetic expression› FOR * SDIGITS

This source part functions similarly to a source part of the form

```
‹arithmetic expression› FOR ‹arithmetic expression› SDIGITS
```

except that the stack-integer-counter is assigned a value equal to the minimum number of characters required to express accurately the value of the stack-source-operand.

Examples

In the following examples, the first column shows the source part, the second column shows the resulting string when the destination is an 8-bit pointer, and the third column shows the resulting string when the destination is a 4-bit pointer.

Source part	8-bit destination	4-bit destination
1234 FOR * SDIGITS	4"F1F2F3C4"	4"C1234"
-1234 FOR * SDIGITS	4"F1F2F3D4"	4"D1234"

<arithmetic expression> FOR <count part> NUMERIC

A source part of this form is processed as follows:

1. If the arithmetic expression in the count part yields a positive value, this value is rounded to an integer, if necessary, and assigned to the stack-integer-counter; otherwise, zero is assigned to the stack-integer-counter.

2. The first arithmetic expression is evaluated, and an internal procedure is called. This procedure generates an EBCDIC character string representing the decimal value of the arithmetic expression as precisely and concisely as possible given the field width specified by the stack-integer-counter.

3. If the character size of the stack-destination-pointer is eight bits, the string is copied to the destination without translation. If the character size is six bits, the string is copied with EBCDIC-to-BCL translation. If the character size is four bits, the program is discontinued with a fault.

If a residual count does not appear in the count part, the string is copied to the destination, right-justified with blank fill, in a field with a width equal to the value of the stack-integer-counter. If a residual count does appear in the count part, the string is copied to the destination, left-justified, and the simple variable is assigned the difference between the initial value of the stack-integer-counter and the number of characters copied.

The form of the decimal representation is determined by the operand type (single or double precision), whether or not the operand value is an integer, the magnitude of the operand, the number of significant digits in its decimal representation, and the field width. The basic rule is that the number is represented as compactly as possible using integer, simple decimal, or exponential notation, as appropriate.

For example, the following source parts generate the decimal representations shown:

```
12345678 FOR 8 NUMERIC          12345678
12345678 FOR 6 NUMERIC          1.23+7
123/100 FOR N:6 NUMERIC         1.23            (N := 2)
```

<arithmetic expression> FOR * NUMERIC

This source part functions similarly to a source part of the form

<arithmetic expression> FOR <count part> NUMERIC

except that no maximum field width is specified. Thus, the internal procedure that generates the string is allowed to use as many as 36 characters to represent the decimal value of the arithmetic expression.

For example, the following source parts generate the decimal representations shown:

```
123 FOR * NUMERIC           123
1/3 FOR * NUMERIC           0.333333333333333333333
```

Pointer Expression (<source>) Source Parts

<source> FOR <arithmetic expression>

A source part of this form is processed as follows:

1. The pointer expression in the source is evaluated and assigned to the stack-source-pointer.

2. If the arithmetic expression yields a positive value, this value is rounded to an integer, if necessary, and assigned to the stack-integer-counter; otherwise, zero is assigned to the stack-integer-counter.

3. The character sizes of the stack-source-pointer and the stack-destination-pointer are compared. If they are not equal, the program is discontinued with a fault. If both the source and the destination are specified by non-character array rows or array elements, the character size of both the stack-source-pointer and the stack-destination-pointer is eight bits.

4. The number of characters specified by the stack-integer-counter are copied from the location specified by the

stack-source-pointer to the destination specified by the stack-destination-pointer.

Example

REPLACE P BY Q FOR 20

The 20 EBCDIC characters pointed to by Q are copied to the location pointed to by P.

<source> FOR <arithmetic expression> WORDS

A source part of this form is processed as follows:

1. The pointer expression in the source is evaluated and assigned to the stack-source-pointer.

2. If the arithmetic expression yields a positive value, this value is rounded to an integer, if necessary, and assigned to the stack-integer-counter; otherwise, zero is assigned to the stack-integer-counter.

3. The stack-source-pointer and the stack-destination-pointer are moved forward, if necessary, to the nearest word boundary.

4. The number of 48-bit words specified by the stack-integer-counter are copied from the location specified by the stack-source-pointer to the destination specified by the stack-destination-pointer.

The character sizes of the source and destination pointer expressions are irrelevant.

Example

REPLACE P BY Q FOR 20 WORDS

Both P and Q are advanced to the nearest word boundary, if necessary, and 20 words are copied from the location pointed to by Q to the location pointed to by P.

<source> FOR <arithmetic expression> WITH <translate table>

This construct retrieves characters from a source location, translates each character (through the use of the specified translate table) into a possibly different character with a possibly different character size, and stores each resulting character in the location indicated by the stack-destination-pointer.

The value of the pointer expression in the source points to the first character to be translated. The stack-source-pointer is initialized to this value. The stack-destination-pointer and the stack-source-pointer need not have the same character size. Instead, the stack-source-pointer must have a character size equal to that of the characters being translated, and the stack-destination-pointer must have a character size equal to that of the resulting translated characters.

The value of the arithmetic expression indicates the number of characters to be translated and written to the destination. This value is integerized, if necessary, and assigned to the stack-integer-counter. The stack-auxiliary-pointer is initialized to point to the first character of the first word of the translate table, and its character size is absent. Normally, when a pointer is used and its character size is absent, a default value of six or eight is used, depending on the default character type. However, the character size of the pointer used to initialize the stack-auxiliary-pointer is irrelevant. The translate table is not examined sequentially (one character at a time); instead, the data in the table is accessed by special indexing techniques implemented in the hardware, as follows:

<intrinsic translate table>

If the translate table is of this form, the stack-auxiliary-pointer is initialized to point to the appropriate intrinsic translate table. The function of each translate table can be deduced from its name. For example, the HEXTOEBCDIC table is used to translate characters from hexadecimal to EBCDIC.

<translate table identifier>

If the translate table is of this form, a translate table must have been declared in a TRANSLATETABLE declaration. For a detailed discussion regarding the construction of a translate table, refer to "TRANSLATETABLE Declaration."

‹subscripted variable›

If the translate table is of this form, the programmer is responsible for creating a properly structured translate table that is contained entirely in the array row and begins with the word in the array row indicated by the subscripted variable. Refer to "Translate Table Indexing" under "TRANSLATETABLE Declaration."

Examples

```
REPLACE POINTER(B,4) BY POINTER(A,8) FOR 20 WITH EBCDICTOHEX

    A = 8"0123456789ABCDEFGHIJ"
    B = 4"0123456789ABCDEFFFFF"


REPLACE POINTER(B,7) BY POINTER(A,8) FOR 14 WITH EBCDICTOASCII

    A = 4"F0F1F2F3F4F5F6F7F8F9C1C2C3C4"
    B = 4"30313233343536373839414243 44"


REPLACE POINTER(B,8) BY POINTER(A,4) FOR 12 WITH HEXTOEBCDIC

    A = 8"012345" = 4"F0F1F2F3F4F5"
    B = 8"F0F1F2F3F4F5"
```

‹source› WITH ‹picture identifier›

The character data specified by the source (which must be a pointer) is processed under control of the picture specified by the picture identifier. The source and destination pointers must be 4-bit, 8-bit, or word-oriented. If the source is a word-oriented pointer, it is changed to a 4-bit pointer if the destination is a 4-bit pointer; otherwise, it is changed to an 8-bit pointer. If the destination is a word-oriented pointer, it is changed to a 4-bit pointer if the source is a 4-bit pointer; otherwise, it is changed to an 8-bit pointer. If neither the source nor the destination pointer is a word-oriented pointer, the source and destination pointers must either both be 4-bit pointers or both be 8-bit pointers. Details regarding the formation and action of pictures are described under "PICTURE Declaration."

Source Parts with Boolean Conditions

The next eight forms of the source part copy characters from the source to the destination until a source character fails or passes the specified test. The number of characters copied can also be limited by an optional count part. For more information on the use of these Boolean conditions, refer to "SCAN Statement."

In the source parts containing a condition of either of the forms

```
WHILE <relational operator> <arithmetic expression>
UNTIL <relational operator> <arithmetic expression>
```

the source characters are tested against bits [7:8], [5:6], or [3:4] of the arithmetic expression, depending on the character size of the source. In all cases, the stack-source-pointer is left pointing to the character that failed or passed the test.

The count part consists of an arithmetic expression and, optionally, a residual count. The value of the arithmetic expression specifies the maximum number of characters to be copied. The residual count, when it appears, is a simple variable in which is stored the difference between the value of the arithmetic expression and the number of source-part characters copied.

<source> WHILE <relational operator> <arithmetic expression>

The stack-source-pointer is initialized to the source pointer. Characters are then copied from the source to the destination as long as source characters pass the test.

A paged (segmented) array error fault could occur at run time if all of the following conditions occur:

1. The stack-destination-pointer references the first character beyond the end of the destination array.
2. The stack-source-pointer references the first character to fail the test.
3. The stack-integer-counter is nonzero.

Example

```
REPLACE P BY Q WHILE NEQ " "

    Q = "LONG STRING"
    P = "LONG"
```

<source> UNTIL <relational operator> <arithmetic expression>

The stack-source-pointer is initialized to the source pointer. Characters are then copied from the source to the destination until a source character passes the test.

A paged (segmented) array error fault could occur at run time if all of the following conditions occur:

1. The stack-destination-pointer references the first character beyond the end of the destination array.
2. The stack-source-pointer references the first character to pass the test.
3. The stack-integer-counter is nonzero.

Example

```
REPLACE P BY Q UNTIL = "."

    Q = "FILE/TITLE ON PACK.XXX"
    P = "FILE/TITLE ON PACK"
```

<source> WHILE IN <truth set table>

The stack-source-pointer is initialized to the source pointer. Characters are then copied from the source to the destination as long as the source characters are members of the truth set. For further information on truth sets, see "TRUTHSET Declaration."

Example

```
REPLACE P BY Q WHILE IN ALPHA8

    Q = "ABCD1234.56"
    P = "ABCD1234"
```

<source> UNTIL IN <truth set table>

The stack-source-pointer is initialized to the source pointer. Characters are then copied from the source to the destination until a source character is encountered that is a member of the truth set. For further information on truth sets, see "TRUTHSET Declaration."

Example

```
REPLACE P BY Q UNTIL IN ALPHA8

    Q = ",  *,$1234"
    P = ",  *,$"
```

<source> FOR <count part> WHILE <relational operator> <arithmetic expression>

The stack-source-pointer is initialized to the source pointer. The stack-integer-counter is initialized to the value of the arithmetic expression in the count part. Characters are then copied from the source to the destination and the stack-integer-counter is decremented for each character copied as long as the stack-integer-counter is not zero and the source characters pass the test.

A paged (segmented) array error fault could occur at run time if all of the following conditions occur:

1. The stack-destination-pointer references the first character beyond the end of the destination array.

2. The stack-source-pointer references the first character to fail the test.

3. The stack-integer-counter is nonzero.

Example

```
REPLACE P BY Q FOR N:11 WHILE NEQ " "

    Q = "LONG STRING"
    P = "LONG"          (and N = 7)
```

<source> FOR <count part> UNTIL <relational operator> <arithmetic expression>

The stack-source-pointer is initialized to the source pointer. The stack-integer-counter is initialized to the value of the arithmetic expression in the count part. Characters are then copied from the source to the destination and the stack-integer-counter is decremented for each character copied until either the stack-integer-counter is zero or a source character passes the test.

A paged (segmented) array error fault could occur at run time if all of the following conditions occur:

1. The stack-destination-pointer references the first character beyond the end of the destination array.

2. The stack-source-pointer references the first character to pass the test.

3. The stack-integer-counter is nonzero.

Example

```
REPLACE P BY Q FOR N:22 UNTIL = "."

    Q = "FILE/TITLE ON PACK.XXX"
    P = "FILE/TITLE ON PACK"      (and N = 4)
```

<source> FOR <count part> WHILE IN <truth set table>

The stack-source-pointer is initialized to the source pointer. The stack-integer-counter is initialized to the value of the arithmetic expression in the count part. Characters are then copied from the source to the destination and the stack-integer-counter is decremented for each character copied as long as the stack-integer-counter is not zero and the source characters are members of the truth set. For further information on truth sets, see "TRUTHSET Declaration."

Example

```
REPLACE P BY Q FOR N:11 WHILE IN ALPHA8

    Q = "ABCD1234.56"
    P = "ABCD1234"      (and N = 3)
```

<source> FOR <count part> UNTIL IN <truth set table>

The stack-source-pointer is initialized to the source pointer. The stack-integer-counter is initialized to the value of the arithmetic expression in the count part. Characters are then copied from the source to the destination and the stack-integer-counter is decremented for each character copied until either the stack-integer-counter is zero or a source character is a member of the truth set. For further information on truth sets, see "TRUTHSET Declaration."

Example

```
REPLACE P BY Q FOR N:10 UNTIL IN ALPHA8

    Q = ",  *,$1234"
    P = ",  *,$"      (and N = 4)
```

Other Source Parts

<pointer-valued attribute>

The string of characters that forms the value of the pointer-valued attribute is copied to the location indicated by the stack-destination-pointer. The string of characters is formatted in the destination array row in a form suitable to serve in a replace pointer-valued attribute statement that assigns a value to the same attribute. The character string ends with an EBCDIC period (8"."). For example, if P is a pointer identifier and F1 and F2 are file identifiers, then the following sequence of statements is valid:

```
REPLACE P BY F1.TITLE;
REPLACE F2.TITLE BY P;
```

All pointer-valued attributes have a character size of eight bits. At run time, if the destination pointer does not also have a character size of eight bits, the program is discontinued with a fault.

If a pointer-valued attribute appears as the source part in a REPLACE statement, a call is made on a Master Control Program (MCP) procedure to perform this part of the REPLACE statement.

<string expression>

When a string expression appears as the source part in a REPLACE statement, it is evaluated and stored in a pool array. The stack-source-pointer is initialized to point to the first character of the string in the pool array. The entire string is copied to the destination.

Example

```
STR := "ABCDEFG";
REPLACE P BY STR;
```

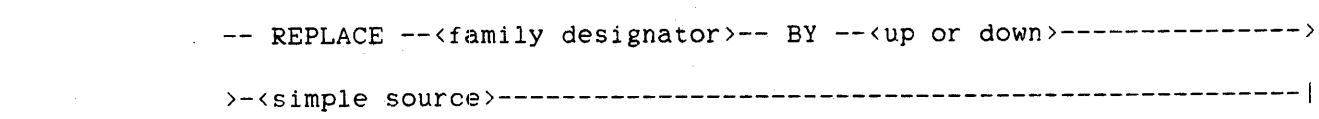
Copies the EBCDIC string ABCDEFG to the location pointed to by P.

REPLACE FAMILY-CHANGE STATEMENT

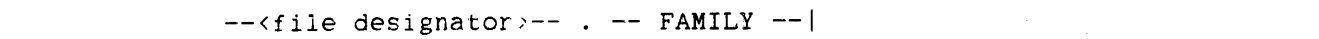
The replace family-change statement adds stations to, or deletes stations from, the family of an open, remote file.

Syntax

<replace family-change statement>

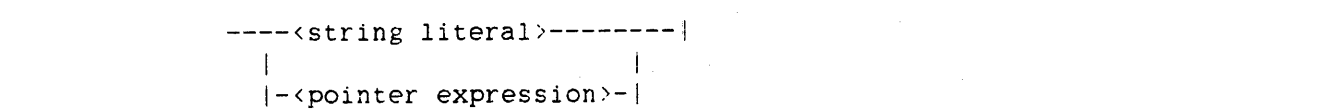


<family designator>



<up or down>

<simple source>



See also
<file designator> 189

Semantics

The file designator specifies the file whose FAMILY file attribute is to be changed. If a station is to be added to the family, <up or down> is "*+". If a station is to be deleted from the family, <up or down> is "*-". The simple source specifies the title of the station involved. Because the simple source is a value for a pointer-valued attribute, its value must end with a period (.). For more specific information, refer to "Replace Pointer-Valued Attribute Statement" in this chapter.

Pragmatics

If the simple source does not reference a valid station title (as specified for the current network in the Network Definition Language II (NDLII) description), then the replace family-change statement has the following effects:

1. <file designator>.FAMILY is unchanged.

2. <file designator>.ATTERR is given the value TRUE, and <file designator>.ATTYPE is set to the appropriate value.

3. An appropriate error message is displayed on the Operator Display Terminal (ODT).

4. The program continues.

If <up or down> is "*-" and the simple source specifies a valid station as defined by the current NDLII description, but the specified station is not currently a member of the family, then the replace family-change statement makes no change to the specified family. No error condition is indicated (such a situation is not considered to be an error), and control passes to the next statement of the program.

If, after execution of a replace family-change statement, the remote file is closed with release and later re-opened, the family reverts to its NDLII-specified value. However, if the remote file is closed with retention and later re-opened, the family retains its changed value.

When the replace family-change statement is executed, a call is made on a Master Control Program (MCP) procedure to perform the desired function.

Examples

```
REPLACE NETWORK.FAMILY BY *+ "ACCT7."
```

Adds the station with the title "ACCT7" to the family of the remote file NETWORK.

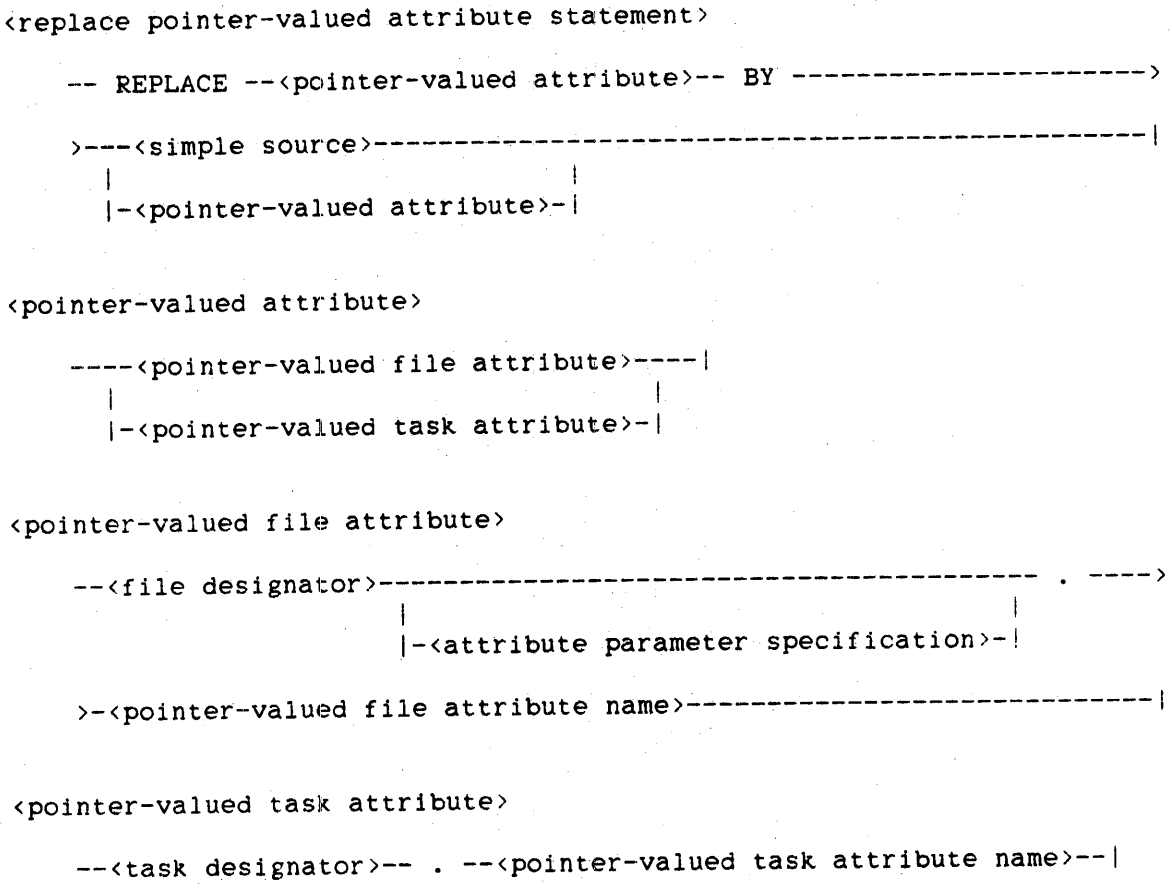
```
REPLACE DATACOLLECTORS.FAMILY BY *- STATIONNAMEPTR
```

Deletes the station with the title given by the pointer STATIONNAMEPTR from the family of the remote file DATACOLLECTORS.

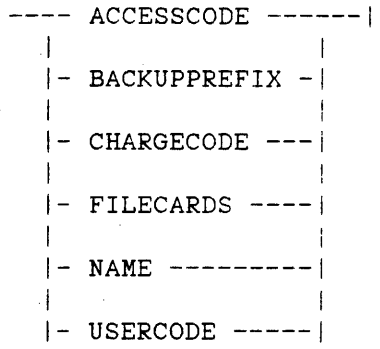
REPLACE POINTER-VALUED ATTRIBUTE STATEMENT

The replace pointer-valued attribute statement changes the value of a pointer-valued attribute.

Syntax



<pointer-valued task attribute name>



See also

Semantics

The simple source specifies the string of characters that is to become the new value of the pointer-valued attribute.

If the simple source is a string literal, the last character of the string literal must be a period (.). The "effective" part of the string literal is terminated by the first period in the string. A maximum string length is associated with each pointer-valued attribute. If the effective part of the string literal has a string length that is greater than the maximum length allowed for the pointer-valued attribute, then the new value of the pointer-valued attribute is the value of the string literal truncated on the right to the required length.

If the simple source is a pointer expression, the pointer expression must point to the string of characters that is to become the new value of the pointer-valued attribute. Starting with the first character pointed to by the pointer expression, characters are copied as the new value of the pointer-valued attribute until a period is encountered, the maximum number of characters for the attribute are copied, or the end of the array row is encountered. The last case results in a run-time error.

If a pointer-valued task attribute is used as the destination and the source is a pointer-valued attribute, the source attribute and the destination attribute must be the same attribute. If a pointer-valued

file attribute is used as the destination, the source must be a simple source.

Pragmatics

When the replace pointer-valued attribute statement is executed, a call is made on a Master Control Program (MCP) procedure to perform the desired function.

Examples

```
REPLACE FYLE.TITLE BY "MASTER/PAYROLL."
```

The TITLE attribute of file FYLE is assigned "MASTER/PAYROLL.".

```
REPLACE TSK.NAME BY "SECOND/STACK."
```

The NAME attribute of the task TSK is assigned "SECOND/STACK.".

```
REPLACE T.NAME BY TS.NAME
```

The NAME attribute of task T is assigned the value of the NAME attribute of task TS.

RESET STATEMENT

The RESET statement sets the happened state of the designated event to FALSE (not happened).

Syntax

<reset statement>

```
-- RESET -- ( --<event designator>-- ) --|
```

See also
<event designator>. 78

Semantics

The RESET statement does not change the status of any tasks waiting on the event.

Pragmatics

If a RESET statement is used after a WAIT statement to restore the happened state of an event to FALSE (not happened), a period of time exists during which another task could cause the event. For this reason, a WAITANDRESET statement might prove to be more useful than a WAIT statement followed by a RESET statement.

Examples

```
RESET(EVNT)
```

Sets the happened state of the event EVNT to FALSE (not happened).

```
RESET(EVNTARAY[INDX])
```

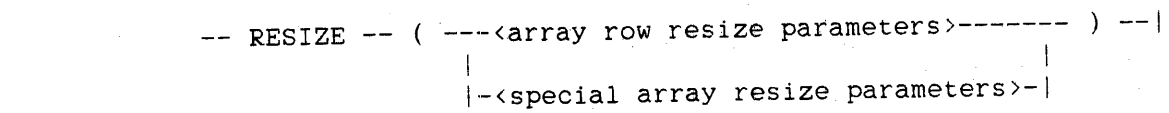
Sets the happened state of the event designated by EVNTARAY[INDX] to FALSE (not happened).

RESIZE STATEMENT

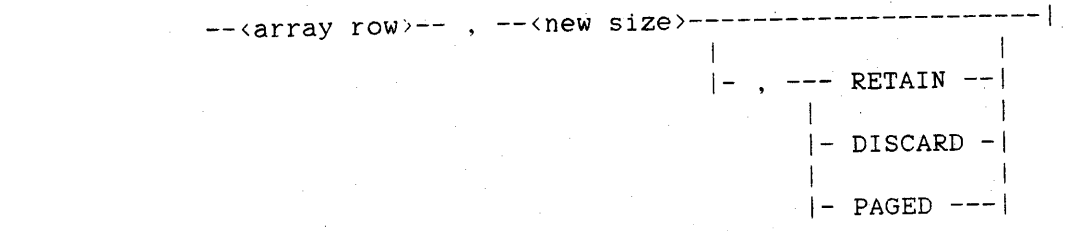
The RESIZE statement modifies the size of the designated array row, subarray, or array.

Syntax

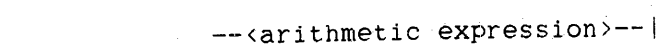
<resize statement>



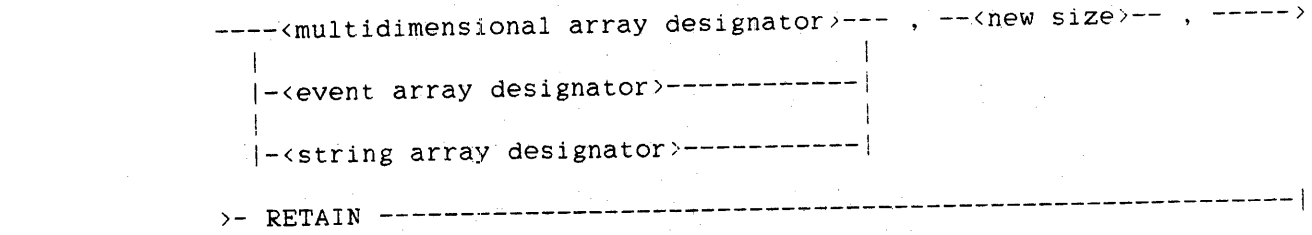
<array row resize parameters>



<new size>



<special array resize parameters>



<multidimensional array designator>

An <array designator> with dimensionality greater than one; that is, a multidimensional <array name>, optionally suffixed by a <subarray selector> with at least two asterisks (*).

See also

Semantics

The RESIZE statement changes the upper bounds of the appropriate dimensions of an array. The resize parameters designate the array row or rows to be changed and the new sizes of those rows.

There are two forms of the RESIZE statement: the <array row resize parameters> form and the <special array resize parameters> form.

<array row resize parameters>

In this form of RESIZE statement, the first parameter is an array row, a one-dimensional array whose elements are of some array class: BOOLEAN, COMPLEX, DOUBLE, INTEGER, REAL, or a character type.

The RESIZE statement causes the size of the designated row to be modified as specified by the new size. The resize options have the following effects:

DISCARD

The current contents of the array row are discarded--the new contents of the array row are undefined.

RETAIN

As much of the current information in the array row is retained as fits in the new size. If the new size is smaller than the old, data in the lost elements is discarded. If the new size is larger, the data in the new elements is undefined.

PAGED

The resized array is to be a paged (segmented) array. The new paged array is considered to be "touched" (referenced) after the resize is complete. "PAGED" also implies "RETAIN". (Note: If a program is compiled to use this feature and is run on a pre-Mark 3.5 Master Control Program (MCP), an attempt is made to resize the array, leaving it unpaged (unsegmented). This

attempt might succeed, or might result in "WORDS REQUIRED" action, or might fail with an "ARRAY TOO LONG" error.)

If no third parameter appears, DISCARD is assumed.

If the new array row size is less than the old, any pointer variable that now points beyond the end of the array row is set to the uninitialized state.

If the array row designates a value array or a referenced paged (segmented) array, the program is discontinued with a "BAD RESIZE/DEALLOCATE" error.

The value of the new size is integerized with rounding, if necessary, to specify a new size, Sn, which is interpreted as a number of elements for the resized array row. If the array row is an original array, then its size is changed to Sn. If the array row is a referred array and the original array has a different element size, the original array is resized to have just enough elements to hold Sn elements of the referred array row.

When an original array is resized, any referred arrays with element widths different from those of the original array are assigned the size they would have had if the original array had been declared at its new size and the referred array had been created from the original by array equivalence or array reference assignment.

When a resize of a referred array causes a resize of an original array, the size calculations are performed with the element widths Wn for the resized referred array and Wo for the original array. The new size of the original array, So, is

```
So := (Sn * Wn + Wo - 1) DIV Wo
```

So * Wo can exceed Sn * Wn. The new size, Sr, of any referred array with element width Wr that is based on the resized original array is

```
Sr := (So * Wo) DIV Wr
```

Wr = Wn for the explicitly resized referred array, and the net calculation is

```
Sr := (((Sn * Wn + Wo - 1) DIV Wo) * Wo) DIV Wn
```

which can exceed Sn. That is, the explicitly resized array row can be slightly larger than requested, if the original array has a wider element width. For example, if arrays RA and EA are declared as follows:

```
REAL ARRAY RA[0:5];
EBCDIC ARRAY EA[0] = RA;
```

then EA contains 36 elements. If the statement

```
RESIZE(EA,50,RETAIN)
```

is executed, the original array, RA, is resized to a new size of 9 words, calculated from

```
So = (50 * 8 + 48 - 1) DIV 48 = 9
```

The actual new size of the referred array, EA, is then 54, calculated from

```
Sr = (9 * 48) DIV 8 = 54
```

Because the second calculation truncates, Sr * Wr can be less than So * Wo, just as with array row equivalence or array reference assignment.

For example, consider the statement

```
RESIZE(H,29,DISCARD)
```

where H is a hexadecimal array row. The following table shows the size assigned to referred arrays for several combinations of referred and original classes. (The diagonal of the table shows the size assigned to each original.)

	Referred			
Original	Hexadecimal	EBCDIC	Real	Double
Hexadecimal	29	14	2	1
EBCDIC	30	15	2	1
Real	36	18	3	1
Double	48	24	4	2

<special array resize parameters>

In this form of RESIZE statement, the first parameter is an array whose elements do not have an array class. Events and strings are special classes of objects. A multidimensional array can be considered an array of arrays.

The RESIZE statement sets the size of the parameter array to the new size, unless the new size is less than the existing size, in which case the RESIZE statement is ignored and the warning message "ATTEMPTED DOWNWARD RESIZE IGNORED" is generated.

If the first parameter includes a subarray selector, the dimension corresponding to the first asterisk (*) is changed; otherwise, the first dimension of the designated array is changed. Whenever a higher-order dimension of an array is enlarged, new subarrays are created with the same dimensions as in the original ARRAY declaration. Any existing subarrays are unaffected by the resize operation. For example, given the declaration

```
DOUBLE ARRAY A[1:2,0:5,-4:4]
```

the statement

```
RESIZE(A[1,*,*],8,RETAIN)
```

increases the size of A[1,*,*] from 6 to 8 (new bound pair = 0:7); it causes array rows A[1,6,*] and A[1,7,*] to be established as one-dimensional double arrays of size 9 (even if all existing rows of A had already been resized to some other size).

If the array to be resized is specified by a <multidimensional array designator>, then the new subarrays have the same type as the original array; their contents are undefined.

If the array to be resized is specified by a <event array designator>, then enlarging the low-order dimension creates new events with the happened state equal to FALSE (not happened) and the available state equal to TRUE (available); existing elements are unaffected.

If the array to be resized is specified by a <string array designator>, then enlarging the low-order dimension creates new empty strings; existing elements are unaffected.

Pragmatics

RETAIN is typically used for an array being employed as a stack. When the array is not large enough to accept a "push" of the next entry, the array can be enlarged without losing the data already present. If no data has been assigned to the array, or if the old data is no longer relevant, DISCARD is more efficient for the resize of an array row.

Note that if the RESIZE statement is used on other than the highest-order dimension of an array, the array can contain subarrays of different sizes.

When the initial size of an array is to be chosen dynamically by a program, the most efficient technique is to declare the array with a variable upper bound, the bound being a global variable or a parameter computed before the procedure or block is entered.

It is not possible to resize a referenced paged array. An array is referenced (touched) if a statement referring to the array has been executed in the block. An array row is paged if its declared length exceeds the array segmentation start size, unless it is declared LONG or DIRECT. The array segmentation start size is typically 1024 words. The start size can be displayed or set with the Operator Display Terminal (ODT) command SEGARRAYSTART. Note that if an array is initially declared shorter than the array segmentation start size, then it is unpaged, and resizing it larger without using the PAGED option does not cause it to become paged.

An array that is initially declared to be shorter than the array segmentation start size and is, therefore, an unpaged array can be resized to become a paged array by using the PAGED option. The new paged array is considered to be "touched" (referenced) after the resize is complete. This implies that the array can never be resized again. (Note: If a program is compiled to use this feature and is run on a pre-Mark 3.5 Master Control Program (MCP), an attempt is made to resize the array, leaving it unpaged (unsegmented). This attempt might succeed, or might result in "WORDS REQUIRED" action, or might fail with an "ARRAY TOO LONG" error.

The PAGED option is useful in cases where the desired size of a paged array is not known at the time the array is declared. The PAGED option offers an alternative to declaring an array larger than the array segmentation start size and avoiding references to the array until the desired size is known. The PAGED option achieves the same results and makes errors less likely. The only restrictions on the use of the PAGED option are that the array row must not already be paged and the new size of the resized array must be larger than the array segmentation start size.

Examples

```
RESIZE(A,NEWSZ,DISCARD)
```

The size of one-dimensional array A is changed to NEWSZ, and the previous contents of A are discarded.

```
RESIZE(ARAY,NEWSZ,PAGED)
```

The size of one-dimensional array ARAY is changed to NEWSZ, and ARAY is changed to a paged array. The contents of ARAY are retained.

```
RESIZE(INPUTDATA,F.MAXRECSIZE,DISCARD)
```

The size of one-dimensional array INPUTDATA is changed to equal the value of the MAXRECSIZE attribute of file F. The previous contents of INPUTDATA are discarded.

```
RESIZE(A[2,*],5,DISCARD)
```

The size of the specified row of array A is changed to 5, and the previous contents of that row are discarded. The other rows of A are not affected.

```
RESIZE(A,SIZE(A)+100,RETAIN)
```

The size of one-dimensional array A is increased by 100 elements, and the previous contents of A are retained.

```
RESIZE(EVENTARRAY,20,RETAIN)
```

The size of the one-dimensional event array EVENTARRAY is changed to 20, and the previous contents of the array are retained. Note that RETAIN must be specified for event arrays.

```
RESIZE(STUFF[I,*,*],M,RETAIN)
```

The size of the second dimension of array STUFF is changed to M. New array rows are created for the new size of the second dimension. The previous contents of the array are retained.

```
RESIZE(STUFF[I,J,*],N,RETAIN)  % RESIZE an array row
```

The size of the specified row of array STUFF is changed to N, and the previous contents of that row are retained.

REWIND STATEMENT

The REWIND statement causes the designated file to be closed and the file buffer areas to be returned to the system.

Syntax

<rewind statement>

```
-- REWIND -- ( --<file designator>-- ) --|
```

See also
<file designator> 189

Semantics

If the file is a paper tape or magnetic tape file, it is rewound. For disk files, the record pointer is set to the first record of the file.

Card reader, card punch, and line printer units are released from program control. When the REWIND statement is used for a magnetic tape file that is positioned past the first reel of a multireel file, the second and subsequent reels are released from program control. Other kinds of units remain under program control.

For paper tape files, the REWIND statement can be used only on input.

Pragmatics

For random access files, if the file is to be reused immediately, the statement "SEEK(<file designator>[0])" positions the file at its first record while avoiding the overhead of closing the file and then re-opening it. For more information, refer to "SEEK Statement."

Pragmatics

Because array and file parameters cannot be call-by-value, procedures with array or file parameters cannot be invoked with a RUN statement. Also, a procedure that has a pointer as a parameter, whether or not it is specified as call-by-value, cannot be invoked with a RUN statement.

See also

Examples

```
RUN SIMPL [TSK]
```

Invokes procedure SIMPL, which has no parameters, as an independent program. The task TSK is copied by the MCP for SIMPL to use as its task variable.

```
RUN DOOER(X,Y,Z,"ABCD") [TSKARRAY[INDEX]]
```

Invokes procedure DOOER as an independent program, passing the four parameters X, Y, Z, and the string literal "ABCD". The task designated by TSKARRAY[INDEX] is used by DOOER as its task variable.

SCAN STATEMENT

The SCAN statement examines a contiguous portion of character data in an array row, one character at a time, in a left-to-right direction.

Syntax

<scan statement>

```
-- SCAN --<source>--<scan part>--|
```

NOTE

The full syntax for the SCAN statement is presented in the description of the REPLACE statement.

See also
<scan part> 381
<source>. 381

Semantics

The source is always a pointer expression, and at the completion of the SCAN statement, the final value of the stack-source-pointer can be stored in a pointer variable.

The scan part is basically a testing operation that determines when the SCAN statement is to stop. The scan part can specify that scanning is to stop after a given number of source characters, or when a source character fails or passes a specified test.

The count part is used in a scan part when a limited number of source characters are to be scanned. <residual count> can be used, in which case the value of the remaining count is stored in the specified simple arithmetic variable at the completion of the SCAN statement.

The relational operator in <condition> specifies the comparison to be made between the arithmetic expression and the source characters. The arithmetic expression can be of any valid form, but most often takes the form of a one-character string literal.

Before the scan operation begins, the arithmetic expression in <condition> is evaluated and the value of bits [7:8], [5:6], or [3:4] (depending on the character size of the source pointer) of the arithmetic expression is assigned to the stack-source-operand.

Examples of SCAN Statement Syntax

```
SCAN PTR WHILE = " "

SCAN PTR UNTIL NEQ 48"00"

SCAN PTR:PTR WHILE IN ALPHA

SCAN PTR UNTIL IN ALPHA8

SCAN PTR:PTR WHILE IN ACCEPTABLE[0]

SCAN PTR FOR 50 WHILE > "Z"

SCAN PTR:PTR FOR X:80 UNTIL = "."

SCAN PTR FOR RMNDR:960 WHILE NEQ 48"1D"

SCAN PTR:PTR FOR ZED:ZED WHILE IN ALPHA8

SCAN PTR FOR 80 UNTIL IN GOODSTUFF[5]
```

<scan part> Combinations

The formal syntax of the <scan part> can be reduced to the following combinations:

```
WHILE <relational operator> <arithmetic expression>
UNTIL <relational operator> <arithmetic expression>

WHILE IN <truth set table>
UNTIL IN <truth set table>

FOR <count part> WHILE <relational operator> <arithmetic expression>
FOR <count part> UNTIL <relational operator> <arithmetic expression>

FOR <count part> WHILE IN <truth set table>
FOR <count part> UNTIL IN <truth set table>
```

The remainder of the information about the SCAN statement is organized according to the above combinations. Because all combinations of the SCAN statement begin with <source>, each description of a combination begins with the assumption that the stack-source-pointer has been initialized to the source pointer.

The scan parts that contain a count part scan (examine) source characters until either the number of characters specified by the arithmetic expression in the count part have been examined or a source character fails or passes the test specified by the <condition> syntax. The scan parts that do not contain a count part examine source characters until either a source character fails or passes the test specified by the <condition> syntax or the end of the array is reached. If the end of the array is reached, the program is discontinued with a paged (segmented) array error.

Scan Parts Without Count Parts

WHILE <relational operator> <arithmetic expression>

Characters are scanned as long as they pass the test. For example,

```
SCAN P WHILE NEQ "."
```

scans the characters pointed to by P as long as a period (.) is not encountered.

UNTIL <relational operator> <arithmetic expression>

Characters are scanned until a source character passes the test. For example,

```
SCAN P:P UNTIL = " "
```

scans the characters pointed to by P until a blank character is encountered. P is updated to point to the blank character that passed the test.

WHILE IN <truth set table>

Characters are scanned as long as they are members of the truth set. For example,

```
SCAN P:P WHILE IN ALPHA8
```

scans the characters pointed to by P as long as they are members of the truth set ALPHA8. P is updated to point to the first character that is not a member of ALPHA8.

UNTIL IN <truth set table>

Characters are scanned until a source character is found that is a member of the truth set. For example,

```
SCAN P:P UNTIL IN ALPHA8
```

scans the characters pointed to by P until a member of the truth set ALPHA8 is encountered. P is updated to point to the first character that is a member of ALPHA8.

Scan Parts with Count Parts

FOR <count part> WHILE <relational operator> <arithmetic expression>

The stack-integer-counter is initialized to the value of the arithmetic expression in the count part. Characters are then scanned and the stack-integer-counter is decremented for each character as long as the stack-integer-counter is not zero and a source character passes the test. For example,

```
SCAN P FOR N:20 WHILE NEQ "."
```

scans the first 20 characters pointed to by P as long as a period (.) is not encountered. Because N reflects how many of the 20 characters have yet to be scanned, it can be used to determine whether a period was encountered and, if so, where the period is.

FOR <count part> UNTIL <relational operator> <arithmetic expression>

The stack-integer-counter is initialized to the value of the arithmetic expression in the count part. Characters are then scanned and the stack-integer-counter is decremented for each character until either the stack-integer-counter is zero or a source character passes the test. For example,

```
SCAN P:P FOR N:N UNTIL NEQ " "
```

scans the first N characters pointed to by P until the first nonblank character is encountered. If, when the statement is invoked, the value of N is the number of characters between P and the end of the array row, then because both P and N are updated in this statement, at the completion of the statement, N gives the number of characters between the updated P and the end of the array row.

FOR <count part> WHILE IN <truth set table>

The stack-integer-counter is initialized to the value of the arithmetic expression in the count part. Characters are then scanned and the stack-integer-counter is decremented for each character as long as the stack-integer-counter is not zero and source characters are members of the truth set. For further information on truth sets, see "TRUTHSET Declaration."

For example,

```
SCAN P:P FOR N:20 WHILE IN ALPHA8
```

scans the first 20 characters pointed to by P as long as they are members of the truth set ALPHA8. P is updated to point to the first character that is not a member of ALPHA8, or, if all of the 20 characters scanned are members of ALPHA8, to the character that is 20 characters beyond the initial position of P. N is assigned the number of characters yet to be scanned.

FOR <count part> UNTIL IN <truth set table>

The stack-integer-counter is initialized to the value of the arithmetic expression in the count part. Characters are then scanned and the stack-integer-counter is decremented for each character until either the stack-integer-counter is zero or a source character is a member of the truth set. For further information on truth sets, see "TRUTHSET Declaration."

For example,

```
SCAN P:P FOR 20 UNTIL IN ALPHA8
```

scans the first 20 characters pointed to by P until a member of the truth set ALPHA8 is encountered. P is updated to point to the first character that is a member of ALPHA8, or, if none of the 20 characters scanned are members of ALPHA8, to the character that is 20 characters beyond the initial position of P.

SEEK STATEMENT

The SEEK statement positions the record pointer for the designated file at the specified record. This record is read or written by the next serial I/O operation.

Syntax

<seek statement>

```
-- SEEK -- ( --<file designator>-- [ --<record number>-- ] -- ) --|
```

<record number>

```
--<arithmetic expression>--|
```

See also
<file designator> 189

Semantics

A serial I/O operation is a READ statement or WRITE statement that does not include a record number in the <record number or carriage control> part. The SEEK statement does not affect any nonserial I/O statements. The value of the record pointer is not saved when the file is closed.

The SEEK statement can be used as a Boolean function. When the statement fails, the value TRUE is returned. When the statement is successful, the value FALSE is returned. Specifically, the SEEK statement returns a value identical to that returned by the file attribute STATE. For more information, refer to the discussion of the STATE attribute in the "I/O Subsystem Reference Manual."

The file designator must not reference a direct file or a direct switch file.

Example

```
SEEK(FILEA[X+2*Y])
```

Positions the record pointer of file FILEA to record number X + 2 * Y.

SET STATEMENT

The SET statement sets the happened state of the designated event to TRUE (happened).

Syntax

<set statement>

```
-- SET -- ( --<event designator>-- ) --|
```

See also
<event designator>. 78

Semantics

The SET statement does not activate any tasks waiting on the event.

Pragmatics

To set the happened state of an event to TRUE (happened) and activate the tasks waiting on the event, use the CAUSE statement.

Examples

SET(EVNT)

Sets the happened state of EVNT to TRUE (happened).

SET(EVNTARAY[INDX])

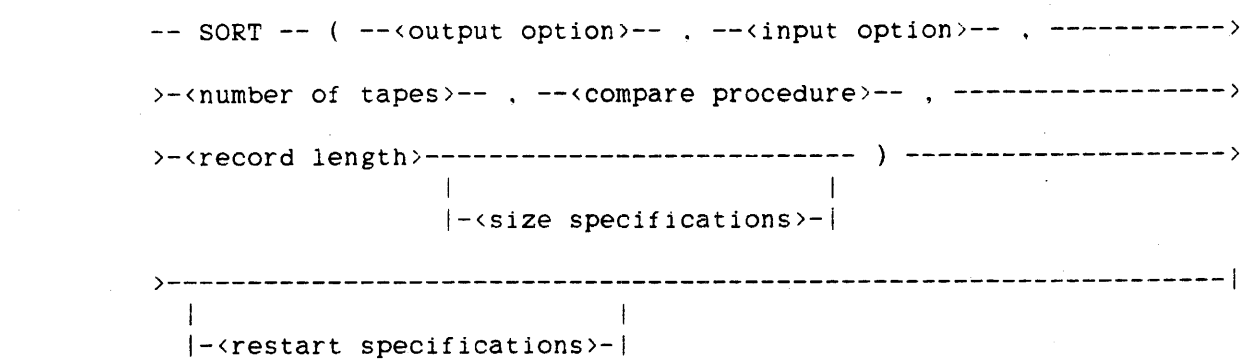
Sets the happened state of the event designated by EVNTARAY[INDX] to TRUE (happened).

SORT STATEMENT

The SORT statement invokes the sort intrinsic. which provides a means for designated data to be sorted and placed in a file or returned to a procedure.

Syntax

<sort statement>



<output option>

<output procedure>

<input option>

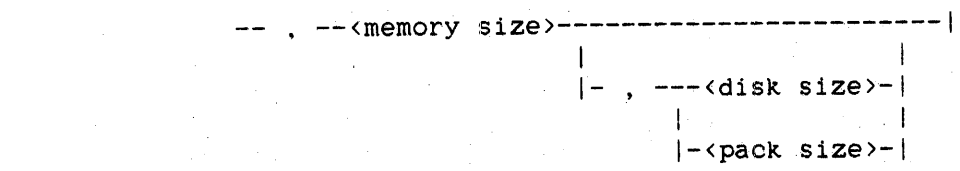
<input procedure>

<number of tapes>

<compare procedure>

<record length>

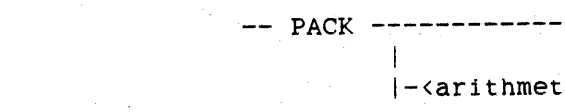
<size specifications>



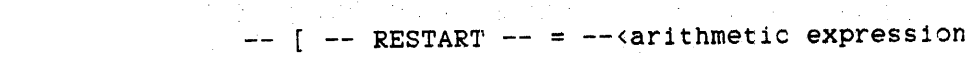
<memory size>

<disk size>

<pack size>



<restart specifications>



See also

Semantics

The data to be sorted is indicated by the input option. The output option indicates where the sorted data is to be placed. The order in which the data is sorted is determined by the compare procedure.

<output option>

If a file designator is specified as the output option, the sort intrinsic writes the sorted output to this file. When sorting is completed, the sort intrinsic closes the file. If the file is a disk file for which the file attribute SAVEFACTOR has a nonzero value, it is closed and locked. The output file must not be open when it is passed to the sort intrinsic by the program.

If an output procedure is specified as the output option, the sort intrinsic calls the output procedure once for each sorted record and once to allow end-of-output action. This procedure must be untyped, must not be declared EXTERNAL, and must have two parameters. The first parameter must be a call-by-value Boolean variable, and the second parameter must be a one-dimensional array with a lower bound of zero. The Boolean parameter is FALSE as long as the second parameter contains a sorted record. When all records are returned, the first parameter is TRUE and the second parameter must not be accessed.

The following is an example of an output procedure:

```
PROCEDURE OUTPROC(B,A);
VALUE B;
BOOLEAN B;
ARRAY A[0];
   BEGIN
   IF B THEN
      CLOSE(FILEID,RELEASE)
   ELSE
      WRITE(FILEID,RECSIZE,A[*]);
   END OUTPROC;
```

<input option>

If a file designator is used as the input option, the file supplies input records to the sort intrinsic. This file is closed after the last record is read. Disk files are closed with regular close action, and non-disk files are closed with release action. The input file must not be open when it is passed to the sort intrinsic by the program.

If an input procedure is used as the input option, the procedure is called to furnish input records to the sort intrinsic. The input procedure must be a Boolean procedure, must not be declared EXTERNAL, and must have a one-dimensional array with a lower bound of zero as its only parameter. This procedure, on each call, either inserts the next record to be sorted into its array parameter or returns the value TRUE, which indicates the end of the input data.

When TRUE is returned by the input procedure, the sort intrinsic does not use the contents of the array parameter and does not call the input procedure again.

The following is an example of an input procedure that can be used when sorting N elements of array Q:

```
BOOLEAN PROCEDURE INPROC(A);
ARRAY A[0];
   BEGIN
   N := *-1;
   IF N GEQ 0 THEN
      A[0] := Q[N]
   ELSE
      INPROC := TRUE;
   END INPROC;
```

<number of tapes>

The value of <number of tapes> specifies the number of tape files that can be used, if necessary, in the sorting process. If the value of the arithmetic expression is zero, no tapes are used. If the value of the arithmetic expression is between 1 and 3, inclusive, three tapes are used. If the value of the arithmetic expression is between 3 and 8, the specified number of tapes are used. If the value of the arithmetic expression is 8 or more, a maximum of eight tapes are used.

<compare procedure>

The compare procedure is called by the sort intrinsic to apply the appropriate sort criteria to a pair of input records. The procedure must be a Boolean procedure, must not be declared EXTERNAL. and must have exactly two parameters. Each of the parameters must be a one-dimensional array with a lower bound of zero. Every time two input records are to be compared, the sort intrinsic calls the compare procedure and passes the two records to the compare procedure through the array parameters. If the compare procedure returns TRUE, the record passed to the first array precedes, in the sorted output, the record passed to the second array. If the compare procedure returns FALSE, the

record passed to the second array precedes the record passed to the first array.

The following is an example of a compare procedure that can be used to sort arithmetic data in ascending sequence:

```
BOOLEAN PROCEDURE CMP(A,B);
ARRAY A,B[0];
   BEGIN
   CMP := A[0] < B[0];
   END CMP;
```

For alphanumeric comparisons, the following compare procedure can be used to sort data in ascending sequence:

```
BOOLEAN PROCEDURE CMP(A,B);
ARRAY A,B[0];
   BEGIN
   CMP := POINTER(A) LSS POINTER(B) FOR 6;
   END CMP;
```

The CMP procedures above return TRUE if the value in A[0] compares as less than the value in B[0] and return FALSE if the value in A[0] compares as greater than or equal to the value in B[0]. Therefore, if A[0] is less than B[0], the content of array A is passed to the output file or procedure before the content of array B, and if A[0] is greater than or equal to B[0], the content of array B is passed to the output file or procedure before the content of array A. If either of these compare procedures is used, word zero of the input records is considered to be the "key" on which sorting is done.

For the actual comparison, a string relation can be used to compare a string from each record (according to the EBCDIC collating sequence), or an arithmetic relation can be used to compare an arithmetic value from each record. The comparison can be done on one or more fields (called "keys") from each record or on the entire record. The manner in which the comparison is done is specified entirely by the programmer.

<record length>

The record length specifies the length, in words or characters (depending on whether the array parameters of the procedure are word or character arrays, respectively) of the largest item that is to be sorted. If the value of the arithmetic expression is not a positive integer, the largest integer that is not greater than the absolute value of the expression is used; for example, a record length of 12 is used if the expression has a value of -12.995. If the value of the arithmetic expression is zero, the program terminates.

<size specifications>

The size specifications allow the programmer to specify the maximum amount of main memory and disk storage to be used by the sort intrinsic.

The memory size specifies the maximum amount (in words) of main memory that is to be used. If the memory size is unspecified, a value of 12,000 is assumed.

The disk size specifies the maximum amount (in words) of disk storage that can be used. If the disk size is unspecified, a value of 600,000 is assumed.

If the pack size is specified, temporary files created by the sort intrinsic have PACK, instead of DISK, as the value of their FAMILYNAME attribute. For an explanation of the FAMILYNAME attribute, refer to the "I/O Subsystem Reference Manual." If the <arithmetic expression> option does not appear in the <pack size> element, a value of 600,000 words is assumed.

<restart specifications>

The restart specifications allow the sort intrinsic to resume processing at the most recent checkpoint after discontinuation of a program. The program must provide logic to restore and maintain variables, arrays, files, pointers, and so forth, which are defined for, and by, the program. In other words, the program must provide the means to restore everything that is necessary for the program to continue from the point of interruption. The restart capability is implemented only for disk sorts.

The sort intrinsic inspects the least significant (rightmost) five bits of the value of the arithmetic expression in the restart specifications to determine the course of action it is to take. To control the sort, these bits can be set by the program. The meanings of these bits are explained in the following table.

Bit	Value	Description
0	1	The program is restarting a previous sort. The sort intrinsic tries to open its two disk files and obtain restart information. If it is successful in obtaining this information, the sort intrinsic tries to continue from the most recent restart point.

Bit	Value	Description
0	0	The sort is starting from the beginning. If the sort is restartable, and previous sort files with identical titles exist, they are removed and replaced by new sort files.
1	1	The program is requesting a restartable sort. The sort intrinsic saves its two internal files and can be restarted on program request. If bit 2 is 1, bit 1 is set to 1 by default.
1	0	A normal sort is requested, and no sort files are saved (unless bit 2 is 1, which sets bit 1 to 1 by default).
2	1	The program is requesting a restartable sort and desires extensive error recovery from I/O errors. If bit 2 is 1, the sort intrinsic attempts to backtrack and remerge strings, as necessary, when I/O errors occur during the accessing of either of the two sort files. To use this option, the program must provide at least three times as much disk space as required to contain the input data. If less disk space is provided, the sort intrinsic emits an error message, changes to restartable-only mode, and continues the sort without further use of backtracking capability.
2	0	Recovery from internal errors is not requested.
3	--	Bit 3 has meaning only if a restartable sort is requested. The use of this option controls the sort during the stringing phase as the user input is being read by the sort intrinsic. Use of this bit determines how the sort restarts (when a restart is requested) only if the restart occurs while the sort is in the stringing phase.
3	1	The program requires that the sort restart at the beginning of the user input. This restart is the equivalent of starting an entirely new sort. In case the restarted sort passes from the stringing phase into the merge phase, it continues from the merge phase. This bit can be set to 1 during a restart, even if it is not 1 initially. Once set to 1, it cannot be set to 0 by subsequent restarts.
3	0	The program requires the ability to restart at the last restart point that occurred during the stringing phase. If the sort is still in the stringing phase, it skips over the records already processed and continues from the last restart point. If the sort is in the merge phase, it continues from the last merge phase restart point. If bit 3 is 0, the sort is normally less efficient because more strings are created during the stringing phase.

Bit	Value	Description
4	--	This bit is reserved for expansion and is not currently used by the sort intrinsic.

Arrays in Sort Procedures

The array parameters used by the input procedure, output procedure, and compare procedure must be similarly specified. For example, if one procedure declares its array parameter as an EBCDIC array, then all must declare their array parameters as EBCDIC arrays.

When character arrays are used in the procedures passed to the sort intrinsic, the record length parameter is interpreted as a length in characters.

For more detailed information about the sort intrinsic, refer to "SORT" in the "System Software Utilities Reference Manual."

SORT Mode

The combination of the <disk size> and <number of tapes> determines the sort mode as follows:

<number of tapes>	<disk size>	Sort Mode
NEQ 0	0	Tape Only
NEQ 0	NEQ 0	Integrated-Tape-Disk (ITD)
0	NEQ 0	Disk Only
0	0	Core Only

Examples

```
SORT(FILEOUT,FILEIN,3,COMPARE,10)
```

Sorts the records of file FILEIN according to compare procedure COMPARE and writes the sorted data to file FILEOUT. Three tapes are used in the sort and the record length is 10.

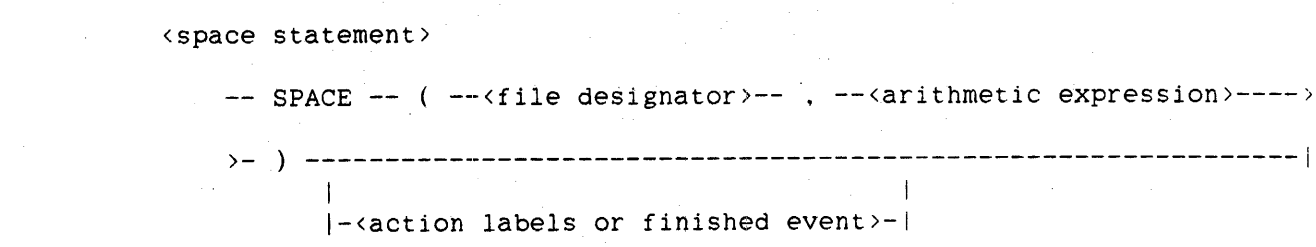
```
SORT(OUTPROC,INPROC,NUMOFTAPES,COMPARER,DSKSZ) [RESTART = PARAM]
```

Sorts the records provided by procedure INPROC according to compare procedure COMPARER, and writes sorted data out according to procedure OUTPROC. The number of tapes is given by NUMOFTAPES, and the record size is given by DSKSZ. A restart specification is given by PARAM.

SPACE STATEMENT

The SPACE statement is used to bypass records in a file without reading those records.

Syntax



See also
<action labels or finished event> 362
<file designator> 189

Semantics

The value of the arithmetic expression determines the number of records to be spaced and the direction of the spacing. If the value of the arithmetic expression is positive, the records are spaced in a forward direction; if it is negative, the records are spaced in the reverse direction.

The SPACE statement can be used as a Boolean function. When the statement fails, the value TRUE is returned. When the statement is successful, the value FALSE is returned. Specifically, the SPACE statement returns a value identical to that returned by the file attribute STATE. For more information, refer to the discussion of the STATE attribute in the "I/O Subsystem Reference Manual."

The file designator must not reference a direct file or a direct switch file.

Examples

```
SPACE(FYLE,50)
```

Spaces file FYLE forward 50 records.

```
SPACE(FILEID,N) [LEOF]
```

Spaces file FILEID a number of records and a direction given by the value of N. If an end-of-file condition occurs, the program continues execution with the statement associated with the label LEOF.

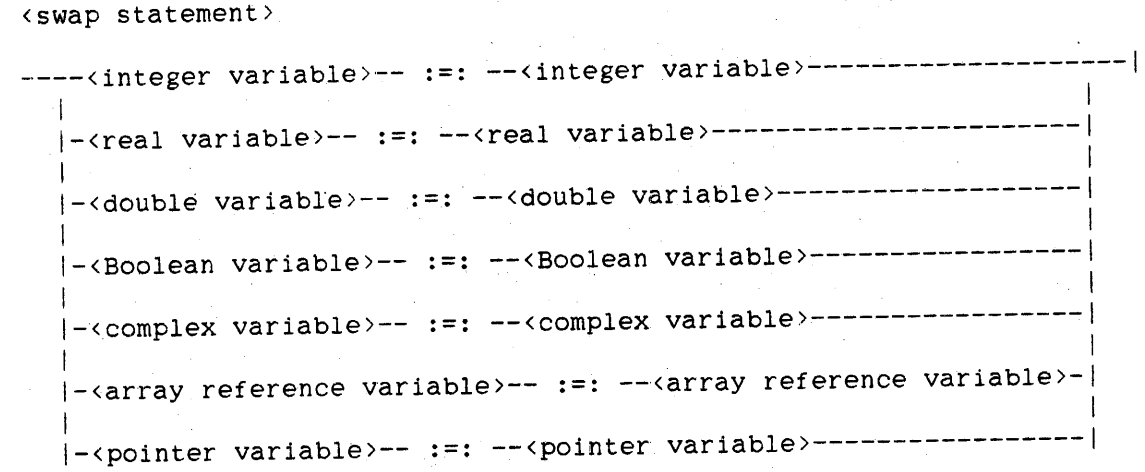
```
B := SPACE(FILEID,-3) [LEOF]
```

Spaces file FILEID backward 3 records. A value is assigned to B indicating the success or failure of the spacing. If an end-of-file condition occurs, the program continues execution with the statement associated with the label LEOF.

SWAP STATEMENT

The swap statement assigns the value of the variable on the right side of the swap operator (:=:) to the variable on the left side of the swap operator, and assigns the value of the variable on the left side of the swap operator to the variable on the right side of the swap operator.

Syntax



<integer variable>

A <variable> of type INTEGER.

<real variable>

A <variable> of type REAL.

<double variable>

A <variable> of type DOUBLE.

See also

<array reference variable>. 231
<Boolean variable>. 234
<complex variable>. 237
<pointer variable>. 241
<variable>. 225

Semantics

The declared types of the variables on either side of the swap operator (:=:) must be the same. Partial word swaps are not permitted.

Descriptions of the processes of an assignment are found under "Assignment Statement."

Example

This example program uses the swap statement to sort a real array.

```
BEGIN
   FILE REM(KIND=DISK,TITLE="SORT/OUT.",PROTECTION=SAVE);
   BOOLEAN SWAP_DONE;
   INTEGER I,J;
   DEFINE LASTONE = 5#;
   INTEGER ARRAY ARY[0:LASTONE];

   PROCEDURE SORTER;
      BEGIN

      BOOLEAN PROCEDURE NEED_TO_SWAP(A,B);
      VALUE A,B;
      INTEGER A,B;
         BEGIN
         IF (A < B) THEN
            NEED_TO_SWAP := TRUE
         ELSE
            NEED_TO_SWAP := FALSE;
         END NEED_TO_SWAP;

      SWAP_DONE := TRUE;
      FOR I := 0 STEP 1 WHILE (I < LASTONE AND SWAP_DONE) DO
         BEGIN
         SWAP_DONE := FALSE;
         FOR J := I+1 STEP 1 UNTIL LASTONE DO
            IF (NEED_TO_SWAP(ARY[I],ARY[J])) THEN
               BEGIN
               SWAP_DONE := TRUE;
               ARY[I] :=: ARY[J];
               END;
         END FORLOOP;
      END SORTER;
```

```
   ARY[0] := "   SAM";
   ARY[1] := "    AL";
   ARY[2] := "   HAL";
   ARY[3] := "   BOB";
   ARY[4] := "   TOM";
   ARY[5] := "   SID";
   SORTER;
   FOR I := 0 STEP 1 UNTIL LASTONE DO
      WRITE(REM,<A6>,ARY[I]);
END.
```

THRU STATEMENT

The THRU statement executes a statement a specified number of times.

Syntax

<thru statement>

```
-- THRU --<arithmetic expression>-- DO --<statement>--|
```

See also
<statement> 219

Semantics

The absolute value of the arithmetic expression is evaluated and rounded to an integer, if necessary. This value determines the number of times the statement following "DO" is executed. The upper limit of this value is 2**39 - 1. Figure 5-6 illustrates the THRU loop.

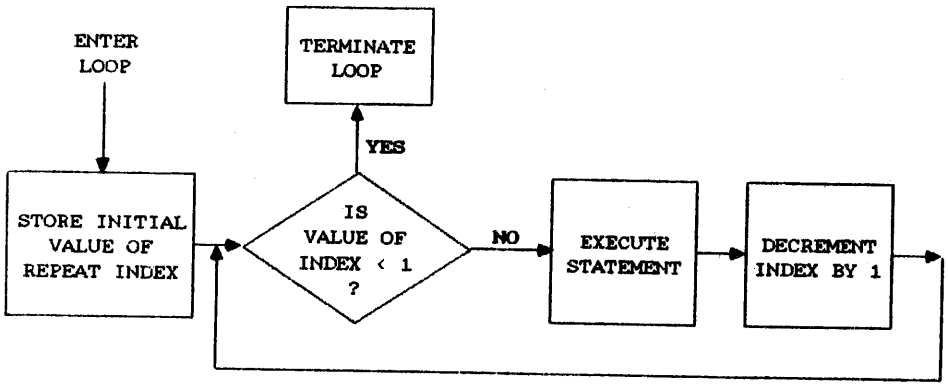


Figure 5-6. THRU Loop

Examples

```
THRU 255 DO
   LOADCHAR
```

The statement LOADCHAR is executed 255 times.

```
THRU MAXI := REAL(PTR,3) DO
   SKIP1
```

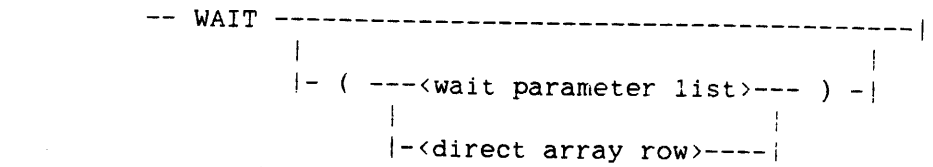
The REAL function is evaluated, the value is assigned to MAXI, and the statement SKIP1 is executed MAXI times.

WAIT STATEMENT

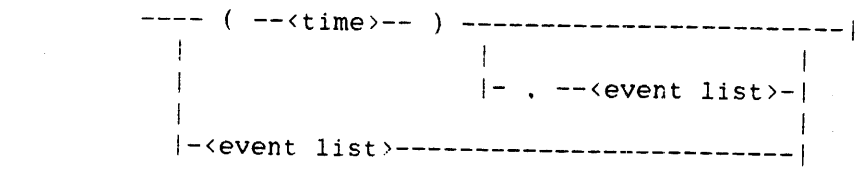
The WAIT statement suspends the program until a designated condition occurs.

Syntax

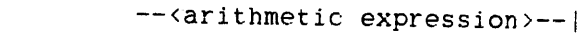
<wait statement>



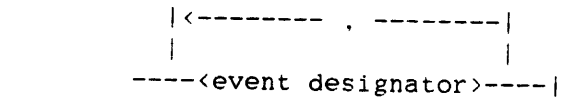
<wait parameter list>



<time>



<event list>



See also

Semantics

The program can be suspended until a given length of time elapses, an event is caused, a previously initiated direct I/O statement is finished, or a software interrupt occurs.

If the WAIT statement consists solely of "WAIT", a Master Control Program (MCP) procedure is called that suspends the program until an attached and enabled interrupt is invoked as a result of the associated event being caused. For more information, refer to "INTERRUPT Declaration." This form of the WAIT statement cannot be used as a function.

When a statement of the form "WAIT(<event designator>)" is executed, the event is examined to determine whether its happened state is TRUE (happened) or FALSE (not happened). If the happened state of the event is TRUE, the program continues executing with the next statement. If the happened state of the event is FALSE, the program is suspended until the event is caused.

When a statement of the form "WAIT((<time>))" is executed. execution of the program is suspended for <time> seconds. Refer to "WHEN Statement" for a discussion of <time>.

When the statement includes event designators in the wait parameter list. the program is suspended until any one event in the event list is caused or until <time> seconds, if specified, have elapsed.

The "WAIT(<wait parameter list>)" form can be used as an arithmetic function that returns an integer value, starting at 1, that represents the position in the wait parameter list of the item that caused the program to be activated. For example, in the statement

```
T := WAIT((.001),E1,E2)
```

the value of T is 1 if elapsed time caused the program to be activated. In the statement

```
T := WAIT(E1,E2,E3)
```

the value of T is 2 if a cause operation on event E2 activated the program. Only one parameter activates the program.

The WAIT statement with a wait parameter list and the WAITANDRESET statement are identical, except for the state to which the caused event is set during the cause process. If a program is waiting on an event

because of the WAIT statement, then the happened state of the event is set to TRUE (happened). If a program is waiting on an event because of a WAITANDRESET statement, then the happened state of the event is set to FALSE (not happened).

The form "WAIT(<direct array row>)" is one of the ways in which a program can determine if a previously initiated direct I/O statement has finished. This form can be used as a Boolean function. When the I/O statement fails, the value TRUE is returned. When the statement is successful, the value FALSE is returned. Specifically, this form of the WAIT statement returns a value identical to that returned by the file attribute STATE. Refer to the discussion of the STATE attribute in the "I/O Subsystem Reference Manual."

Examples

```
WAIT(EVNT)
```

If the happened state of event EVNT is TRUE (happened), the program continues with the next statement. Otherwise, the program is suspended until EVNT is caused, and the happened state of EVNT is set to TRUE (happened).

```
WAIT(EVNT1,EVNT2,EVNT3)
```

If the happened state of event EVNT1, EVNT2, or EVNT3 is TRUE (happened), the program continues with the next statement. Otherwise, the program is suspended until one of the events EVNT1, EVNT2, or EVNT3 is caused, and the happened state of that event is set to TRUE (happened).

```
X := WAIT((NAPTIME),WAKEUP,GOAWAY)
```

If the happened state of WAKEUP or GOAWAY is TRUE (happened), the program continues with the next statement. Otherwise, the program is suspended until NAPTIME seconds have elapsed or until event WAKEUP or GOAWAY is caused. If WAKEUP or GOAWAY is caused, its happened state is set to TRUE (happened). The value stored in X is 1, 2, or 3, indicating which of the three items reactivated the program.

RSLT := WAIT(DIRINPUT)

If DIRINPUT is a direct array row, the program is suspended until the direct I/O operation associated with DIRINPUT is completed. If the I/O operation fails, the value TRUE is assigned to RSLT. If the operation is successful, the value FALSE is assigned to RSLT.

WAIT

The program is suspended until an attached and enabled interrupt is invoked as a result of the associated event being caused.

WAITANDRESET STATEMENT

The WAITANDRESET statement suspends the program until a designated condition occurs.

Syntax

<waitandreset statement>

```
-- WAITANDRESET -- ( --<wait parameter list>-- ) --|
```

See also
<wait parameter list> 452

Semantics

The WAITANDRESET statement and the WAIT statement with a wait parameter list are identical, except for the state to which the caused event is set during the cause process. If a program is waiting on an event because of a WAITANDRESET statement, then the happened state of the event is set to FALSE (not happened). If a program is waiting on an event because of the WAIT statement, then the happened state of the event is set to TRUE (happened).

The WAITANDRESET statement can be used as an arithmetic function that returns an integer value, starting at 1, that represents the position in the wait parameter list of the item that caused the program to be activated. For example, in the statement

```
T := WAITANDRESET((.001),E1,E2)
```

the value of T is 1 if elapsed time caused the program to be activated. In the statement

```
T := WAITANDRESET(E1,E2,E3)
```

the value of T is 2 if a cause operation on event E2 activated the program. Only one parameter activates the program.

Note that the <direct array row> syntax is not allowed as a parameter to the WAITANDRESET statement.

Examples

```
WAITANDRESET(EVNT)
```

If the happened state of event EVNT is TRUE (happened), the program continues with the next statement. Otherwise, the program is suspended until EVNT is caused, and the happened state of EVNT is set to FALSE (not happened).

```
WAITANDRESET(EVNT1,EVNT2,EVNTARAY[INDX])
```

If the happened state of event EVNT1, EVNT2, or the event designated by EVNTARAY[INDX] is TRUE (happened), the program continues with the next statement. Otherwise, the program is suspended until one of the three events is caused, and the happened state of that event is set to FALSE (not happened).

```
WAITANDRESET((.5),FINI,GOAWAY)
```

If the happened state of event FINI or GOAWAY is TRUE (happened), the program continues with the next statement. Otherwise, the program is suspended until .5 second has elapsed or until event FINI or GOAWAY is caused. If FINI or GOAWAY is caused, its happened state is set to FALSE (not happened).

```
REASON := WAITANDRESET((SLEEPMAX),WAKEUP,LOOKAROUND)
```

If the happened state of event WAKEUP or LOOKAROUND is TRUE (happened), the program continues with the next statement. Otherwise, the program is suspended until SLEEPMAX seconds have elapsed or until event WAKEUP or LOOKAROUND is caused. The value stored in REASON is 1, 2, or 3, indicating which of the three items reactivated the program. If WAKEUP or LOOKAROUND is caused, its happened state is set to FALSE (not happened).

WHEN STATEMENT

The WHEN statement suspends processing of the program for the specified number of seconds.

Syntax

<when statement>

```
-- WHEN -- ( --<time>-- ) --|
```

See also
<time>. 452

Semantics

Program processing is suspended for <time> seconds. The value of <time> need not be an integer. If <time> is a double-precision value, it is rounded to single precision. If <time> is less than approximately 0.0000023, the program resumes execution immediately. If <time> is larger than this value, then the number of seconds that the program is suspended is the smaller of <time> and (2**32-1)*2.4 microseconds (approximately 2.86 hours).

Pragmatics

Depending on the amount of multiprocessing being performed and the priorities of other programs in execution, the actual time that a program is suspended can vary widely with respect to <time> but is at least <time> seconds.

Examples

```
WHEN(10)
```

The program is suspended for 10 seconds.

```
WHEN(2*Y+Z)
```

The program is suspended for 2 * Y + Z seconds.

WHILE STATEMENT

The WHILE statement executes a statement as long as a specified condition is met.

Syntax

<while statement>

```
-- WHILE --<Boolean expression>-- DO --<statement>--|
```

See also
<statement> 219

Semantics

The iterative WHILE statement is executed as follows: the Boolean expression is evaluated and, if the result is TRUE, the statement following "DO" is executed. This sequence of events continues until the value of the Boolean expression is FALSE or until the statement following "DO" transfers control outside the WHILE statement. Figure 5-7 illustrates the WHILE-DO loop.

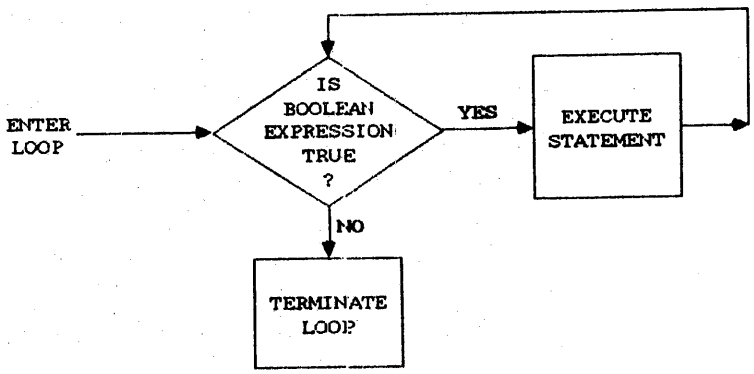


Figure 5-7. WHILE-DO Loop

Examples

```
WHILE INDX LEQ MAXVAL DO
   X := *+A[INDX];
```

As long as INDX is less than or equal to MAXVAL, the value of X is incremented by the value A[INDX].

```
WHILE J LSS LIMIT DO
   BEGIN
   SU[J] := SVALUES[J];
   J := *+1;
   END;
```

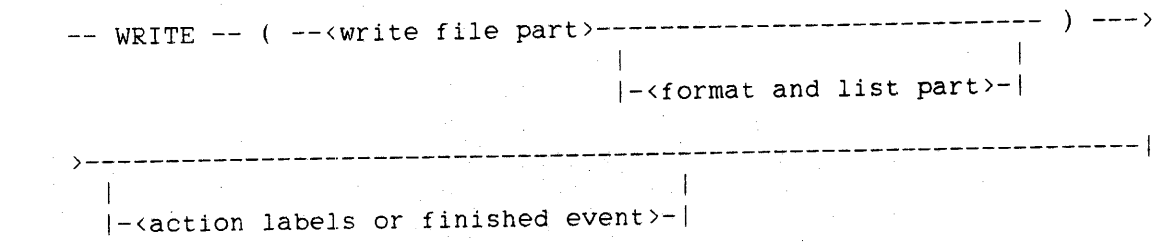
As long as J is less than LIMIT, the compound statement is executed. Contiguous elements of array SU are assigned the values of the elements of array SVALUES with the same indexes.

WRITE STATEMENT

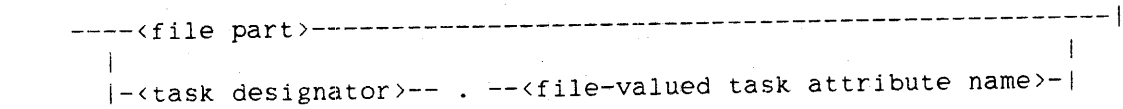
The WRITE statement causes data to be transferred from various program variables to a file.

Syntax

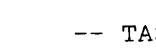
<write statement>



<write file part>



<file-valued task attribute name>



NOTE

The syntax of the WRITE statement and the syntax of the READ statement are nearly identical. Differences in the semantics are discussed separately in the semantics for each statement. See the "READ Statement" in this chapter for a more detailed breakdown of those syntactic elements of the WRITE statement that are not discussed here.

See also

Semantics

The action of the WRITE statement depends on the form of the <write file part> element and on the form of the <format and list part> element.

The WRITE statement can be used as a Boolean function. When the write operation fails, the value TRUE is returned. When the write operation succeeds, the value FALSE is returned. Specifically, the WRITE statement returns a value identical to that returned by the file attribute STATE. For more information, refer to the discussion of the STATE attribute in the "I/O Subsystem Reference Manual."

For Burroughs Network Architecture (BNA) Host Services, error results for WRITE statements are reported one WRITE statement after the WRITE statement that reuses the buffer that originally had the error. That is, the error is reported one buffer later than normal. Normally, error results are reported exactly at the WRITE statement that reuses the buffer having the error.

Pragmatics

WRITE statements that do not contain format designators or editing specifications provide a faster output operation than those that specify that data is to be edited.

Examples of WRITE Statement Syntax

```
WRITE(FILEID)

WRITE(SPOFILE,FMT,LISTID)

WRITE(FILEID[NO],FMT)

WRITE(SPOFILE,10,ARRY[3,*])

WRITE(SWFILEID[0],X+Y-Z,ARRY[X,I,*])

WRITE(SPOFILE,/,LISTID)

WRITE(FILEID,FMT,LISTID)

WRITE(SWFILEID[3][PAGE])

WRITE(FILEID,/,A,B,C)

WRITE(FILEID,SWFMT[A*I])
```

```
WRITE(FILEID,*,LISTID)

WRITE(FILEID[5+I],/,SWLISTID[4])

WRITE(FILEID,/,LISTID)

WRITE(FILEID,*,A,B,C)

WRITE(FILEID,FMT,A,B,C,D+SIN(X)) [:PARL]

WRITE(FILEID,FMT,LISTID) [:PARSWL[M]]

WRITE(SWFILEID[1],SWFMT[2],SWLISTID[3]) [:PARSWL[4]]

WRITE(DIRFYLE,30,DIRARAY) [EVNT]

WRITE(MYSELF.TASKFILE,<"ABOVE DUMP BEFORE TRANSACTION">)

WRITE(OUT,10,S1 || S2)

WRITE(OUT,<2A10>,TAKE(S1,2),S1 || "ABC")
```

<write file part>

The write file part indicates where the data is to be written.

<record number or carriage control>

If the <record number or carriage control> element is "[LINE <arithmetic expression>]" and the file is a printer file, then the printer spaces forward to the specified line before printing. The PAGESIZE file attribute of the file must be nonzero. Because the default action for ALGOL is to print before carriage action, a subsequent WRITE statement can overprint the line. For more information on the PAGESIZE attribute, refer to the "I/O Subsystem Reference Manual."

The "[SKIP <arithmetic expression>]" construct causes the printer to skip to the channel indicated by the value of the arithmetic expression after printing the current record. The LINENUM file attribute of the file is re-initialized to 1 when [SKIP 1] is used. For more information on the LINENUM attribute, refer to the "I/O Subsystem Reference Manual."

The "[SPACE <arithmetic expression>]" construct causes the printer to space the number of lines specified by the arithmetic expression after printing the current record. On other types of devices, this construct causes the number of records specified by the value of the arithmetic expression to be spaced.

If the specified file is a remote file, the "[STOP]" construct causes the normal line feed and carriage return action to be omitted.

The "[STACKER <arithmetic expression>]" construct allows pocket selection for card punch files. Valid values for the arithmetic expression are 0 and 1: 0 selects the normal pocket, and 1 selects the alternative pocket.

The "[TIMELIMIT <arithmetic expression>]" construct is meaningful only for remote files. The write operation is terminated with a timelimit error if the buffer is not available within the number of seconds specified by the value of the arithmetic expression.

The "[STATION <arithmetic expression>]" construct is meaningful only for remote files. It assigns the value of the arithmetic expression to the LASTSUBFILE file attribute of the file. For more information on the LASTSUBFILE attribute, refer to the "I/O Subsystem Reference Manual."

<subfile specification>

If the file to be written is a port file (a file for which the KIND attribute is equal to PORT), an array row write containing a subfile specification must be used. Refer to "Array Row Write" in this section.

The subfile specification is meaningful only for port files. It is used to specify the subfile to be used for the write operation and the type of write operation to be performed.

If a subfile index is used, the value of the subfile index is assigned to the LASTSUBFILE attribute of the file. It specifies the subfile to be used for the write operation. For a WRITE statement, if the subfile index is zero, a broadcast write is performed. If the subfile index is nonzero, then a write to the specified subfile is performed. The result variable, if specified, is assigned the resultant value of the LASTSUBFILE attribute. For more information on the LASTSUBFILE file attribute, refer to the "I/O Subsystem Reference Manual."

If DONTWAIT is specified and no buffer is available, the program is not suspended.

<core-to-core part>
<core-to-core file part>
<core-to-core blocking part>

Refer to "READ Statement" for a discussion of these constructs.

<format and list part>

The <format and list part> element indicates which variables contain the data and how the data is to be interpreted.

If the <format and list part> element is omitted in a WRITE statement, a logically empty record is written. The actual output is device-dependent. Printers and card punches interpret this as a blank record; disks and tapes interpret this as a record with undefined contents.

Formatted Write

A WRITE statement that contains a format designator, editing specifications, or a free-field part is called a "formatted write."

A format designator without a list indicates that the referenced format contains one or more string literals that constitute the entire output of the WRITE statement.

A format designator with a list indicates that the variables in the list are to be written in the format described by the referenced format.

Editing specifications can appear in place of a format designator and have the same effect as if they had been declared in a FORMAT declaration and had been referenced through a format designator. For more information, refer to the "FORMAT Declaration."

Binary Write

A WRITE statement of the form

```
WRITE(<write file part>,*,<list>)
```

is called a "binary write."

An asterisk (*) followed by a list specifies that the elements in the list are to be processed as full words and are to be written without being edited. The number of words written is determined by the number of elements in the list or the maximum record size, whichever is smaller. When unblocked records are used, the block size is the maximum record size.

When writing a character array, only full words are written. If there is a partial word left at the end of the array, it is ignored. For example, if A is an EBCDIC array that contains the characters "12345678", the statement

```
WRITE(FILEID,*,A)
```

writes only the characters "123456".

When a string variable occurs in the list of a binary WRITE statement, a word containing the string length is written to the file before the contents of the string are written. This feature allows the program to write string information that can later be read through a binary READ statement. For more information, see "Binary Read" under "READ Statement."

See also
Binary read 369

Array Row Write

A WRITE statement of any of the forms

```
WRITE(<write file part>,<arithmetic expression>,<array row>)
WRITE(<write file part>,<arithmetic expression>,
                                        <subscripted variable>)
WRITE(<write file part>,<arithmetic expression>,
                                          <pointer expression>)
WRITE(<write file part>,<arithmetic expression>,<string variable>)
```

is called an "array row write."

The first three forms of the array row write specify that the elements of the designated array row, subscripted variable, or item referenced by the pointer expression are to be processed as full words and are to be written without being edited. The number of words written is determined by the smallest of

- the number of elements in the array row, subscripted variable, or item referenced by the pointer expression
- the maximum record length
- the absolute value of the arithmetic expression

If the FILETYPE attribute if the file has a value of 6, then the maximum record length is ignored and records span block boundaries. When unblocked records are used, the block size is the maximum record size. If the UNITS attribute equals CHARACTERS and the INTMODE attribute does not equal SINGLE, then all counts represent characters, not words.

A WRITE statement of the form

```
WRITE(<write file part>,<arithmetic expression>,<string variable>)
```

specifies that the characters in the string variable are to be written without being edited. The number of characters written is determined by the maximum record size, the absolute value of the arithmetic expression, or the length of the string, whichever is smallest. If the UNITS attribute equals CHARACTERS and the INTMODE attribute does not equal SINGLE, then all counts represent characters, not words.

<free-field part>

The free-field part allows output to be performed with editing but without using editing specifications. The appropriate format is selected automatically, but variations of the free-field part give the programmer some control over the form of the output.

On output, each value is edited into an appropriate format. An edited item is never split across a record boundary. If the record is too short to hold the representation of the item, a string of pound signs (#) is written in place of the item.

When a complex expression appears in the list of a free-field WRITE statement, two values are written. The first value corresponds to the real part, and the second value corresponds to the imaginary part.

Data items are normally separated by a comma and a space (,). If the free-field part contains two slashes, data items are separated by two spaces.

If the optional asterisk (*) is used, the name of the data item and an equal sign (=) are written to the left of the value of the data item. If the data item is not a variable, then the expression is written as the name of the data item.

If the free-field part includes the <number of columns> and <column width> elements, each list element is written in a separate column. This process is controlled by two column factors: the number of columns per record (r) and the width of each column (w), where w is measured in characters. Both r and w are integerized, if necessary.

If r is zero, the number of columns per record is determined from the value of w and the record length. If w is zero, the width of each column is determined from the value of r and the record length. If both r and w are zero, the output has no column structure. If r and w are such that r columns of w characters cannot fit on one record, adjustments are made to both r and w. The width of a column does not include the two-character delimiter; therefore, r*(w+2) must be less than or equal to the length of the record.

Example

```
BEGIN
   FILE DCOM(KIND=REMOTE,MAXRECSIZE=12,MYUSE=IO);
   INTEGER I;
   REAL R;
   DOUBLE D;
   STRING S;
   I := 25;
   R := 1002459;
   D := 25@@5;
   S := "string";
   WRITE(DCOM,*/, I, R, D, S, I+R, R-D, S||" ABCDE", 7.2);
END.
```

The following output results when this program is executed:

```
   I=25, R=1002459.0, D=2.5D+6, S=string, I+R=1002484.0, R-D=-1497541.0,
   S||" ABCDE"=string ABCDE, <CNST>=7.2,
```

NOTE

Additional information about I/O operations can be found under "I/O Statement" and "READ Statement."

<action labels or finished event>

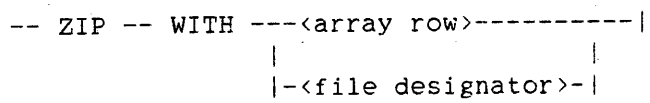
This construct provides a means of transferring program control from a READ statement, WRITE statement, or SPACE statement when exception conditions occur (for normal I/O) or when the I/O is complete (for direct I/O). Exception conditions can also be handled by using the WRITE statement as a Boolean function. For more information, refer to "READ Statement."

ZIP STATEMENT

The ZIP statement causes the Work Flow Language (WFL) compiler to begin compiling the designated source code.

Syntax

<zip statement>



See also
<array row> 43
<file designator> 189

Semantics

The ZIP statement passes to the WFL compiler the source code in the array row or in the file referenced by the file designator. The source code in the array row or file must be valid WFL source input: otherwise, it is not executed. WFL syntax requirements are described in the "Work Flow Language (WFL) Reference Manual."

ZIP WITH <array row>

The array row can be a BCL or EBCDIC array row or a row of a word array. If the array row is a word array, the character type of the contents of the array row is the default character type. (For more information, refer to "Default Character Type" in the "Data Representation" appendix.) The first character of the array row must be a question mark (EBCDIC 48"6F" or BCL 36"14"). The information in the array row must be terminated by the word "END" or the words "END JOB". The array row is processed as one record, but it can include more than 72 characters. A semicolon (;) is used to separate statements within the array row. Only one question mark character can appear in the array row.

WFL examines the contents of the array row for correct syntax, and if errors occur, it reports this fact to the Operator Display Terminal (ODT). If no errors are detected, the compiled job is run. In either case, program control passes to the next statement in the program.

See also
Default Character Type. 817

ZIP WITH <file designator>

When this form of the ZIP statement is executed, the file referenced by the file designator is passed to the WFL compiler. The file is compiled in the same manner as any other WFL source file. If the source compiles without syntax errors, it is executed, and control passes to the statement following the ZIP statement. If syntax errors occur when the source is compiled, a message is displayed on the ODT, the WFL job is not executed, and control passes to the statement following the ZIP statement.

On execution of a ZIP statement, control of the file referenced by the file designator is passed to the Master Control Program (MCP).

Examples

```
ZIP WITH ARAY
```

The WFL source input in array ARAY is compiled and executed.

```
ZIP WITH FYLE
```

The WFL source input in file FYLE is compiled and executed.