

UNISYS

A Series

NEWP

Programming
Reference Manual

Release 3.8.0
Priced Item

May 1989
Distribution Code SE
Printed in U.S.A.
5044233.380

UNISYS

A Series

NEWP

**Programming
Reference Manual**

Copyright © 1989 Unisys Corporation
All rights reserved.
Unisys is a trademark of Unisys Corporation

Release 3.8.0
Priced Item

May 1989
Distribution Code SE
Printed in U.S.A.
5044233.380



Product Information Announcement

New Release Revision Update New Mail Code

Title

A Series NEWP Programming Reference Manual

This Product Information Announcement announces the release of Update 1 to the *A Series NEWP Programming Reference Manual*, dated May 1989, relative to the Mark 3.8.0 System Software Release.

This manual provides reference information on the structure and components of NEWP programs for use in systems programming.

Update 1 contains additional information on the MODULE declaration, PROCEDURE declaration, DECIMALCONVERT function, SEPCOMP and the STANDALONE and XREF compiler control options. It also adds the MODSTRICT compiler control option, and the TESTRASD and TESTWASD UNSAFE procedures.

Changes to the text are indicated by vertical bars in the margins of the replacement pages.

Remove

iii through iv
ix through xvi
4-1 through 4-2

4-17 through 4-18

4-31 through 4-34
5-3 through 5-4
5-11 through 5-12
6-11 through 6-12
7-5 through 7-6

8-5 through 8-6

8-7 through 8-12

8-13 through 8-14
9-27 through 9-28

Index-1 through 10

Insert

iii through iv
ix through xvi
4-1 through 4-2
4-2A through 4-2B
4-17 through 4-18
4-18A through 4-18B
4-31 through 4-34
5-3 through 5-4
5-11 through 5-12
6-11 through 6-12
7-5 through 7-6
7-6A through 7-6B
8-5 through 8-6
8-6A through 8-6B
8-7 through 8-12
8-12A through 8-12B
8-13 through 8-14
9-27 through 9-28
9-28A through 9-28B
Index-1 through 10

Retain this Product Information Announcement as a record of changes made to the basic publication.

Announcement only:

Announcement and attachments:

AS205

System:

A Series

Release:

3.9.0 September 1991

Part Number: 5044233-001

To order additional copies of these manuals,

- United States customers call Unisys Direct at 1-800-448-1424.
- All other customers contact your Unisys Subsidiary Librarian.
- Unisys personnel use the Electronic Literature Ordering (ELO) system.

To receive the update package only, order 5044233-001. To receive the complete guide, order 5044233.380.

Page Status

Page	Issue
iii through iv	-001
v through vii	.380
viii	Blank
ix through xv	-001
xvi	Blank
xvii	.380
xviii	Blank
1-1	.380
1-2	Blank
2-1 through 2-2	.380
3-1 through 3-5	.380
3-6	Blank
4-1 through 4-2A	-001
4-2B	Blank
4-3 through 4-16	.380
4-17 through 4-18A	-001
4-18B	Blank
4-19 through 4-30	.380
4-31 through 4-34	-001
4-35 through 4-45	.380
4-46	Blank
5-1 through 5-2	.380
5-3 through 5-4	-001
5-5 through 5-10	.380
5-11 through 5-12	-001
6-1 through 6-10	.380
6-11 through 6-12	-001
6-13 through 6-19	.380
6-20	Blank
7-1 through 7-4	.380
7-5 through 7-6A	-001
7-6B	Blank
7-7 through 7-9	.380
7-10	Blank
8-1 through 8-4	.380
8-5 through 8-6A	-001
8-6B	Blank
8-7 through 8-12A	-001
8-12B	Blank
8-13 through 8-14	-001
8-15 through 8-19	.380
8-20	Blank
9-1 through 9-26	.380

Page Status

9-27 through 9-28A	-001
9-28B	Blank
9-29 through 9-35	.380
9-36	Blank
A-1 through A-7	.380
A-8	Blank
B-1 through B-5	.380
B-6	Blank
C-1 through C-9	.380
C-10	Blank
Glossary-1 through Glossary-4	.380
Bibliography-1	.380
Bibliography-2	Blank
Index-1 through Index-10	-001

About This Manual

Purpose

This language reference manual for the NEWP product presents the programmer with information on how to use various features of NEWP to do systems programming. NEWP is an ALGOL-based language, and this manual indicates the ways in which NEWP features are similar to or different from those in ALGOL.

Scope

Along with information on the structure and components of NEWP programs, the manual provides syntax and explanations for use of declarations, statements, expressions, functions, and compiler controls. The manual also gives guidance in using the UNSAFE mode. Appendixes provide information on ALGOL features not included in NEWP, information on reserved words, and instructions for reading syntax diagrams.

Audience

This manual is directed to system software programmers.

Prerequisites

The programmer should know the ALGOL language and should be familiar with the A Series architecture.

How to Use This Manual

This document contains reference information for each NEWP feature, which can be accessed either through the index or the table of contents. Cross references are provided within each section. This manual should be used in conjunction with the *A Series ALGOL Programming Reference Manual, Volume 1: Basic Implementation*.

Organization

This manual is divided into nine sections and three appendixes, supplemented by a glossary, a bibliography, and an index.

Section 1. Introduction to NEWP

This section gives an overview of the relationship between ALGOL and NEWP.

Section 2. Program Structure

The basic structure of a NEWP program and the scope of its variables are covered here.

Section 3. Language Components

The building blocks of NEWP are outlined here.

Section 4. Declarations

This section provides information on differences between NEWP and ALGOL declarations. It also provides information on declarations that are available only in NEWP.

Section 5. Statements

This section provides information on differences between NEWP and ALGOL statements. It also provides information on statements that are available only in NEWP.

Section 6. Expressions and Functions

This section provides information on differences between NEWP and ALGOL expressions and functions. It also provides information on expressions and functions that are available only in NEWP.

Section 7. Compiling NEWP Programs

Various input and output files used by the NEWP compiler are presented here.

Section 8. Compiler Controls

This section provides information on differences between NEWP and ALGOL compiler controls. It also provides information on compiler control options that are available only in NEWP.

Section 9. UNSAFE Mode

The various programming constructs that are considered unsafe for general use are presented in this section, as well as some advice on their proper use.

Appendix A. Reserved Words

A reference list of reserved words and keywords, along with their types, is provided.

Appendix B. ALGOL Features Not Implemented in NEWP

For reference purposes, a list of ALGOL features not provided in NEWP is presented.

Appendix C. Understanding Railroad Diagrams

Information on the specifics of railroad, or syntax, diagrams is provided.

Related Product Information

The information in this manual is supplemented by the following documents:

A Series ALGOL Programming Reference Manual, Volume 1: Basic Implementation (form 1169844)

This manual describes the basic features of the Extended ALGOL programming language. This manual is written for programmers who are familiar with programming concepts.

A Series Work Flow Administration and Programming Guide (form 1170149)

This guide discusses the facilities for task initiation and control, including an overview of tasking, a discussion of process management, information on controlling specific processes, and a detailed discussion of task attributes. This guide is written for system administrators, operators, and programmers.

A Series Work Flow Language (WFL) Programming Reference Manual (form 1169802)

This manual presents the complete syntax and semantics of the Work Flow Language (WFL). This language is used to construct jobs that compile or run programs written in other languages, and is used to construct jobs that perform library maintenance such as copying files. This manual is written for individuals who have some experience with programming in a block-structure language such as ALGOL and who know how to create and edit files using CANDE or the Editor.

Contents

	About This Manual	v
Section 1.	Introduction to NEWP	
Section 2.	Program Structure	
	Program Unit	2-1
	Elements of a NEWP Program	2-1
Section 3.	Language Components	
	Basic Symbols	3-1
	Reserved Words	3-1
	Numbers, Numeric Constants, and String Constants	3-1
	Numbers	3-1
	Numeric Constants	3-2
	String Constants	3-4
Section 4.	Declarations	
	ARRAY Declaration	4-2A
	ARRAY REFERENCE Declaration	4-3
	CONSTANT Declaration	4-4
	EXCEPTION PROCEDURE Declaration	4-5
	EXCEPTION PROCEDURE FORWARD Declaration	4-8
	EXPORT Declaration	4-8
	INTERLOCK and INTERLOCK ARRAY Declarations	4-9
	INTRINSIC Declaration	4-10
	LABEL Declaration	4-14
	LIBRARY Declaration	4-14
	MODULE Declaration	4-15
	MODULE Declaration (Old)	4-18A
	ON Declaration	4-22
	OUTPUTMESSAGE ARRAY Declaration	4-23
	POINTER Declaration	4-24
	PROCEDURE Declaration	4-24
	Parameter Passing	4-26
	In-Line Procedures	4-28
	Procedure Value	4-30
	Dynamic Procedure Specification	4-31
	PROCEDURE REFERENCE Declaration	4-34
	SEGMENT Declaration	4-34

Contents

SIMPLE VARIABLE Declaration	4-35
STRUCTURE TYPE Declarations	4-35
SCALAR TYPE Declarations	4-35
Enumerated Types	4-36
Subtypes	4-39
Descendant Types	4-40
Assignment Compatibility	4-41
Range Checking	4-41
SET TYPE Declaration	4-42
STRUCTURE TYPE VARIABLE Declaration	4-43
SUPPLY Declaration	4-44
VALUE ARRAY Declaration	4-45

Section 5. Statements

ASSIGNMENT Statement	5-3
Differences between ALGOL and NEWP ASSIGNMENT Statements	5-3
Array Reference Assignment	5-4
Procedure Reference Assignment	5-4
Procedure Reference Array Assignment	5-6
Set Assignment	5-7
CASE Statement	5-8
FOR Statement	5-8
FREEZE Statement	5-9
PROCEDURE REFERENCE Statement	5-10
REPLACE Statement	5-11
SELECT Statement	5-12
SWAP Statement	5-12

Section 6. Expressions and Functions

Arithmetic Expressions	6-3
Boolean Expressions	6-3
Precedence in Boolean Expressions	6-4
Set Relation	6-5
Function Expressions	6-5
Arithmetic Function Designator	6-5
Boolean Function Designator	6-6
Pointer Expressions	6-6
Scalar Type Expressions	6-7
Set Expression	6-8
String Expressions	6-10
DECIMALCONVERT Function	6-10
DINTEGERT Function	6-11
INTERLOCK Functions	6-11
ARROGATE Function	6-11
BREAK Function	6-12
LOCK Function	6-13
LOCKSTATUS Function	6-14

UNLOCK Function	6-15
PACKDECIMAL Function	6-16
SCALAR TYPE Functions	6-16
LOWER BOUND Function	6-16
MAPPING Function	6-16
PREDECESSOR Function	6-18
SUCCESSOR Function	6-18
UPPER BOUND Function	6-19

Section 7. Compiling NEWP Programs

Full Compilations	7-1
Host Compilations	7-1
Compiling with SEPCOMP	7-2
SEPCOMP Background	7-3
Performing a SEPCOMP	7-4
SEPCOMP Guidelines	7-6
SEPCOMP MERGE	7-8
Compiling for Syntax Only	7-9

Section 8. Compiler Controls

Compiler Control Options	8-1
ALGOL Options Duplicated in NEWP	8-2
Additional NEWP Options	8-2
ASD	8-2
CLEAR	8-3
INCLLIST	8-3
INSTALLATION	8-3
INTERLOCKOPS	8-3
LIST	8-4
LISTO	8-4
LIST1	8-4
MAKEHOST	8-5
MCP	8-5
MERGE	8-5
MODSTRICT	8-5
NEW	8-6
NOCOUNT	8-6
PROCREF	8-6
READLOCK	8-6
READLOCKTIMEOUT	8-7
SEPCOMP	8-7
SEPCOMP MERGE	8-7
SINGLE	8-7
STANDALONE	8-8
STATISTICS	8-10
TADS	8-11
UNDERLINE	8-11
VERSION	8-12

Contents

VOID	8-12
XREF	8-12A
XREFFILES	8-12A
\$	8-12A
Block Directives	8-13
ASDSpace	8-14
CONTROLSTATE	8-14
FIRSTFREEDOCELL	8-14
FIRSTSEGDESC	8-15
INHERITSTATE	8-15
INLINE	8-15
INTERLOCKOPS	8-15
MEMIMAGEBOUND	8-16
NORANGECHECK	8-16
NORMALSTATE	8-16
PROTECTED	8-17
RANGECHECK	8-17
SAFE	8-17
SEGMENT	8-17
SEGMENTLEVEL	8-18
SEPCOMPLEVEL	8-18
STATSUMMARY	8-18
<target option>	8-18
UNSAFE	8-19

Section 9. UNSAFE Mode

Declarations (UNSAFE)	9-1
Address Equation	9-1
DESCRIPTOR Declaration	9-3
PROCEDURE Declaration	9-4
SAVE ARRAY Declaration	9-4
SEGMENT Declaration	9-4
WORD Declaration	9-5
Statements (UNSAFE)	9-6
FORK Statement	9-6
PROCESS Statement	9-6
REPLACE Statement	9-7
OVERWRITE Option	9-7
FLOAT Option	9-7
WAIT and WAITANDRESET Statements	9-8
Expressions and Functions (UNSAFE)	9-9
DESCRIPTOR Expressions	9-9
SETACTUALNAME Function	9-9
SIZE Function	9-10
WORD Expressions	9-10
Intrinsics (UNSAFE)	9-11
ASDTABLE [MACHINEOPS]	9-12
AT [REFERENCE]	9-13
BMASKSEARCH [MACHINEOPS]	9-14
BUZZ [MISC]	9-14

BUZZ47 [MISC]	9-15
CALLIO [MACHINEOPS]	9-15
CHECKHASH [MACHINEOPS]	9-15
DAWDLE [MISC]	9-15
DESCRIPTOR [DESCRIPTOR]	9-16
DLL [REGISTERS]	9-16
DREADMEMORYCONTROL [MACHINEOPS]	9-16
EVAL [MACHINEOPS]	9-17
EXIT [MACHINEOPS]	9-17
FAILREGISTER [MACHINEOPS]	9-17
FMMRREADLOCK [MACHINEOPS]	9-18
IGNOREPARITY [MACHINEOPS]	9-18
INTERRUPTCHANNEL [MACHINEOPS]	9-18
INTERRUPTCOUNTZERO [MACHINEOPS]	9-18
LEXLEVEL [MISC]	9-18
LEXOFFSET [MISC]	9-19
LISTLOOKUP [MACHINEOPS]	9-19
LOADEVENT [MACHINEOPS]	9-19
MAKEPCW [MACHINEOPS]	9-19
MEMORY [MEMORY]	9-20
MOVESTACK [MACHINEOPS]	9-20
PAUSE [MACHINEOPS]	9-20
POINTER [DESCRIPTOR or WORD]	9-21
READANDCLEAREXTERNALS [MACHINEOPS]	9-21
READMEMORYCONTROL [MACHINEOPS]	9-21
READPROCESSORSTATE [MACHINEOPS]	9-21
READTIMEOFDAY [MACHINEOPS]	9-22
READXMEMORYTABLE [MACHINEOPS]	9-22
RECEIVEFROMREQUESTOR [MACHINEOPS]	9-22
REFERENCE TO [REFERENCE and WORD]	9-22
REGISTERS [REGISTERS]	9-23
RETURN [MACHINEOPS]	9-23
RETURNORIGINALS [MACHINEOPS]	9-23
RUNNINGLIGHT [MACHINEOPS]	9-24
SCALERIGHTS [MACHINEOPS]	9-24
SCANIN [MACHINEOPS]	9-24
SCANOUT [MACHINEOPS]	9-24
SENDTOREQUESTOR [MACHINEOPS]	9-25
SETINHIBIT [MACHINEOPS]	9-25
SETLIMITS [MACHINEOPS]	9-25
SETTIMEOFDAY [MACHINEOPS]	9-26
SHOW [MACHINEOPS]	9-26
SIGNALPROCESSOR [MACHINEOPS]	9-27
STACKNUMBER [MACHINEOPS]	9-27
STOP [MACHINEOPS]	9-28
STOP77 [MACHINEOPS]	9-28
SUSPEND [MACHINEOPS]	9-28
SYSTEMCONTROL [MACHINEOPS]	9-28
TESTRASD [MACHINEOPS]	9-28A
TESTWASD [MACHINEOPS]	9-28A
TIMER [MACHINEOPS]	9-28A
VECTOR INTRINSICS [MACHINEOPS]	9-29

Contents

Untyped Intrinsic That Act on Two Vectors	9-29
Untyped Intrinsic That Act on Two Vectors and a Scalar	9-30
VSCAT Intrinsic	9-31
VGATH Intrinsic	9-31
VSUM and VSUMA Intrinsic	9-31
VDOT and VDOTX Intrinsic	9-31
VSEQ Intrinsic	9-32
VPOLY Intrinsic	9-32
VCHEK Intrinsic	9-32
VFMX, VFMN, and VFMXA Intrinsic	9-33
VIA [REFERENCE]	9-33
WHATAMI [MACHINEOPS]	9-34
WHOAMI [MACHINEOPS]	9-34
WORD [WORD]	9-34
WRITEMEMORYCONTROL [MACHINEOPS]	9-34
WRITEPROCESSORSTATE [MACHINEOPS]	9-34
WRITEXMEMORYTABLE [MACHINEOPS]	9-35
 Appendix A. Reserved Words	
Types of Reserved Words	A-1
Reserved Words List	A-2
 Appendix B. ALGOL Features Not Implemented in NEWP	
General Features	B-1
Specific Features	B-1
Declarations	B-1
Statements	B-2
Expressions	B-3
Compiler Control Options	B-4
Miscellaneous	B-5
Product Interfaces	B-5
 Appendix C. Understanding Railroad Diagrams	
 Glossary	 1
 Bibliography	 1
 Index	 1

Figures

8-1.	Diagnostic Information Format	8-6A	
C-1.	Railroad Constraints	C-5	

Tables

3-1.	Character Representation: Differences between ALGOL and NEWP	3-5
4-1.	Intrinsic Identifiers	4-11
8-1.	Unsafe Constructs Permitted	8-19
9-1.	Untyped Intrinsic: Two Vectors	9-29
9-2.	Untyped Intrinsic: Two Vectors and a Scalar	9-30

Section 1

Introduction to NEWP

NEWP is an ALGOL-based language designed for your use in implementing system software. Its major application is the A Series and B 7900 Master Control Program (MCP). The code files produced by the compiler are executable unless you use unsafe features. For more information on unsafe features, see Section 9, "UNSAFE Mode."

Although NEWP is based on ALGOL, there are some significant differences between the two languages. One major difference is that NEWP performs more rigorous type checking than ALGOL does, which helps prevent errors caused by unexpected type conflicts.

As a result, a program that compiles with the ALGOL compiler sometimes receives syntax errors from the NEWP compiler. For example, ALGOL allows a REAL variable to be passed to an INTEGER by-reference formal parameter and allows this formal parameter to be passed to a REAL variable. NEWP does not allow either, unless the actual parameter has had its type explicitly converted with a type transfer function.

Another difference is that NEWP provides a number of features that ALGOL lacks, such as modules, user-defined scalar types, and sets.

NEWP does not provide all the features of ALGOL, nor does NEWP include the environmental software interfaces that ALGOL supports, such as interfaces to Data Management System II (DMSII), Communications Management System (COMS), and Semantic Information Manager (SIM).

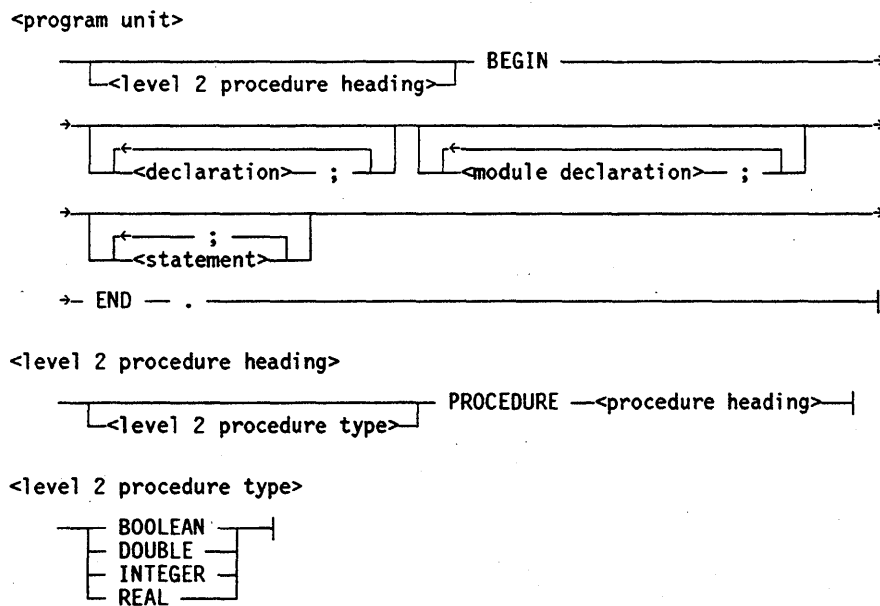
Because the syntax and semantics of most NEWP constructs are the same as the syntax and semantics of the corresponding ALGOL constructs, only those features of NEWP that are not the same as in ALGOL are described in this manual. Elements of NEWP that are not outlined in this document are identical in operation to ALGOL as described in the *A Series ALGOL Programming Reference Manual, Volume 1: Basic Implementation*. Specific information on ALGOL features not supported in NEWP can be found in Appendix B, "ALGOL Features Not Implemented in NEWP."

Section 2

Program Structure

Program Unit

A program unit is a group of NEWP constructs that can be compiled as a whole. The following diagram shows the elements that you can include in a NEWP program.



Elements of a NEWP Program

The simplest valid NEWP program is just a BEGIN/END pair. The BEGIN/END pair can enclose a list of declarations, a list of modules, or a list of statements. If the BEGIN/END pair is preceded by a procedure heading, the entire program is a procedure, which can be typed or untyped and can have one or more parameters.

A NEWP program cannot include the <global part> or <separate procedure> list that are valid in ALGOL.

In NEWP, the lexical level is increased only by nested procedures, not by both nested procedures and nested blocks, as in ALGOL. Thus, by default, any nested procedure causes a change in lexical level, while a nested block that is not a procedure does not.

Segmentation is based upon lexical levels. By default, a new segment is assigned to each procedure declared at lexical level 15 or below, whether or not you declare local variables. You can change the lexical level at which segmentation occurs by

using the `SEGMENTLEVEL` block directive. In addition, you can override automatic segmentation on a block-by-block basis by using the `SEGMENT` block directive.

For the semantics of the `<procedure heading>`, see “PROCEDURE Declaration” in Section 4, “Declarations.” For information on block directives, see “Block Directives” in Section 8, “Compiler Controls.”

Section 3

Language Components

Language components are the building blocks of NEWP. These components consist of basic symbols, reserved words, and constants.

Basic Symbols

Basic symbols are the same in NEWP as in ALGOL, except that both uppercase and lowercase letters are allowed in NEWP. Compiler control options and any token in a program can be written in lowercase letters. Any mixture of uppercase and lowercase is legal; the NEWP program does not make a distinction between uppercase and lowercase letters.

Reserved Words

Reserved words are described and listed in Appendix A, "Reserved Words."

Numbers, Numeric Constants, and String Constants

In ALGOL, there is a syntactic ambiguity between string constants and numeric constants. For example, the following ALGOL symbols could be interpreted as the 8-bit character A when you use the symbols in a string concatenation or as the 48-bit operand represented by the 12 hexadecimal characters 0000000000C1 when you use the symbols in an arithmetic expression:

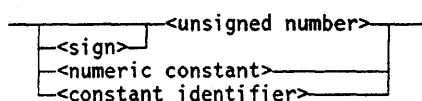
8"A"

In NEWP, string constants and numeric constants are syntactically distinct; string constants are delimited by quotation mark characters ("), and numeric constants are delimited by apostrophe characters (').

The following is information on NEWP numbers, numeric constants, and string constants.

Numbers

<number>

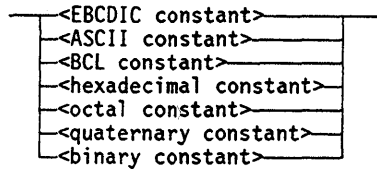


Explanation

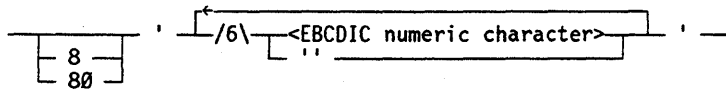
NEWP extends the definition of a <number> to include <numeric constant> and <constant identifier>. For more information, refer to "Numeric Constants" in this section and to the "CONSTANT Declaration" in Section 4, "Declarations."

Numeric Constants

<numeric constant>



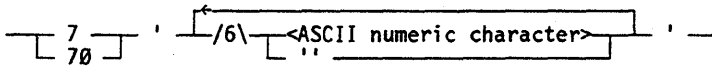
<EBCDIC constant>



<EBCDIC numeric character>

Any EBCDIC character except an apostrophe (').

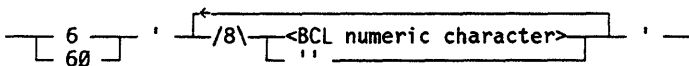
<ASCII constant>



<ASCII numeric character>

Any ASCII character except an apostrophe (').

<BCL constant>

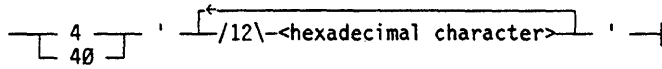


<BCL numeric character>

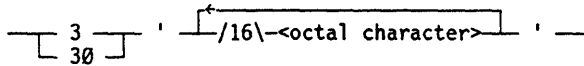
Any BCL character except an apostrophe (').

Note: *The BCL data type is not supported on all A Series and B 7900 systems.*

<hexadecimal constant>



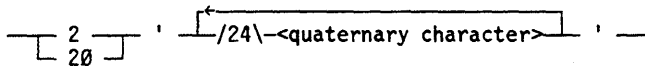
<octal constant>



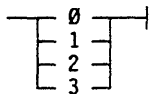
<octal character>

Any digit from 0 through 7.

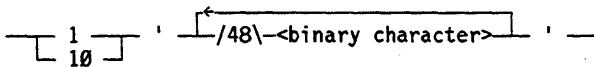
<quaternary constant>



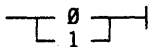
<quaternary character>



<binary constant>



<binary character>



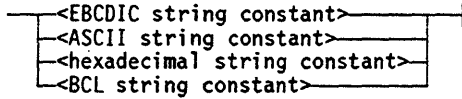
Explanation

A numeric constant is a representation of a 48-bit value. You can specify the value in EBCDIC, ASCII, BCL, hexadecimal, octal, quaternary, or binary notation. An apostrophe in a BCL, EBCDIC, or ASCII numeric constant is represented by two adjacent apostrophe characters. For example, 8'''' is the EBCDIC numeric constant consisting of a single apostrophe character.

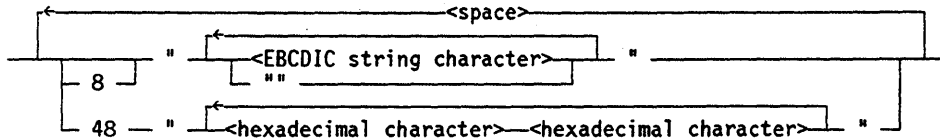
A character code that ends in zero (that is, 10, 20, 30, 40, 60, 70, or 80) indicates that the bits represented by the constant are to be left-justified within the 48-bit arithmetic value, with binary-zero fill. All other character codes generate right-justified, binary-zero-filled values.

String Constants

<string constant>



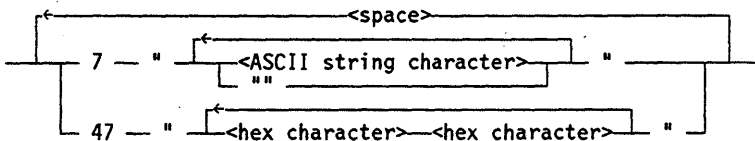
<EBCDIC string constant>



<EBCDIC string character>

Any EBCDIC character except a quotation mark (").

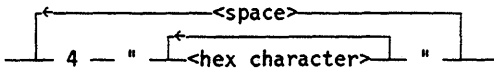
<ASCII string constant>



<ASCII string character>

Any ASCII character except a quotation mark (").

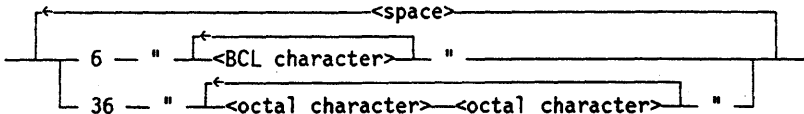
<hexadecimal string constant>



<hexadecimal character>

Any digit from 0 through 9 or any letter from A through F.

<BCL string constant>



<BCL character>

Any BCL character except a quotation mark (").

Note: The BCL data type is not supported on all A Series and B 7900 systems.

<octal character>

Any digit from 0 through 7.

Explanation

String constants are primaries of type string and of subtype EBCDIC, ASCII, hexadecimal, or BCL. A string constant of a particular subtype must include only characters defined for that subtype. Quotation mark (") characters can be included in strings if you represent each embedded quotation mark as two adjacent quotation marks in the syntax.

No more than 256 characters can appear between one pair of quotation marks in a string constant; however, as many as 4095 characters can appear in an EBCDIC string constant, ASCII string constant, or hexadecimal string constant.

You can concatenate string constants by using implicit string concatenation or by using the <string concatenation operator> (see "String Expressions" in Section 6, "Expressions and Functions"). You can type-transfer a string constant to an arithmetic value by using the REAL function. Table 3-1 shows differences in character representation between NEWP and ALGOL.

Table 3-1. Character Representation: Differences between ALGOL and NEWP

Actual String	NEWP Representation	ALGOL Representation
ABC	"ABC"	"ABC"
A"B	"A""B"	"A""B"
"	""	""
null string	not implemented	EMPTY

Examples of legal <string constant> s include the following:

```
"ABCD"
48"Ø1Ø9"
48"Ø1Ø9" "ERROR MESSAGE"
8"A MESSAGE OF TEXT"
```


Section 4

Declarations

NEWP provides several kinds of declarations that are not supported by ALGOL. The NEWP-specific forms of declarations are discussed in this section and in the "Declarations (UNSAFE)" portion of Section 9, "UNSAFE Mode."

The following table describes the differences between declarations that are supported in both NEWP and ALGOL. The table also refers you to more detailed information on differences in particular declarations. In addition, information about ALGOL declarations that NEWP does not support can be found in Appendix B, "ALGOL Features Not Implemented in NEWP:"

Declaration	NEWP Information
ARRAY	Refer to "ARRAY Declaration" and "INTERLOCK and INTERLOCK ARRAY Declarations" later in this section and to "SAVE ARRAY Declaration" in Section 9, "UNSAFE Mode."
ARRAY REFERENCE	Refer to "ARRAY REFERENCE Declaration" later in this section.
BOOLEAN	The OWN clause is not supported; use globally declared variables instead.
DIRECT ARRAY	The OWN clause is not supported; use globally declared variables instead.
DOUBLE	The OWN clause is not supported; use globally declared variables instead.
EXPORT	Refer to "EXPORT Declaration" later in this section.
INTEGER	The OWN clause is not supported; use globally declared variables instead.
LIBRARY	Refer to "LIBRARY Declaration" later in this section.
OUTPUTMESSAGE ARRAY	Refer to "OUTPUTMESSAGE ARRAY Declaration" later in this section.
POINTER	The OWN clause is not supported; use a globally declared variable instead. For information on other differences, refer to "POINTER Declaration" later in this section.
PROCEDURE	Refer to "PROCEDURE Declaration" later in this section and in Section 9, "UNSAFE Mode."
PROCEDURE REFERENCE ARRAY	Only the <local procedure reference array declaration> form is supported in NEWP.
REAL	The OWN clause is not supported; use a globally declared variable instead.
SIMPLE VARIABLE	Refer to "SIMPLE VARIABLE Declaration" later in this section.

Declarations

Declaration

TRANSLATETABLE

VALUE ARRAY

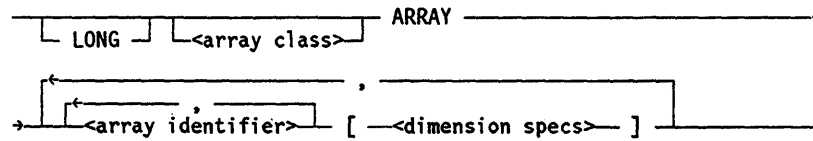
NEWP Information

The <translate table identifier> form of the <translation specifier> is not supported.

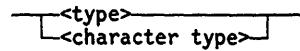
Refer to "VALUE ARRAY Declaration" later in this section.

ARRAY Declaration

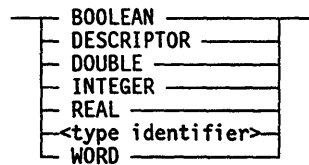
<array declaration>



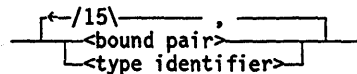
<array class>



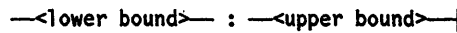
<type>



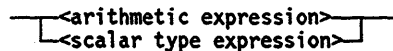
<dimension specs>



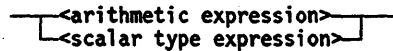
<bound pair>



<lower bound>



<upper bound>



(All other syntax is identical to ALGOL.)

Explanation

NEWP allows one or all dimensions of an array to be unspecified. The < bound pair > `0:-1` indicates an unspecified bound (for example, `ARRAY A[0:-1]`). Except for this special case, the lower bound can never exceed the upper bound.

For a one-dimensional array, the unspecified bounds indication causes the compiler to build a zero-length descriptor for the array. For a multidimensional array, if any bounds are unspecified, either all bounds or only the last bound must be unspecified. If all bounds are unspecified, a zero-length descriptor is built. If only the last bound is unspecified, the dope vectors for the array are built normally and only the row descriptors are built as zero-length descriptors.

Declarations

For a multidimensional array, a zero-length descriptor has to be used under the UNSAFE (DESCRIPTOR) construct.

If a dimension is specified by a < type identifier >, the type of the dimension is defined to be that of type identifier. The type identifier must denote an ordered, bounded, discrete type. The lower bound of this dimension is taken from the smallest value defined for that type (the .LBOUND value), and the upper bound is taken from the largest value defined (the .UBOUND value). When an array is referenced, a subscript that corresponds to a scalar type dimension must be of the same type as the dimension.

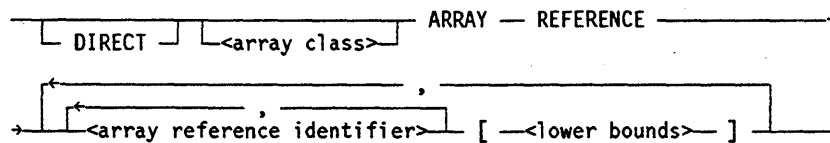
If a dimension is declared by a < bound pair >, then the < lower bound > and the < upper bound > must be either both arithmetic expressions or both scalar type expressions.

If both bounds are given by < scalar type expression > s, the expressions must be of the same ordered type. The type of the dimension is defined to be that of the < scalar type expression > s.

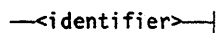
For more information on ARRAY declarations, see "INTERLOCK and INTERLOCK ARRAY Declarations" and the discussion of < type identifier > in "SCALAR TYPE Declaration" later in this section. Also see "DESCRIPTOR [DESCRIPTOR]" and "WORD [WORD]" in Section 9, "UNSAFE Mode."

ARRAY REFERENCE Declaration

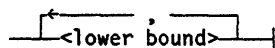
<array reference declaration>



<array reference identifier>



<lower bounds>



Explanation

The ARRAY REFERENCE declaration in NEWP has the same syntax and semantics as the ARRAY REFERENCE declaration in ALGOL, except that in NEWP, array references can be declared to have scalar types.

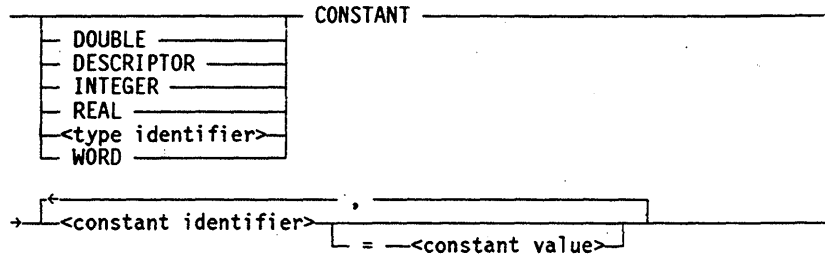
In addition, each dimension of a NEWP array reference can be specified to have a scalar type. The scalar type must be an ordered, bounded discrete type. If the lower bound of a dimension is defined by a < scalar type expression >, the type of the dimension is the type of the expression. When an array reference is used, a subscript that corresponds to a scalar type dimension must be of the same type as was used in the declaration for that dimension.

Declarations

For more information on scalar types, see "SCALAR TYPE Declaration" later in this section and "Scalar Type Expressions" in Section 6, "Expressions and Functions."

CONSTANT Declaration

<constant declaration>



<constant identifier>

—<identifier>—

<constant value>

┌<constant arithmetic expression>┐
└<constant scalar type expression>┘

Explanation

The CONSTANT declaration can be used to declare arithmetic or enumerated constants. This declaration is particularly useful for declaring a series of INTEGER constants where each value is to be one greater than the previous value. If no type is provided in the <constant declaration>, REAL is assumed.

The declaration and use of WORD and DESCRIPTOR constants do not require the use of the UNSAFE WORD or DESCRIPTOR block directives because constants are treated as literals.

If a <constant identifier> appears with a <constant value>, the identifier is associated with the specified value. If no constant value is given, the compiler assigns the identifier a value one greater than the value of the previous identifier. The compiler assigns the value 0 (zero) to the identifier if it is the first identifier in the list. If the type of a CONSTANT declaration is an enumerated type identifier, or a subtype of one, any constant value must be a <constant scalar type expression>. All other types of CONSTANT declarations can use only a <constant arithmetic expression> as the constant value.

The constant value is evaluated in the context of the CONSTANT declaration (unlike defines, for which the text is expanded in the context of the invocation).

If the type identifier is a subtype, it must be a descendant type of INTEGER or an enumerated type. Each constant scalar type expression must be of the same type as the type identifier. If the type identifier is a bounded type, the constant value assigned to the <constant identifier> is checked to verify that the value falls within the valid range for the type.

Examples

```

CONSTANT
  TASKMSCW,           % AUTOMATICALLY ASSIGNED 0
  TASKPARAMS,        % 1
  CODEHEADERINDEX = 0, % STARTS AGAIN AT 0
  RUNNINGCOUNT,    % 1
  CODELINKS,         % 2
  MARKER = CODELINKS, % 2 ALSO
  COMPILERINFO,      % 3
  TOFFSET = MARKER+5, % 7
  NEXTWORD;          % 8

```

```

TYPE SUBCLASS = SUBTYPE INTEGER 0..63;

```

```

SUBCLASS CONSTANT
  UNSPECIFIED = 0,
  BY_VALUE,           % 1
  BY_NAME,            % 2
  BY_REFERENCE,       % 3
  EXTERNAL_PROC,     % 4
  FORWARD_PROC,      % 5
  LIBRARY_PROC,      % 6
  INLINE_PROC,        % 7
  ORDINARY_PROC;     % 8

```

```

TYPE COLOR = (RED, GREEN, BLUE, YELLOW);

```

```

COLOR CONSTANT
  MYFAVORITE = BLUE,
  YOURFAVORITE = RED,
  SCREENCOLOR = GREEN;

```

EXCEPTION PROCEDURE Declaration

<exception procedure declaration>

```

— EXCEPTION PROCEDURE —<exception procedure identifier>— ; —————>
→<compound statement>—————|

```

<exception procedure identifier>

```

—<identifier>—|

```

Explanation

Using the <exception procedure declaration>, you can specify a procedure to be invoked automatically before any abnormal exit of the block in which the EXCEPTION PROCEDURE declaration is contained. An exception procedure is not invoked automatically for a normal exit of the containing block, but you can invoke it directly.

In most respects, an exception procedure is like any other untyped procedure. It can be exported from a module or library and it can be called directly. However, an exception procedure is invoked automatically by the MCP when an abnormal exit of the containing block occurs. This special capability is available only to the containing block. Abnormal

Declarations

exits are all terminations of the block, except in cases of normal completion of the block, or the RETURN and EXIT statements. Examples of abnormal exits include unhandled faults, a DS system command, and going to a label global to the block in which the exception procedure is declared.

If a fault occurs, and an < on declaration > exists to handle the fault, the exception procedure is invoked only if an abnormal exit of the block occurs. Thus, the exception procedure is not invoked if the ON declaration includes a GO TO clause linked to a label inside the block in which the EXCEPTION PROCEDURE declaration occurs.

Usually, an exception procedure is subject to interrupts, including a DS system command. However, you can use the PROTECTED block directive with exception procedures for which an interruption is not acceptable. For more information on the PROTECTED block directive, see "PROTECTED" in Section 8, "Compiler Controls."

The following restrictions apply to exception procedures:

- The body of an exception procedure can contain a < compound statement > only. The exception procedure body cannot be a NULL, EXTERNAL, a < library entry point specification >, or a < dynamic procedure specification >. Exception procedures are not permitted to have parameters.
- Bad GO TO clauses are not allowed. Any attempt to exit the procedure by the way of a GO TO clause to an outer block is flagged as a syntax error.
- The INLINE block directive cannot be used for an exception procedure. If this block directive is used, a syntax error is issued.
- An exception procedure cannot contain an EXCEPTION PROCEDURE declaration. A block or procedure cannot have more than one EXCEPTION PROCEDURE declaration.
- An exception procedure cannot be specified as a formal parameter. It can be passed as an actual parameter, however, if the corresponding formal parameter is declared as an untyped procedure, without parameters.
- Exception procedures can be exported from modules by declaring them as FORWARD in the < module head >, exactly as in any other procedure. When the exception procedure is imported, it acts like any other untyped procedure; an abnormal exit in the importing module does not cause the MCP to invoke the exception procedure automatically. Automatic invocation can occur only in the block in which the exception procedure is originally declared.
- If the MCP option is set, the exception procedure cannot be declared in such a way that its Program Control Word (PCW) is placed in the segment dictionary, since this block is never exited. This means that an exception procedure cannot be declared in the outer block (lexical level 0) of the MCP.

When the MCP option is not set, an exception procedure in the outer block is permitted.

- An exception procedure cannot be address equated.
- An exception procedure identifier can be included in a library EXPORT list. However, any programs that import this entry point must declare it as a normal, untyped procedure without parameters, not as an exception procedure. The following export is an example:

```
EXCEPTION PROCEDURE CLEANUP;  
  BEGIN  
    %procedure body  
  END; % of exception procedure  
EXPORT  
  CLEANUP;
```

Any programs that import this entry point must include the following declaration, assuming library MYLIB has already been declared:

```
PROCEDURE CLEANUP;  
  LIBRARY MYLIB;
```

Considerations for Use

One stack cell is saved if the exception procedure has no FORWARD declaration and is declared as the last item in the block. This is because the exception procedure Program Control Word (PCW) must be directly below the Software Control Word (SCW) for the block, and the compiler generates a second PCW, when necessary, to ensure this ordering. The use of duplicate PCWs is a concern only in an environment in which the conservation of stack cells is important.

It is possible that an exception procedure might be invoked automatically while the program is in the middle of a direct call to the same exception procedure. For example, if a program calls CLEANUP (an exception procedure) and is terminated by a DS system command, the exception procedure is invoked a second time because of the abnormal exit of the block in which the exception procedure was declared. Exception procedures that are called directly need to be written with this possibility in mind.

Example

```
PROCEDURE P1;  
  BEGIN  
    REAL A, B;  
    FILE MYFILE (KIND=DISK);  
    EXCEPTION PROCEDURE CLEANUP;  
      BEGIN  
        CLOSE (MYFILE, LOCK);  
      END; % of exception procedure cleanup  
  
    IF MYFILE.AVAILABLE THEN  
      BEGIN  
        OPEN (MYFILE);  
        CLEANUP; % a direct call to the exception procedure  
      END;  
    A := 17 * (B + 4);  
  END; % of procedure P1. The procedure cleanup will be  
    % invoked automatically if P1 is exited abnormally.
```

EXCEPTION PROCEDURE FORWARD Declaration

```

<exception procedure forward declaration>
  - EXCEPTION PROCEDURE —<exception procedure identifier>—————>
  -> FORWARD ——————|
  
```

Explanation

An exception procedure can be declared forward in the same manner as in other procedures. If an <exception procedure forward declaration> exists, then references to the procedure can occur before the EXCEPTION PROCEDURE declaration.

EXPORT Declaration

```

<export declaration>
  - EXPORT —————>
    [ — PROTECTED —<linkage class> ]
  -> <export object specification> —————>
    <export options>
  
```

<linkage class>

```

—<integer>—|
  
```

<export object specification>

```

—<procedure identifier>—|
  | <array identifier> — AS —<EBCDIC string literal>—|
  | <file identifier> —|
  
```

<export options>

```

— ( — LINKCLASS — = — PROTECTED —<linkage class> ) —|
  
```

Explanation

The EXPORT declaration in NEWP is similar to the EXPORT declaration in ALGOL, except that NEWP allows different export objects, and the LINKCLASS can be provided in two ways. If the linkage class for all objects exported in the EXPORT declaration is the same, the linkage class can be specified just after EXPORT; it then applies to the entire list of objects. If the objects in the export list have different linkage classes they can be specified individually after each <export object specification>.

The <linkage class> is an integer between 0 and 15. A library entry point can be exported with protection by including the PROTECTED clause in the EXPORT declaration for that entry point. Only programs belonging to appropriate linkage classes can be linked to a protected entry point. The linkage classes are defined in a table maintained by MCP LIBRARIAN. If the PROTECTED clause is not included, the linkage class is 0 (zero).

There are 16 linkage classes ranging from 0 to 15. Classes 0 and 1 are used by the MCP such that tasks of linkage class 0 can link only to objects of linkage class 0 and tasks of linkage class 1 are allowed to link to objects of any linkage class. The remaining classes are reserved for use by Unisys systems software.

EXPORT declarations are not allowed within module alternatives.

In-line procedures, procedures that are declared to have a scalar type, and procedures with scalar type parameters cannot appear in an EXPORT declaration.

Note: The keyword EXPORT is interpreted as a module <export list> when it appears in the module head of a MODULE declaration (old). EXPORT in any other context is interpreted as a library <export declaration>.

For information related to the EXPORT declaration, see "LIBRARY Declaration," "MODULE Declaration (Old)," and "In-Line Procedures" later in this section.

Examples

```
EXPORT MYPROG;

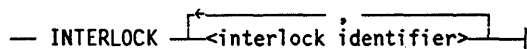
EXPORT [PROTECTED] YOURPROC, THEIRPROC;

EXPORT [PROTECTED 1] OURPROC;

EXPORT THEPROC (LINKCLASS = 2),
        APROC (LINKCLASS = 6);
```

INTERLOCK and INTERLOCK ARRAY Declarations

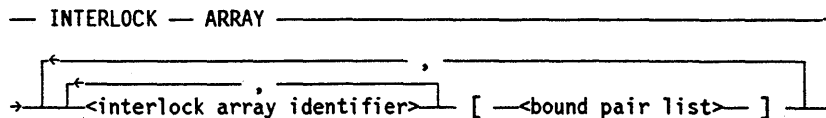
<interlock declaration>



<interlock identifier>



<interlock array declaration>



<interlock array identifier>



Explanation

An < identifier > declared to be an < interlock identifier > or an element of an interlock array can be used to protect a resource that is shared among several participating

processes. In many respects, the use of an interlock identifier or <interlock array identifier> is similar to the use of an event with the PROCURE and LIBERATE statements. However, using interlocks can improve significantly the run-time performance when compared with using the PROCURE and LIBERATE statements on events. For more information on PROCURE and LIBERATE statements, see the *A Series ALGOL Programming Reference Manual, Volume 1: Basic Implementation*.

The initial state of an interlock is FREE. For a complete list of the possible states of an interlock, refer to "LOCKSTATUS Function" in Section 6, "Expressions and Functions."

INTRINSIC Declaration

<intrinsic declaration>

— INTRINSIC <intrinsic identifier> <equation part>

Explanation

INTRINSIC declarations are provided especially for use in the MCP, where implicit references to MCP addresses can cause problems. The compiler is directed to use the address associated with the INTRINSIC declaration in place of a newly allocated cell in the segment dictionary (normal program) or to use a direct reference to the global environment within the MCP. This direct reference is disallowed and results in a compilation error.

Each intrinsic declared is assigned a stack cell. If you do not use an address equation, the cell is stored in the segment dictionary and contains an appropriate intrinsic binding word. If you supply an address through address equation, that address is used for references and no initialization is done by the compiler. The <equation part> is an UNSAFE construct; see "Address Equation" in Section 9, "UNSAFE Mode," before proceeding.

INTRINSIC declarations are associated by name to the references made by the compiler for implicitly called MCP intrinsics. When the compiler generates code to refer to an MCP intrinsic, it looks for a specified identifier, derived from the intrinsic name in the scope of the code being compiled. A list of these recognized intrinsic identifiers is shown in Table 4-1.

If the identifier is found and is of type INTRINSIC, the compiler uses the address associated with the identifier. If no useable intrinsic identifier is found, a compilation error is issued for MCP compilations. For non-MCP compilations, an intrinsic binding word is allocated in the segment dictionary.

The following table shows the recognized intrinsic identifiers, along with associated MCP identifiers and MCP intrinsic numbers, in hexadecimal form.

Table 4-1. Intrinsic Identifiers

Intrinsic Identifier	MCP Identifier	Number
INTRINSIC_STACKVECTOR	STACKVECTOR	02
INTRINSIC_MEMORY	MEMORY	04
INTRINSIC_ARRAYDEC	ARRAYDEC	07
INTRINSIC_BLOCKEXIT	BLOCKEXIT	0A
INTRINSIC_BADGOTO	GOTOSOLVER	0B
INTRINSIC_INTRINSICINFO	INTRINSICINFO	11
INTRINSIC_MYJOB	MYJOB	12
INTRINSIC_FREEZELIBRARY	FREEZELIBRARY	16
INTRINSIC_PROGRAMDUMP	PROGRAMDUMP	17
INTRINSIC_TIME	TIMEINTRINSIC	18
INTRINSIC_CLOSE	CLOSE	1B
INTRINSIC_POTL	POTL	23
INTRINSIC_POTC	POTC	24
INTRINSIC_POTH	POTH	25
INTRINSIC_ATTRIBUTEGRABBER	ATTRIBUTEGRABBER	29
INTRINSIC_TRUTHSETS	TRUTHSETS	26
INTRINSIC_ATTRIBUTEHANDLER	ATTRIBUTEHANDLER	2A
INTRINSIC_USERIOERROR	USERIOERROR	2E
INTRINSIC_LOADCONTROL	LOADCONTROL	32
INTRINSIC_SET_GET_LIBRARYSTATUS	SET_GET_LIBRARYSTATUS	36
INTRINSIC_SORT	SORT	45
INTRINSIC_RESIZEANDDEALLOCATE	RESIZEANDDEALLOCATE	46
INTRINSIC_CANCELLIBRARY	CANCELLIBRARY	4C
INTRINSIC_OPENP	OPENP	4E
INTRINSIC_DELIVERY	DELIVERY	4E
INTRINSIC_DESCRIPTOR_SIZE	DESCRIPTOR_SIZE	5D
INTRINSIC_LINKLIBRARY	LINKLIBRARY	63
INTRINSIC_EBCTOHEX	EBCTOHEX	65

continued

Table 4-1. Intrinsic Identifiers (cont.)

Intrinsic Identifier	MCP Identifier	Number
INTRINSIC_UNRAVEL	UNRAVEL	66
INTRINSIC_MUTATE	MUTATE	67
INTRINSIC_MYSELFER	MYSELFER	68
INTRINSIC_CONTINUER	CONTINUER	69
INTRINSIC_CLOSEP	CLOSEP	70
INTRINSIC_FORKCONTROLCARD	FORKCONTROLCARD	76
INTRINSIC_FIXHANDLER	FIXHANDLER	8E
INTRINSIC_DIRECTOR	DIRECTOR	95
INTRINSIC_CAUSEP	CAUSEP	96
INTRINSIC_SETORRESET	SETORRESET	99
INTRINSIC_PROCUREP	PROCUREP	9A
INTRINSIC_LIBERATEP	LIBERATEP	9C
INTRINSIC_COMBINEPPBS	COMBINEPPBS	AB
INTRINSIC_FORKHANDLER	FORKHANDLER	AF
INTRINSIC_EBCTOASC	EBCTOASC	B1
INTRINSIC_ASCTOHEX	ASCTOHEX	B2
INTRINSIC_ASCTOEB	ASCTOEB	B4
INTRINSIC_HEXTOEBDIC	HEXTOEBDIC	BD
INTRINSIC_HEXTOASCII	HEXTOASCII	BE
INTRINSIC_READLOCKTIMEOUT	READLOCKTIMEOUT	C2
INTRINSIC_CLOCKOFFPCW	CLOCKOFFPCW	C3
INTRINSIC_CLOCKONPCW	CLOCKONPCW	C4
INTRINSIC_CLOCKRESUMEPCW	CLOCKRESUMEPCW	C5
INTRINSIC_CLOCKSUSPENDPCW	CLOCKSUSPENDPCW	C6
INTRINSIC_GETSTRINGAREA	GETSTRINGAREA	C9
INTRINSIC_GETSTRINGPOOLSIZE	GETSTRINGPOOLSIZE	CC
INTRINSIC_RESETSTRINGPOOLSIZE	RESETSTRINGPOOLSIZE	CD

continued

Table 4-1. Intrinsic Identifiers (cont.)

Intrinsic Identifier	MCP Identifier	Number
INTRINSIC_ARRAYSEARCHP	ARRAYSEARCHP	D5
INTRINSIC_GETLIBATTRIBUTES	GETLIBATTRIBUTES	D5
INTRINSIC_SETLIBATTRIBUTES	SETLIBATTRIBUTES	DA
INTRINSIC_HIGHESTPNUM	HIGHESTPNUM	DE
INTRINSIC_HAPPENEDP	HAPPENEDP	E4
INTRINSIC_AVAILABLEP	AVAILABLEP	E5
INTRINSIC_MULTWAIT	MULTWAIT	E9
INTRINSIC_SIMPLEWAIT	SIMPLEWAIT	EA
INTRINSIC_DELINKLIBRARY	DELINKLIBRARY	EB
INTRINSIC_DESC_HIDING	DESC_HIDING	EE
INTRINSIC_ILOK_OK	ILOK_OK	F0
INTRINSIC_ILOK_STATUS	ILOK_STATUS	F1
INTRINSIC_ILOK_BREAK	ILOK_BREAK	F2
INTRINSIC_ILOK_ARROGATE	ILOK_ARROGATE	F3
INTRINSIC_ILOK_LOCKING	ILOK_LOCKING	F4
INTRINSIC_ILOK_UNLOCKING	ILOK_UNLOCKING	F5
INTRINSIC_FA_JACKET	FA_JACKET	FA

Note: Address equating another declaration to a declared intrinsic identifier has one unusual effect. If the stack cell was associated with the intrinsic (the intrinsic was the first declaration assigned to the cell), the normal initialization is overridden and the new declaration is used to initialize the cell.

Examples

```
PROCEDURE CAUSEP=( 0,150 ) (E,HOW);

INTRINSIC INTRINSIC_CAUSEP = CAUSEP;

INTRINSIC INTRINSIC_EBCTOHEX;

TRANSLATETABLE EBCTOHEX = INTRINSIC_EBCTOHEX (. . .);
```

LABEL Declaration

<label declaration>

— LABEL [BAD] <label identifier>

Explanation

In NEWP, a bad GO TO is a GO TO statement that branches out of the segment or procedure in which the GO TO statement appears. A label that is the object of a bad GO TO statement must be declared in a label declaration that includes the [BAD] syntax.

Example

```
LABEL [BAD] ENDITALL, ERROREXIT;
```

LIBRARY Declaration

<library declaration>

— LIBRARY <library identifier> (<library attribute specifications>)

<library attribute specifications>

<Boolean library attribute specification>
 <string library attribute specification>
 <mnemonic library attribute specification>

<Boolean library attribute specification>

— <Boolean-valued library attribute name> = TRUE | FALSE

<Boolean-valued library attribute name>

— SYSTEMLIB —

(All other syntax is identical to ALGOL.)

Explanation

In NEWP, a value for the Boolean-valued library attribute SYSTEMLIB can be specified. When SYSTEMLIB is TRUE, the associated library is to be initiated as a system library, which allows it to access protected library entry points. For more information, see “EXPORT Declaration” in this section. Use of this attribute requires that the MCP compiler control option be set.

In NEWP, a maximum of 150 libraries can be declared in a single program.

For more information related to the LIBRARY declaration, see "EXPORT Declaration" in this section and Section 8, "Compiler Controls."

MODULE Declaration

Note: Wherever possible, use this version of the MODULE declaration. If you are unable to use this version for work on previous releases, use the version described in "MODULE Declaration (Old)" later in this section.

<module declaration>

```

MODULE <module identifier> ; <module head>
BEGIN <module identifier> ;
<module body>
END <module identifier> ;
    
```

<module identifier>

```
<identifier>
```

<module head>

```
<declaration> ;
```

<module body>

```

<declaration> ;
<initialization procedure>
<alternative>
    
```

<interface declaration>

```
INTERFACE <interface>
```

<interface>

```
<interface identifier> <interface body>
```

<interface body>

```
( <exportable identifier> <interface identifier> )
```

Declarations

<moduleexport declaration>

— MODULEEXPORT —<interface identifier>—

<moduleimport declaration>

— MODULEIMPORT —<interface identifier>—

<remote module declaration>

— INCLUDE —<module identifier>—

<interface identifier>

—<identifier>—

<exportable identifier>

—<identifier>—

Explanation

A module, in NEWP, is a self-contained package that includes a number of declarations. By default, none of its declarations is visible outside the bounds of the module, and no identifiers declared in any other modules (nested, sibling, or enclosing) are visible within the module. The module import and export declarations provide the means to allow visibility of declarations across module boundaries.

On the export side, INTERFACE declarations collect items declared within the module and group them under the interface identifier. The interface can then be exported with the MODULEEXPORT declaration, which makes it available for import into modules outside the scope of the exporting module. No MODULEEXPORT declaration is necessary to import an interface into a nested module.

The following restrictions govern the items that can appear in an interface:

- Items imported from outside the module cannot be reexported.
- The enumerated literals of an enumerated data type cannot themselves be included in an interface but are automatically included when the type, descendant of the enumerated type, or structure type variable descended from the enumerated type is included.

An item can appear in more than one interface. An item imported by a nested module can be included in an interface formed by the parent or grandparent, for example. An item must be declared before it can be included in an interface.

If an interface identifier appears in the list of items (for another interface), all the items contained in the referenced interface are also contained in the interface being declared. This provides a convenient mechanism for building a large interface out of smaller ones. Components of a composite interface can also be overlapped. Only interfaces already defined within the same module, or imported from a nested module, can be used to build this kind of composite interface.

If an interface identifier from a nested submodule appears in the `MODULEEXPORT` declaration, the entire interface is reexported, under its own name, from the parent module. This provides a mechanism for grouping modules, while still retaining their individual identities.

The `MODULEIMPORT` declaration makes the items collected in the named interface visible within the importing module. An interface appears as a window across the boundary between two modules. If a module imports several interfaces exported from another module and items appear in more than one of these interfaces, there is no conflict or ambiguity. The windows simply overlap.

A `MODULEIMPORT` declaration is required for all uses of an identifier within a module different from that in which it was declared. This means that interfaces must be declared and imported from parent to child, child to parent, and sibling to sibling. Alternatives within modules are not considered to be nested modules for this purpose, and therefore the alternatives directly inherit the environment of the containing module.

Interfaces exported from a nested module can be imported by the parent or a sibling. Items exported from a nested module, in the head of the parent, can be included in interfaces exported from the parent. In addition, entire submodule interfaces are available for reexport under the original interface name.

The `REMOTE MODULE` declaration is a feature intended for use on large modularized programs, such as the MCP. This declaration reorganizes the relationships among the modules without physically reorganizing the source files. It is not intended as a tool for construction of new programs.

The effect of the `REMOTE MODULE` declaration is as if the actual text of the designated module appeared at the point of the `INCLUDE` portion of the declaration. All scope rules are applied as if this were the case. The module included is then no longer eligible for compilation again either through another `INCLUDE` portion or in the normal location defined by the physical order of appearance. The included module is effectively removed from the source. The `REMOTE MODULE` declaration is not related to the compiler control option `$INCLUDE`.

The target module must appear in the source as a member of the same contiguous group of modules as the module that invokes the `REMOTE MODULE` declaration. The `REMOTE MODULE` declaration can appear only as a declaration in the heading or the body of the host module, not within a nested module or procedure. However, a nested module can include another nested module from its own contiguous group.

Scope Rules

The declaration of an interface makes the interface identifier known throughout the rest of the module declaring it. This includes any later nested modules. The export of an interface identifier makes that identifier known throughout the rest of the environment containing the exporting module. The exporting module includes later sibling modules. If the exporting module is an outermost module, the interface name is known throughout the rest of the program. If a module is declared within a procedure or block, the scope rules restrict the visibility of any exported interfaces to the rest of the body of the procedure or block.

The ALGOL rules apply to the declaration of interfaces with the same name. A potential conflict can arise if two or more interfaces with the same name appear in the same scope. This can occur by local declaration, by export from a nested or sibling submodule, or by import from the outer environment. If one of the conflicting declarations appears within a scope properly nested within the scope containing the other interface, the outer name becomes invisible in the inner scope. If the conflicting declarations arise in the same scope, the situation is flagged as a compilation error. For more information on scope, refer to the *A Series ALGOL Programming Reference Manual, Volume 1: Basic Implementation*.

The declaration and export of an interface does not, however, make the items collected in the interface automatically visible outside of the declaring module, or within any nested submodules. When an interface identifier appears in a MODULEIMPORT declaration, the identifiers contained within the interface become visible through the rest of the scope within which the import declaration appears. These identifiers appear to belong to that scope. The ALGOL scope rules apply to the point of import.

MODULE, INTERFACE, MODULEEXPORT, and MODULEIMPORT declarations are true declarations. As such, they can appear anywhere that declarations are allowed. However, if an interface is to be exported, its declaration must appear in the heading of the module, not nested within any procedure or block.

The order of compilation and the scope definition of *after* in NEWP is first line to last line, with the following exception related to modules: contiguous sibling modules are compiled as a group. Within the group, the order of compilation is heads (first to last), then bodies. Nested submodules, including groups of adjacent sibling submodules, are compiled where they appear within this order. If a group of modules is nested in the head of the parent module, the compilation of the bodies of the nested modules is deferred to the beginning of the body of the parent.

Imports must be specified before they are used, when establishing a compilation order. Appearance of an identifier within a DEFINE body does not constitute a use. However, an invocation of such a DEFINE would be a use. A given interface can be imported safely more than once into the same environment.

MODULE Declaration (Old)

Note: Do not use this version of the *MODULE* declaration unless your work on previous releases requires it. Wherever possible, use the version described in "MODULE Declaration" in this section.

<module declaration>

```

MODULE <module identifier> ; <module head>
BEGIN <module identifier> ;
<module body>
END <module identifier> ;

```

<module identifier>

```
<identifier>
```

<module head>

```

<export list> ; <supply declaration>
<declaration> ;

```

<export list>

```
EXPORT <identifier>
```

<module body>

```

<declaration> ;
<initialization procedure>
<alternative>

```


Declarations

```

<alternative>
  → ALTERNATIVE —<alternative identifier>————→
  → ┌ SELECT —<constant Boolean expression> ───┐ ; ───→
  → └──────────────────────────────────────────┘
  → BEGIN —<alternative identifier>— ; ───→
  → ┌──────────────────────────────────────────┐
  → │ ┌ <declaration> ───┐ ; ───┘
  → └──────────────────────────────────────────┘
  → END —<alternative identifier>— ; ───→

```

```

<alternative identifier>
  → <identifier> ───┘

```

```

<initialization procedure>
  → ┌──────────────────────────────────────────┐
  → │ ┌ <procedure type> ───┐ INITIALIZATION PROCEDURE ───→
  → └──────────────────────────────────────────┘
  → <procedure heading> ───<procedure body> ───┘

```

Explanation

The **MODULE** declaration allows logically related declarations to be grouped together. Items declared within a module are protected, in the sense that they are not visible to other modules unless specified in an **EXPORT** list.

Exported identifiers must be declared in the <module head>. The form of the declarations for procedures to be exported depends on whether or not the procedure is declared to be **INLINE** (see “In-Line Procedures” in this section). Non-in-line procedures to be exported are declared in the module head as they would be declared anywhere else, except that the <procedure body> must be **LIBRARY**, **EXTERNAL**, **NULL**, or **FORWARD**; if the procedure body is **FORWARD**, the procedure must be fully declared in the <module body>. In-line procedures to be exported cannot be declared **EXTERNAL**, **NULL**, or **FORWARD** and must be fully declared in the module head.

Note: The keyword **EXPORT** is interpreted as a module <export list> when it appears in the module head of a **MODULE** declaration. **EXPORT** in any other context is interpreted as a library **EXPORT** declaration. When a library entry point is declared in the module head by using the <library entry point specification>, any other entry points to the library to be used in the module must also be declared in the module head. If no entry points are to be used in the module head, then this restriction is inapplicable.

MODULE <module identifier>, **BEGIN** <module identifier>, and **END** <module identifier> must be the first tokens on the records on which these module identifiers appear. In a single <module declaration>, the three occurrences of the module identifier must all be the same identifier. The **BEGIN** <module identifier> and the **END** <module identifier> cannot be part of an **INCLUDE** file, nor part of a **define**.

Modules can be nested up to 50 deep. Selectable modules can be used at any point in the nesting. For more information, see the “**SELECT** Statement” in Section 5, “Statements.”

Declarations

Within a given outer level module, the code for all initialization procedures is placed in one single segment. For example, the initialization procedure of a selectable submodule is in the same segment as the initialization procedure of its selectable parent when modules are configured as such.

Items declared to have scalar types can be exported by and imported into modules.

Alternatives

The specification of alternatives allows the compile-time or run-time selection of one group of declarations from a list of one or more alternative groups. The selection is made at compile-time by providing a **SELECT** clause on the alternative declaration whose Boolean expression evaluates to **TRUE**. Only one alternative can be selected at compile time. Run-time selection is made by an `< initialization procedure >`, which must be declared if alternatives are declared (and can be declared even if no alternatives are declared).

EXPORT declarations are not allowed within alternatives.

If an initialization procedure occurs in a module, all items declared in the module, except the initialization procedure, are unavailable until the initialization procedure is entered. At that time, the items declared in the module but outside any alternatives are initialized. Items declared in an alternative are not available until, and unless, a **SELECT** statement for that alternative is executed. For more information, see "SELECT Statement" in Section 5, "Statements."

Items declared in a module but outside all alternatives are available inside all alternatives in that module.

If a procedure is declared inside the alternatives but is visible outside the alternatives, either because it is exported or because it is declared **FORWARD**, its actual declaration must appear in every alternative in the module and must have the same procedure heading in every declaration. That is, the procedure type, number of parameters, type of parameters, and all other information contained in the procedure heading (except formal parameter identifiers) must be identical.

In a single `< alternative >` specification, the three occurrences of `< alternative identifier >` must all be the same identifier. Alternatives cannot contain procedures declared to be **EXTERNAL**.

The **NEWP** compiler attempts to reuse **D[0]** cells for different alternatives within the same module. Cells that cannot be reused are those holding segment descriptors, value arrays, library templates and markers, and double-precision items. The attempt to reuse **D[0]** cells occurs only when the **MCP** control option is set.

Initialization Procedures

An initialization procedure is declared much like a standard procedure, except that the keyword **INITIALIZATION** must follow the procedure type (if any) and must precede the keyword **PROCEDURE**. Initialization procedures are the only procedures that can contain **SELECT** statements.

Initialization procedures are subject to the following restrictions:

- An initialization procedure can occur only as the last declaration in a module body.
- An initialization procedure can be executed only once. An INVALID OPERATOR fault occurs if an attempt is made to execute an initialization procedure a second time.
- The SEGMENT block directive cannot be specified in the block directives for an initialization procedure.
- The INLINE block directive cannot be specified in the block directives for an initialization procedure.

For more information related to the MODULE declaration, see “EXPORT Declaration,” “In-Line Procedures,” and “SUPPLY Declaration” in this section and “SELECT Statement” in Section 5, “Statements.”

Example

```

BEGIN
MODULE PHYSICALIO;
  EXPORT PHYSICALIO_INITIALIZATION,
         DOCHARIO,
         T;
  BOOLEAN INITIALIZATION PROCEDURE
    PHYSICALIO_INITIALIZATION(WHICHONE);
    VALUE WHICHONE; BOOLEAN WHICHONE; FORWARD;
  PROCEDURE DOCHARIO; FORWARD;
  INTEGER T;
BEGIN PHYSICALIO;
  REAL R;
  ALTERNATIVE MLIP_PHYSICALIO;
  BEGIN MLIP_PHYSICALIO;
    INTEGER I;
    PROCEDURE DOCHARIO;
      BEGIN
      END DOCHARIO;
  END MLIP_PHYSICALIO;
  ALTERNATIVE MPX_PHYSICALIO;
  BEGIN MPX_PHYSICALIO;
    PROCEDURE DOCHARIO;
      BEGIN
      END DOCHARIO;
    REAL PROCEDURE IOFINISH68;
      BEGIN
      END IOFINISH68;
  END MPX_PHYSICALIO;
  BOOLEAN INITIALIZATION PROCEDURE
    PHYSICALIO_INITIALIZATION(WHICHONE);
    VALUE WHICHONE; BOOLEAN WHICHONE;
  BEGIN
    IF WHICHONE THEN
      SELECT(MLIP_PHYSICALIO)

```


In addition, the following faults are available in NEWP:

- LOCKEDFAULT(20)
- LIBLINKFAULT(21): Occurs if an attempted library linkage is unsuccessful
- MEMORYFAIL1(23): B 7900 fault
- PRIVILEGEDINSTRUCTION(24): B 7900 fault
- PARITYFAIL1(25): B 7900 fault

If the B 7900 compiler control option is TRUE, the following faults are not included in ANYFAULT fault list, although they can be specified as individual fault names:

- LOOP
- MEMORYPARITY
- INVALIDADDRESS
- SCANPARITY
- INVALIDPROGRAMWORD
- MEMORYFAIL1
- PARITYFAIL1

For information about the ANYFAULT fault name, see the *A Series ALGOL Programming Reference Manual, Volume 1: Basic Implementation*. For information about the B 7900 compiler control option, see Section 8, "Compiler Controls."

Example

```
ON ANYFAULT,
  BEGIN
  RUNSTATUS := FIRSTPROC_FAULT;
  GO TO ERRLABEL;
  END;
```

OUTPUTMESSAGE ARRAY Declaration

Outputmessage arrays are declared as in ALGOL except for the following differences:

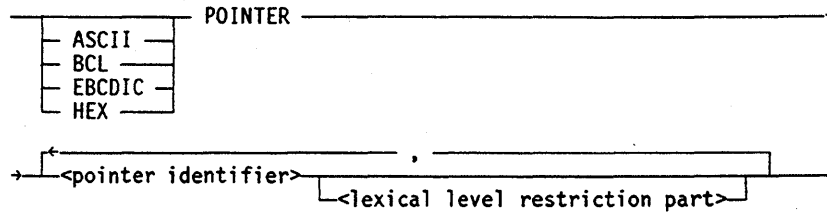
- Implicit string concatenation in NEWP requires at least one blank, or the use of a <string concatenation operator> .
- NULL or EMPTY strings are not allowed, but a string of one or more blanks is allowed.

For example:

```
" "
```

- A <numeric constant> is accepted at all places where a number is accepted.

POINTER Declaration



Explanation

Pointers can be declared with a size specification (for example, HEX). If the character size is not specified, the default is EBCDIC.

The <lexical level restriction part> construct has the same syntax and semantics in NEWP as it does in ALGOL. Up-level pointers are allowed only if the MCP compiler control option is set or the block is in UNSAFE (UPLEVEL) mode.

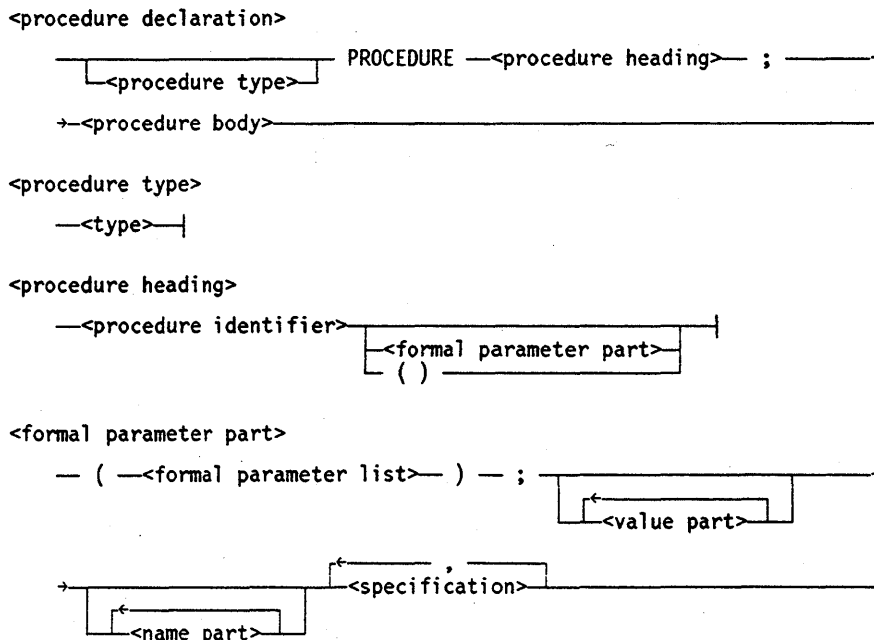
Syntax errors are given for pointer and string size mismatches. For example, if PTR is declared as an EBCDIC pointer, the following statement causes a syntax error:

```
REPLACE PTR BY 4"FF00"; % SHOULD BE 48"FF00"
```

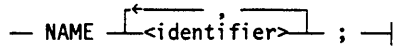
Example

```
EBCDIC POINTER PTRIN, PTROUT;
```

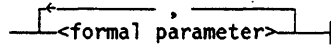
PROCEDURE Declaration



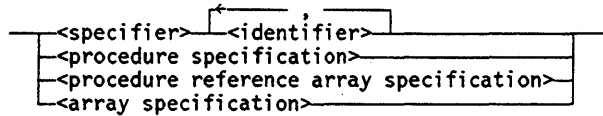
<name part>



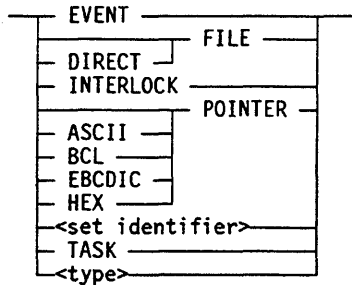
<formal parameter list>



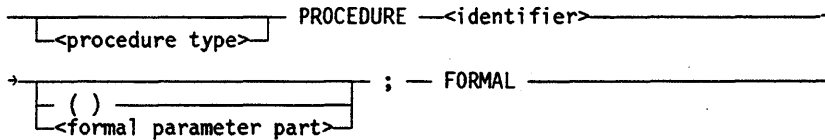
<specification>



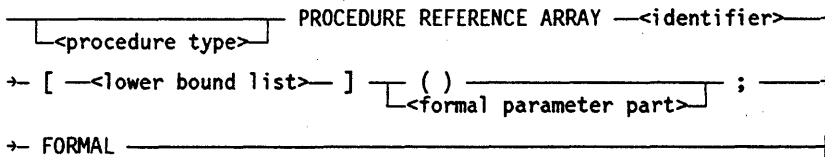
<specifier>



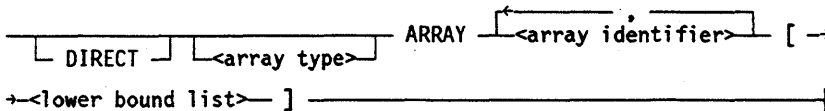
<procedure specification>



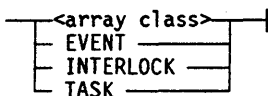
<procedure reference array specification>



<array specification>

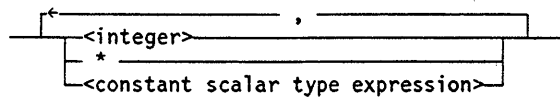


<array type>

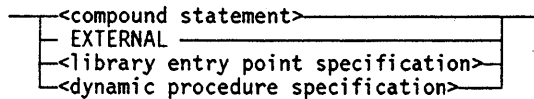


Declarations

<lower bound list>



<procedure body>



(All other syntax is identical to ALGOL.)

Explanation

Procedures in NEWP are similar to procedures in ALGOL, with the following exceptions:

- Parameters can be passed as call-by-reference, call-by-value, or call-by-name. Parameters passed as call-by-name are permitted only for in-line procedures.
- Procedures can be declared `INLINE` through the `INLINE` block directive.
- The procedure value of a typed procedure can be accessed explicitly within the scope of the procedure through the `<procedure identifier>.VALUE` construct.
- Use of `< name part >` is supported for `INLINE` procedures.
- The names of formal parameters to a procedure cannot be used again as local identifier declarations in the outer block of the procedure.

Call-by-name parameters cannot be used in the `FOR` specification of `POINTER` declarations.

`REAL`, `INTEGER`, `BOOLEAN`, `DOUBLE`, `POINTER`, `DESCRIPTOR`, and `WORD` can be call-by-name parameters. `DESCRIPTOR` and `WORD` parameters can be used only in the appropriate `UNSAFE` mode.

Parameter Passing

Parameters can be passed as call-by-value, call-by-reference, or call-by-name. The default is call-by-reference. No more than 63 parameters are allowed on one procedure. Thunks are not implemented in NEWP. To pass a parameter as call-by-value, the `< value part >` must appear in the procedure heading. To pass a parameter as call-by-name, the `< name part >` must appear in the procedure heading of an `INLINE` procedure.

Actual parameters passed to call-by-reference formal parameters must generate address references. Constants and arithmetic expressions do not generate address references. However, conditional and case expressions are allowed if each branch generates an address reference. For parameters passed as call-by-reference, the types of the actual and formal parameters must agree. For example, a variable of type `REAL` cannot be passed as call-by-reference to a formal parameter of type `DOUBLE` or `INTEGER`.

If a call-by-reference formal parameter has a scalar type, the actual parameter passed to that formal parameter must be of the same type as the formal parameter. If a call-by-value formal parameter has a scalar type, the type of the actual parameter passed to that formal parameter must be assignment-compatible with the type of the formal parameter.

If a call-by-reference formal parameter is a set type, the actual parameter and the formal parameter must have been declared with the same < set identifier >. Sets cannot be passed as call-by-value.

Call-by-name parameters do not cause anything to be stacked upon entry to the `INLINE` procedure. Instead, the actual parameter is evaluated each time the formal parameter is used.

If you store into a given call-by-name formal parameter or you use the parameter as the subject of nonevaluative type transfer (for example, `WORD AT X`), then its corresponding actual parameter must be of a form suitable for the actual parameter to a call-by-reference formal parameter. The actual parameter must be capable of generating an address reference. Implicit type coercion because of type mismatch is allowed to the same extent as for call-by-reference parameters and is handled with nonevaluative type transfer, producing the same semantics. For example, if `WORD W` is passed to `BOOLEAN` parameter `B`, it is treated as if `BOOLEAN AT W` had been passed. For information on implicit type coercion, see the *A Series ALGOL Programming Reference Manual, Volume 1: Basic Implementation*.

If you do not store into a given call-by-name formal parameter or you do not use the parameter as the subject of nonevaluative type transfer, then its corresponding actual parameter must be in a form suitable for the actual parameter to a call-by-value formal parameter. The actual parameter need not be capable of generating an address reference. Implicit type coercion because of type mismatch is allowed to the same extent as for call-by-value parameters and is handled with evaluative type transfer, producing the same semantics. For example, if `WORD W` is passed to `BOOLEAN` parameter `B`, it is treated as if `BOOLEAN(W)` had been passed.

The syntax for specifying procedures as formal parameters differs between `NEWP` and `ALGOL`. `NEWP` does not support run-time parameter checking; therefore, all parameters of formal procedures must be specified.

Each dimension of a formal array parameter can be a scalar type. The scalar type must be a bounded, discrete type. If the lower bound of a dimension is specified by a < constant scalar type expression >, the type of the dimension is the type of the expression. When the procedure is invoked, the dimension of the actual array passed to the formal array must have the same scalar type as the corresponding dimension of the formal array. A lower bound specified by an asterisk (*) is of type `INTEGER`.

In `NEWP` (unlike in `ALGOL`), `DEFINES` are not expanded when the compiler is processing the individual identifiers in the < value part > or < specification part > of a procedure heading.

In-Line Procedures

In-line procedures combine some of the efficiency of DEFINEs with the semantics of procedures. Each invocation of an in-line procedure results in an in-line expansion of its code at the point of the invocation.

In-line procedures with local ARRAY or EVENT declarations are executed as procedures, that is, as block entry by means of a Program Control Word (PCW). Unlike normal procedures, the code for an enterable in-line procedure is always placed in the same segment as the code that invokes the in-line procedure. There is a performance penalty associated with enterable in-line procedures, so the declaration of local arrays and events in in-line procedures is discouraged.

An in-line procedure is declared by including the keyword `INLINE` as a block directive associated with the first `BEGIN` of the procedure body (see “Block Directives” in Section 8, “Compiler Controls”). In-line procedures are subject to the following restrictions:

- An in-line procedure that is exported from a module must be fully declared in the module head. For more information, refer to “MODULE Declaration” in this section.
- The only imported items an exported in-line procedure can use are those exported from modules declared prior to the module containing the exported in-line procedure.
- In-line procedures must not be recursive.
- An in-line procedure cannot be passed as a formal parameter to a procedure that is not an in-line procedure. However, both in-line and other procedures can be passed as formal parameters to in-line procedures.
- The `<procedure identifier>` referred to in a `RUN`, `PROCESS`, or `FORK` statement must not be that of an in-line procedure.
- The procedure identifier referred to in the `CONTROL` form of the `<freeze statement>` must not be that of an in-line procedure.
- An in-line procedure cannot be exported as a library entry point.
- Noninvocation references to in-line procedures (such as in a `MAKEPCW` or `LEXOFFSET` call) are, in general, not allowed. However, `<procedure identifier>.VALUE` is a noninvocation reference that can reference an in-line procedure from within the body of the procedure.
- The `RETURN` and `EXIT` functions for `NEWP` are not allowed within the body of an in-line procedure. For more information, refer to “Intrinsics (UNSAFE)” in Section 9, “UNSAFE Mode.”
- An in-line procedure cannot be declared `FORWARD`, `EXTERNAL`, or `NULL`.
- An initialization procedure cannot be an in-line procedure. For more information, see “MODULE Declaration” in this section.
- An in-line procedure cannot be used in a `SORT` statement.

In-line procedures are similar in many respects to both DEFINEs and regular procedures. However, there are differences that should be taken into account when you are deciding which should be used.

The primary difference between an in-line procedure and a DEFINE is that in-line procedures apply regular scope rules, evaluate parameters like regular procedures do (except for NAME parameters), and are treated as procedures for LINEINFO and XREF. For information on LINEINFO and XREF, see the *A Series ALGOL Programming Reference Manual, Volume 1: Basic Implementation*.

In-line procedures differ from regular procedures in several ways. Functionally, in-line procedures provide NAME parameters and the INHERITSTATE block directive. Regular procedures always require a PCW, while in-line procedures generally do not; therefore, no stack cell is allocated for one. The code of an in-line procedure is always present in the segment that contains the call, so you do not need to think about where to take presence bits (pbits). The TADS option can be used with regular procedures, but not with in-line ones (a syntax error is received), though in many cases simply omitting the [INLINE] block directive when TADS is set can help you to avoid receiving an error.

Because all the code for an in-line procedure exists in each place it is invoked (rather than just having the procedure entry code), using a large in-line procedure in many places affects the code file size.

The cost of doing an ENTR/EXIT operation is avoided with an in-line procedure (except when ARRAY or EVENT declarations are used), but having a large number of parameters or locals offsets this advantage. This is because each parameter and local must be deleted off the stack at the end of the in-line. The point at which the cost of deleting the parameters and locals exceeds the savings from not doing the ENTR/EXIT operations differs on each machine.

Finally, in-line procedures cannot be passed as parameters, and if exported from a module, these procedures must be fully declared in the module head.

In general, you must consider a number of these factors when you are deciding whether to use a define, in-line, or regular procedure. The functionality (that is, the use of NAME parameters, the use of INHERITSTATE block directives, the acceptability of p-bits, the use of TADS, and so on) should be considered first. This should be weighed against the PCW requirement of a regular procedure (some primitive software can have problems addressing PCWs). The final factor should be the size and performance of the code file. If you take all these factors into account, you can make an appropriate choice.

The LINEINFO references for an expanded in-line procedure include both the sequence number or numbers of the invoking code and the sequence number of the invoked code. These sequence numbers are listed in order from most recently invoked procedure or block to least recently invoked procedure or block. For more information on LINEINFO, see the *A Series ALGOL Programming Reference Manual, Volume 1: Basic Implementation*.

Declarations

More efficient code is emitted for certain typed in-line procedures. The value returned is generated on top of the stack and no cell to hold the value is pushed to the stack when the following conditions apply:

- No parameters to the in-line procedure need be stacked. Call-by-name parameters never stack their actual parameters. Call-by-reference parameters do not stack their actual parameters when simple variables are passed to them. Call-by-value parameters do not stack their actual parameters when constants are passed to them.
- The in-line procedure has no local variables requiring stack cells. (DEFINES and CONSTANTS are acceptable.)
- The only reference to the procedure value is the final statement, which stores to it. This statement must be executed unconditionally and must not be spanned by a looping structure or followed by a label.

Note: If the in-line procedure ends with a combination of IF THEN . . . ELSE statement constructs that depend on a constant expression or expressions, then only the statement or compound statement that is executed need meet these constraints.

Procedure Value

Within the body of a typed procedure, the value of the procedure can be accessed through the following syntax:

<procedure identifier>.VALUE

This construct is treated as a primary of the same type as the declared procedure and, as such, can be used in expressions, assignments, and address equations.

The value returned by a scalar type procedure is undefined if no value is assigned to the procedure during the execution of the procedure.

Example

```
REAL PROCEDURE PROC;  
  BEGIN  
    BOOLEAN B;  
    PROC.VALUE := 10;  
    IF PROC.VALUE=20 THEN  
      .  
      .  
    END PROC;
```

Dynamic Procedure Specification

Dynamic procedure specification is the same in NEWP as it is in ALGOL, except that the <selection procedure identifier> must specify an untyped procedure with three parameters. The first parameter must be a real array, specified with a constant (not star-bounded) lower bound. The second parameter must be a call-by-value integer. The third parameter must be a fully specified untyped procedure with one parameter that is a task. When the MCP invokes the 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.

The following example shows the use of a <dynamic procedure specification>. The example assumes that its object file is named OBJECT/SAMPLE/DYNAMICLIB and that a library called OBJECT/SAMPLE/LIBRARY that exports the procedure DATEANDTIME as DAYTIME also must be available.

Examples

The following NEWP library, compiled as OBJECT/SAMPLE/LIBRARY, provides its entry points directly.

```

BEGIN
  ARRAY MSG[0:120];

  INTEGER PROCEDURE FACT(N);
  INTEGER N;

  BEGIN
    IF N LSS 1 THEN
      FACT := 1
    ELSE
      FACT := N * FACT(N - 1);
    END; % OF FACT

  PROCEDURE DATEANDTIME(TOARRAY,WHERE);
  ARRAY TOARRAY[*];
  INTEGER WHERE;

  BEGIN
    REAL T;
    POINTER PTR;

    T := TIME(7);
    PTR := POINTER(TOARRAY,8) + WHERE;
    CASE T.[5:6] OF
      BEGIN
        0: REPLACE PTR:PTR BY "SUNDAY, ";
        1: REPLACE PTR:PTR BY "MONDAY, ";
        2: REPLACE PTR:PTR BY "TUESDAY, ";
        3: REPLACE PTR:PTR BY "WEDNESDAY, ";
        4: REPLACE PTR:PTR BY "THURSDAY, ";
      END
    END
  END

```

Declarations

```
5: REPLACE PTR:PTR BY "FRIDAY, ";
6: REPLACE PTR:PTR BY "SATURDAY, ";
END;
REPLACE PTR BY T.[35:6] FOR 2 DIGITS, "-",
          T.[29:6] FOR 2 DIGITS, "-",
          T.[47:12] FOR 4 DIGITS, ", ",
          T.[23:6] FOR 2 DIGITS, ":",
          T.[17:6] FOR 2 DIGITS, ":",
          T.[11:6] FOR 2 DIGITS;
END; % OF DATEANDTIME

EXPORT FACT,DATEANDTIME AS "DAYTIME";
REPLACE POINTER(MSG) BY
  " - SAMPLE LIBRARY STARTED",
  " " FOR 94;
DATEANDTIME(MSG,60);
FREEZE(TEMPORARY);
END.
```

The following NEWP library, compiled as OBJECT/SAMPLE/DYNAMICLIB, illustrates dynamic and indirect library linkage.

```
BEGIN [UNSAFE (FORK)]
TASK LIB1TASK, LIB2TASK;

PROCEDURE DYNLIB1;
% LIBRARY PROVIDED DYNAMICALLY AND INDIRECTLY
BEGIN % PRINTS DATE WITH TIME
LIBRARY SAMLIB (TITLE = "OBJECT/SAMPLE/LIBRARY.");
PROCEDURE DAYTIME (TOARRAY, WHERE);
  ARRAY TOARRAY [*];
  INTEGER WHERE;
  LIBRARY SAMLIB;
EXPORT DAYTIME;
FREEZE (TEMPORARY);
END; % OF DYNLIB1

PROCEDURE DYNLIB2;
% LIBRARY PROVIDED DYNAMICALLY
BEGIN % PRINTS OUT DATE WITHOUT TIME
PROCEDURE DAYTIME (TOARRAY, WHERE);
  ARRAY TOARRAY [*];
  INTEGER WHERE;

  BEGIN
  REAL T;
  T := TIME (7);
  REPLACE POINTER (TOARRAY, 8) + WHERE
  BY T.[35:06] FOR 2 DIGITS, "-",
  T.[29:06] FOR 2 DIGITS, "-",
  T.[47:12] FOR 4 DIGITS;
  END; % OF DAYTIME
```

```

EXPORT DAYTIME;
  FREEZE (TEMPORARY);
  END; % OF DYNLIB2

% THE SELECTION PROCEDURE
PROCEDURE THESELECTIONPROC (LIBPARAM, LIBPARAMLEN, NAMINGPROC);
  VALUE LIBPARAMLEN;
  ARRAY LIBPARAM [0];
  INTEGER LIBPARAMLEN;
  PROCEDURE NAMINGPROC (LIBTASK);
    TASK LIBTASK; FORMAL;

  BEGIN
    IF POINTER(LIBPARAM) EQL "WITH TIME" THEN
      BEGIN
        IF LIB1TASK.STATUS NEQ VALUE (FROZEN) THEN
          PROCESS DYNLIB1 [LIB1TASK];
        NAMINGPROC (LIB1TASK);
      END
    ELSE
      BEGIN
        IF LIB2TASK.STATUS NEQ VALUE (FROZEN) THEN
          PROCESS DYNLIB2 [LIB2TASK];
        NAMINGPROC (LIB2TASK);
      END;
    END; % OF THE SELECTION PROCEDURE

  PROCEDURE DAYTIME (TOARRAY, WHERE);
    ARRAY TOARRAY [*];
    INTEGER WHERE;
    BYCALLING THESELECTIONPROC;

  EXPORT DAYTIME; % PROVIDED DYNAMICALLY
  FREEZE (TEMPORARY);
  END.

```

The following example invokes the library in the preceding example:

```

BEGIN
  LIBRARY MYLIB (TITLE = "OBJECT/SAMPLE/DYNAMICLIB.");

  PROCEDURE DAYTIME (A, W);
    ARRAY A [*];
    INTEGER W;
    LIBRARY MYLIB;

  REAL T;
  INTEGER X,Y;
  ARRAY DATIME [0:120];

  REPLACE MYLIB.LIBPARAMETER BY "WITH TIME.";

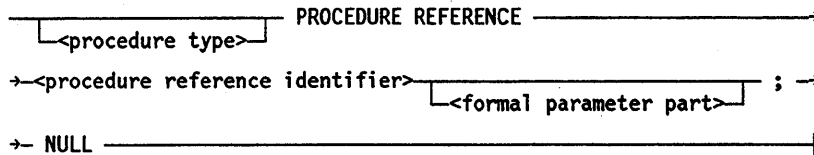
```



```
X := 13;
Y := 40;
DAYTIME (DATIME [*], Y);
END.
```

PROCEDURE REFERENCE Declaration

<procedure reference declaration>



Explanation

A <procedure reference identifier> can appear anywhere that an element of a procedure reference array can appear. For information on the <procedure reference array declaration> see the *A Series ALGOL Programming Reference Manual, Volume 1: Basic Implementation*.

Before a procedure reference identifier can be used as a parameter, as a primary, or in a procedure reference statement, the identifier must be initialized in a procedure reference assignment. If no initialization is done, a run time error occurs.

A <procedure reference declaration> cannot appear in the formal parameter part of a procedure declaration, of a procedure reference array declaration, or of another procedure reference declaration. However, a procedure reference identifier can be passed as an actual parameter to a formal procedure that is of the same type and that has the same parameter descriptions.

SEGMENT Declaration

<segment declaration>



<segment identifier>



Explanation

The SEGMENT declaration defines one or more identifiers for use in referring to code segments. OUTERBLOCK is a predeclared segment identifier that refers to the code segment containing the code for the outer block of the program.

Segment identifiers can be used in the SEGMENT block directive to specify the segment into which the compiler is to put the code for that block. For more information, refer to "Block Directives" in Section 8, "Compiler Controls."

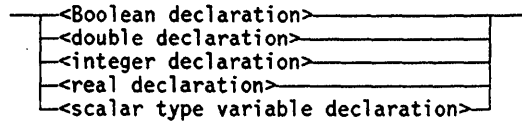
Example

```

SEGMENT SCANSEG,
      PARSESEG,
      EMITTERSEG;
    
```

SIMPLE VARIABLE Declaration

<simple variable declaration>



<scalar type variable declaration>

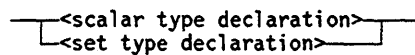
A <structure type variable declaration> used to declare subtype variables or enumerated type variables.

Explanation

For information on the semantics of a <simple variable declaration>, refer to its definition in the *A Series ALGOL Programming Reference Manual, Volume 1: Basic Implementation* and to the definition of the <scalar type declaration> in this section.

STRUCTURE TYPE Declarations

<structure type declaration>

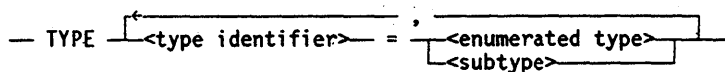


Explanation

A <structure type declaration> defines a user-specified data type by associating a type definition with an identifier.

SCALAR TYPE Declarations

<scalar type declaration>



<type identifier>



Explanation

A SCALAR TYPE declaration defines a user-specified data type by associating a type definition with a <type identifier>. The type identifier can then be used in contexts

Declarations

in which a data type must be specified, such as in declarations of variables, arrays, procedures, constants, and formal parameters. The type identifier can also be used in such contexts as declarations of array dimensions, other SCALAR TYPE declarations, and STRUCTURE TYPE VARIABLE declarations.

The SCALAR TYPE declaration allows the creation of entirely new types and allows types to be defined in terms of the predefined types (REAL, INTEGER, and BOOLEAN) and types from previous SCALAR TYPE declarations.

Two kinds of type definition are possible: enumerated types and subtypes. Each is discussed in this section under its own heading.

Enumerated Types

<enumerated type>

[ORDERED] (—<enumerated literal list>—) —

<enumerated literal list>

—<enumerated literal>— , —<enumerated literal>—
[= —<nonnegative constant integer expression>—]

<enumerated literal>

—<identifier>—

<nonnegative constant integer expression>

A <constant integer expression> that evaluates to an integer greater than or equal to 0 (zero).

Explanation

An <enumerated type> declaration defines a new, bounded, discrete data type. An enumerated type is unordered by default, or it can be declared as an ORDERED enumerated type.

The purpose of an enumerated type is to enable you to declare a data type that has a range of valid values that can be enumerated specifically. For example, valid values of an enumerated type called DEVICES might include DISKPACK, FLOPPY, GCRTAPE, and TERMINAL.

Each <enumerated literal> in an enumerated type declaration has a unique nonnegative integer value associated with it. You can assign these associated values explicitly in the enumerated type declaration. Those enumerated literals to which you do not assign values are given associated values by the compiler. The associated values are considered to be constants of the enumerated type being declared, and the enumerated literals denote these constant values.

The associated values assigned to the literals by the compiler are assigned to all the literals from left to right in ascending order; that is, the associated value assigned by the

compiler is the associated value of the preceding literal incremented by one. When you do not assign a value to the first literal in the list, the compiler assigns it a value of 0 (zero).

If you assign an associated value to a literal, the value must be exactly one greater than the associated value of the preceding literal. If the literal is the first in the list, the value must be 0 (zero) or greater.

The values of ordered enumerated types can be compared using the following relational operators:

- <
- LSS
- <=
- LEQ
- =
- EQL
- ^=
- NEQ
- >
- GTR
- >=
- GEQ

If X and Y are two literals of an ordered enumerated type, then X is less than Y if and only if the value associated with X is arithmetically less than the value associated with Y. The other relations are similarly defined.

Unordered enumerated types can be compared for equality and inequality only by the following operators:

- =
- EQL
- ^=
- NEQ

No other relational operators are defined for unordered enumerated types.

Example

```
BEGIN
  TYPE COIN = ORDERED (NICKEL, DIME, QUARTER),
  BILL = ORDERED (TEN, TWENTY, FIFTY, HUNDRED),
  DAY = ORDERED (MON=1, TUE, WED, THU, FRI, SAT, SUN),
  FLOWER = (JASMINE, LILY, ROSE);          % UNORDERED
```

Declarations

```
COIN BIT, CHANGE;
BILL WAD;
FLOWER BOUQUET;
BOOLEAN RELATION;

INTEGER PROCEDURE HOURS_WORKED (WORKINGDAY);
    VALUE WORKINGDAY;
    DAY WORKINGDAY;
BEGIN
    IF WORKINGDAY <= FRI THEN
        HOURS_WORKED := 8
    ELSE
        HOURS_WORKED := 0;
END HOURS_WORKED;

WAD := FIFTY;
BIT := QUARTER;
CHANGE := DIME;
BOUQUET := ROSE;

WAD := NICKEL;      % SYNTAX ERROR

RELATION := TEN < TWENTY AND % TRUE
           NICKEL < DIME AND % TRUE
           DIME > QUARTER AND % FALSE
           LILY > ROSE;      % SYNTAX ERROR

RELATION := BOUQUET = JASMINE; % FALSE

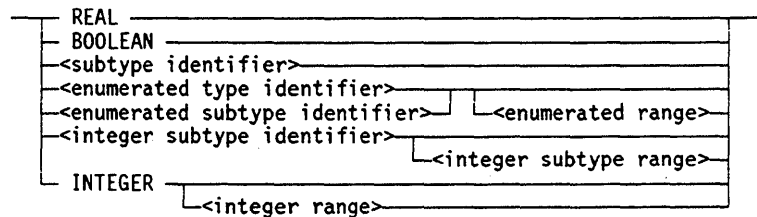
RELATION := CHANGE < BIT;      % TRUE
END.
```

Subtypes

<subtype>

— SUBTYPE —<base type identifier>—|

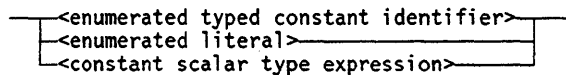
<base type identifier>



<enumerated range>

—<enumerated endpoint> .. —<enumerated endpoint>—|

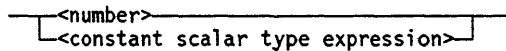
<enumerated endpoint>



<integer subtype range>

—<integer subtype endpoint> .. —<integer subtype endpoint>—|

<integer subtype endpoint>



<integer range>

—<integer endpoint> .. —<integer endpoint>—|

<integer endpoint>

—<number>—|

Explanation

A < subtype > declaration defines a data type that takes its characteristics and valid operations from the < base type identifier >. In certain cases, the valid values for the subtype can be less than the valid values for the base type identifier. However, the range for the subtype being created cannot be larger than the valid range for the base type identifier.

A subtype is not a new type; its type is the same as that of the base type identifier. When the base type identifier is a discrete, ordered type, the range of valid values can be limited by the inclusion of a range. The valid operators for a subtype are those allowed for the specified base type.

The primary purpose of subtypes is to enable you to separate different types of data that might have similar fundamental characteristics. For example, a height and a weight are both numbers and they share a set of valid operators (addition, subtraction, and so on).

Declarations

However, height might be measured in inches, and weight might be measured in pounds. In this case, HEIGHT and WEIGHT could be declared as subtypes descended from INTEGER. In this way, the programmer informs the compiler that the two types are conceptually different, and the compiler must ensure that the types are not accidentally mixed.

A range can be specified if the base type identifier is INTEGER, an ordered enumerated type, or a subtype of either of these two. The range endpoint specifications must be of the base type or of a direct ancestor of the base type. If a constant identifier is used as an <integer endpoint>, the constant must have been declared as an INTEGER CONSTANT.

Example

```
BEGIN
TYPE MONEY = ORDERED (PENNY, NICKEL, DIME, QUARTER,
                      DOLLAR, FIVEDOLLAR, TENDOLLAR),
COINS = SUBTYPE MONEY PENNY..QUARTER,
BILLS = SUBTYPE MONEY DOLLAR..TENDOLLAR;

TYPE TOTAL_MONEY = SUBTYPE INTEGER,
POCKET_CHANGE = SUBTYPE TOTAL_MONEY 2..15;

COINS YOUR_LARGEST, MY_LARGEST;
TOTAL_MONEY YOUR_TOTAL, MY_TOTAL;
POCKET_CHANGE YOUR_CHANGE, MY_CHANGE, TOTAL_CHANGE;

YOUR_LARGEST := QUARTER;
MY_LARGEST := DIME;
YOUR_CHANGE := 15;
MY_CHANGE := 7;
TOTAL_CHANGE := MY_CHANGE + YOUR_CHANGE;

END.
```

Descendant Types

An enumerated type is a completely new type, but a subtype descends from its base type. A base type can, in turn, descend from yet another type. This chain of ancestry can be followed all the way up to the root type, which is not descended from another type. The root type is either a predefined type (INTEGER, REAL, and BOOLEAN) or an enumerated type. A subtype is said to be a descendant type of its root type.

Examples

In the following examples, all of the types declared are descendant types of INTEGER.

```
TYPE FIRST = SUBTYPE INTEGER,  
    SECOND = SUBTYPE FIRST,  
    THIRD  = SUBTYPE SECOND;  
  
TYPE A     = SUBTYPE THIRD;
```

Assignment Compatibility

Two types are said to be assignment compatible if a value of one type can be assigned directly to a variable of the other type. Among the scalar types, the following are the only cases of assignment compatibility:

- A type is always assignment compatible with itself.
- A subtype or typed constant is assignment compatible with any of its ancestor types.
- A constant or enumerated literal is assignment compatible with its base type or with any descendant of its base type.

If an assignment is to be made between two types that are not assignment compatible, explicit type conversion using the mapping function is necessary. For more information, see “MAPPING Function” in Section 6, “Expressions and Functions.”

Range Checking

By default, range checking is performed whenever there is a chance of assigning to a variable a value that is not defined for the type of that variable.

No run-time range checking is performed on the simple assignment of types that are assignment compatible (that is, a direct assignment not involving any kind of expression). Therefore, it is possible for the <structure type identifier> to assume a value that is outside its range without any interrupt occurring if the source of the assignment was never initialized.

A range error is reported whenever a range check fails. Range errors can occur during expression evaluation or during an assignment operation.

The manner in which range errors are reported depends on the target computer family for which the code is compiled. (See the discussion of the <target option> block directive in Section 8, “Compiler Controls.”) If a range error occurs within code compiled for LEVEL1 machines, a false assertion is reported. If a range error occurs within code compiled for LEVEL0 machines, a divide-by-zero fault is reported.

The error message indicates only that the value is not in range; it does not indicate whether the value is too large or too small.

Range checking can be disabled through the use of the block directive NORANGECHECK. Within a block for which range checking is disabled, range checking can be enabled through the use of the block directive RANGECHECK.

SET TYPE Declaration

<set type declaration>

— TYPE — $\overbrace{\text{<set identifier>}}^{\leftarrow}$ = — SET — $\underbrace{\text{<set base type>}}_{\text{OF}}$ —

<set identifier>

—<identifier>—

<set base type>

— $\overbrace{\text{<type identifier>}}^{\leftarrow}$ —
 — (— $\overbrace{\text{<enumerated literal list>}}^{\leftarrow}$ —) —
 — INTEGER —

Explanation

A <set type declaration> defines a structured type for which the range of values is all possible subsets of the specified <set base type>. In mathematical terms, a <set identifier> defines the powerset of its set base type. A variable of set identifier type can contain any subset of the set, including the null set and the entire set.

The ordinal numbers associated with the set base type must be within the range 0 through 1000.

When the set base type is a type identifier, then the type of the type identifier must be an enumerated type, a subtype descended from an enumerated type, or a subtype descended from INTEGER.

When the set base type is an <enumerated literal list>, the enumerated literals must not have been declared previously. The literals are treated as unordered enumerated literals. If a <nonnegative constant integer expression> is present for the first <enumerated literal>, the expression must be a nonnegative number less than or equal to 1000. If the nonnegative constant integer expression is present for literals other than the first one in the list, the expression must be equal to one more than the value associated with the previous literal.

When the set base type is INTEGER, the maximum possible range (0 through 1000) is assumed. An assignment to a set is allowed if the root type of the right-hand side and the left-hand side are the same. For example, INTEGER values or values with a type descended from INTEGER can be assigned to a set variable that has a set base type of INTEGER.

If the maximum value in the set is 47 or less, a single word is allocated for set variables of that set type. If the maximum value in the set is greater than 47, an array is allocated for set variables of that set type. As a result, set variable of a set type with a maximum value greater than 47 should not be used in places that cannot handle a p-bit.

Example

```

TYPE COLOR      = {RED, BLUE, GREEN, YELLOW};
TYPE COLORSET   = SET OF COLOR;
TYPE MONEY      = ORDERED (PENNY, NICKEL, DIME, QUARTER,
                          DOLLAR, FIVEDOLLAR, TENDOLLAR);
TYPE COINS      = SUBTYPE MONEY PENNY..QUARTER,
BILLS          = SUBTYPE MONEY DOLLAR..TENDOLLAR;
TYPE COINSET    = SET OF COINS,
CARDSET        = SET OF (CLUB = 2, DIAMOND, HEART = 4,
                        SPADE),
WEIGHTS        = SET OF INTEGER;

```

STRUCTURE TYPE VARIABLE Declaration

<structure type variable declaration>

```

—<structure type identifier>—<identifier>—|

```

<structure type identifier>

```

—<set identifier>—|
—<type identifier>—|

```

Explanation

The <structure type variable declaration> defines variables of data types that have been declared previously in <structure type declaration> s.

The initial value of a SET structure type variable is the null set. The initial value of a scalar type variable is undefined.

The identifier declared in a structure type variable declaration is referred to in this document as a <set variable identifier>, <enumerated variable identifier>, or <subtype variable identifier>, depending on the type of the declared variable.

Example

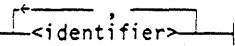
```

TYPE MONEY = ORDERED (PENNY, NICKEL, DIME, QUARTER,
                     DOLLAR, FIVEDOLLAR, TENDOLLAR);
TYPE COINS = SUBTYPE MONEY PENNY..QUARTER;
TYPE COINSET = SET OF COINS;
COINSET POCKETCHANGE; % POCKETCHANGE is a <set variable identifier>
MONEY PAY; % PAY is an <enumerated variable identifier>
COINS SPARECHANGE; % SPARECHANGE is a <subtype variable identifier>

```

SUPPLY Declaration

<supply declaration>

— SUPPLY —  <identifier>

Explanation

The SUPPLY declaration allows a module nested within another module to supply the body of a procedure declared in the outer module.

Each identifier must be that of a procedure that has been declared FORWARD, and that could have occurred at the same syntactic location as the module containing the SUPPLY declaration. The specified procedures must occur within the module containing the SUPPLY declaration.

Example

```
MODULE PHYSICALIO;
  INTERFACE PHYSICALIO_INTERFACE (IOFINISH);
  MODULEEXPORT PHYSICALIO_INTERFACE;
  PROCEDURE IOFINISH (...); ... FORWARD;
BEGIN PHYSICALIO;
  MODULE CPIO;
    SUPPLY IOFINISH;
  BEGIN CPIO;
    PROCEDURE IOFINISH (...); ...
    BEGIN
      .
      .
      .
    END IOFINISH;
  END CPIO;
END PHYSICALIO;
```

VALUE ARRAY Declaration

In addition to the VALUE ARRAY declarations permitted in ALGOL, NEWP allows the VALUE ARRAY declaration to be used to declare value arrays of a scalar type. The elements of a scalar type value array are specified by < constant scalar type expression > s of the same type as the value array.

String literals that are less than 48 bits long are left-justified and padded with zeros on the right to make them 48 bits long when they are used in the < constant list > of a < value array declaration > . Note that this practice differs from that of the ALGOL compiler, which right-justifies string literals less than 48 bits long when used in value arrays. If the type of the value array is DOUBLE, the low-order word is all zeros.

For information related to this type of declaration, see "Scalar Type Expressions" in Section 6, "Expressions and Functions."

Section 5

Statements

NEWP provides several kinds of statements that are not supported by ALGOL. These NEWP-specific forms of statement are discussed in this section and in "Statements (UNSAFE)" in Section 9, "UNSAFE Mode."

The following table briefly describes some of the differences in statements that are supported in both NEWP and ALGOL. More extensive differences in statements are described in the rest of this section. For information on ALGOL statements that are not supported in NEWP, refer to Appendix B, "ALGOL Features Not Implemented in NEWP."

Statement	NEWP Information
ACCEPT	Only the <pointer expression> form of the <accept statement> is supported.
ASSIGNMENT	For information on differences in all forms of the <assignment statement>, refer to "ASSIGNMENT Statement" later in this section.
CASE	NEWP does not allow a <string literal> to be used as an <arithmetic primary>. Numeric constants should be used instead. In addition, CASE labels in ALGOL are limited to the range 0 through 1023. NEWP does not have this restriction. However, in NEWP, the difference between the highest and lowest valued labels cannot exceed 1023. For example, if the smallest label is 100, the largest label can be no more than 1123.
CLOSE	Like ALGOL, the NEWP compiler emits code to call the MCP for non-MCP programs that do not use the CLOSE statement as a function. In the MCP, no additional code is emitted; but at compile time a message is issued, which warns that the result is not handled. In addition, NEWP does not support the REWIND clause of the <close option> for the CLOSE statement. Instead, NEWP provides a RETAIN clause for the CLOSE option, as shown in the following: CLOSE (<file designator>, RETAIN);
FOR	In the <iteration part>, NEWP allows the STEP phrase to be omitted, even if an UNTIL clause is present. If you omit the STEP <arithmetic expression> phrase, a default of STEP 1 is assumed.
FREEZE	This statement has more options than are available in ALGOL. Refer to "FREEZE Statement" later in this section.
GO TO	NEWP does not allow branching into FOR loops or THRU loops. Note that branching within FOR and THRU loops is allowed, as long as you declare the label within the loop.
I/O	NEWP does not allow the <rewind statement> and <space statement> forms of the I/O statement. For more information, refer to Appendix B, "ALGOL Features Not Implemented in NEWP."

Statements

Statement	NEWP Information
MLSaccept	NEWP does not support a <string variable> or <subscripted string variable> as part of this statement. Use the <pointer expression> form of the statement.
ON	NEWP provides a declaration instead of a statement. Refer to "ON Declaration" in Section 4, "Declarations."
OPEN	Like ALGOL, the NEWP compiler emits code to call the MCP for non-MCP programs that do not use the OPEN statement as a function. In the MCP, no additional code is emitted; but at compile time a message is issued, which warns that the result is not handled.
PROCEDURE INVOCATION	NEWP does not support the following types of <actual parameters>: <ul style="list-style-type: none">• <string array designator>• <direct switch file identifier>• <switch file identifier>• <format designator>• <switch label identifier>• <list designator>• <switch list identifier>• <picture identifier>
PROCESS	In NEWP, the <process statement> is unsafe. Refer to "PROCESS Statement" in Section 9, "UNSAFE Mode," for information on the differences between NEWP and ALGOL.
READ	NEWP does not support several features that ALGOL allows. Note that all limitations mentioned here also apply to the NEWP WRITE statement. NEWP does not allow the <core-to-core part> of the file part. In addition, the <format and list part> cannot be a format designator, editing specification, asterisk (*), or <free field part>. The <format and list part> cannot include a string variable. The list cannot include an <iteration clause> that includes a WHILE loop embedded in a FOR loop. The update replacement construct (:=*) cannot be used on the variable of a FOR iteration clause.
REPLACE	There are a number of differences between ALGOL and NEWP <replace statement> syntax and semantics. Refer to "REPLACE Statement" later in this section and to "REPLACE Statement" in Section 9, "UNSAFE Mode."
REPLACE POINTER-VALUED ATTRIBUTE	Only the <simple source> form of this statement is supported in NEWP. If a pointer-valued attribute is to be replaced by the value of another pointer-valued attribute, replace the value into a temporary array, and then replace the destination attribute with the contents of the array.
RESET	The RESET statement has been renamed the RESETEVENT statement in NEWP.

Statement	NEWP Information
RESIZE	NEWP does not support a <string array designator> as a <special array resize parameter>. However, NEWP does allow an interlock array to be resized, provided you do not specify DISCARD.
SEEK	NEWP allows the keyword SPACE to appear before the <arithmetic expression>. This form of the SEEK statement is used to perform the function of the ALGOL <space statement>.
SET	The SET statement has been renamed the SETEVENT statement in NEWP to avoid conflicts with the data type SET.
SWAP	Refer to "SWAP Statement" later in this section for information on the differences between NEWP and ALGOL.
WAIT	NEWP does not require parentheses around the <time> specification as ALGOL does. NEWP does not support the form of the WAIT statement that includes no parameters (wait for interrupt). Refer to "WAIT Statement" in Section 9, "UNSAFE Mode," for information about unsafe options for the <wait statement>.
WAITANDRESET	NEWP does not require parentheses around the <time> specification as ALGOL does. Refer to "WAIT Statement" in Section 9, "UNSAFE Mode," for information about unsafe options for the <waitandreset statement>.
WRITE	Refer to the discussion of the <read statement> earlier in this table.

ASSIGNMENT Statement

The ASSIGNMENT statement causes the item on the right-hand side of the assignment operator (:=) to be evaluated and the resulting value to be assigned to the item on the left-hand side of the assignment operator.

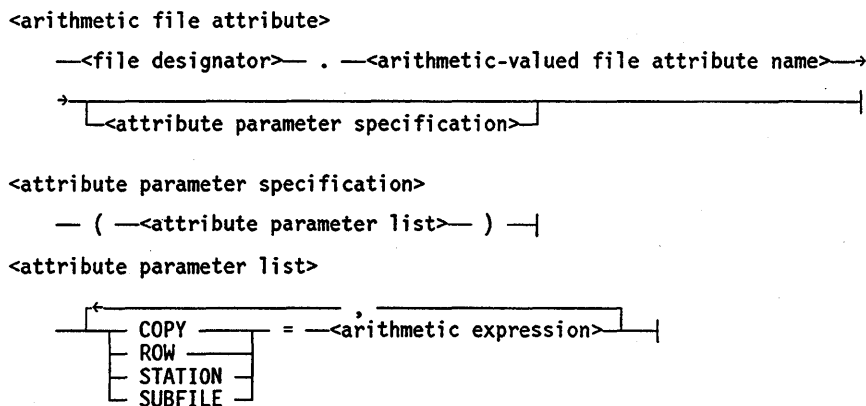
Differences between ALGOL and NEWP ASSIGNMENT Statements

The following are some differences between ALGOL and NEWP in regard to various kinds of ASSIGNMENT statements:

- NEWP does not support the <complex assignment> and <string assignment> forms of the ASSIGNMENT statement. However, in NEWP, every <string-valued library attribute> is treated as a <pointer-valued attribute>. For more information, refer to "REPLACE Statement" later in this section.
- NEWP allows a <partial word part> on the left-hand side of an <arithmetic update assignment>, but ALGOL does not.
- NEWP requires that parameters for attributes follow the attribute name (ALGOL requires that the parameters be placed before the attribute name).

The following is the NEWP syntax for the <arithmetic file attribute>:

Statements



The differences between the ALGOL and the NEWP syntax are shown in the following example:

```
I := DISKFILE(COPYNUM).ERRORTYPE;           % ALGOL syntax
I := DISKFILE.ERRORTYPE(COPY=COPYNUM);      % NEWP syntax
```

Data comm file attributes must include the phrase *STATION* = as part of the <attribute parameter specification>:

```
I := TERMFILE(1).WIDTH;                       % ALGOL syntax
I := TERMFILE.WIDTH(STATION=1);               % NEWP syntax
```

In addition, keywords are needed to identify some types of attribute parameters. For example, the ALGOL statement *REPLACE MYFILE(1).TITLE BY PTR* would be *REPLACE MYFILE.TITLE(SUBFILE=1) BY PTR* in NEWP.

- NEWP does not allow the <arithmetic update assignment> form (:=*) of the ASSIGNMENT statement to be used with file or task variables.
- NEWP does not fully support the use of an <arithmetic type transfer variable> on the left-hand side of an <arithmetic assignment>.

Array Reference Assignment

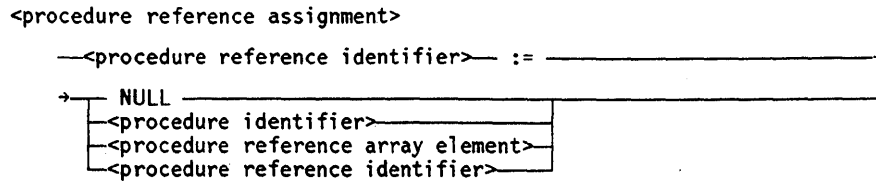
Array reference assignment in NEWP is similar to that in ALGOL. The differences are as follows.

In the ARRAY REFERENCE ASSIGNMENT statement, the corresponding dimension of the <array reference variable> and the <array designator> must have the same type.

In addition, the element size of the array designator must match that of the array reference variable. In the context of ARRAY REFERENCE ASSIGNMENT statements, subtypes and enumerated types are considered to have an element size of a single word.

Procedure Reference Assignment

A procedure reference assignment associates a procedure reference with a procedure reference identifier. The identifier can then be used to refer to the procedure.



Explanation

The procedure reference identifier on the left-hand side of the assignment operator (:=) and the procedure, the procedure reference array element, or the procedure reference identifier on the right-hand side must be of the same type and have the same parameter descriptions.

The procedure reference identifier on the left-hand side of the assignment operator cannot be global to the procedure, the procedure reference array element, or the procedure reference identifier on the right-hand side.

If the procedure reference array element or procedure reference identifier on the right-hand side of the assignment operator is uninitialized, then a later attempt to use the statement on the left-hand side will result in an error.

If NULL is specified and there is an environment called NULL, then a reference to the procedure called NULL is assigned. If NULL is specified and there is no environment called NULL, then a NULL value is assigned to the procedure reference array element. When a NULL reference is assigned, the previous contents are overwritten with a tag 0 (zero). If the procedure reference array element is invoked while it is NULL, a program interrupt occurs.

Example

In the following example, P is a REAL procedure and RA is a REAL procedure identifier. Neither P nor RA have parameters. The program sample assigns a reference to procedure P to the procedure reference identifier RA.

```

BEGIN
  REAL PROCEDURE P;
  BEGIN
    REAL A;
    A := T * T;
    P := A;
  END;

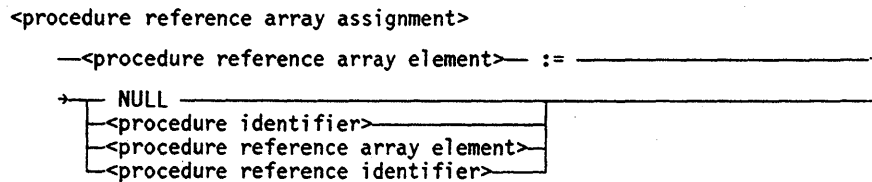
  REAL PROCEDURE REFERENCE RA;
  NULL;

  RA := P;
END.

```

Procedure Reference Array Assignment

A procedure reference array assignment associates a procedure reference with a procedure reference array element. The element can then be used to refer to the procedure.



Explanation

The procedure reference array element on the left-hand side of the assignment operator (:=) and the procedure, the procedure reference array element, or the procedure reference identifier on the right-hand side must be of the same type and have the same parameter descriptions.

The procedure reference array on the left-hand side of the assignment operator cannot be global to the procedure, the procedure reference array element, or the procedure reference identifier on the right-hand side. If the procedure reference array element on the left-hand side of the assignment operator is a formal parameter, a procedure reference array element on the right-hand side can only be another element of the same procedure reference array that appears on the left-hand side.

A procedure reference array that is declared to be part of a library cannot appear on the left-hand side of a procedure reference array assignment. An attempt to assign into such a procedure reference array results in an error at compile time or at run time.

If the procedure reference array element on the right-hand side of the assignment operator is uninitialized, then a later attempt to use the statement on the left-hand side will result in an error.

If NULL is specified and there is an environment called NULL, then a reference to the procedure called NULL is assigned. If NULL is specified and there is no environment called NULL, then a NULL value is assigned to the procedure reference array element. When a NULL reference is assigned, the previous contents are overwritten with a tag 0 (zero). If the procedure reference array element is invoked while it is NULL, a program interrupt occurs.

Example

In the following example, P and Q are REAL procedures and RA is a REAL procedure reference array. Neither P, nor Q, nor RA have parameters. The program sample assigns references to elements 1 through 4 of the procedure reference array RA.

Example

```

TYPE COLORSET = SET OF (RED, BLUE, GREEN, YELLOW);
COLORSET C1,C2;

C1 := [RED, YELLOW];      % C1 is assigned the set whose
                          % members consist of the elements
                          % "RED" and "YELLOW".
C2 := * + [GREEN];        % The element "GREEN" is added as a
                          % member to the previous value of C2.
C1 := [];                 % C1 is assigned the NULL set.
    
```

CASE Statement

In addition to the ALGOL syntax for the CASE statement, NEWP allows extensions to <case head> and to <number list>. For more information on the CASE statement, see the *A Series ALGOL Programming Reference Manual, Volume 1: Basic Implementation*.

```

<case head>
  — CASE —<scalar type expression>— OF —|
<number list>
  ┌──────────────────────────────────────────────────────────┐
  │ <constant scalar type expression> : ───────────────────┘
  └── ELSE ───────────────────────────────────────────────────┘
    
```

Explanation

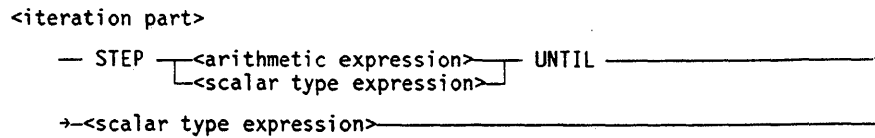
The <constant scalar type expression> s in the number list must be assignment-compatible with the <scalar type expression> s in the case head.

FOR Statement

In addition to the ALGOL syntax for the FOR statement, NEWP allows extensions to the <for statement>, the <initial part>, and the <iteration part>. For more information on the FOR statement, see the *A Series ALGOL Programming Reference Manual, Volume 1: Basic Implementation*.

```

<for statement>
  — FOR —<scalar type variable>— := ┌──<for list element>──┐ DO ─→
  └──────────────────────────────────────────────────────────┘
  →<statement>──────────────────────────────────────────────────┘
<initial part>
  —<scalar type expression>—|
    
```



Explanation

When a < scalar type variable > is used in the FOR statement, the < scalar type expression > s specified in the < for list element > part must be assignment-compatible with the scalar type variable. An exception to this is that an < arithmetic expression > that is not assignment-compatible can be used following the STEP clause.

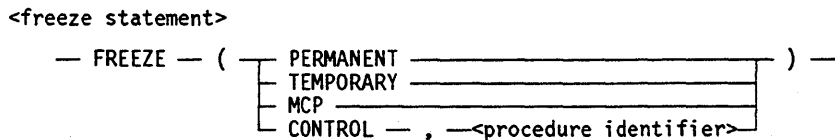
Example

```

TYPE COLOR = ORDERED (RED, BLUE, GREEN, YELLOW);
COLOR COLORVAR;

FOR COLORVAR := BLUE STEP 1 UNTIL YELLOW DO
    
```

FREEZE Statement



Explanation

In addition to the values supported by ALGOL, the MCP option is allowed as a value for the parameter to the FREEZE statement. If you specify the MCP option, the compiler control option MCP must be set.

Unlike the other forms of the FREEZE statement, FREEZE(MCP) is allowed in a block with no library EXPORT declarations. FREEZE(MCP) exports all entry points appearing in library EXPORT declaration that occur at D[0] up to the FREEZE statement in the symbolic. No subsequent library EXPORT declarations are allowed.

The options TEMPORARY, PERMANENT, and CONTROL work exactly as they do in ALGOL.

The < procedure identifier > required in the CONTROL form of the FREEZE statement must be a previously declared untyped procedure with no parameters.

If a D[0] library EXPORT declaration is changed during a Separately Compiled Procedure (SEPCOMP), all procedures containing FREEZE(MCP) statements must be recompiled. For more information on SEPCOMP, refer to Section 7, "Compiling NEWP Programs."

PROCEDURE REFERENCE Statement

A PROCEDURE REFERENCE statement causes the procedure referenced by the specified procedure reference identifier to be executed as a procedure invocation.

<procedure reference statement>

—<procedure reference identifier> ————|
 └<actual parameter part>┘

Explanation

If the procedure reference identifier has not been assigned a procedure reference in a procedure reference assignment, the program is terminated with the message *Invalid Stack Argument*.

When a typed procedure reference identifier is used in a PROCEDURE REFERENCE statement, the value returned by the procedure reference identifier is discarded.

The actual parameter part of a PROCEDURE REFERENCE statement must have the same number of entries as the formal parameter list in the declaration of the procedure reference. The formal and actual parameters are compared in the manner in which the formal and actual parameters are compared in a procedure invocation statement.

Invoking a procedure through a procedure reference identifier in a PROCEDURE REFERENCE statement is equivalent to invoking the procedure directly in a procedure invocation statement.

Example

The following example assigns a reference to procedure SWAPPER into the procedure reference identifier PROCREF and then invokes SWAPPER through PROCREF.

```
BEGIN
.
.
.
REAL
  SORT1,
  SORT2;

PROCEDURE REFERENCE PROCREF (A,B);
                      REAL  A,B;
                      NULL;

PROCEDURE SWAPPER(X,Y);
  REAL  X,Y;
  BEGIN
  X ::= Y;
  END;

PROCREF := SWAPPER;
```

```

READ(MYFILE,*,SORT1,SORT2);
IF SORT2 > SORT1 THEN
  PROCREF (SORT1,SORT2);
END.

```

REPLACE Statement

NEWP does not support the SDIGITS (signed digits) clause of the <replace statement>. Instead, you can simply test the value that is being replaced and use the REPLACE statement to put in the appropriate sign. Then use the DIGITS clause to add the absolute value of the item.

When an <arithmetic expression> appears as the <source part> in a REPLACE statement, and no FOR clause is included, exactly 48 or 96 bits of data (depending on whether the <arithmetic expression> is single-precision or double-precision) are transferred as characters. The size of the characters is determined by the <destination part>. This differs from the ALGOL implementation in which either 6 or 8 characters are transferred, depending on the value of the BCL compiler control option.

NEWP does not support the <intrinsic translate table>s that refer to BCL. If such translate tables are needed, they can be declared directly.

Note: The BCL data type is not supported on all A Series and B 7900 systems. The appearance of a BCL construct that can cause the creation of a BCL descriptor, such as a BCL string literal more than 96 bits long, causes the program to receive a compile-time warning message.

NEWP provides an extension to the <transfer part> of the REPLACE statement. The syntax for the extension to the transfer part is as follows:

```

<transfer part>
  — FOR —<length>— WITH —<edit micros>—|

<length>
  —<arithmetic expression>—|

<edit micros>
  — INSERT — ( —<insert character>— ) —|

<insert character>
  —<EBCDIC constant>—|

```

The transfer part of the REPLACE statement has been extended to include the INSERT option. INSERT (<insert character>) translates and transfers <length> characters from the source string to successive character positions in the destination string. Each leading zero source character is replaced with an insert character; the remaining characters are transferred unchanged.

Statements

The source and destination pointer expressions must denote 8-bit characters. Furthermore, the resulting destination string is undefined if the source string contains characters other than the EBCDIC digits 0 through 9.

In NEWP, every <string-valued library attribute> (TITLE, FUNCTIONNAME, INTNAME, and LIBPARAMETER) is treated as a <pointer-valued attribute>. As a result, the REPLACE statement is used to assign values to these attributes, rather than the <string assignment> form of the ASSIGNMENT statement.

Example

```
REPLACE LIBRARY1.FUNCTIONNAME BY POINTERVAR;  % NEWP setting of
                                                % FUNCTIONNAME attribute

LIBRARY1.FUNCTIONNAME := STRINGVAR;          % ALGOL setting of
                                                % FUNCTIONNAME attribute
```

It is required that the value contained in the array pointed to by the pointer variable must contain a termination character, or period (.). Therefore, the termination character cannot be part of the array value for LIBPARAMETER except as the terminating character.

SELECT Statement

```
<select statement>
  — SELECT — ( — <alternative identifier> — ) —|

<alternative identifier>
  —<identifier>—|
```

The SELECT statement initializes an alternative module, which includes the items you declare inside the alternative specified by the <alternative identifier>. For more information, see “MODULE Declaration (Old)” in Section 4, “Declarations.”

A SELECT statement can occur only in an initialization procedure. Only one SELECT statement can be executed in an initialization procedure; an INVALID OPERATOR fault occurs if a second SELECT statement is executed. You must declare in the same module the initialization procedure and the alternative identifier it selects.

SWAP Statement

The SWAP statement is implemented with the same syntax and semantics as in ALGOL, with the following exceptions:

- The <complex variable> s are not supported by NEWP and therefore cannot appear in the SWAP statement.
- The <enumerated variable identifier> s and <subtype variable identifier> s can be swapped as long as the variables on both sides of the swap operator (:=) are of the same type.

Section 6

Expressions and Functions

NEWP provides several kinds of expressions and functions that are not supported by ALGOL. These NEWP-specific expressions and functions are discussed in this section and in "Expressions and Functions (UNSAFE)" in Section 9, "UNSAFE Mode."

The following table describes some of the differences in expressions and functions that are supported in both NEWP and ALGOL. For information on ALGOL functions and expressions that are not supported in NEWP, refer to Appendix B, "ALGOL Features Not Implemented in NEWP."

Expression or Function	NEWP Information
ARITHMETIC FUNCTION DESIGNATOR	NEWP provides the <packdecimal function>, which is not supported by ALGOL. For more information, refer to "PACKDECIMAL Function" later in this section.
ARITHMETIC OPERATOR	The keyword TIMES as a synonym for an asterisk (*), or multiplication sign, is not supported in NEWP.
ARITHMETIC PRIMARY	The NEWP definition of <arithmetic primary> is different from that in ALGOL. For the NEWP definition, refer to "Arithmetic Expressions" later in this section.
BOOLEAN FUNCTION DESIGNATOR	NEWP provides the Boolean <decimalconvert function>, which is not supported by ALGOL. For more information, refer to "DECIMALCONVERT Function" later in this section.
BOOLEAN OPERATOR	The NEWP definition of <Boolean operator> is different from that in ALGOL. For more information, refer to "Boolean Expressions" later in this section.
BOOLEAN PRIMARY	The <complex relation> form of <Boolean primary> is not supported by NEWP.
CASE EXPRESSION	The <complex case expression> and <designational case expression> forms of the <case expression> are not supported in NEWP.
CONDITIONAL EXPRESSION	The <conditional complex expression> and <conditional designational expression> forms of <conditional expression> are not supported in NEWP.
DESIGNATIONAL EXPRESSION	The only valid <designational expression> in NEWP is a <label identifier>.
DINTEGER FUNCTION	NEWP does not support the form of this function that accepts a <pointer expression>. The DOUBLE function can be used instead.
EXPRESSION	The <complex expression> form of <expression> is not supported in NEWP.

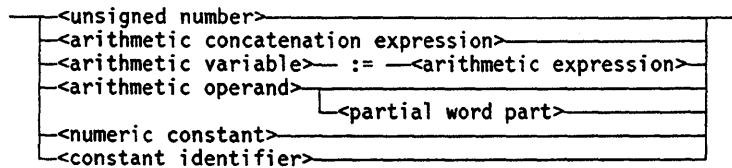
Expressions and Functions

Expression or Function	NEWP Information
FUNCTION EXPRESSION	The <complex function designator> and <string function designator> forms of the <function expression> are not supported in NEWP. However, NEWP supports <interlock function>s and <scalar type function>s, which are not supported by ALGOL. For more information, refer to "INTERLOCK Functions" and "SCALAR TYPE Functions" later in this section.
INTEGER FUNCTION	In addition to the parameters accepted by the ALGOL <integer function>, NEWP accepts the following as parameters: <ul style="list-style-type: none">• <enumerated literal>• An expression of an enumerated type• An expression of a subtype that has an enumerated type or a root type of INTEGER or REAL
LABEL DESIGNATOR	The only valid <label designator> in NEWP is a <label identifier>.
LISTLOOKUP FUNCTION	This function is unsafe in NEWP. Refer to Section 9, "UNSAFE Mode," for more information.
NORMALIZE FUNCTION	The syntax of the <normalize function> is identical to the ALGOL syntax, but the semantics are slightly different. NORMALIZE is an arithmetic-valued procedure that returns the result of the normalize (NORM) machine operator. If the parameter is single-precision, the result returned is of type REAL. If the parameter is double-precision, the result returned is of type DOUBLE.
POINTER EXPRESSION	Refer to "POINTER Expressions" later in this section.
REAL FUNCTION	The <complex expression> is not supported as a parameter to the <real function> in NEWP. In addition to the other parameters supported by ALGOL, the NEWP REAL function accepts parameters listed earlier for <integer function> and a <string expression> as a parameter (see "String Expressions"). The function returns, as a REAL value, the bit image of the string expression. The bit image is right-justified with binary-zero fill. All bits in each character are used. The string expression cannot exceed 48 bits in length. Following are examples of valid REAL functions: R:=REAL (48"04" & 8"NAME" & 48"00"); % NEWP R:=REAL (48"04D5C1D4C500"); % ALGOL Equivalent
SIZE FUNCTION	The <pointer identifier> is not supported as a parameter to the <size function> in NEWP.
STRING EXPRESSION	The definition of <string expression> is different in NEWP than it is in ALGOL. Refer to "String Expressions" later in this section.

Arithmetic Expressions

Arithmetic expressions in NEWP function just as in ALGOL, except for the specifics outlined as follows:

<arithmetic primary>



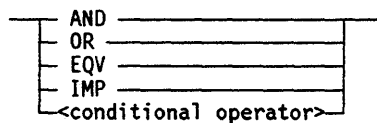
Unlike ALGOL, NEWP does not allow a <string literal> to be used as an <arithmetic primary>. However, you can achieve the same functionality by using a <numeric constant>. In addition, NEWP allows <constant identifier>s to be used as constant arithmetic primaries.

Note: In NEWP, an <arithmetic operand> can be <subtype variable identifier> descended from INTEGER or REAL in addition to any of the elements allowed in ALGOL.

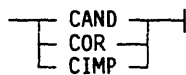
For more information on <arithmetic primary>, see the discussion of <constant identifier>, <numeric constant>, and <structure type variable identifier> in Section 4, "Declarations."

Boolean Expressions

<Boolean operator>



<conditional operator>

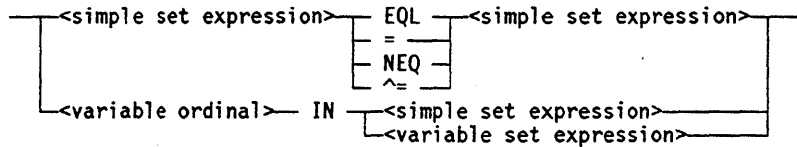


Expressions and Functions

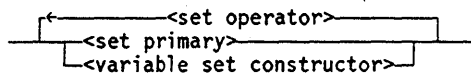
<Boolean primary>

In NEWP, a <set relation> is allowed as a <Boolean primary> in addition to those items allowed by ALGOL.

<set relation>



<variable set expression>



Explanation

NEWP includes <conditional operator>s, which are similar to their corresponding standard <Boolean operator>s. However, when a conditional operator is used, a second operand is not evaluated if the value of the first operand is sufficient to determine the value of the expression. The value returned is either 1 (TRUE) or 0 (FALSE) in the low-order bit of the resulting operand; the remaining 47 bits of the operand are undefined. The following table illustrates the functions of these conditional operators:

Operands		Operations		
L	R	L CAND R	L COR R	L CIMP R
TRUE	bool	bool	TRUE	bool
FALSE	bool	FALSE	bool	TRUE

CAND has the same precedence as AND, while COR has the same precedence as OR, and CIMP has the same precedence as IMP.

Following are examples of Boolean operators:

```

B := R1 NEQ 0 CAND R2/R1 EQL R3;
B := R1 GEQ 0 AND R1 LSS SIZE(A) COR R2 NEQ A[R1];
B := R1 GTR 0 CIMP A[R1-1] NEQ R1;
    
```

Precedence in Boolean Expressions

Occasionally, the precedence of terms within primary parts of expressions should be specified by parentheses. For example, when you use the <pointer table membership> primary, enclose it in parentheses whenever it is followed by a Boolean operator. In ALGOL, the following statement compiles:

```

IF PTR_ID IN TRUTH_ID FOR 1 AND BOOLEAN_ID THEN
    
```

In NEWP, to get the statement to compile you need to change it as follows:

```
IF (PTR_ID IN TRUTH_ID FOR 1) AND BOOLEAN_ID THEN
```

Set Relation

There are two kinds of set relations; both return a Boolean value. The first compares the two < simple set expression > s for equality (= or EQL) or inequality (^ = or NEQ). The second determines whether or not the value of the specified < variable ordinal > is a member of (is IN) the set specified by the set expression. When simple set expressions are compared, they must be of compatible types. Equality or inequality is determined by doing a bit-by-bit comparison of two sets. Two sets of different lengths are considered equal if every bit that is set in one is also set in the other.

Example

```
TYPE COLOR      = (RED, BLUE, GREEN, YELLOW, ORANGE);
TYPE COLORSET   = SET OF COLOR;

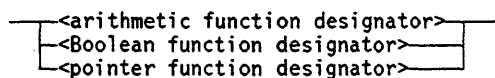
COLORSET CSET1, CSET2, CSET3;
BOOLEAN B1, B2;

CSET1 := [RED] + [BLUE];
CSET2 := CSET1 * [YELLOW, BLUE, GREEN]; % CSET2 IS [BLUE]
CSET1 := CSET1 - CSET2;                % CSET1 IS [RED]
CSET3 := UNIVERSE (COLOR);

B1 := CSET1 EQL [RED, BLUE];           % FALSE
B2 := BLUE IN CSET2;                  % TRUE
```

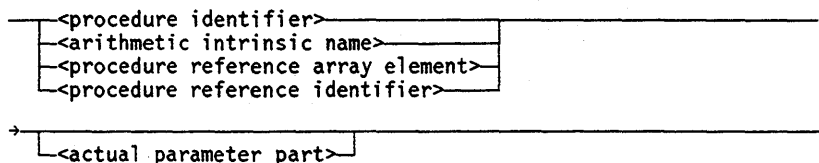
Function Expressions

<function expression>



Arithmetic Function Designator

<arithmetic function designator>

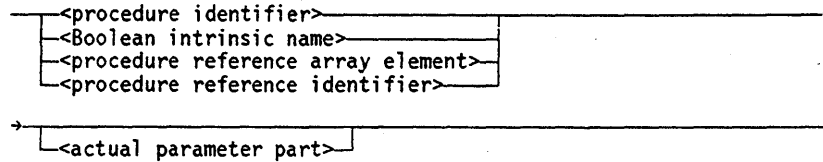


Explanation

The procedure specified by the procedure identifier, the procedure reference array element, or the procedure reference identifier must be of type INTEGER, REAL, DOUBLE, or of a subtype descended from INTEGER or REAL.

Boolean Function Designator

<Boolean function designator>



Explanation

The procedure specified by the procedure identifier, the procedure reference array element, or the procedure reference identifier must be of type BOOLEAN or of a subtype descended from BOOLEAN.

Pointer Expressions

Unlike ALGOL, NEWP does not allow a fully subscripted, noncharacter array that is used as a <pointer primary>. For example, the following program block compiles in ALGOL, but causes a syntax error on the REPLACE statement in NEWP:

```
BEGIN
  ARRAY B[0:10];
  REPLACE B[0] BY 0 FOR 11 WORDS;
END;
```

An alternative is to use the POINTER type transfer function as follows:

```
BEGIN
  ARRAY B[0:10];
  REPLACE POINTER(B[0]) BY 0 FOR 11 WORDS;
END;
```

Scalar Type Expressions

<scalar type expression>

An expression that evaluates to a scalar type value.

<simple scalar type expression>

A <scalar type expression> that consists only of scalar type variables and scalar type constants.

<constant scalar type expression>

A <scalar type expression> that consists only of scalar type constants.

Explanation

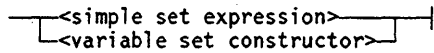
The operands in a scalar type expression must have the same root type. The resulting type of the expression is based on the operation and the two operands. If all the operands are the same type, the resulting type of operations involving addition (+), subtraction (-), multiplication (*), MOD, and the Boolean operators is that of the operands.

For those operations involving operands of different types and union, difference, intersection, MOD, and the Boolean operators, the resulting type is that of the common ancestor between the operands. An exception to this occurs when one of the operands is a constant. In this case, the resulting type is that of the nonconstant operand. For all other operations, the resulting type is INTEGER, REAL, or BOOLEAN, based on the type of operation.

Enumerated types and subtypes descended from enumerated types are not allowed in arithmetic expressions. Exceptions to this are in the CASE and FOR statements. For more information on CASE and FOR statements, see Section 5, "Statements."

Set Expression

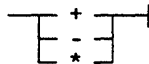
<set expression>



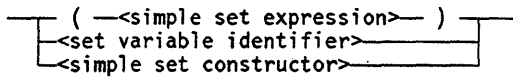
<simple set expression>



<set operator>



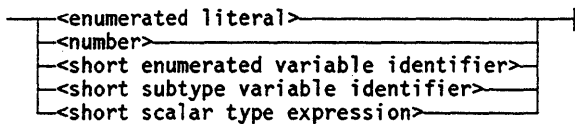
<set primary>



<simple set constructor>



<ordinal>



<short enumerated variable identifier>

<short subtype variable identifier>

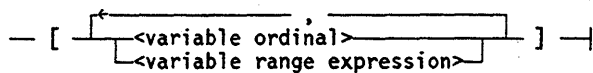
<short scalar type expression>

A variable or scalar expression for which the highest valid value is 47 or less, and for which the lowest valid value is 0 (zero) or greater.

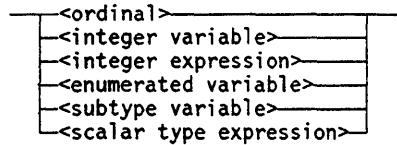
<range expression>



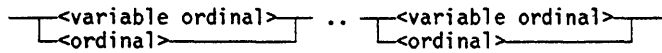
<variable set constructor>



<variable ordinal>



<variable range expression>



Explanation

A < set expression > generates a set. The < set operator > s perform the set operations of union (+), difference (-), and intersection (*).

The operators can be applied to declared < set variable identifier > s or to sets that are defined within the expression by the use of the < simple set constructor > syntax (these sets are referred to as anonymous sets). The < set primary > s within a set expression must be of compatible types.

A set constructor defines a value of an implied set type (an anonymous set). The members of the anonymous set are specified by the list of ordinal and range expressions, which must all be of the same type or subtypes of the same base types.

The associated value of an enumerated literal cannot be used in place of the literal; integers can be used only when the base type of the set being constructed is INTEGER or a subtype of INTEGER. If a range expression is used, the base type must be a discrete, ordered data type; that is, an ordered enumerated type, INTEGER, or one of their descendant types. The members denoted are those values from the first ordinal through the second ordinal. If the second ordinal is less than the first ordinal, the range expression evaluates to the null set.

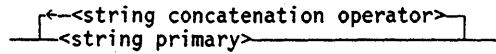
The UNIVERSE function returns a set in which all possible members of the type indicated by the type identifier or set identifier are present. The type must be a discrete data type; that is, an enumerated type, a subtype of INTEGER, or one of their descendant types.

The result of a set constructor takes its type from the elements used. In the case of the UNIVERSE function, it is the type of the type identifier or set identifier. Otherwise, the result is the type of the ordinals. An empty set constructor (that is, []) has no specific type and can be used in any set expression.

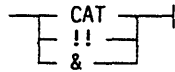
If the MCP compiler option is set, < variable range expression > s cannot be used in set expressions.

String Expressions

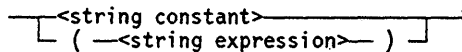
<string expression>



<string concatenation operator>



<string primary>



Explanation

You can concatenate two or more string constants by using the < string concatenation operator >. The concatenation of two strings yields a new string whose length is the sum of the lengths of the two original strings. The value of the new string is formed by joining the second string immediately onto the end of the first string.

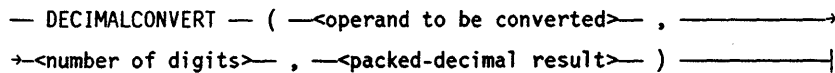
Only < string constant >s of the same character type can be concatenated. For more information on string constants, see "String Constants" in Section 3, "Language Components."

Examples of string expressions are as follows:

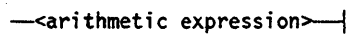
```
"LONG STRING CONSTANT"
"NUMBER" & 48"F1"
47"3138" !! 7"ASCII CHARACTERS"
"" CAT "DOESNT DO MUCH"
```

DECIMALCONVERT Function

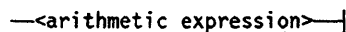
<decimalconvert function>



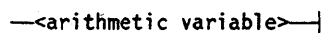
<operand to be converted>



<number of digits>



<packed-decimal result>



DECIMALCONVERT is a Boolean function that takes three arguments. The first argument, V, is a single- or double-precision value that is to be converted to

packed-decimal form. The second argument, SF, specifies the number of packed-decimal digits to be converted. SF must be in the range 0 through 24. The third argument, D, must be a variable of type REAL or DOUBLE. The remainder of $V \text{ DIV } 10^{**SF}$ is returned by way of D as a left-justified, packed-decimal operand; if D is a double-precision value, the value returned is first extended to double precision. The function result is TRUE if the quotient $V \text{ DIV } 10^{**SF}$ is nonzero; otherwise, the result is FALSE. The value returned in D can later be converted to a string of EBCDIC numeric characters by using the UNPACK intrinsic.

DINTEGERT Function

The DINTEGERT function has the following syntax:

— DINTEGERT — (—<arithmetic expression>—) —|

DINTEGERT is a DOUBLE function that returns the value of the arithmetic expression as a double-precision integer value, with truncation. It differs from INTEGERT in that it returns a double value and differs from DINTEGER in that it uses truncation instead of rounding. For more information on INTEGERT, see the *A Series ALGOL Programming Reference Manual, Volume 1: Basic Implementation*.

INTERLOCK Functions

INTERLOCK functions use interlocks or interlock array elements to protect a resource that is shared between several participating processes. The use of these functions is similar to the use of an event with the PROCURE and LIBERATE statements except that interlock functions are often considerably faster than PROCURE and LIBERATE statements. The initial state of an interlock is FREE. For a complete list of the possible states of an interlock, refer to the “LOCKSTATUS Function” later in this section.

ARROGATE Function

<arrogate function>

— ARROGATE — (—<interlock designator>—) —|

<interlock designator>

—<interlock identifier>—
 —<interlock array identifier> [—<subscript>—] —|

The <arrogate function> claims the specified interlock for the caller, regardless of the previous state, but does not modify the contender list. At the completion of the ARROGATE function the interlock owner is the process that issued the ARROGATE function, and the state is either LOCKED_UNCONTENDED or LOCKED_CONTENTENDED. The ARROGATE function is of type REAL, and the prior status of the interlock is returned as the result of an ARROGATE function. Refer to the LOCKSTATUS function for the format of the result of the ARROGATE function.

This operation can be used in several different circumstances. First, it can be used to “steal” the interlock. This might be desirable when you detect a correctable problem

with the protected resource. A process can steal the interlock, correct the condition, and then return the interlock to normal use with the `<unlock interlock function>`. In this case, some action must be taken to notify or eliminate the process that held the interlock when the interlock was stolen. Otherwise, both the original owner and the new owner (after the interlock is returned to use) would believe that they were the owner. Because the contender list is not modified, an `ARROGATE` function should be used instead of a `<break function>` if the condition is correctable.

A second use of the `ARROGATE` function is to return a `BROKEN` interlock to normal use. You can accomplish this by issuing an `ARROGATE` function followed by an `UNLOCK INTERLOCK` function on an interlock whose state is `BROKEN`.

If the interlock is `FREE` when the `ARROGATE` function is issued, it becomes `LOCKED_UNCONTENDED`.

If the interlock is `LOCKED_UNCONTENDED` or `LOCKED_CONTENDED`, the state remains unchanged, but the owner process identifier is changed to reflect the new owner.

If the interlock is `BROKEN`, it becomes `LOCKED_UNCONTENDED`.

The `ARROGATE` function should be used primarily when you can determine that the entity protected by the interlock is temporarily corrupted or unavailable, but can be corrected. The `ARROGATE` function can be used to force acquisition of the interlock so that the condition can be corrected before any other contender is given access to the interlock (and the protected resource).

Caution

Few complications should arise if the lock is not held by any process when the `ARROGATE` function is used. Otherwise, extreme caution should be exercised. In many cases, it might be desirable to terminate the process that owned the lock. This action prevents the original owner from interfering with recovery efforts.

Examples of `ARROGATE` functions are as follows:

```
I := ARROGATE (MYLOCK)
ARROGATE (YOURLOCKS [3])
```

BREAK Function

```
<break function>
— BREAK — ( —<interlock designator>— ) —|
```

The `<break function>` is used to remove an interlock from normal use and to cause an error to be returned to all contenders for the interlock. This function is desirable if the resource that the interlock is protecting becomes permanently unavailable or hopelessly

corrupted. You can use this function regardless of the state of the interlock at the time the operation is performed.

When a BREAK function is executed, the state of the interlock is set to BROKEN, regardless of the prior state. In addition, any contenders waiting for the interlock receive an error result. The interlock is marked as owned by the process that issued the BREAK function.

The BREAK function is of type REAL, and the prior status of the interlock is returned as the result of a BREAK function. Refer to the "LOCKSTATUS Function" for the format of the result of the BREAK function.

The only way to return a BROKEN interlock to normal use is by using the ARROGATE function, followed by the UNLOCK INTERLOCK function.

Refer to "ARROGATE Function" for information on temporarily removing an interlock from normal operation. The BREAK function should be used only if it is desirable to have all contenders return from the <lock interlock function> with an error.

Caution

Few complications should arise if the lock is not held by any process when the BREAK function is used. Otherwise, extreme caution should be exercised. In many cases, it might be desirable to terminate the process that owned the lock when the lock was broken. This action prevents the original owner from further corrupting the protected resource.

Following are examples of the BREAK function:

I := BREAK (MYLOCK)

I := BREAK (YOURLOCKS [3])

LOCK Function

<lock function>

— LOCK — (—<interlock designator> — , —<timeout> —) —

<timeout>

—<arithmetic expression>—

The <lock function> attempts to acquire the interlock. If you use the LOCK function as a statement, the process is discontinued when the result is not 1 (successfully acquired). The <timeout>, if present, specifies the amount of time the caller can wait if the interlock cannot be acquired immediately. The timeout is specified in seconds, and a value less than zero indicates that the program can wait indefinitely. If no timeout is

supplied, a timeout of -1 is assumed. If the timeout is 0 (zero), the caller cannot wait, and the lock succeeds only if the interlock is FREE.

The LOCK function is of type INTEGER, and the following values can be returned:

Value	Meaning
1	The interlock was successfully acquired.
2	The timeout elapsed before the interlock could be acquired.
4	The interlock has a state of BROKEN and cannot be acquired. This occurs when the BREAK operation is used on the interlock. Refer to "BREAK Function" earlier in this section for more information on BROKEN interlocks.

The following conditions cause various values to be returned:

- If the interlock has a state of FREE, it becomes LOCKED_UNCONTENDED and a result of 1 is returned.
- If the interlock is LOCKED_UNCONTENDED, it becomes LOCKED_CONTENTENDED and the caller is placed in the contender list. When the owner unlocks the interlock, and the caller is at the head of the contender list, a result of 1 is returned. If the timeout expires before the caller can acquire the interlock, then a result of 2 is returned.

Note: If the caller that timed out is the only contender, the interlock becomes LOCKED_UNCONTENDED.

- If a BREAK operation is performed on the interlock while the caller is in the contender list, a result of 4 is returned.
- If the interlock is LOCKED_CONTENTENDED, its state does not change, and the caller is added to the contender list. A result of 4 is returned.
- If the interlock has a state of BROKEN, its state does not change, and a result of 4 is returned.

Examples of the LOCK function are as follows:

```
I := LOCK (MYLOCK, 17)
I := LOCK (MYLOCKS [3])
LOCK (MYLOCK)
LOCK (OURLOCKS [2], 4)
```

LOCKSTATUS Function

<lockstatus function>

— LOCKSTATUS — (—<interlock designator>—) —|

The <lockstatus function> returns the status of the specified interlock. If the LOCKSTATUS function is used as a statement, a warning is issued at compile time, and no code is generated for the LOCKSTATUS. The LOCKSTATUS function is of type REAL, and the result has the following subfields:

```

[47:24] Owner's process ID; 0 if none
[23:22] Undefined
[01:02] Current state:
        0 FREE
        1 LOCKED_UNCONTENDED
        2 LOCKED_CONTENDED
        3 BROKEN
    
```

The owner portion of the interlock status can be compared to the task attribute `STACKNUMBER` and the function `PROCESSID`.

Examples of the `LOCKSTATUS` function are as follows:

```

R := LOCKSTATUS (MYLOCK);

IF I := LOCKSTATUS (YOURLOCKS [3]).[47:24] = PROCESSID THEN
    GOFORIT;
    
```

UNLOCK Function

<unlock function>

— UNLOCK — (—<interlock designator>—) —|

The <unlock function> is normally used to relinquish an interlock that was used earlier in a `LOCK` function. If the `UNLOCK` function is used as a statement, the process is discontinued when the result is not 1. The `UNLOCK` function is of type `INTEGER`, and the following values can be returned:

Value	Meaning
1	The interlock was successfully unlocked.
4	The interlock has a state of <code>BROKEN</code> and cannot be unlocked. Refer to "ARROGATE Function" for information on how to return a <code>BROKEN</code> interlock in normal operation.
6	The interlock has a state of <code>FREE</code> and therefore cannot be unlocked.

The following conditions cause various values to be returned:

- If the interlock is `FREE`, the state is not changed and a result of 6 is returned.
- If the interlock is `LOCKED_UNCONTENDED` when this operation is performed, the state is changed to `FREE` and a result of 1 is returned.
- If the interlock is `LOCKED_CONTENDED`, a result of 1 is returned, and the interlock is given to the first contender in the contender list. If there is only one contender in the list, the state is changed to `LOCKED_UNCONTENDED`; otherwise, it is not changed.
- If the interlock is `BROKEN`, the state is not changed and a result of 4 is returned.

Examples of the `UNLOCK` function are as follows:


```
I := UNLOCK (MYLOCK)

UNLOCK (YOURLOCKS [2])
```

PACKDECIMAL Function

```
— PACKDECIMAL — ( —<arithmetic expression>— , —————>
→<arithmetic expression>— ) —————|
```

PACKDECIMAL takes two arguments: a dividend V and a scale factor S. Each argument is first converted to an integer, if necessary. The resulting value of S must then be an integer in the range 0 through 12. PACKDECIMAL then returns a single-precision, left-justified, packed-decimal operand that represents the remainder of the following equation:

$$V \text{ DIV } 10^{**S}$$

The rightmost 12 - S digits are undefined.

SCALAR TYPE Functions

The following functions return scalar type values. The function value is undefined whenever any of its arguments has an undefined value.

LOWER BOUND Function

```
<lower bound function>
—<type identifier>— . — LBOUND —|
```

The <type identifier > must denote one of the following discrete, ordered data types that has a defined lower bound:

- Ordered enumerated types
- Subtypes of ordered enumerated types
- Subtypes of INTEGER that include a range specification

For a subtype, the LOWER BOUND function returns the smallest value permitted in that type. For an ordered, enumerated type, this function returns the literal with the lowest associated value. The type of the function value is the same as the type identifier.

MAPPING Function

```
<mapping function>
—<type identifier>— ( —<expression>— ) —|
```

The MAPPING function is a type transfer function that changes the type of the <expression > to the type of the type identifier. Within the valid combinations given in the following paragraphs, if the value of the expression is not valid for the type identifier, the result of the MAPPING function is undefined.

The expression provided as a parameter to the MAPPING function must evaluate to a value that is valid for the type identifier. The valid range for an enumerated type ranges from the numeric value associated with the first literal for the type to the numeric value associated with the last literal for the type. The valid range for a subtype is either specified with the declaration of the subtype or inherited from the parent type.

Example

As an example, assume the following declarations and assignments:

```

TYPE COLOR = (RED=1, GREEN, BLUE, BLACK, WHITE, YELLOW=6);
TYPE MONEY = ORDERED (PENNY=6, NICKEL, DIME, QUARTER, DOLLAR);
TYPE COINS = SUBTYPE MONEY PENNY. .QUARTER;
TYPE WEIGHTS = SUBTYPE INTEGER;
TYPE LARGEWEIGHTS = SUBTYPE WEIGHTS 1000. .10000;
INTEGER I;
COLOR MYFAVORITE, YOURFAVORITE, THEIRFAVORITE;
MONEY TOTALWORTH, DESIREDWORTH;
COINS POCKETCHANGE;
WEIGHTS BABYWEIGHT;
LARGEWEIGHTS HEAVY;

I := 2;
MYFAVORITE := BLUE;
YOURFAVORITE := YELLOW;
TOTALWORTH := DOLLAR;
POCKETCHANGE := DIME;
BABYWEIGHT := 10;
HEAVY := 1001;
    
```

In this particular case, the valid numeric ranges for the various types are as follows:

Type	Valid Range
COLOR	1 through 6
MONEY	6 through 10
COINS	6 through 9
WEIGHTS	Range for INTEGERS
LARGEWEIGHTS	1000 through 10000

Given these valid ranges and the initial assignments shown in the previous example, the following uses of the MAPPING function all return valid results:

```

THEIRFAVORITE := COLOR (3);           % THEIRFAVORITE gets BLUE = 3
THEIRFAVORITE := COLOR (PENNY);      % Ok because PENNY = 6 = YELLOW
THEIRFAVORITE := COLOR (1);         % THEIRFAVORITE gets GREEN = 2
DESIREDWORTH := MONEY (YELLOW);     % Ok because YELLOW = 6 = PENNY
DESIREDWORTH := MONEY (BABYWEIGHT); % Ok because BABYWEIGHT = 9
                                     % which equals QUARTER
BABYWEIGHT := WEIGHTS (COINS);      % COINS = DIME, which is 8
    
```

Expressions and Functions

The following assignments produce syntax errors because the compiler can detect that they all produce values outside the valid range for the type identifier in the MAPPING function:

```
THEIRFAVORITE := COLOR (15);           % Maximum COLOR is 6
THEIRFAVORITE := COLOR (QUARTER)      % QUARTER is a literal = 9;
                                       % too large for COLOR
DESIREDWORTH := MONEY (GREEN);       % GREEN = 2; too small for MONEY
POCKETCHANGE := COINS (DOLLAR);      % DOLLAR is a literal = 10;
                                       % too large for COINS
THEIRFAVORITE := COLOR (HEAVY);       % HEAVY is of type LARGEWEIGHTS,
                                       % and the valid range for COLOR
                                       % does not overlap the valid
                                       % range for LARGEWEIGHTS, so
                                       % a syntax error can be given.
```

By contrast, the following assignment is valid but produces a run-time error because a variable, rather than a constant, is involved. In addition, the valid ranges of the types involved overlap:

```
THEIRFAVORITE := COLOR (POCKETCHANGE); % POCKETCHANGE contains
                                       % DIME = 8; too large
                                       % for COLOR
```

PREDECESSOR Function

<predecessor function>

```
—<type identifier>— . — PRED — ( —————→
→<simple scalar type expression>— ) —————|
```

The type identifier must denote a discrete, ordered data type. The type of the function value is the same as type identifier.

If type identifier is a descendant type of INTEGER, then the <simple scalar type expression> must be an expression of type identifier. If the value of the expression is higher than the lowest value defined for that type (if any), the function returns the next lowest value of that type. This function is undefined if the value of the expression is less than or equal to the lowest value defined for that type (if any).

If type identifier is an ordered enumerated type or a descendant of an ordered enumerated type, then the simple scalar type expression must be an enumerated literal of that type. If the literal does not denote the lowest associated value for that type, the function returns the next lowest associated value of that data type. This function is undefined for the literal with the lowest associated value.

SUCCESSOR Function

<successor function>

```
—<type identifier>— . — SUCC — ( —————→
→<simple scalar type expression>— ) —————|
```

The type identifier must denote a discrete, ordered data type. The type of the function value is the same as type identifier.

If type identifier is a descendant type of INTEGER, then the simple scalar type expression must be an expression of type identifier. If the value of the expression is less than the highest value defined for that type (if any), the function returns the next highest value of that type. This function is undefined if the value of the expression is higher than or equal to the highest value defined for that type (if any).

If type identifier is an ordered enumerated type or a descendant of an ordered enumerated type, then the simple scalar type expression must be an enumerated literal of that type. If the literal does not denote the highest associated value for that type, the function returns the next highest associated value of that data type. This function is undefined for the literal with the highest associated value.

UPPER BOUND Function

<upper bound function>

—<type identifier>— . — UBOUND —|

The type identifier must denote one of the following discrete, ordered data types that has a defined upper bound:

- Ordered enumerated types
- Subtypes of ordered enumerated types
- Subtypes of INTEGER that include a range specification

For a subtype, the UPPER BOUND function returns the largest value permitted in that type. For an ordered enumerated type, this function returns the literal with the highest associated value. The type of the function value is the same as type identifier.

Section 7

Compiling NEWP Programs

In NEWP, as in ALGOL, you can perform three kinds of compilations. The general term *compile* refers to any of these three types:

- A full compilation, which parses all source lines and translates them into object code. This default compilation type occurs unless you set the control option MAKEHOST or SEPCOMP.
- A host compilation, which is performed when you set the control option MAKEHOST. A host compilation is a superset of a full compilation. In addition to processing all source lines, it puts extra information in the code file that is necessary for the use of SEPCOMP.
- A SEPCOMP, which is performed when you set the control option SEPCOMP. A SEPCOMP compiles only the minimum necessary source lines, primarily those areas changed by a patch. The code and symbol files from a host compilation are used as the basis for a SEPCOMP. SEPCOMP is considerably faster than either a full compilation or a host compilation, but cannot be used in all cases. For further information on the limitations of SEPCOMP, see “SEPCOMP Guidelines” later in this section.

This section provides information about all three types of compilations and states the rules for using each one.

Note: The term *host compile* is often used as a general term for both *full* and *host* compilations. Because the *host* compilation takes longer and produces a larger code file, you should use a *SEPCOMP* or a *full* compilation whenever possible.

For information about the input and output files for each type of compile, refer to the *A Series ALGOL Programming Reference Manual, Volume 1: Basic Implementation*.

Full Compilations

A full compilation is performed by default if you do not set the control options MAKEHOST or SEPCOMP. This type of compilation accepts one or more source files and, if there are no syntax errors, produces a code file.

Host Compilations

A host compilation is performed if you set the control option MAKEHOST. This type of compilation accepts one or more source files and, if there are no syntax errors, produces a code file that contains both the object code and the additional information that is necessary to do a SEPCOMP. This object code file is often referred to as a host file. Because of the inclusion of extra information, the code file produced by a host

compilation is always larger than the code file produced by a full compilation. However, the actual object code produced is identical in both cases.

The amount of information saved during a host compilation is under your control through the block directive `SEPCOMPLEVEL`. This block directive includes an integer that is the highest lexical level for which declaration information is stored in the host file. Each declaration for which this information exists can be separately compiled during a `SEPCOMP`. As a result, the value of the `SEPCOMPLEVEL` block directive during a host compilation determines the “granularity” of the separate compilation done during a `SEPCOMP`. A small value causes larger areas to be recompiled during a `SEPCOMP`, and a large value causes smaller areas to be recompiled. For more information about `SEPCOMPLEVEL`, refer to “`SEPCOMP` Background” in this section.

One piece of information stored in a host file is the title of a symbolic associated with the compile. This information is required during a `SEPCOMP` and is determined as follows.

If a new symbolic is being produced during the host compilation (that is, the control option `NEW` is set), the title of the symbolic is saved. Otherwise, if you set the `MERGE` option, the title associated with the `TAPE` file is saved because the bulk of the symbolic is usually found there. However, if you set neither the `NEW` option nor the `MERGE` option, the title associated with the `CARD` file is saved because that file contains the only input.

Note: When the `MERGE` option is set, but the `NEW` option is not, the symbol file associated with the host is not complete, because it does not contain the changes made by the patch file (`CARD`). In this case, you should do one of two things when a `SEPCOMP` is performed with the host. You should either include the host-compiled patches with those to be compiled, or you should use the `SEPCOMP` merge capability. More information on both `SEPCOMP` and `SEPCOMP` merge is provided later in this section.

Compiling with `SEPCOMP`

A `SEPCOMP` is performed when you set the `SEPCOMP` control option. This type of compilation requires at least three input files: a patch, a base symbol file, and the host file (object) from a previous host compile or `SEPCOMP`. The result is a complete code file, produced by taking a copy of the host file and changing only the sections modified by the patch. The base symbol file is necessary to allow the recompilation of the source code that was changed.

Performing a `SEPCOMP` is not difficult, and it offers significant savings in time when compared to full and host compiles. However, some types of changes cannot be compiled with `SEPCOMP`, and you should become familiar with the rules of `SEPCOMP` to ensure a reliable result. For information on these rules, see “`SEPCOMP` Guidelines” later in this section.

`SEPCOMP` is implemented by the `NEWP` compiler itself and does not require invocation of `BINDER`, as it does in `ALGOL`.

Note: The object file produced by a NEWP SEPCOMP is a host file; another SEPCOMP can be done against it. However, the object code contained in the file might not be identical to the object code that would have been produced by a full or host compile.

Certain types of syntax errors cannot be detected by a SEPCOMP because of the limited nature of the compilation. Although these errors are unusual, they do occasionally occur. In addition, if the guidelines listed under "SEPCOMP Guidelines" are not followed, a full or host compile and a SEPCOMP can produce code files that are different. These problems can be avoided by following the guidelines.

Because of the limited compilation that is done during a SEPCOMP, certain other control options cannot be used. For example, neither a new symbol file (\$NEW) nor xref files (\$XREF) can be produced by a SEPCOMP. Both these options require that all the source input be processed; therefore, these options have no meaning during a SEPCOMP.

The user compiler control options that had a TRUE value when the host file was created are preserved and reinstated during a SEPCOMP. The VERSION option supplied for the host is also reinstated.

As mentioned earlier, it is possible to use the output from one SEPCOMP as a host file for another one. The title of the default symbolic file is updated, as previously described for host compiles. Because there is not a symbol file that matches the host file exactly, you should read the information on the NEWP SEPCOMP merge capability later in this section before using the output of a SEPCOMP as a host file.

SEPCOMP Background

The NEWP SEPCOMP facility is based on the concept of a region, which is a section of the program that can be separately compiled. Any change within a region causes the entire region to be recompiled. The size of a region is determined by the value of the SEPCOMPLEVEL block directive during a host compile.

The executable statements of the outer block are always a region. In addition, a declaration list (for example, REAL A,B,C;), a procedure heading, and a procedure body can be regions, depending on their lexical level and the value of the SEPCOMPLEVEL block directive.

An easy way to understand what is and is not a region is to consider the lexical level portion of the address of an item. If the lexical level is less than or equal to the SEPCOMPLEVEL, then the declaration is a region. Otherwise, the item is part of some other region.

For example, if a declaration in the outer block of a non-MCP program is REAL A,B;, then A and B are assigned the addresses (2,x) and (2,x+1) respectively. Therefore, this declaration list is a region if SEPCOMPLEVEL is 2 or higher. On the other hand, if the same declaration appeared inside a procedure, the Program Control Word (PCW) of the procedure would be at (2,x), but A and B would be at lexical level 3 (for example, 3,x). If the SEPCOMPLEVEL is 2, the procedure would be a region, and the declaration of

A and B would simply be part of that region. Any change to the declaration of A and B causes a recompilation of the entire region (the procedure).

The default value of SEPCOMPLEVEL is 2 when the MCP control option is not set; the default value is 0 (zero) when the MCP option is set.

NEWP records the starting and ending sequence numbers of each region. A declaration list begins on the line where the type (REAL, BOOLEAN, and so forth) is stated, and includes everything up to the semicolon (;). A procedure heading region begins on the line where the keyword PROCEDURE appears, and includes all the parameter declarations. A procedure body begins on the line where the BEGIN appears, and includes everything until the procedure END statement.

In some cases, the recompilation of one region causes the recompilation of another region, even though the regions can be compiled separately. For example, if a declaration in a procedure is changed, the body of the procedure is also recompiled because the stack building code for the declarations needs to be regenerated. This recompilation occurs even when the nested declaration is at or below the lexical level specified by the SEPCOMPLEVEL block directive. The determination of that which needs to be recompiled is therefore based primarily on the patch and the SEPCOMPLEVEL, but is also affected by the relationships between regions.

During SEPCOMP, if the LIST compiler control option is set, this option causes a source listing to be generated for every patched region. Since any given area of a program can be made up of a number of regions, a complete listing of the area occurs only if every region in the area is changed. For example, at or below the lexical level specified by the SEPCOMPLEVEL block directive, a procedure heading and the body of the procedure are two different regions. If a SET LIST is placed before the start of the procedure declaration and if a POP LIST is placed after the end of the procedure, then a complete listing is obtained only if both the heading and the body are changed. To get a listing of the procedure body, you must put the SET LIST just after the procedure BEGIN and the POP LIST just before the procedure END.

The level of the SEPCOMP information is stored in the host file. This level determines the contents and format of the SEPCOMP information. The level of the SEPCOMP information produced during the host compile must match the level of the information expected by the compiler doing the SEPCOMP. Since it is periodically necessary to improve the SEPCOMP information, not all combinations of host file, patch, and compiler succeed. If such a mismatch occurs, an error message is issued to indicate the mismatch and provide more information to resolve the error. In general, there are two choices: use the same version of the compiler for the SEPCOMP that was used for the host compile, or recompile the host with the compiler that will be used for the SEPCOMP.

Performing a SEPCOMP

Performing a SEPCOMP consists of providing the necessary input files and ensuring that the SEPCOMP option is set. The compiler can supply default titles for all of the required files, but ordinarily you supply the actual titles. Although SEPCOMPs can be performed from Command and Edit (CANDE), they are generally started from Work Flow Language (WFL) instead. Therefore, WFL examples are used here. For

further information on the WFL programs, see the *A Series Work Flow Language (WFL) Programming Reference Manual*.

The internal names for the three required input files are as follows:

CARD	The patch file.
TAPE or SOURCE	The name of the base symbolic associated with the host file.
HOST	The name of the host file (the object). The host file must be compatible with NEWP.

If you do not supply names, the compiler uses the names CARD and HOST for the patch and host file respectively. If the name of the base symbolic has been file-equated to SOURCE or TAPE, then that name is used. Otherwise, the base symbolic file name is taken from the host file, where it is stored during the creation of the host.

As an example, assume that the following program has been compiled to produce a host file (the object) and a base symbolic:

```

100 $ SET MAKEHOST
200 $ SET NEW "SYMBOL/MYPROGRAM"
300 BEGIN [SEPCOMPLEVEL = 2]
400
500 REAL X, Y, Z;
600
700 PROCEDURE P;
800   BEGIN
900     BOOLEAN B;
1000    X := Y * Z;
1100    IF X EQL 7 THEN
1200      B := TRUE
1300    ELSE
1400      B := FALSE;
1500    END P;
1600 Y := 12;
1700 Z := 14;
1800 P;
2000 END; % OF PROGRAM

```

The compilation of this program produces both a host file named *SYSTEM/MYPROGRAM* and a new base symbolic file named *SYMBOL/MYPROGRAM*. The name of the new symbolic is stored in the host file, along with information about all the regions that occur at lexical level 2 and below.

If a program patch is developed that includes *\$ SET SEPCOMP* and is named *MYPROGRAM/PATCH*, the following WFL deck can be used to do a SEPCOMP:

```

BEGIN JOB SEPCOMP;
TASK T;
COMPILE SYSTEM/MYPROGRAM/SEPCOMPED WITH NEWP [T] LIBRARY;
  NEWP FILE CARD (KIND = DISK, TITLE = MYPROGRAM/PATCH);
  NEWP FILE HOST (TITLE = SYSTEM/MYPROGRAM);
END JOB;

```

Because the deck does not file-equate a TAPE file title, the default title *SYMBOL/MYPROGRAM* stored in the host file is used. The output is a host file named *SYSTEM/MYPROGRAM/SEPCOMPED* that is a copy of the original host file (*SYSTEM/MYPROGRAM*), except in those areas changed by the patch.

Note that it is possible to supply the name of the host file with the *SEPCOMP* option. In that case, the patch could include the following card and the file equation to the file *HOST* in the WFL deck could be dropped:

```
$ SET SEPCOMP "SYSTEM/MYPROGRAM"
```

A host file title provided this way has precedence over any file equation to the *HOST* file.

SEPCOMP Guidelines

Though *SEPCOMP* can be very useful, it does have some limitations. Certain types of changes can be compiled only with a full or host compilation. Other kinds of changes can be compiled with *SEPCOMP*, but require extra attention. Note that these restrictions only apply to changes at or below the *SEPCOMPLEVEL*. Any changes above this level are completely recompiled; therefore, the restrictions have no impact.

The following types of changes at or below the *SEPCOMPLEVEL* cannot be compiled with *SEPCOMP*; a full or host compilation must be done:

- Changes to library entry points.
- The addition of executable statements in front of the first executable statement in a region.
- Reordering of declaration regions.
- Modification of a *TYPE* declaration.
- Modification of the beginning or ending sequence numbers of a region. You can change the text on the beginning and ending lines of a region as long as the change follows the other rules listed here.
- Modification or addition of multiple regions on the same line (for example *REAL A; BOOLEAN B*).
- Modification of the type of a region (for example, *REAL X,Y* is changed to *BOOLEAN X,Y*).
- Deletion of a module.

- Addition of any text between adjacent modules (members of the same group of modules).
- The setting of the `INLINE` block directive; you cannot change a regular procedure to an in-line procedure, or vice versa, during a `SEPCOMP`.

Note: If the patch being compiled with `SEPCOMP` includes changes to an area that are not omitted in the host, testing cannot be completed until the patch is compiled with `SEPCOMP` and tested against another host in which that area was included. Note that during `SEPCOMP`, a warning is issued if the patch includes changes to omitted areas.

During a `SEPCOMP`, NEWP reads the `xref` files for the `HOST` and uses any that are present to recompile references to any changed or deleted declarations.

Compiling NEWP Programs

To accomplish this, the compiler automatically looks for the following files:

```
XREFFILES/<hostname>/REFS
XREFFILES/<hostname>/DECS
```

If present, they are used to ensure that all references to changed and deleted declarations are recompiled. If they are not present, you must follow the rules listed for changes at or below the SEPCOMP level.

When a patch to a declaration is encountered and the xref files are loaded, the old declaration is compared to the new declaration. If the declaration has changed enough to invalidate the references to the original declaration, the xref files are used to mark the referenced lines. Each of these lines is then recompiled. Some types of variables are not yet supported in this implementation. These include format variables, type variables, type declarations, translate tables, truthsets, files, and items in an export or interface list. When the xref files are loaded and one of these declarations is changed, a warning that the xref files were not used is generated, and the old SEPCOMP rules, shown in the following guidelines, apply.

The following guidelines must be observed if the designated types of changes occur at or below the SEPCOMP level and the xref files are not available. In most cases, you are responsible for ensuring that specific areas are recompiled. One way to accomplish this is to use the Editor command *CHANGE* to mark the specified lines as changed, even if no actual change was made. Another method is to include a *\$%* card in each region that needs recompilation.

- If an identifier is deleted, you must ensure that all references to it are also deleted. If all references are not deleted, a run-time error occurs when the code for the undeleted reference is executed.
- If the type of an identifier is changed, or its declaration is moved from one region to another, you must ensure that all references to the identifier are recompiled. If this is not done, the host file will contain sections of code that handle the variable according to the old type as well as code that generates code based on the new type.
- If the declaration of any of the following types of items is changed, you must ensure that all references to the identifier are recompiled. If this guideline is not followed, the resulting code file will not execute as expected, because some areas will use the old definition of the item and some will use the new.
 - Value array
 - Array bounds
 - Constant
 - Define
 - File
 - In-line procedure
 - Translate table
 - Truthset
- In the MCP, if a D[0] library EXPORT declaration is changed, you must ensure that all procedures containing a FREEZE(MCP) statement are recompiled.

SEPCOMP MERGE

NEWP provides a capability known as SEPCOMP MERGE that allows a SEPCOMP to be done, even when no complete symbolic exists. The need for SEPCOMP MERGE could arise in two different instances.

First, if a new symbol file is not created during the host compile, and a patch was included in the compilation, then no symbol file exists that includes the patch.

Second, since a new symbol file cannot be created during a SEPCOMP, using the results of a SEPCOMP as the host file for a later SEPCOMP leads to the same problem: no symbol file exists that includes the patch originally compiled with SEPCOMP.

SEPCOMP MERGE is very similar to SEPCOMP, except that it creates a virtual symbol. You provide two patch files: one patch that is used only to create a virtual symbol, and one normal SEPCOMP patch that is used to determine which regions are to be recompiled. Note that the virtual symbol patch does not need to be compiled, since it is already part of the host file.

The SEPCOMP MERGE capability saves time because there is no need to compile over and over again patches previously compiled with SEPCOMP. It is particularly useful when several patches are developed that patch the same area. Each one can be compiled in turn with SEPCOMP and then can be used to create the virtual symbol when the next patch is compiled with SEPCOMP.

The following WFL deck assumes that a second patch named MYPROGRAM/PATCH/2 is developed for the example program discussed earlier. The second patch includes the card \$ SET SEPCOMP MERGE.

```
BEGIN JOB SEPCOMP MERGE;  
TASK T;  
COMPILE SYSTEM/MYPROGRAM/SEPMERGED WITH NEWP [T] LIBRARY;  
  NEWP FILE CARD (KIND = DISK, TITLE = MYPROGRAM/PATCH/2);  
  NEWP FILE SOURCEP (TITLE = MYPROGRAM/PATCH);  
  NEWP FILE HOST (TITLE = SYSTEM/MYPROGRAM/SEPCOMPED);  
END JOB;
```

Because no title is equated for the TAPE file (base symbolic), the title stored in the host file is used as the default. The actual patch (CARD file) is applied against the base symbolic updated with the virtual source patch.

The SOURCEP file is the patch that is applied first to the base symbolic to obtain the virtual source. The title of this file can be provided as part of the control option, if desired. In that case, the second patch should include the following and the file equation to the SOURCEP file can be dropped from the WFL deck:

```
$ SET SEPCOMP MERGE "MYPROGRAM/PATCH"
```

A file title provided in this way has precedence over any file equation to the SOURCEP file.

The SEPCOMP MERGE capability is subject to the same rules and guidelines listed earlier for SEPCOMP.

Compiling for Syntax Only

WFL provides an option that can be used with some compilers to check only the syntax of a source program. However, NEWP does the same work during a syntax compilation as it does for any other type of compilation. The only difference is that the object file is removed just before the compilation terminates. Therefore, there is no point in setting this option in a NEWP compilation. For more information on compiling for syntax, see the *A Series Work Flow Language (WFL) Programming Reference Manual*.

Section 8

Compiler Controls

Compiler Control Options

Compiler control options in NEWP are similar in format and function to compiler control options in ALGOL. A list of ALGOL compiler control options not recognized in NEWP can be found in Appendix B, "ALGOL Features Not Implemented in NEWP." Any compiler control option item not listed in this section can be used as a user-control option. Option expressions are evaluated from right to left, unless parentheses are used to force a particular evaluation order. The following features differ from those in ALGOL:

- The specification of an option without including SET, RESET, or POP causes no action other than setting that option. Standard options are not reset. For example, \$MERGE does not reset LIST. The CLEAR option, as shown in the following example, can be used to reset all standard options:

```

$ MERGE          % IN NEWP HAS THE SAME EFFECT AS:
$ SET MERGE     % IN ALGOL OR NEWP

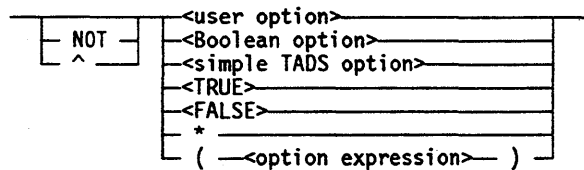
$ CLEAR MERGE   % IN NEWP HAS THE SAME EFFECT AS:
$ MERGE        % IN ALGOL
  
```

- The dollar sign (\$) must appear in column 1, 2, or 3.
- The simple TADS option and other compiler-provided Boolean options are recognized as <option primary> s for use in option expressions. The following expression with the use of AND sets the user option TADSLIST by including the current values of the compiler options TADS and LIST:

```
$SET TADSLIST = TADS AND LIST
```

The ALGOL definition of <option primary> is extended as follows:

<option primary>



ALGOL Options Duplicated in NEWP

The following compiler control options are implemented in NEWP exactly as they are in ALGOL:

- ASCII
- CODE
- DELETE
- ERRLIST, ERRORLIMIT, ERRORLIST
- INCLNEW, INCLSEQ, INCLUDE
- LIMIT, LINEINFO, LISTDOLLAR, LISTINCL, LISTOMITTED, LISTP
- MAP
- NEWSEQERR
- NOBINDINFO
- OMIT
- PAGE
- SEGS, SEQ, SEQUENCE, <sequence base>, <sequence increment>
- SHARING, STACK, STRINGS
- TARGET
- TIME
- <user option>
- VOIDT

Additional NEWP Options

The following options either are in ALGOL but are different in NEWP or are not in ALGOL and have been added to NEWP.

ASD

— ASD —

(Type: BOOLEAN, Default: FALSE)

This option is meaningful only if you set the MCP or STANDALONE option. If you set the ASD option, the code that is produced should be used only on a machine capable of supporting Master Control Program/Advanced Systems (MCP/AS). If you also set the MCP option, but not the ASD option, the resulting MCP can be used only on a non-ASD system. This condition is enforced when an attempt is made to change MCPs using the CM system command.

If you set both the ASD and STANDALONE options, the code can be used only on a machine capable of supporting MCP/AS. The memory image that the compilation produces in this case is tailored for ASD systems.

When the compiler control option ASD is set, memory access uses the Absolute Store Reference Word (ASRW). This setting builds the ASRW for stand-alone programs.

This option can appear only before the first source statement.

CLEAR

— CLEAR —

(Type: IMMEDIATE)

The CLEAR option resets all compiler-provided options that can be set.

INCLLIST

— INCLLIST —

(Type: BOOLEAN, Default: FALSE)

This option is a synonym in NEWP for the LISTINCL option.

INSTALLATION

— INSTALLATION — = <numeric constant> —————→
 → —————→
 : —————→
 - - <numeric constant>

(Type: BOOLEAN, Default: FALSE)

When TRUE, the INSTALLATION option causes the compiler to recognize a group of installation intrinsics so that they can be referred to in a NEWP program.

The <numeric constant> must be an unsigned integer from 1 through 2047. The first numeric constant must be less than the second numeric constant. Numbers larger than 2047 are treated as if they were equal to 2047. Numbers less than 1 are treated as if they were equal to 1.

INTERLOCKOPS

— INTERLOCKOPS — = HARDWARE CONDITIONAL

(Type: VALUE, Default: CONDITIONAL)

You can use this option to specify that when the <lock function> or <unlock function> is used, only hardware locking code is to be generated, or that the conditional locking

Compiler Controls

code is desired. The **HARDWARE** option is effective only when the **TARGET** is **LEVEL1**. If the **TARGET** is not **LEVEL1**, then **INTERLOCKOPS = HARDWARE** is ignored. Conditional locking code is generated by default, so the **CONDITIONAL** option explicitly states the default. This option can appear only before the first statement in the program.

The **HARDWARE** option should be used only on machines that support the **LOK** and **UNLK** operators.

The actual locking code generated depends on both the use of this control option and the use of the **INTERLOCKOPS** block directive. The most local setting of **INTERLOCKOPS** has precedence, regardless of whether the control option or the block directive is used.

LIST

```
-- LIST [ " <file name> " ]
```

(Type: **BOOLEAN**, Default: **FALSE** for compiles originated in **CANDE**; **TRUE** otherwise)

The **LIST** option in **NEWP** is identical to the **LIST** option in **ALGOL** except that **NEWP** allows you to specify a file name, which is used to set the **TITLE** attribute of the file **LINE** of the compiler. The **<file name>** must not include embedded quotation marks (").

LISTO

```
-- LISTO --
```

(Type: **BOOLEAN**, Default: **FALSE**)

If **LISTO** is **TRUE**, all records from the secondary input file (**TAPE**) that are voided or replaced and all records from the primary input file (**CARD**) that are omitted are printed.

LIST1

```
-- LIST1 [ " <file name> " ]
```

(Type: **BOOLEAN**, Default: **FALSE**)

If **LIST1** is **TRUE**, a listing is produced during the first pass that the compiler makes while it is compiling module headings. The **LIST1** option allows you to specify a file name, which is used to set the **TITLE** attribute of the file **LINE1** of the compiler. The **<file name>** must not contain any embedded quotation marks (").

MAKEHOST

— MAKEHOST —|

(Type: BOOLEAN, Default: FALSE)

The MAKEHOST option in NEWP is similar to the MAKEHOST option in ALGOL. It must be set before the first source statement. This option controls the collection of information necessary for a SEPCOMP. When this option is set, compilation takes longer and produces a larger code file than when it is reset. Therefore, the MAKEHOST option should be set only when the result of the compilation is to be used as input to a later SEPCOMP.

For more information about the use of this option and about SEPCOMP, refer to Section 7, "Compiling NEWP Programs."

MCP

— MCP —|

(Type: BOOLEAN, Default: FALSE)

The MCP option indicates whether the program being compiled is an MCP. If the MCP option is TRUE, the compiler allocates the segment dictionary and global variables at lexical level 0. Also, the code file generated when this option is TRUE contains a bootstrap in segment zero.

This option can appear only before the first source statement.

MERGE

— MERGE —|
 └ " <file name> " ┘

(Type: BOOLEAN, Default: FALSE)

The MERGE option in NEWP is identical to the MERGE option in ALGOL except that NEWP allows you to specify a file name, which is used to set the TITLE attribute of the file SOURCE of the compiler. The <file name> must not include embedded quotation marks (").

MODSTRICT

— MODSTRICT —|

(Type: BOOLEAN, Default: FALSE)

Setting the MODSTRICT option disables the automatic import of interfaces or user-defined types and values. The MODSTRICT option can be set only before the first source statement. This option controls automatic import of interfaces into module bodies when the interfaces were explicitly imported into the corresponding module head. The option also controls the automatic import of user-defined types and enumerated values

Compiler Controls

into a module when only a variable or the user-defined type itself was explicitly placed into the interface.

NEW

```
— NEW —|  
  | " <file name> " |
```

(Type: BOOLEAN, Default: FALSE)

The NEW option in NEWP is identical to the NEW option in ALGOL except that NEWP allows you to specify a file name, which is used to set the TITLE attribute of the file NEWSOURCE of the compiler. The <file name> must not include embedded quotation marks (").

NOCOUNT

```
— NOCOUNT —|
```

(Type: BOOLEAN, Default: FALSE)

The NOCOUNT option is used to control the accounting of statistical information when the STATISTICS option is TRUE. If NOCOUNT is TRUE at the end of the compilation of a procedure, the compiler emits code to cause the statistics handlers not to add the accumulated time for the execution of that procedure into the accumulated time for the procedure that called it.

This option applies only to MCP gathering statistics.

PROCREF

```
— PROCREF —|
```

(Type: BOOLEAN, Default: FALSE)

If PROCREF is TRUE, each source record that references a procedure is listed with the line number of the declaration of that procedure. If the procedure body has not yet been compiled, the line number of the FORWARD declaration of the procedure is listed, preceded by an F. If LIST is FALSE, the setting of PROCREF is ignored.

READLOCK

```
— READLOCK —|
```

(Type: BOOLEAN, Default: FALSE)

The READLOCK option, when TRUE, changes the value that the BUZZ and BUZZ47 intrinsics exchange with the contents of the variables specified in the BUZZ intrinsic. Instead of exchanging a value of all zeros except for one bit, the BUZZ intrinsic with READLOCK TRUE exchanges a modified Program Control Word (PCW) that identifies

the task, and location within the task, where the BUZZ task is being performed. The following is an example:

- [31:12] Stack number of the BUZZ task
- [19: 3] Program Syllable Index for the location at which the BUZZ appears
- [16:13] Program Word Index for the location at which the BUZZ appears
- [0: 1] (BUZZ) or [47: 1] (BUZZ47) = 1, to lock the lock

In addition, if READLOCK is TRUE, the contents of the variable specified in the BUZZ intrinsic, which were obtained during the first interchange operation, are left on the top of the stack for the duration of the BUZZ loop.

When the READLOCK and/or READLOCKTIMEOUT MCP compile time options are set, diagnostic information is placed in the word used when performing a BUZZ task on a lock. The format of that information is as follows:

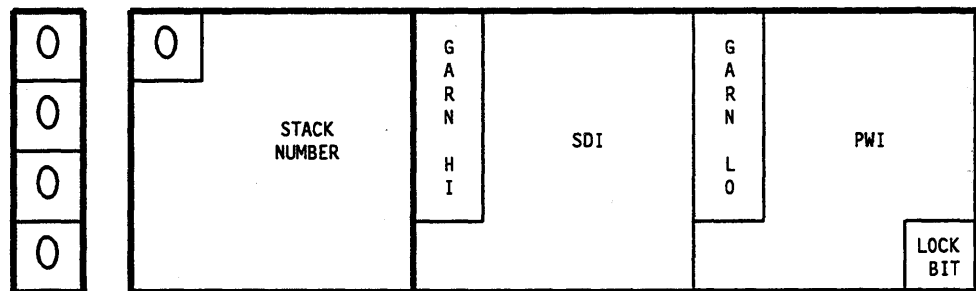


Figure 8-1. Diagnostic Information Format

The following is an explanation of the fields and bits of the word:

Field or Bit	Contents
Stack number	This field contains the value of the Stack Number Register (SNR), right justified.
GARN	This field contains the Global Activation Record Name (GARN). The GARN specifies the multiple MCP D1 activation records that contain the code segment descriptor to which the SDI field refers. This 6-bit name is split into two 3-bit fields. The high-order 3-bit field is in the GARN HI field, and the low-order 3-bit field is in the GARN LO field.
SDI and PWI	These fields contain the Segment Dictionary Index (SDI) of the code segment and the Program Word Index (PWI) of the first operator of the BUZZ intrinsic code, respectively. Note that the low-order bit of the PWI field has been truncated to allow space for the lock bit.
Lock bit	This bit is a constant 1 and indicates the state of the hard lock. The location and use of this bit is identical to previous implementations.

Compiler Controls

READLOCKTIMEOUT

— READLOCKTIMEOUT —|

(Type: BOOLEAN, Default: FALSE)

If READLOCKTIMEOUT is TRUE, the MCP procedure READLOCKTIMEOUT is called whenever the BUZZ or BUZZ47 intrinsic returns the contents of the variable specified in the BUZZ intrinsic with the tested bit (bit 0 or bit 47, respectively) equal to 1. If the option is FALSE, the BUZZ intrinsic loops internally.

SEPCOMP

→ SEPCOMP [" <file name> "]|

(Type: BOOLEAN, Default: FALSE)

The SEPCOMP option allows you to use the separate compilation capability, thus reducing the amount of time required to compile. This option must be set before the first source statement. You can specify a file name, which is used to set the TITLE attribute of the HOST file. The <file name> cannot include embedded quotation marks (").

For more information about the use of this option, refer to Section 7, "Compiling NEWP Programs."

SEPCOMPMERGE

→ SEPCOMPMERGE [" <file name> "]|

(Type: BOOLEAN, Default: FALSE)

The SEPCOMPMERGE option allows you to use an extended form of the separate compilation capability. This option should be used when no symbol file exists that exactly matches the host file. You can specify a file name, which is used to set the TITLE attribute of the SOURCEP file. The <file name> cannot include embedded quotation marks (").

For more information about the use of this option and about the SOURCEP file, refer to Section 7, "Compiling NEWP Programs."

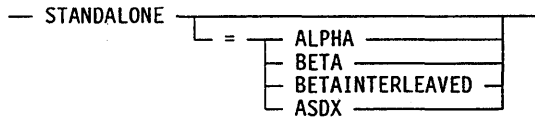
SINGLE

— SINGLE —|

(Type: BOOLEAN, Default: TRUE)

The <single option> causes the printout of a listing to be single-spaced. When the <single option> is FALSE, the printout is double-spaced.

STANDALONE



(Type: VALUE, Default: NONE)

STANDALONE is used to specify that a stand-alone memory image is to be generated when the program is compiled. An example of a program requiring a stand-alone memory image is SYSTEM/LOADER. To take effect, this option must be assigned a value prior to the beginning of the program, and the MCP compiler control option must be TRUE. When the MCP option is TRUE, the OUTERBLOCK segment descriptor is placed at segment (0,5) by default (that is, unless overridden by a SEGMENT block directive). If STANDALONE is set to BETA or BETAINTERLEAVED, the Actual Segment Descriptor (ASD) or Actual Segment Descriptor Extended (ASDX) should also be set. Otherwise, the resulting program cannot be executed.

The STANDALONE option causes the compiler to prepare a complete memory image of the program so that the program is ready to run when loaded into the system (starting at memory location zero). The maximum size of this memory image is 22,000 words.

The STANDALONE option implicitly resets the LINEINFO option. If the LINEINFO option is subsequently set, an additional code file with a suffix of */LISTCODE* is generated. This file can be used by the Editor in conjunction with its LOAD CODE, GO RCW, WHERE RCW, and LISTCODE commands.

When STANDALONE is used by itself and is not assigned a value, STANDALONE assumes the value BETA if the ASD option is set. If ASD is reset, STANDALONE assumes the value ALPHA. All other values must be specified explicitly.

When STANDALONE is specified to a setting of ALPHA, the compiler sets ASD equal to FALSE. For all other settings of STANDALONE the compiler sets ASD equal to TRUE. The compiler setting of ASD overrides the user's setting of ASD. This relationship avoids the nonexecutable code files being generated by incompatible settings for the two options.

Depending on the value that is assigned to STANDALONE, different memory images and code files are generated. The following items are common to all variants in the memory image:

- The D[0] stack image
- All code segments
- All value arrays
- All pool data items (for example, translate tables and truthsets)
- The storage space for all SAVE arrays declared at a D[0] location
- The proper entry PCW for the outer block at location (0,3)
- A memory descriptor at location (0,4)

The memory image does not include allocated data storage for any array declared within a procedure or any D[0] array not declared to be either VALUE or SAVE. No errors or warnings are given for these conditions, as it is assumed that you will provide a proper presence-bit handling routine. No memory links are provided to separate any of the allocated storage areas.

Depending on the value that is assigned to STANDALONE, the code file and memory image generated by the NEWP compiler differ substantially.

- If STANDALONE is set to ALPHA, the code file format is as follows:
 - A bootstrap is in code segment 0.
 - A valid segment 0 (SEG0) is in code segment 1.
 - The memory image starts in code segment 2 and continues to the end of the file. The SEG0[18] word (the segment dictionary pointer) properly describes the D[0] image. The SEG0[1].[23:4] (MCP/stand-alone type) field contains the number 0, indicating a non-ASD code file.
- If STANDALONE is set to BETA, the code file format is as follows:
 - A valid SEG0 is in code segment 0.
 - An ASD table starts in code segment 1. The table is formatted as contiguous vectors of ASD1, ASD2, and ASD3. Following the ASD table is the memory image.
 - The SEG0[1].[23:4] (MCP/stand-alone type) field contains a 1, indicating an ASD code file that uses a 3-word linear ASD table.
- If STANDALONE is set to BETAINTERLEAVED, the code file format is the same as that for BETA, except for the following:
 - The ASD or ASDX table starts in code segment 1. The table is formatted as interleaved vectors of ASD1, ASD2, ASD3, and ASD4, so that all ASD or ASDX table elements for a given entry are contiguous rather than all ASD or ASDX entries for a given element being contiguous.
 - The stand-alone stack contains not only all D[0] variables, but also a correctly formatted D1 frame with a Return Control Word (RCW) at the beginning of the outer block on top of the stack. The D1 frame contains a properly formatted and entered D1 Mark Stack Control Word (MSCW), a dummy RCW, and two words of zero before the RCW. The Top of Stack Control Word (TOSCW) at the base of the stack points at the RCW and the D1 MSCW.
 - The SEG0[1].[23:4] (MCP/stand-alone type) field contains a 2, indicating an ASD or ASDX code file that uses a 4-word interleaved ASD or ASDX table.
- If STANDALONE is set to ASDX, the code file format is the same as for the BETAINTERLEAVED option, except for the following:
 - All word formats found in the STANDALONE code file meet the MCP/AS (Extended) specifications.
 - The TOSCW and copy TOSCW in the base of the stand-alone stack contain an MCP/AS (Extended) TOSCW word with the number 1 (SNR) in the [46:15] field.

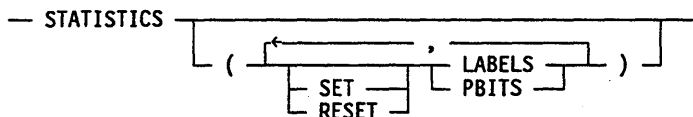
- The stand-alone stack contains not only all the D[0] variables but also a correctly formatted D1 frame and D2 frame. The D1 frame contains an entered D1 MSCW and a dummy RCW. The D2 frame contains an entered D2 MSCW, an RCW that points to the code stream to be executed, and an FCW on top of the stack.
- The SEG0[1].[23:4] (MCP/stand-alone type) field contains a 3, indicating a MCP/AS (Extended) code file that meets all requirements for the code file to run on an ASDX system.

The code file does not contain any SEPCOMP information, BINDINFO, or LINEINFO. For information on BINDINFO and LINEINFO, see the *A Series ALGOL Programming Reference Manual, Volume 1: Basic Implementation*.

If the STACK option is TRUE, the STANDALONE option prints a table of the code segment descriptors and data descriptors in the D[0] location, indicating which ones remain absent and showing the memory locations of the ones made present. Additionally, if STANDALONE is set to BETA, BETAINTERLEAVED, or ASDX, the ASD table is printed to indicate which ASDs are in use and to indicate their contents.

Note: STANDALONE = BETAINTERLEAVED must be used on the Unisys A 17 system.

STATISTICS



(Type: BOOLEAN, Default: FALSE)

When the MCP option is FALSE, the STATISTICS option functions exactly as it does in ALGOL, with the following exceptions:

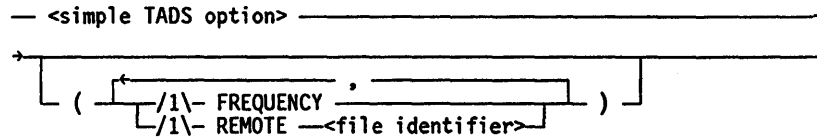
- When the STATISTICS option is TRUE, the NOBINDINFO option is assigned the value TRUE. A warning is given.
- When the STATISTICS option is TRUE, the MAKEHOST option is assigned the value FALSE. A warning is given.
- When the TADS option is TRUE, the STATISTICS option cannot be assigned the value TRUE. A warning is given.
- Statistics are not generated separately for in-line procedures, but are included with the normal statistics gathering for the procedures in which they are invoked.

When the MCP option and the STATISTICS option are TRUE, the code to accumulate MCP statistics is added to each procedure entry, exit, and invocation. The code consists of calls on procedures at fixed address couples; these procedures accumulate the statistical information and must be declared in the MCP. Calls on the statistics procedures are also emitted at the entry and exit of each block in which the block directive STATSUMMARY appears.

The LABELS and PBITS options do not apply to MCP statistics and are therefore ignored when the MCP control option is set.

The STATSUMMARY block directive applies to MCP gathering statistics only.

TADS



<simple TADS option>

— TADS —

(Type: BOOLEAN, Default: FALSE)

When the TADS option is set, special debugging code and tables are generated as part of the object code file. The use of the TADS option is incompatible with the use of certain other options, including MCP, SEPCOMP, and STATISTICS.

Caution

Use caution when you run the TADS option on a program that contains UNSAFE blocks. Avoid stepping through or executing breakpoints in unsafe CONTROLSTATE blocks, as this can cause the running program to lose CONTROLSTATE.

The semantics for this option are the same as for ALGOL, with the exception that the TADS option is supported only for ALGOL type constructs in NEWP.

UNDERLINE

— UNDERLINE —

(Type: BOOLEAN, Default: FALSE)

If UNDERLINE is TRUE, all procedure names in procedure declarations are underlined on the output listing. The setting of the UNDERLINE option is ignored if the LIST option is FALSE.

VERSION

— VERSION —<release number>— . —<cycle number>— . —————>
-><patch number>—————|

(Type: VALUE, Default: 00.000.000)

The <version option> enables you to specify a version number for a source program or to replace an existing version number. Each occurrence of the <version option> in the CARD file updates the value of the version number. A warning is issued if the new version number is lower than the old version number.

If the current value of the version number is 00.000.000, the version number is set to the value specified by the <version option>. Otherwise, the occurrence of the <version option> is updated with the current value of the version number.

The updated record then replaces the original SOURCE record in the NEWSOURCE file (if the <new option> is TRUE) and in the program listing (if the control option LISTDOLLAR is TRUE or if the compiler control record begins in column 2 or 3). A warning is issued if the current value of version number is lower than the value originally specified by the <version option>.

*Note: All occurrences of <version option> in INCLUDE files are ignored.
Unpredictable results occur if you attempt to specify a version number
that exceeds 99.999.9999.*

During a compilation, the current value of each component of the version number can be accessed through the arithmetic functions COMPILETIME(20), COMPILETIME(21), and COMPILETIME(22), which return the release, cycle, and patch numbers, respectively.

VOID

— VOID —|

(Type: BOOLEAN, Default: FALSE)

If the <void option> is TRUE, all input other than compiler control records from the SOURCE and CARD files is ignored by the compiler until the VOID option is disabled. The ignored input is neither listed nor included in the updated symbolic file, regardless of the values of the <list option> and the <new option>. Once the VOID option is set to TRUE, it can be recalled using RESET or POP in a compiler control record in either the SOURCE or CARD file.

XREF

— XREF —|

(Type: BOOLEAN, Default: FALSE)

Cross-reference generation is performed by the NEWP compiler itself and is controlled by two compiler options, XREF and XREFFILES. If XREF is TRUE, a cross-reference listing is produced (the listing is written to the file XREFLINE). If XREFFILES is TRUE, cross-reference files are generated. These options can be set or reset independently of each other.

Alternatives within modules are cross-referenced in the same manner as modules are cross-referenced. If a procedure has its FORWARD declaration within a module— but outside any alternatives— and its actual body is in each alternative, each alternative procedure body is cross-referenced as an alias of each of the other alternative procedure bodies.

Summary information, including the number of references and sort times, is printed if either XREF or both XREFFILES and TIME are set.

XREFFILES

— XREFFILES —|

(Type: BOOLEAN, Default: FALSE)

XREFFILES controls the generation of cross-reference information in disk file form. For more information, see “XREF” earlier in this section.

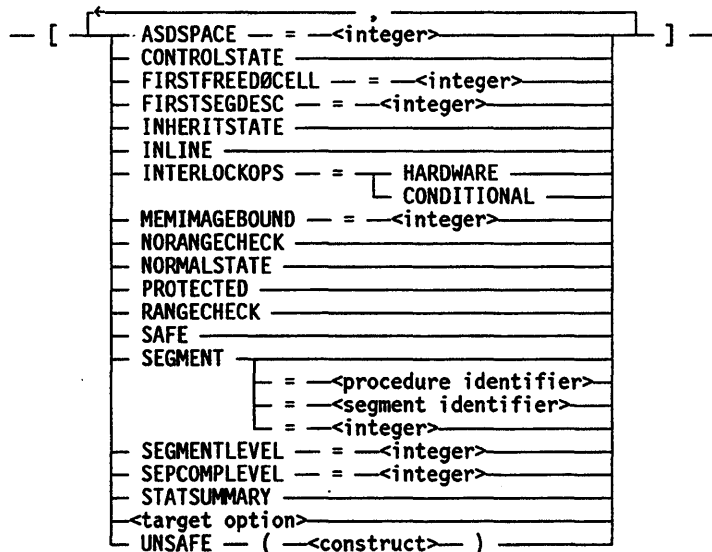
\$

— \$ —|

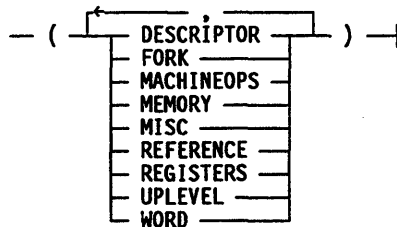
(Type: BOOLEAN, Default: FALSE)

If both the LIST option and the \$ option are TRUE, the printer listing includes all compiler control options. If the LIST option is TRUE but the \$ option is FALSE, only compiler control records with an initial dollar sign (\$) in any column from 2 through 72 appear in the printer listing. If the LIST option is FALSE, the value of the \$ option is ignored. The \$ option can appear only in columns 4 through 72, inclusive. The control option LISTDOLLAR is the preferred synonym for the \$ control option.

Block Directives



<construct>



Explanation

Through the use of block directives, you can control segmentation, the use of potentially dangerous constructs, and other compilation concerns. Block directives can occur inside brackets immediately after any BEGIN statement. In the following discussion, the word *block* is used for the concept of block or procedure.

In general, block directives are inherited by nested blocks unless overridden by block directives appearing in the nested blocks. However, CONTROLSTATE is inherited only by blocks, not by procedures.

Example

```
BEGIN [SEGMENT=OUTERBLOCK, SEPCOMPLEVEL=5]
...
PROCEDURE P(X);
  VALUE X; REAL X;
  BEGIN
    [UNSAFE(MEMORY,WORD), SEGMENT=5, CONTROLSTATE]
    ...
  END P;
...
END.
```

The following keywords are recognized as directives to the compiler.

ASDSPACE

The <integer> specifies the number of ASDs to be allocated in addition to the preallocated data and code segments and reserved ASD slots. Additional ASDs are necessary for any program that contains arrays. The default number of additional ASDs allocated is 0 (zero).

CONTROLSTATE

This directive can be used when the block must be run in control state. CONTROLSTATE is inherited by all nested nonprocedure blocks, except those explicitly specifying NORMALSTATE. It is not inherited by procedures, including in-line procedures.

Explicit use of the CONTROLSTATE block directive causes the block to be a new environment, resulting in the usual limitations: a GO TO statement from outside the block cannot branch to a label within the block, and a GO TO statement to a label outside the block is considered a bad GO TO. All bad GO TO statements outside of a CONTROLSTATE block cause a loss of control, even when the statements are branching to another CONTROLSTATE block.

If the CONTROLSTATE block directive is used, the code file is marked as nonexecutable and unsafe.

FIRSTFREEDOCCELL

FIRSTFREEDOCCELL specifies the first location (displacement) at lexical level 0 that can be allocated freely by the compiler. Locations below the FIRSTFREEDOCCELL are reserved and can be allocated by means of an address equation.

The default value of FIRSTFREEDOCCELL is 10 (decimal), and it cannot be assigned a value less than 10. FIRSTFREEDOCCELL is valid only when the MCP compiler control option is TRUE and only in the outer block of the program.

FIRSTSEGDESC

FIRSTSEGDESC is valid only for the outer block of a program being compiled with the MCP compiler control option set to TRUE. FIRSTSEGDESC accepts a constant integer value. The <integer> is used as the D[0] location where allocation for segment descriptors is to begin (similar to the ALGOL SEGDESCABOVE compiler control option).

When this directive is set, segment descriptors (except those declared by an explicit SEGMENT declaration) are placed at D[0] locations, starting at FIRSTSEGDESC and proceeding to higher addresses. All other items placed in the D[0] stack are allocated as usual, starting at FIRSTFREED0CELL.

If, in the course of compilation under this block directive, the allocation point for normal items (not segment descriptors) reaches the beginning of the segment descriptors, a warning is issued and all further D[0] allocation occurs above the area currently occupied by segment descriptors.

The MAKEHOST/SEPCOMP facilities in NEWP preserve the allocation points from the host and continue allocation in the same way. The BINDER adds only segment descriptors to the D[0] location and always adds them to the end, or top. The BINDER also preserves the MAKEHOST information for the SEPCOMP of a bound host.

FIRSTSEGDESC is designed to relieve crowding in the addressable portion of the D[0] stack, to allow adding new variables to the D[0] stack during a SEPCOMP, and to encourage breaking up of large MCP code segments to improve memory utilization.

This block directive allows items that must be addressed from lexical levels higher than 1 to be given stack addresses that are visible from those higher lexical levels, even if the items are added by a separate compilation. The value given for FIRSTSEGDESC should be high enough to allow some spare D[0] cells below the segment descriptors. No benefit results from using a value higher than 4096.

INHERITSTATE

INHERITSTATE is allowed only for in-line procedures. It specifies that each invocation of the in-line procedure should run in the CONTROLSTATE/NORMALSTATE environment of the invoking code.

INLINE

INLINE specifies that the procedure is to be compiled as an in-line procedure. For more information, see "In-Line Procedures" in Section 4, "Declarations."

INTERLOCKOPS

This block directive can be used to specify that when the <lock function> or <unlock function> is used, only hardware locking code is to be generated, or that conditional locking code is needed. The INTERLOCKOPS = HARDWARE option takes effect

only when the TARGET is LEVEL1. If TARGET is not LEVEL1, the block directive is ignored.

The HARDWARE option should be used only when the code file is to be run on a machine that supports the LOK and UNLK operators.

The INTERLOCKOPS block directive is inherited, and conditional locking code is generated by default. The CONDITIONAL option is required only when the HARDWARE option is set and conditional code is needed for a block that would normally inherit the HARDWARE option.

The actual locking code generated depends on both the use of this block directive and the use of the INTERLOCKOPS control option. The most local setting of INTERLOCKOPS has precedence, regardless of whether the block directive or control option is used.

MEMIMAGEBOUND

MEMIMAGEBOUND specifies the maximum size allowed for the memory image of a stand-alone program. If the size of the memory image exceeds the size specified by the block directive, a syntax error is given. This block directive is allowed only for the outer block and can be used only if the compiler control option STANDALONE is set. If the MEMIMAGEBOUND block directive is not used, the maximum size allowed for a memory image is 8192 words.

NORANGECHECK

When this directive is used, dynamic range checking for scalar types and sets, which is done by default, is disabled in the block. Code is not emitted to check ranges at run time.

Dynamic range checking can be enabled for a block nested within this block through the use of the block directive RANGECHECK.

When the NORANGECHECK block directive is set, range checking code is not emitted for the DIGITS phrase of the REPLACE statement. You should verify that the < arithmetic expression > that provides the number of digits is in the range 0 through 24. In addition, no dynamic range checking is done for user-defined scalar types and sets.

NORMALSTATE

The block is run in normal state by default, unless nested in other blocks for which the CONTROLSTATE block directive is specified. Explicit use of the NORMALSTATE block directive causes the block to be a new environment, resulting in the usual limitations: a GO TO statement from outside the block cannot branch to a label within the block, and a GO TO statement to a label outside of the block is considered a bad GO TO.

PROTECTED

PROTECTED is valid only for an **EXCEPTION PROCEDURE** and is not inherited. This block directive indicates that the **EXCEPTION PROCEDURE** is protected from an entire class of interruptions, when it is invoked automatically by the system.

Interruptions from which the **EXCEPTION PROCEDURE** is protected include software interrupts, a **DS** system command, a **STOP** command, and stack stretching. You should be particularly aware of the system's inability to stretch the stack during execution of a protected **EXCEPTION PROCEDURE**. Use of this block directive makes the code file unsafe and nonexecutable.

For more information about **EXCEPTION PROCEDURES** and the way in which they can be invoked, see "EXCEPTION PROCEDURE Declaration" in Section 4, "Declarations."

RANGECHECK

This block directive is used to enable dynamic range checking for scalar types and sets. Code is emitted to check ranges at run time.

Dynamic range checking is done by default for scalar types and sets. The block directive **RANGECHECK** is necessary only to enable range checking within a block for which range checking has been disabled by the block directive **NORANGECHECK**.

SAFE

The compiler generates syntax errors for all **UNSAFE** constructs used in this block, except those enabled by subsequent **UNSAFE** specifications. Such syntax errors are generated by default, and this directive is necessary only to enable the unsafe feature checking within a block for which the checks have been disabled by the block directive **UNSAFE**.

SEGMENT

Segmentation information for the block can be specified by one of the following phrases:

- **SEGMENT**
Makes a new segment.
- **SEGMENT** < procedure identifier >
Adds the code for this block to the segment that contains the specified procedure. The < procedure identifier > must have appeared previously in a **PROCEDURE** declaration or a < forward procedure declaration >.
- **SEGMENT** < segment identifier >
Adds the code for this block to the segment associated with this segment name. The < segment identifier > must have appeared previously in a **SEGMENT** declaration.

- `SEGMENT <integer>`
Adds to the segment number `<integer>`.

SEGMENTLEVEL

The `SEGMENTLEVEL` block directive specifies the highest lexical level at which procedure declarations cause the compiler to generate a separate code segment. At lexical levels higher than the specified `<integer>`, procedure declarations are compiled into the code segment of their containing block or procedure (unless overridden by a `SEGMENT` block directive). For example, `SEGMENTLEVEL = 2` causes all procedures declared at or below lexical level 2 to be put in separate code segments; all procedures declared above level 2 are, by default, compiled into the same segment as the block or procedure in which they are declared. The default `SEGMENTLEVEL` is 15.

SEPCOMPLEVEL

The `SEPCOMPLEVEL` block directive is valid only when `MAKEHOST` or `SEPCOMP` is also set. This block directive specifies the highest lexical level at which information is collected for use during a `SEPCOMP`. When the `MCP` option is reset, the default `SEPCOMPLEVEL` is 2. When the `MCP` option is set, the default is 0 (zero).

For more information about this block directive, refer to the Section 7, "Compiling NEWP Programs."

STATSUMMARY

The `STATSUMMARY` block directive causes the compiler to emit calls on the statistics procedures at the entry and exit of the block in which the directive appears. For more information, refer to "STATISTICS" earlier in this section.

Compound statements that specify `STATSUMMARY` are treated as blocks, or new environments.

<target option>

This compiler control option can also be used as a block directive. The `<target option>` specifies the computer family for which the generated code is to be optimized. There is no code generated to verify that the machine on which the code is running is compatible with the code generated.

Note: Use of this construct is considered safe as long as none of the families is incompatible with the compilation option. See the A Series ALGOL Programming Reference Manual, Volume 1: Basic Implementation for a description of the compiler control option. If this use is not considered safe, then `UNSAFE(MACHINEOPS)` must be in effect. Without the `UNSAFE` declaration, an error is issued.

UNSAFE

UNSAFE specifies that certain potentially dangerous constructs can be used in the block. UNSAFE is inherited by nested blocks and procedures. The following keywords are used to specify the particular type of unsafe constructs permitted.

Table 8-1. Unsafe Constructs Permitted

Keyword	Unsafe Construct
DESCRIPTOR	Allows the DESCRIPTOR data type, DESCRIPTOR expressions, and DESCRIPTOR type transfer function.
FORK	Allows FORK statements and PROCESS statements.
MACHINEOPS	Allows the FLOAT option in the REPLACE statement and the use of the intrinsics requiring MACHINEOPS, which are listed in "Intrinsics (UNSAFE)" in Section 9, "UNSAFE Mode."
MEMORY	Allows the MEMORY array intrinsic.
MISC	Allows address equations, NULL as a <procedure body>, <option list>s on WAIT and WAITANDRESET statements, the OVERWRITE option on the REPLACE statement, SAVE ARRAY declarations, extensions to the SETACTUALNAME function, and the following intrinsics: BUZZ, BUZZ47, DAWDLE, LEXLEVEL, and LEXOFFSET.
REFERENCE	Allows the REFERENCE TO intrinsic, VIA intrinsic, and AT intrinsic.
REGISTERS	Allows the REGISTERS intrinsic and the DLL intrinsic.
UPLEVEL	Allows up-level pointer assignments and up-level procedure reference assignments.
WORD	Allows the WORD data type, WORD type transfer function, POINTER(<word expression>), and TAG.

If you use the = *<identifier>* form of address equation, the item being declared is assigned the same address as the identifier following the equal sign. The identifier must have been declared previously and must have an associated stack address. If the identifier is followed by the + *<unsigned integer>* syntax, the item being declared is assigned to the address at the same lexical level as the specified identifier. However, the declared item has a displacement that is *<unsigned integer>* words higher in the stack; the unsigned integer must be in the range 1 through 15.

Note: *If the identifier that specifies the address is a call-by-reference parameter, the identifier being declared is assigned the address of the formal parameter, not the actual parameter.*

You can use the *<address couple>* form of address equation to specify directly the lexical level and displacement to be assigned to the item being declared. If you specify the *<lexical level>* as a simple unsigned integer, the item is assigned an address at that lexical level; the unsigned integer specified must be less than or equal to the current lexical level. If you specify the lexical level with an asterisk (*), the item is assigned an address at the current lexical level. The *<displacement>* determines the offset of the item within the specified lexical level.

For non-MCP programs, the INTRINSIC form of address equation allows direct access to a stack cell containing an intrinsic descriptor for the appropriate intrinsic. If the intrinsic has already been referred to, the existing address is used. Otherwise, a new cell is allocated and initialized with the appropriate intrinsic descriptor.

Note: *Most uses of the EVENT declaration, other than those allowed by ALGOL, might be incompatible with a change to the format of events. For example, neither of the following is acceptable:*

- *The use of a variable declared by address equation to an EVENT or an EVENT ARRAY, for example:*

```
EVENT E; DOUBLE D = E; D:=0;
```

- *The use of an EVENT or EVENT ARRAY declared by address equation. For example:*

```
DOUBLE D; EVENT E = D; CAUSE(E);
```

For more information on EVENT declaration, see the *A Series ALGOL Programming Reference Manual, Volume 1: Basic Implementation*.

Examples

```
BOOLEAN BCOUNTER = (3,92);  
ARRAY COUNTS = BCOUNTS [0];  
INTEGER PROCEDURE COUNTONE = (*,15) (P1,P2);
```

DESCRIPTOR Declaration

<descriptor declaration>

```

— DESCRIPTOR —<descriptor identifier> —<equation part>

```

<descriptor identifier>

```

—<identifier>—|

```

Explanation

Variables of type **DESCRIPTOR** are used to store original or copy descriptors. Simple variables of type **DESCRIPTOR** are declared in a **DESCRIPTOR** declaration. Arrays, procedures, and formal parameters can also be specified as type **DESCRIPTOR**. All uses of type **DESCRIPTOR** require the block directive **UNSAFE(DESCRIPTOR)**.

Before being assigned a value, descriptor variables contain the uninitialized operand value: all zeros with a tag of 6. If a descriptor variable is referenced while it is uninitialized, an **INVALID OPERAND** fault occurs. Descriptor variables are accessed with the **LOAD** (as opposed to the **LODT**) operator. When a descriptor is evaluated, copy-bit action occurs if the target is a data descriptor.

The type transfer function **DESCRIPTOR** can be applied to both **WORD** variables and arrays. Implicit type transfers, or coercions, allow array references to be assigned from **DESCRIPTOR** values and **DESCRIPTOR** data types to be assigned from array references or arrays. The same coercions are applied between formal and actual parameters. **WORDS** and **DESCRIPTORS** are mutually coerced.

An array or **DESCRIPTOR** can be passed by-value to a formal parameter of type **DESCRIPTOR**. In either case, a copy descriptor is passed, and the original descriptor cannot be modified through the formal parameter.

If an array or **DESCRIPTOR** is passed by-reference to a formal parameter of type **DESCRIPTOR**, a reference to the one-word descriptor for the array (that is, a Stuffed Indirect Reference Word (SIRW) or indexed data descriptor to the descriptor) is passed. In this case, the descriptor itself is the object and can be modified directly.

If an array is passed by-reference to a formal array parameter, the compiler ensures that the formal and actual parameters have the same number of dimensions and then passes a copy descriptor.

For information on descriptor declarations, see “**WORD Declaration**,” “**DESCRIPTOR Expressions**,” and “**Intrinsics (UNSAFE)**” later in this section.

PROCEDURE Declaration

```

<procedure declaration>
    PROCEDURE <procedure heading> ;
    <procedure type>
    <procedure body>
    NULL
  
```

Explanation

In UNSAFE(MISC) mode, a <procedure body> can be specified as NULL. The procedure being declared must be address-equated. A null procedure declaration defines a calling sequence, but has no associated code. Before calling such a procedure, a Program Control Word (PCW) or a reference to a PCW must be provided at the address-equated location. For information on procedure declarations, see "Address Equation" earlier in this section.

SAVE ARRAY Declaration

```

<save array declaration>
    SAVE <array declaration>
  
```

Explanation

In UNSAFE(MISC) mode, arrays can be declared SAVE, which causes the compiler to mark the arrays so that they cannot be overlaid.

SEGMENT Declaration

```

<segment declaration>
    SEGMENT
    <segment identifier> ,
    <segment equate>
    <segment MPCWSDI>
  
```

```

<segment equate>
    = <D0 address couple>
  
```

```

<D0 address couple>
    ( - 0 - , <D0 displacement> )
  
```

```

<D0 displacement>
    <unsigned integer>
  
```

```

<segment MPCWSDI>
    [ - MPCWSDI = <address couple> ]
  
```

Explanation

In UNSAFE mode, segment identifiers can be used in address equations. For more information, refer to “Address Equation” earlier in this section. In addition, a <segment identifier> can be address-equated itself, but only to a literal <address couple> with lexical level 0 and displacement less than the FIRSTFREED0CELL.

The <segment MPCWSDI> construct can be used to specify the Segment Dictionary Index (SDI) value used for Make Program Control Word (MPCW) operators that are generated during stack building code for the <segment identifier>.

A <segment identifier> can be used as the parameter to the LEXOFFSET intrinsic and as the <address primary> in the <type> AT<address primary> construct. For more information, see “Intrinsics (UNSAFE)” later in this section and “SEGMENT Declaration” in Section 4, “Declarations.”

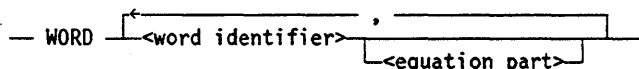
Examples

```

SEGMENT SEG1 = (0,1)    % legal
      SEG2 = (0,1000), % syntax error if 1000 is
                      % above the FIRSTFREED0CELL
      SEGB = (1,2),    % syntax error because it's
                      % not at lex level 0
      SEGC = REALID;  % syntax error because it's
                      % relative, not absolute,
                      % address equation

SEGMENT GETITGOING_SEG [MPCWSDI =(0,1)];
PROCEDURE GETITGOING;
BEGIN [CONTROLSTATE, UNSAFE (MEMORY), SEGMENT=GETITGOING_SEG]
END GETITGOING;
WORD   WSEG1 = SEG1;  % legal
REALID := LEXOFFSET(SEG1);
WORDID := WORD AT SEG1;

```

WORD Declaration**Explanation**

Type WORD is used for transparent examination and manipulation of arbitrary bit patterns, including machine control words. A value of type WORD is treated as an entity with 48 bits of data plus tag information, with no additional type significance. When values of type WORD are evaluated, the exact bit pattern in memory is returned. No copy-bit action occurs if the word accessed is a descriptor, and the second word is undefined if the word is part of a double-precision operand.

Simple variables of type WORD are declared in a WORD declaration. Arrays, procedures, and formal parameters can also be specified as type WORD. All uses of type WORD require the block directive UNSAFE(WORD).

UNSAFE Mode

Variables of type WORD are stored with overwrite operators and are accessed with the LODT operator. For information related to WORD declaration, see "WORD Expressions" later in this section.

Note: Most uses of the EVENT declaration, other than those allowed by ALGOL, might be incompatible with a change to the format of events. User programs should avoid the implicit coercion of an EVENT or EVENT ARRAY element to a WORD, whether by direct assignment or as an actual parameter passed to a formal parameter of type WORD.

Statements (UNSAFE)

The FORK statement is available in UNSAFE mode. In addition, extensions to the PROCESS, REPLACE, and WAIT statements are allowed.

FORK Statement

```
<fork statement>
  — FORK —<procedure invocation statement>— [ _____>
    →<arithmetic expression>— , —<arithmetic expression>— , _____>
    →<arithmetic expression>— [ _____>
      [ , —<pointer expression> ] ] _____>
```

Explanation

The FORK statement is used in the MCP to initiate independent runners. The <procedure invocation statement> specifies the procedure to be initiated and its actual parameter list.

The parameters in brackets are passed to the MCP procedure FORKHANDLER, which interprets these parameters as the box number of the box in which the task is to run, the size of the stack in which the task is to run, the priority of the task, and if the last parameter is present, the name to be displayed in the system mix entry for the task. The name defaults to the <procedure identifier> if the last parameter is omitted. Note that on an ASD system the box number is ignored.

The FORK statement is allowed only in UNSAFE(FORK) mode, and when the MCP dollar option is set and the STANDALONE option is not.

PROCESS Statement

The PROCESS statement is implemented with the same syntax as in ALGOL. You cannot invoke in-line procedures with a PROCESS statement.

The use of this construct requires the UNSAFE(FORK) block directive. In addition, unlike the ALGOL implementation, you must establish and maintain the necessary critical block linkages between the parent and any processed tasks.

REPLACE Statement

OVERWRITE Option

```

<unit count>
  -- FOR --<arithmetic expression>
    [ WORDS
    [ OVERWRITE ] ]

```

Explanation

In NEWP, the OVERWRITE option of the REPLACE statement is allowed in UNSAFE(MISC) mode. OVERWRITE appears in place of the WORDS specification in the syntax for <unit count>.

The default unit size for the <replace statement> is expressed in characters. WORDS and OVERWRITE both indicate units of one word. OVERWRITE also overrides memory protection (odd tags) on both the source and the destination and suppresses memory parity errors on any prior contents of the destination.

FLOAT Option

```

<transfer part>
  -- FOR --<length> WITH --<edit micros>
    [
      <length>
        --<arithmetic expression>
      <edit micros>
        [ INSERT -- ( --<insert character> ) ]
        [ FLOAT --<FLOAT parameters> ]
      <FLOAT parameters>
        -- ( --<insert character> , --<float character> ) --
      <float character>
        --<EBCDIC constant>
    ]

```

Explanation

In UNSAFE(MACHINEOPS) mode, the FLOAT option is allowed as an extension to the <transfer part> of the REPLACE statement. In contrast, INSERT does not require UNSAFE mode. The INSERT option is described in "REPLACE Statement" in Section 5, "Statements."

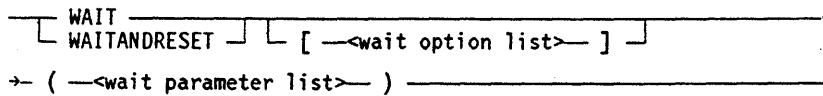
Like INSERT, the FLOAT option replaces the leading zero character. In addition, <float character> is inserted in the destination string before the first nonzero character transferred from the source. If the source string consists of the number of zeros defined in the <length> construct, only that number of insert characters is transferred to the destination string. No characters are transferred if <length> is zero or less.

UNSAFE Mode

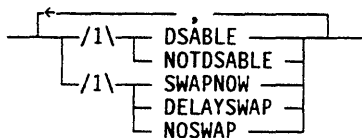
The source and destination pointer expressions must denote 8-bit characters. Furthermore, the resulting destination string is undefined if the source string contains characters other than the EBCDIC digits 0 through 9.

WAIT and WAITANDRESET Statements

<wait statement>



<wait option list>



Explanation

In UNSAFE(MISC) mode, the WAIT and WAITANDRESET statements can include an <option list> to specify whether the waiting process can be terminated by a DS system command or be swapped out while waiting. The following options can be included in the option list:

- **DSABLE:** This option causes the process not to wait if it is already terminated by a DS system command and not to continue to wait if it is externally terminated by a DS system command while waiting. The value returned by the WAIT function (if it occurs in an arithmetic expression) is zero in either case.
- **NOTDSABLE:** This option causes the process to wait even if it is or becomes terminated by a DS system command.
- **DELAYSWAP:** This option causes the process to be swapped out if it waits longer than an interval defined by the MCP.
- **SWAPNOW:** This option causes the process to be swapped out as soon as the waiting starts.
- **NOSWAP:** This option forces the process not to be swapped out while waiting.

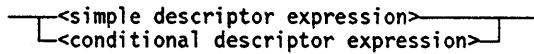
If an <option list> is not given, default values are assigned according to the type of process. System processes (D[0] and pseudo D[0] relative code) default to NOTDSABLE and NOSWAP. User processes default to DELAYSWAP and DSABLE if the processes are not already terminated by a DS system command. If the process has already been terminated by a DS command, it must wait.

Expressions and Functions (UNSAFE)

The expressions and functions described in this section are allowed in UNSAFE mode.

DESCRIPTOR Expressions

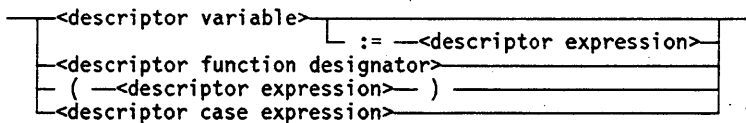
<descriptor expression>



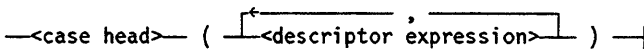
<simple descriptor expression>



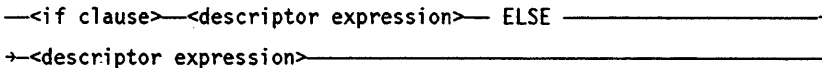
<descriptor primary>



<descriptor case expression>



<conditional descriptor expression>



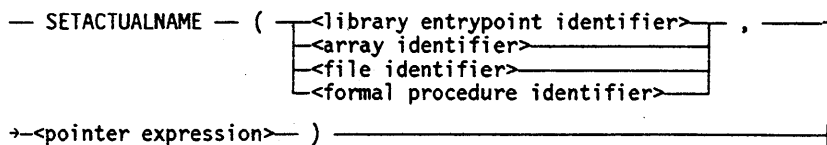
Explanation

DESCRIPTOR expressions generate values of type DESCRIPTOR. UNSAFE intrinsic functions that return values of type DESCRIPTOR include *DESCRIPTOR AT* <address primary>, the DESCRIPTOR type transfer function, and *DESCRIPTOR VIA* <word primary>. Descriptor expressions are valid only in UNSAFE(DESCRIPTOR) mode.

For information related to DESCRIPTOR expressions, see “DESCRIPTOR Declaration” and “Intrinsics (UNSAFE)” in this section.

SETACTUALNAME Function

<setactualname function>

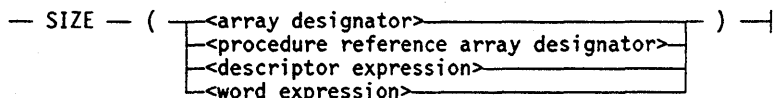


Explanation

In UNSAFE(MISC), a <file identifier>, an <array identifier>, or a procedure identifier that is a formal parameter can be used as an alternative to the <library entrypoint identifier>. However, you must ensure that the file, array, or formal procedure is a library object.

SIZE Function

<size function>

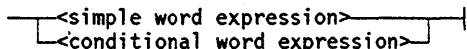


Explanation

In UNSAFE (WORD) and UNSAFE (DESCRIPTOR) a <word expression> and a <descriptor expression> can be used with the <size function>. The descriptor expression or word expression must evaluate to an unindexed, touched data descriptor. If the expression does not, a run-time error can result.

WORD Expressions

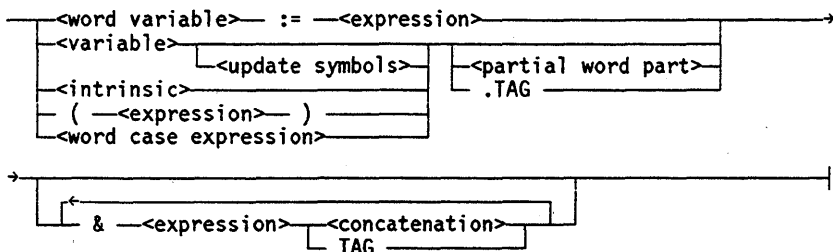
<word expression>



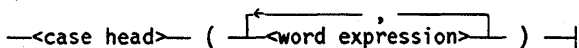
<simple word expression>



<word primary>



<word case expression>



<conditional word expression>



Explanation

WORD expressions are extremely general because almost any type is implicitly type-transferred, or coerced, to type WORD when a WORD expression is expected. While no operators are defined for type WORD, an expression of any other type, including types with defined operators (such as addition in arithmetic expressions), can be assigned to a word variable.

The .TAG clause can be used in place of the <partial word part> or in place of the <concatenation>, and refers to the tag bits of the operand.

Although the result of any intrinsic can be assigned to a WORD variable, the following UNSAFE intrinsics specifically return values of type WORD:

- WORD(<expression>)
- WORD VIA <word primary>
- REFERENCE TO <primary>
- EVAL
- MAKEPCW
- MEMORY
- WORD AT <address primary>

Word expressions are valid only in UNSAFE(WORD) mode.

For information related to WORD expressions, see “WORD Declaration” and “Intrinsics (UNSAFE)” in this section.

Note: Most uses of the EVENT declaration, other than those allowed by ALGOL, might be incompatible with a change to the format of events. User programs should avoid the implicit coercion of an EVENT or EVENT ARRAY element to a WORD, whether by direct assignment or as an actual parameter passed to a formal parameter of type WORD.

Intrinsics (UNSAFE)

The intrinsics described in the following text are recognized only in UNSAFE mode; the UNSAFE category for each intrinsic is shown in brackets.

Caution

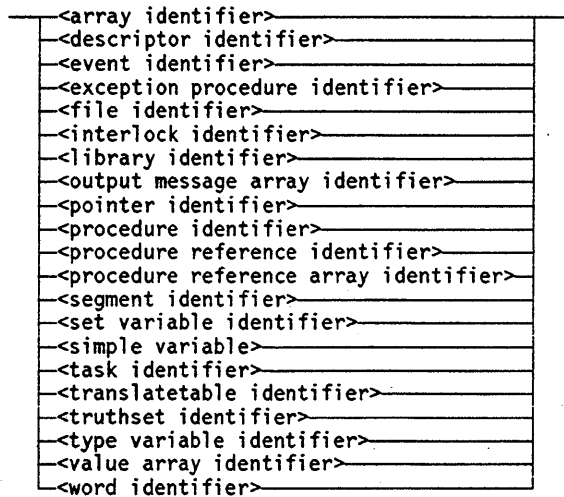
Intrinsics in the MACHINEOPS category are often available only on certain machine types. You should ensure that the operator generated by the intrinsic is supported on the machine on which the program is to run.

UNSAFE Mode

For further information on the use of machine operators, see the *A Series System Architecture Reference Manual, Volume 2*.

In UNSAFE mode, a broader range of data types and expressions is allowed than is defined in safe mode. The following diagrams describe syntactic categories that are used to define valid parameters to the UNSAFE intrinsics:

<addressable identifier>



ASDTABLE [MACHINEOPS]

— ASDTABLE — [—<integer expression>— , —————>
-><constant integer expression>—] —————|

Explanation

ASDTABLE is an intrinsic that accesses the ASD table using the RASD (read) and WASD (write) operators. The first parameter indicates the ASD index, which is found in the lower 20 bits of data and code descriptors. The second parameter is the ASD specifier, which must be a constant in the range 0 through 3 and must correspond to ASD1 through ASD4.

Use of the ASD table is restricted in the following ways:

- The first parameter must evaluate to an integer.
- The second parameter must be an integer constant in the range of 0 through 3. In addition, if the reference is to the left of an assignment, a negative zero is valid and can be used to ensure the integrity of the unaltered bit when writing ASD1.
- The ASD table cannot appear to the left of an embedded assignment.
- The ASD control option must be set.

Examples

```

W := ASDTABLE[K, 2];

ASDTABLE[K, -Ø] := W;

ASDTABLE[K, 3] := * & 1[47:1] & Ø[43:1];

ASDTABLE[K, 1].[47:1] := 1;

```

AT [REFERENCE]

—<type>— AT —<address primary>—|

<address primary>

—<addressable identifier>—|
—<subscripted variable>—|
—<array row>—|

<type>

—BOOLEAN—
—DESCRIPTOR—
—DOUBLE—
—EVENT—
—FILE—
—INTEGER—
—INTERLOCK—
—POINTER—
—REAL—
—TASK—
—<type identifier>—
—WORD—

Explanation

The <type> AT <address primary> syntax allows the item at the location specified by the <address primary> to be referred to or assigned to as if it had been declared of the specified <type>. If the AT syntax is used in an expression, the item is retrieved from the location in the manner appropriate for a value of the target <type>.

For example, if D is a variable of type DESCRIPTOR, the syntax WORD(D) causes D to be retrieved as a DESCRIPTOR, and copy-bit action is performed. The syntax WORD AT D, on the other hand, causes the item at D to be fetched as a WORD, and no copy-bit action is performed. The WORD AT <address primary> syntax is valid only in UNSAFE(WORD) mode. The DESCRIPTOR AT <address primary> syntax is valid only in UNSAFE(DESCRIPTOR) mode.

Note that if the specified <address primary> is a formal parameter, the <type> AT <address primary> syntax references the actual parameter, not the reference (SIRW or indexed data descriptor) that was passed.

Note: Except within the event-handling routines of the MCP, programs should avoid the use of the `EVENT AT <address primary>` or the `<type> AT <address primary>` syntax where the `<address primary>` has type `EVENT`, `EVENT ARRAY`, `REFERENCE`, or `REFERENCE ARRAY`.

Examples

```
WORD W;
DESCRIPTOR D;
W := WORD AT D;
```

BMASKSEARCH [MACHINEOPS]

```
— BMASKSEARCH — ( —<arithmetic expression>— , —————→
→<arithmetic expression>— , —<array row>—————→
                        —<subscripted variable>—
→<arithmetic expression>— ) —————→
```

Explanation

The integer function `BMASKSEARCH` is a bounded `MASKSEARCH` intrinsic that uses the `BMS` operator. This function is similar to `MASKSEARCH`, except that it requires a fourth argument to specify the word limit of the search. For more information on `MASKSEARCH`, see the *A Series ALGOL Programming Reference Manual, Volume 1: Basic Implementation*.

The fourth argument is the length, in words, to be searched within the domain specified by the third argument. The length must be less than or equal to $2^{20}-1$.

The result returned is the same as for `MASKSEARCH`. A failure result is returned under the following additional conditions:

- If the search length is initially zero or negative.
- If the search exhausts the search length.
- If the search has examined word zero of the virtual segment or word zero of memory.

BUZZ [MISC]

```
— BUZZ — ( —<variable>— ) —|
```

Explanation

`BUZZ` is a `REAL` procedure that disables external interrupts and then uses the `Read with Lock (RDLK)` operator to exchange continually the following:

- A value whose low-order bit (bit 0) is equal to 1
- The current contents of the location specified by `<variable>`

BUZZ continues exchanging until bit 0 of the value of the location is 0. This value is returned as the procedure value. BUZZ can be used only in CONTROLSTATE blocks.

The semantics of BUZZ are affected by the values of the READLOCK and READLOCKTIMEOUT compiler control options.

For information related to the BUZZ intrinsic, see “Compiler Control Options” in Section 8, “Compiler Controls.”

BUZZ47 [MISC]

BUZZ47 performs the same function as the BUZZ intrinsic, except that bit 47 of the parameter is tested and set instead of bit 0 (zero). BUZZ47 can be used only in CONTROLSTATE blocks.

CALLIO [MACHINEOPS]

— CALLIO — (—<array row>— —<descriptor expression>—) —

Explanation

CALLIO is an untyped procedure that places the parameter on the top of the stack and executes the Communicate with Universal I/O (CUIO) operator.

CHECKHASH [MACHINEOPS]

— CHECKHASH — (—<array row>— , —<length>—) —

Explanation

CHECKHASH generates the HASH operator. This procedure is of type REAL, with two parameters. The first parameter specifies an array row of type REAL. The second parameter is an integer.

The result is defined by the following relation:

$$\begin{aligned} \text{hash}[\emptyset] &= \emptyset \\ \text{hash}[i] &= (\text{hash}[i-1] \text{ EQV } \text{word}[1]).[46:48] \end{aligned}$$

Word[1] is the first word of the array row.

DAWDLE [MISC]

— DAWDLE — (—<integer constant>—) —

EVAL [MACHINEOPS]

```

— EVAL — ( —<array identifier>————— ) —|
              |—————<descriptor identifier>—————|
              |—————<event identifier>—————|
              |—————<exception procedure identifier>—————|
              |—————<file identifier>—————|
              |—————<interlock identifier>—————|
              |—————<output message array identifier>—————|
              |—————<pointer identifier>—————|
              |—————<procedure array identifier>—————|
              |—————<procedure identifier>—————|
              |—————<procedure reference array identifier>—————|
              |—————<procedure reference identifier>—————|
              |—————<set variable identifier>—————|
              |—————<simple variable>—————|
              |—————<task identifier>—————|
              |—————<translatable identifier>—————|
              |—————<truthset identifier>—————|
              |—————<type variable identifier>—————|
              |—————<value array identifier>—————|
              |—————<word identifier>—————|
              |—————MEMORY — [<expression>]—————|

```

Explanation

EVAL creates a reference to the location specified by the parameter and executes an Evaluate (EVAL) operator. EVAL is an intrinsic that returns a WORD data type containing the last reference in the chain (the reference that points to the final target).

You cannot use the EVAL operator with call-by-name parameters.

EXIT [MACHINEOPS]

```

— EXIT —|

```

Explanation

EXIT generates the EXIT machine operator. The MCP procedure BLOCKEXIT is not called when an EXIT operator is executed. The EXIT intrinsic cannot be used in the body of an in-line procedure.

FAILREGISTER [MACHINEOPS]

```

— FAILREGISTER — ( —<arithmetic expression>— ) —|

```

Explanation

FAILREGISTER returns a type REAL, which is the result of executing the Fetch Main Memory Fail Register (FMFR) operator. The parameter is the memory module number, which is changed to an integer prior to use.

FMMRREADLOCK [MACHINEOPS]

Explanation

FMMRREADLOCK performs the same function as the READLOCK option except that the compiler emits the Fetch Main Memory Reference (FMMR) operator prior to emitting the Read with Lock (RDLK) operator.

IGNOREPARITY [MACHINEOPS]

— IGNOREPARITY —|

Explanation

IGNOREPARITY generates the Ignore Parity (IGPR) operator.

INTERRUPTCHANNEL [MACHINEOPS]

— INTERRUPTCHANNEL — (—<arithmetic expression>—) —|

Explanation

INTERRUPTCHANNEL generates the Interrupt Channel (INCN) operator. The parameter is a mask indicating the channel that is to be interrupted. A parameter of 0 causes the STORE queue to be purged.

INTERRUPTCOUNTZERO [MACHINEOPS]

— INTERRUPTCOUNTZERO —|

Explanation

INTERRUPTCOUNTZERO generates the Zero Interrupt Count (ZIC) operator. The interrupt counter of the processor (which is used to detect interrupt loops) is reset to 0 (zero). If a Stack Overflow interrupt is pending, that interrupt is generated.

LEXLEVEL [MISC]

— LEXLEVEL — (

<addressable identifier>
<set variable identifier>

) —|

Explanation

LEXLEVEL is an integer-valued procedure that returns the lexical level of the specified <addressable identifier>. For example, if X is declared at location (1,9), then LEXLEVEL(X) returns the value 1. The lexical level is known to the compiler at compile time.

LEXOFFSET [MISC]

— LEXOFFSET — (—<addressable identifier>—) —
└─<set variable identifier>─┘

Explanation

LEXOFFSET returns an integer that is the offset (displacement) of the specified < addressable identifier > relative to the Mark Stack Control Word (MSCW). For example, if X is declared at location (1,9), then LEXOFFSET(X) returns the value 9. The offset is known by the compiler at compile time. The < addressable identifier > must not reference an in-line procedure.

You cannot use the LEXOFFSET operator with call-by-name parameters.

LISTLOOKUP [MACHINEOPS]

— LISTLOOKUP — (—<arithmetic expression>— , —<array row>—) —
→ , —<arithmetic expression>—) —

Explanation

The syntax and semantics of LISTLOOKUP in NEWP are similar to LISTLOOKUP in ALGOL except that the array must be present, the array must not be segmented, and the Linked List Lookup (LLU) operator is used instead of an MCP call.

LOADEVENT [MACHINEOPS]

— LOADEVENT — (—<event designator>—) —

Explanation

LOADEVENT is a word procedure that creates a reference to the < event identifier > and applies a Load Protected Object Word (LPOW) operator to it. If the < event identifier > is a formal parameter of a procedure, a LOAD operator is applied to the reference to < event identifier > before the LPOW is applied.

MAKEPCW [MACHINEOPS]

— MAKEPCW — (—<procedure identifier>—) —
└─<exception procedure identifier>─┘
└─<label identifier>─┘

Explanation

MAKEPCW returns a WORD value containing a hardware PCW that points to the code for the specified procedure or label. This PCW has the NCSF field set to 1 for control state procedures and for labels declared in control state environments. The operator MPCW is generated by this intrinsic.

If you specify a <procedure identifier> or an <exception procedure identifier>, the identifier must not reference a procedure declared EXTERNAL, NULL, LIBRARY, BY CALLING, or INLINE. In addition, the procedure must not be declared FORWARD at the time of the MAKEPCW invocation. The MAKEPCW invocation cannot occur within the body of the procedure being passed as the parameter to MAKEPCW.

If you specify a <label identifier>, the declaration of the label must not be more global than the beginning of the code segment in which the label is used as a MAKEPCW parameter. The generated PCW points to the actual label occurrence, rather than to any hidden label generated for optimizations of bad GO TO statements.

MEMORY [MEMORY]

Explanation

MEMORY can be used only when the MCP compiler control option is set. MEMORY is an intrinsic WORD array identifier referencing a one-dimensional array that maps all of memory. MEMORY can be used anywhere that a WORD array identifier is valid; however, NEWP does not allow a character-oriented pointer to be assigned to the memory array. For example, the syntax *POINTER(MEMORY[0],8)* is invalid. The default character size for pointers to the MEMORY array is 0 (word), not 8 (EBCDIC) as it is for other arrays; that is, the syntax *POINTER(MEMORY[I])* is synonymous with *POINTER(MEMORY[I],0)*.

When the compiler control option ASD is set, MEMORY access uses the Absolute Store Reference Word (ASRW) operator. This change builds the ASRW for stand-alone programs.

MOVESTACK [MACHINEOPS]

— MOVESTACK — (—<arithmetic expression>—) —

Explanation

MOVESTACK places the parameter on the top of the stack and performs the Move Stack (MVST) operator. The parameter is changed to an integer before use.

PAUSE [MACHINEOPS]

— PAUSE — [(—<expression1>— , —<expression2>—)] —

Explanation

PAUSE generates the IDLE machine operator. If the parameters are present, they are loaded onto the stack as the top and next-to-top items before the IDLE operator is executed, and these parameters are deleted afterward.

READTIMEOFDAY [MACHINEOPS]

— READTIMEOFDAY —|

Explanation

READTIMEOFDAY returns an INTEGER that contains the result of executing the Read Time of Day (RTOD) operator.

READXMEMORYTABLE [MACHINEOPS]

— READXMEMORYTABLE — (—<function definition>—) —|

Explanation

READXMEMORYTABLE returns a type REAL that contains the result of executing the Read External Memory Table (REMT) operator. The <function definition> must be of type REAL.

The <function definition> has the following layout:

[31:12]	Environment Number
[19:03]	Virtual Page Number
[03:04]	MSM mask

RECEIVEFROMREQUESTOR [MACHINEOPS]

— RECEIVEFROMREQUESTOR — (—<real>—) —|

Explanation

RECEIVEFROMREQUESTOR takes a type REAL parameter, which is the function definition, and returns a type DOUBLE. RECEIVEFROMREQUESTOR loads the parameter value onto the top of the stack and executes the Receive (RECV) operator.

REFERENCE TO [REFERENCE and WORD]

— REFERENCE — TO —<primary>—|

Explanation

The REFERENCE TO function returns a WORD that is equivalent to the value that would be generated to access the <primary> construct if the primary were a call-by-reference parameter. For example, valid WORDs include an SIRW for simple data type, a data descriptor for array rows, and an indexed data descriptor for subscripted variables. The <primary> construct can be of any data type that NEWP allows to be passed as a call-by-reference parameter.

Note: *If the specified <primary> is a call-by-reference parameter, the REFERENCE TO <primary> syntax returns a reference to the actual parameter, not the formal parameter. For example, if X is the actual parameter passed by reference to the formal parameter R, then REFERENCE TO R returns a reference to X.*

Most uses of the EVENT declaration, other than those allowed by ALGOL, might be incompatible with a change to the format of events. Except within the event handling routines of the MCP, programs should avoid the use of REFERENCE TO <address primary> syntax where the <address primary> has type EVENT, EVENT ARRAY, REFERENCE, REFERENCE ARRAY, or INTERLOCKS.

Example

```
WORD W;
REAL A;
W:=REFERENCE TO A;
```

REGISTERS [REGISTERS]

```
— REGISTERS — [ —<constant integer expression>— ] —|
```

Explanation

The REGISTERS intrinsic returns the contents of the processor register specified by the <constant integer expression>.

RETURN [MACHINEOPS]

```
— RETURN — ( —<expression>— ) —|
```

Explanation

RETURN generates the RETN machine operator, leaving the value of the <expression> on the top of the stack. The MCP procedure BLOCKEXIT is not called when a RETURN is performed. The RETURN intrinsic cannot be used in the body of an in-line procedure.

RETURNORIGINALS [MACHINEOPS]

```
— RETURNORIGINALS — ( —<double_the_cache_size>— ) —|
```

```
<double_the_cache_size>
```

```
— <numeric constant> —|
```


Explanation

RETURNORIGINALS generates code that forces the processor to contain only copies of data in its cache and no originals.

RUNNINGLIGHT [MACHINEOPS]

— RUNNINGLIGHT —|

Explanation

RUNNINGLIGHT generates the Running Light (RUNI) operator.

SCALERIGHTS [MACHINEOPS]

— SCALERIGHTS — (—<arithmetic expression>— , —————→
→<arithmetic expression>— , —<arithmetic variable>—) —————|

Explanation

The SCALERIGHTS function is an integer intrinsic that takes three parameters and generates the Scale Right Save (SCRS) operator and the Dynamic Scale Right Save (DSRS) operator. The first parameter is a single- or double-precision value to be scaled (V). The second parameter is the scale-factor (SF), which must be in the range 0 through 12. The third parameter is an output parameter. The function returns the following result as the integer procedure value:

V DIV 10**SF

The remainder of $V \text{ DIV } 10^{**SF}$ is returned in packed decimal form, left-justified, in the third (output) parameter.

SCANIN [MACHINEOPS]

— SCANIN — (—<arithmetic expression>—) —|

Explanation

SCANIN is an intrinsic of type REAL that causes the parameter to be placed on the top of the stack and causes the Scan In (SCNI) operator to be performed. The range of valid parameter values is dependent on the system type.

SCANOUT [MACHINEOPS]

— SCANOUT — (—<expression>— , —<expression>—) —|

Explanation

SCANOUT causes the two parameters to be placed on the top of the stack and causes the Scan Out (SCNO) operator to be performed. The range of valid parameter values is dependent on the system type.

SENDTOREQUESTOR [MACHINEOPS]

— SENDTOREQUESTOR — (—<function definition>— , —<data>—) —|

Explanation

SENDTOREQUESTOR is an untyped intrinsic that applies the Send to External Processing Element (SEND) operator to the two parameters. Both parameters are of type REAL.

The < function definition > has the following layout:

[43:08]	Function code
[07:08]	Source or destination routing
[07:04]	Destination requestor number
[03:04]	Address of link or MSM (B 7900 machines)

SETINHIBIT [MACHINEOPS]

— SETINHIBIT — (—<arithmetic expression>— , —————→
→<arithmetic expression>—) —————|

Explanation

SETINHIBIT generates the Set Memory Inhibits (SINH) operator where the low-order 8 bits of the first parameter contain the inhibit mask, and the low-order 4 bits of the second parameter represent the memory module. The second parameter is changed to an integer before it is used.

SETLIMITS [MACHINEOPS]

— SETLIMITS — (—<arithmetic expression>— , —————→
→<arithmetic expression>—) —————|

Explanation

SETLIMITS generates the Set Memory Limits (SLMT) operator. The first parameter contains the availability mask and the upper and lower addresses. The second parameter is the memory module number; this parameter is changed to an integer before use.

SETTIMEOFDAY [MACHINEOPS]

```
— SETTIMEOFDAY — ( —<arithmetic expression>— ) —|
```

Explanation

SETTIMEOFDAY changes the parameter to an integer, places it on the top of the stack, and executes the Write Time of Day (WTOD) operator.

SHOW [MACHINEOPS]

```
— SHOW — ( —<SHOW source>— ) —|
```

<SHOW source>

```
—<EBCDIC string constant>—
—<arithmetic expression>— FOR —<arithmetic expression>—
—<EBCDIC pointer expression>—
```

Explanation

SHOW is an untyped intrinsic that uses the Primitive Display (SHOW) operator to display text, without involving the Input/Output subsystem.

If the SHOW invocation occurs in a CONTROLSTATE block and the program is not compiled with the STANDALONE option, the length of the < EBCDIC string constant > is restricted to 12 characters; a longer string in this context results in a syntax error. (The compiler enforces this restriction to prevent control-state code from incurring a presence-bit interrupt on an absent string-pool array.)

The number of characters actually displayed is limited by the machine; any additional characters in the source are ignored. A system with no primitive display mechanism disregards the SHOW operator; if a display mechanism exists, it must accommodate at least 24 characters. Depending on the implementation, a SHOW statement can erase a previous SHOW display. The <SHOW source> cannot cross a page boundary in a paged array.

The display mechanism can restrict the character set that can be displayed. At least the following characters and blank can be displayed:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ 0123456789 ,./+==()
```

The token *SHOW* is recognized but not reserved; the SHOW statement cannot be used if you have declared the identifier SHOW.

Examples

```
SHOW("MCP STARTING");           % 12 characters: valid anywhere

SHOW("UTILoader STARTING");% Not valid in CONTROLSTATE
                               % except in STANDALONE program
```

```

SHOW(Ø FOR Ø);           % Erase previous SHOW display

REPLACE P BY ...;

SHOW(P FOR L);

SHOW(DOUBLE((IF WHOAMI > 9 THEN WHOAMI + 'A'-1Ø
              ELSE WHOAMI + 'Ø'
              ) & 'PROC ' [47:4Ø]
              , 'ALARM'
              ) FOR 12);

```

SIGNALPROCESSOR [MACHINEOPS]

— SIGNALPROCESSOR — (—<integer expression>— , —————→
 →<arithmetic expression>—) —————|

Explanation

SIGNALPROCESSOR is an integer intrinsic that uses the Signal Processor (SPES) operator to send a signal to a processing element. The first parameter supplies the signal type in the range 0 through 7. The second parameter is a bit vector of type REAL that specifies the set of potential receiver processing elements. For example, subport S of port P is in the set if bit $[2*P + S:01] = 1$.

The possible result values are as follows:

Value	Meaning
0	No exception; a receiver is to be signaled.
3	None of the designated receiver ports were available.
5	A receiver processing element was not in the sender partition. (This exception might not be noted by all systems.)

STACKNUMBER [MACHINEOPS]

The intrinsic STACKNUMBER is no longer available. The intrinsic function PROCESSID should be used instead. For more information on the <processid function>, see the *A Series ALGOL Programming Reference Manual, Volume 1: Basic Implementation*.

STOP [MACHINEOPS]

— STOP —————→
→ (<expression>, <expression> [, <variable>, <variable>])

Explanation

STOP generates the Conditional Halt (HALT) operator. If the first two parameters are present, the two values are loaded onto the stack as the top and next-to-top items before the HALT operator is executed. If the third and fourth parameters are present, they are interpreted as addresses into which the top-of-stack values are to be stored following execution of the HALT operator. The third and fourth parameter addresses receive the first and second parameter values, respectively; these values might have been modified by a user while the machine was stopped.

STOP77 [MACHINEOPS]

Explanation

The STOP77 intrinsic is similar to the STOP intrinsic except that the STOP operator is generated.

SUSPEND [MACHINEOPS]

Explanation

The SUSPEND intrinsic is similar to the PAUSE intrinsic except that the Pause Until Interrupt (PAUS) operator is generated.

SYSTEMCONTROL [MACHINEOPS]

— SYSTEMCONTROL — (—<arithmetic expression>— , —————→
→<word expression>—) —————→

Explanation

SYSTEMCONTROL is a REAL intrinsic that uses the CSCP operator to communicate with the System Control Subsystem (SCS). The SCS is the subsystem responsible for system initialization and maintenance. The <arithmetic expression> parameter is of type REAL and specifies a function to the SCS. The <word expression> parameter is of type WORD and provides a word of data. The result value is the response from the SCS.

TESTRASD [MACHINEOPS]

— TESTRASD — (—<arithmetic expression>— , —————>
 ↳<integer expression>—) —————|

Explanation

TESTRASD is a word procedure that returns the result of executing the RASD Test (RAST) operator to the two parameters. The RAST operator is intended only for debug use on Level Gamma machines.

TESTWASD [MACHINEOPS]

— TESTWASD — (—<word expression>— , —<real expression>— , —>
 ↳<integer expression>—) —————|

Explanation

TESTWASD is an untyped procedure that applies the WASD Test (WAST) operator to the three parameters. The WAST operator is intended only for debug use on Level Gamma machines.

TIMER [MACHINEOPS]

— TIMER — (—<arithmetic expression>—) —|

Explanation

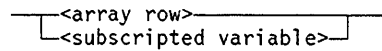
TIMER loads the parameter value onto the top of the stack and executes the Set Interval Timer (SINT) operator. The < arithmetic expression > defines the time value in microseconds. The default is 512; the maximum value is 2047.

UNSAFE Mode

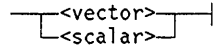
VECTOR INTRINSICS [MACHINEOPS]

The following constructs apply to the syntax diagrams for vector intrinsics:

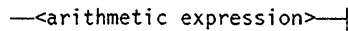
<vector>



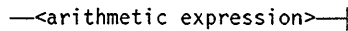
<xvector>



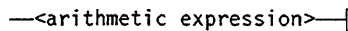
<scalar>



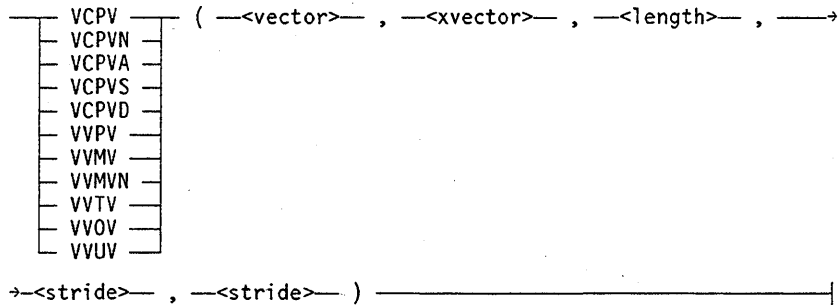
<length>



<stride>



Untyped Intrinsics That Act on Two Vectors



Explanation

As is shown in Table 9-1, these untyped intrinsics act on two vectors. The first five intrinsics copy a function of B into A; the remaining six replace A by a function of A and B. A is the vector defined by the first <vector> and first <stride>; B is the vector or scalar defined by the second <vector> or <xvector> and second <stride>.

Table 9-1. Untyped Intrinsics: Two Vectors

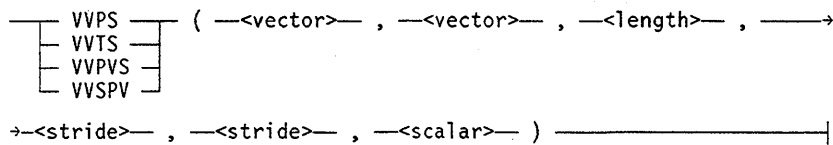
Intrinsic	Operator	Function
VCPV	CPV (Copy Vector)	A := B
VCPVN	CPVN (Copy Vector Negated)	A := -B

continued

Table 9–1. Untyped Ininsics: Two Vectors (cont.)

Intrinsic	Operator	Function
VCPVA	CPVA (Copy Vector Absolute)	$A := B $
VCPVS	CPVS (Copy Vector Single)	$A := \text{single}(B)$
VCPVD	CPVD (Copy Vector Double)	$A := \text{double}(B)$
VVPV	VPV (Vector Plus Vector)	$A := A + B$
VVMV	VMV (Vector Minus Vector)	$A := A - B$
VVMVN	VMVN (Vector Minus Vector Negated)	$A := -A + B$
VTV	VTV (Vector Times Vector)	$A := A * B$
VVOV	VOV (Vector Over Vector)	$A := A / B$
VVUV	VUV (Vector Under Vector)	$A := B / A$

Untyped Ininsics That Act on Two Vectors and a Scalar



Explanation

As shown Table 9-2, these untyped intrinsics act on two vectors and a scalar.

Table 9–2. Untyped Ininsics: Two Vectors and a Scalar

Intrinsic	Operator	Function
VVPS	VPS (Vector Plus Scalar)	$A := B + S$
VVTS	VTS (Vector Times Scalar)	$A := B * S$
VVPVS	VPVS (Vector Plus Vector Times Scalar)	$A := A + B * S$
VCSPV	VSPV (Vector Times Scalar Plus Vector)	$A := A * S + B$

VSCAT Intrinsic

```

— VSCAT — ( —<vector>— , —<xvector>— , —<length>— , —————>
→<vector>— , —<stride>— ) —————|

```

Explanation

This untyped intrinsic accepts vector I with an implicit stride of + 1 in place of the stride of A. The VSCAT intrinsic generates the Scatter (SCAT) operator to compute the following:

$$A[I(i)] := B(i)$$
VGATH Intrinsic

```

— VGATH — ( —<vector>— , —<vector>— , —<length>— , —————>
→<stride>— , —<vector>— ) —————|

```

Explanation

This untyped intrinsic accepts vector I with an implicit stride of + 1 in place of the stride of B. The VGATH intrinsic generates the Gather (GATH) operator to compute the following:

$$A(i) := B[I(i)]$$
VSUM and VSUMA Intronics

```

┌ VSUM ───┐ ( —<vector>— , —<length>— , —<stride>— , —————>
└ VSUMA ─┘ →<expression>— ) —————|

```

Explanation

These REAL or DOUBLE intrinsics compute the sum of the elements of a vector. The type of the result (REAL or DOUBLE) is the same as the type of the < expression >, which represents the initial sum. The value of the result is the initial sum in addition to the sum of each element (or, for VSUMA, the absolute value of each element) of the vector.

VDOT and VDOTX Intronics

```

┌ VDOT ───┐ ( —<vector>— , —<length>— , —<stride>— , —————>
└ VDOTX ─┘ →<expression>— ) —————|

```

Explanation

VDOT is a REAL intrinsic that computes the dot (inner) product of two vectors. The value this intrinsic returns is the sum of the <expression> and the dot product. VDOTX is a similar double intrinsic.

VSEQ Intrinsic

— VSEQ — (—<vector>— , —<xvector>— , —<length>— , —————→
 →<stride>— , —<stride>— , —<expression>—) —————|

Explanation

VSEQ is an untyped intrinsic that generates the Sequential Occurrence (SEQ) operator to compute A as a function of B and an initial < expression > Z as follows:

A(i) := Z + B(i)

VPOLY Intrinsic

— VPOLY — (—<vector>— , —<xvector>— , —<length>— , —————→
 →<stride>— , —<stride>— , —<expression>—) —————|

Explanation

VPOLY is a REAL or DOUBLE intrinsic that generates the Polynomial Recurrence (POLY) operator to compute a result as a function of A, B, and an initial < expression > Z as follows:

result := Z := A(i) + B(i) * Z

VCHEK Intrinsic

— VCHEK — (—<vector>— , —<length>— , —<stride>— , —————→
 →<expression>—) —————|

Explanation

VCHEK is a Boolean intrinsic that generates the Compute Check Hash (CHEK) operator to compute a result as a function of A and an initial single-precision < expression > H as follows:

result := H := (H EQV A(i)).[46:48]

VFMX, VFMN, and VFMXA Intrinsics

```

┌ VFMX ───┐ ( ─<vector>─ , ─<length>─ , ─<stride>─ , ────┐
├ VFMN ───┘
└ VFMXA ───┘
└──<expression>─ , ─<expression>─ ┌───┐ ────┘
                                └───┘ , ─<variable>─┘

```

Explanation

These REAL intrinsics find the index of either the maximum (VFMX), minimum (VFMN), or maximum absolute (VFMXA) value within a vector. The first <expression> is returned as the intrinsic value if no vector element exceeds the second <expression>, which is an initial comparison value. If the <variable> is specified, it is set to the maximum (minimum or maximum absolute) value.

VIA [REFERENCE]

```

──<type>─ VIA ─<word primary>─┘

┌──<procedure identifier>──┐
├──<exception procedure identifier>──┘
└──┬──┐ VIA ─<word primary>─┘
    └──┬──┐
        └──<actual parameter part>──┘

```

Explanation

The syntax <type> VIA <word primary> is used to access the item referenced by <word primary> as if the item were a value of the specified <type>. The word primary is first evaluated as a value of type WORD; the resulting value is then used as a reference, which is evaluated as a value of the specified type. Refer to the AT intrinsic for a description of <type>. The REFERENCE TO intrinsic generates word primaries that are appropriate for use with the VIA constructs.

The form <procedure identifier> VIA <word primary> applies to procedure invocation. The word primary is used as a reference to effect a procedure entry; the contents of word primary should result in an Indirect Reference Word (IRW) to a PCW for the desired code in the appropriate environment. All type checking and parameter matching is performed according to the declared procedure heading, but the address couple of the procedure is irrelevant.

The use of the *EVENT VIA* <word primary> syntax should be avoided except within the event-handling routines of the MCP.

Examples

```

W := WORD VIA M[I]
T := PROC VIA REFERENCE TO W (PARAM)
MYGEORGE VIA WORDSPIB[SNR,SIRWTOPALACE] (WHY)

```

WHATAMI [MACHINEOPS]

— WHATAMI —|

Explanation

WHATAMI generates the Read Machine Identification (WATI) operator and returns a type **DOUBLE** containing information about the machine type. Specifically, bits [07:08] of the most significant word define the machine type.

WHOAMI [MACHINEOPS]

— WHOAMI —|

Explanation

WHOAMI is an integer intrinsic that returns the result of executing the Read Processor Identification (WHOI) operator.

WORD [WORD]

— WORD — (—<expression>—) —|

Explanation

The **WORD** intrinsic performs a type transfer from the type of the < expression > to type **WORD**. Note that the type transfer is performed after the < expression > is fully evaluated. For example, the construct *WORD(D)*, where *D* is of type **DESCRIPTOR**, evaluates *D* as a **DESCRIPTOR** and performs the copy-bit action before performing the type transfer to type **WORD**.

WRITEMEMORYCONTROL [MACHINEOPS]

— WRITEMEMORYCONTROL — (—<function definition>— , —<data>— →
→) —————|

Explanation

WRITEMEMORYCONTROL is an untyped intrinsic that applies the Write External Memory Control (WEMC) operator to the two parameters. Both parameters are of type **REAL**.

WRITEPROCESSORSTATE [MACHINEOPS]

— WRITEPROCESSORSTATE — (—<status identifier>— , —<data>— →
→) —————|

Explanation

WRITEPROCESSORSTATE is an untyped intrinsic that applies the Write Internal Processor State (WIPS) operator to the two parameters. The <status identifier> is of type INTEGER and is the internal status identification of the processor. The <data> construct is of type WORD and is the value to which the specified status is set.

WRITEXMEMORYTABLE [MACHINEOPS]

— WRITEXMEMORYTABLE — (—<function definition>— , —<data>—)
 →) _____|

Explanation

WRITEXMEMORYTABLE is an untyped intrinsic that applies the Write External Memory Table (WEMT) operator to the two parameters. Both parameters are of type REAL. The <function definition> identifies where <data> is to be written and has the following layout:

[39:08] RQIN (mask of requests that are inhibited from initiating control information)

Function Definition Parameter Word:

[31:12] Environment Number
 [19:03] Virtual Page Number
 [15:08] Mask of requestors to be affected
 [03:04] MSM mask

Data Word:

[07:06] Table Data (what is to be written in table)
 [01:01] Validity Bit
 [00:01] Parity Bit

Appendix A

Reserved Words

A <reserved word> in NEWP has the same syntax as an identifier. The reserved words are divided into three types.

Types of Reserved Words

A reserved word of type 1 can never be declared as an identifier; that is, the reserved word has a predefined meaning that cannot be changed. For example, because DO is a type 1 reserved word, the declaration is flagged with a syntax error as follows:

```
ARRAY DO[0:999]
```

A reserved word of type 2 can be redeclared as an identifier; the reserved word then loses its predefined meaning in the scope of that declaration. For example, because IN is a type 2 reserved word, the following declaration is legal:

```
FILE IN(KIND = READER)
```

However, in the scope of the declaration, the following statement is flagged with a syntax error on the word IN:

```
SCAN P WHILE IN ALPHA
```

If a type 2 reserved word is used as a variable but is not declared as one, then the error message that results is not the expected UNDECLARED IDENTIFIER. Instead, the message might be the following:

```
NO STATEMENT CAN START WITH THIS
```

A reserved word of type 3 is context-sensitive. This reserved word can be redeclared as an identifier, and if it is used where the syntax calls for that reserved word, it carries the predefined meaning. Otherwise, it carries the user-declared meaning. The different meanings for the type 3 reserved word TIMELIMIT are illustrated in the following example:

Reserved Words

```
BEGIN
  FILE F (KIND=REMOTE);
  REAL TIMELIMIT;
  ARRAY A[0:49];
  % IN THE NEXT STATEMENT, "TIMELIMIT" IS A REAL VARIABLE
  TIMELIMIT := 4.5;
  % IN THE NEXT STATEMENT, "TIMELIMIT" IS A READ OPTION
  IF READ (F[TIMELIMIT 60],50,A) THEN
    % IN THE NEXT STATEMENT, "TIMELIMIT" IS A REAL VARIABLE
    TIMELIMIT := 60;
END.
```

All file attributes, direct array attributes, and mnemonics described in the *A Series I/O Subsystem Programming Reference Manual* are type 3 reserved words in NEWP. All task attributes and mnemonics described in the *A Series Work Flow Administration and Programming Guide* are type 3 reserved words in NEWP.

Reserved Words List

The following is an alphabetical list of reserved words for NEWP. The number in parentheses following each word indicates the type of the reserved word. For example, *FOR (1)* indicates that FOR is a type 1 reserved word.

Note that the presence of a reserved word in this list does not necessarily imply that the feature is fully supported. Some words appear in this list because they are reserved for future implementation, or for consistency with ALGOL or the predecessor of NEWP (ESPOL).

ACCEPT (2)	ACTUALNAME (3)
AFTER (3)	ALL (3)
ALTERNATIVE (2)	AND (2)
ANYFAULT (3)	ARRAY (1)
ARRAYS (3)	ARROGATE (2)
AS (3)	ASCII (2)
ASDSPACE (3)	AT (1)
ATEND (3)	ATTRIBSEARCHER (2)
AVAILABLE (3)	AVAILATEND (3)
BASE (3)	BCL (2)
BEFORE (3)	BYFUNCTION (3)
BITS (2)	BOOLEAN (1)
BREAK (2)	BUZZ (2)
BUZZ47 (2)	BY (2)
BYINITIATOR (3)	BYTITLE (3)
CALL (2)	CALLING (3)
CALLIO (2)	CANCEL (2)

CAND (2)	CASE (1)
CAT (2)	CAUSE (2)
CAUSEANDRESET (2)	CHECKHASH (2)
CIMP (2)	CLOSE (2)
CODE (2)	COMBINEPPBS (2)
COMMENT (1)	CONDITIONAL (3)
CONSTANT (1)	CONTINUE (2)
CONTROL (3)	CONTROLSTATE (3)
COPY (3)	COR (2)
CRUNCH (3)	DAWDLE (2)
DBS (3)	DEALLOCATE (2)
DEFINE (1)	DELAYSWAP (3)
DELINKLIBRARY (2)	DESCRIPTOR (1)
DETAIL (2)	DIGITS (2)
DIRECT (1)	DISCARD (3)
DISPLAY (2)	DIV (2)
DO (1)	DONTWAIT (3)
DOUBLE (1)	DREADMEMORYCONTROL (2)
DSABLE (3)	DSWAIT (2)
DSWAITANDRESET (2)	EBCDIC (2)
EQL (2)	EQV (2)
ESTABLISH_ASDS (2)	EVENT (1)
EXCEPTION (2)	EXIT (2)
EXPONENTOVERFLOW (3)	EXPONENTUNDERFLOW (3)
EXPORT (2)	EXTERNAL (2)
EXTERNALFUNCTION (2)	FAILREGISTER (2)
FALSE (1)	FI (1)
FILE (1)	FILES (3)
FIRSTFREEDOCCELL (3)	FIRSTSEGDESC (3)
FIX (2)	FMMRREADLOCK (2)
FOR (1)	FORK (2)
FORMAL (2)	FORMAT (1)
FORWARD (2)	FREE (2)
FREEZE (2)	FROM (2)

Reserved Words

FUNCTIONAME (3)	GEQ (2)
GO (2)	GTR (2)
HARDWARE (3)	HEX (2)
HEYOU (2)	IGNOREPARITY (2)
IMP (2)	IN (2)
INHERITSTATE (3)	INITIALIZATION (1)
INLINE (3)	INTEGER (1)
INTEGEROVERFLOW (3)	INTERFACE (2)
INTERLOCK (2)	INTERLOCKOPS (3)
INTERRUPTCHANNEL (2)	INTERRUPTCOUNTZERO (2)
INTNAME (3)	INTRINSIC (2)
INVALIDADDRESS (3)	INVALIDINDEX (3)
INVALIDOP (3)	INVALIDPROGRAMWORD (3)
IS (2)	ISNT (2)
LABEL (2)	LBOUND (2)
LEQ (2)	LIBACCESS (3)
LIBERATE (2)	LIBLINKFALULT (3)
LIBPARAMETER (3)	LIBRARIES (3)
LIBRARY (2)	LIMITED (1)
LINE (3)	LINKCLASS (3)
LINKLIBRARY (2)	LOCK (2)
LOCKSTATUS (2)	LONG (2)
LOOP (3)	LSS (2)
MACHINEOPS (3)	MCP (3)
MEMIMAGEBOUND (3)	MEMORY (3)
MEMORYFAIL1 (3)	MEMORYPARITY (3)
MEMORYPROTECT (3)	MESSAGESEARCHER (2)
MISC (3)	MLSACCEPT (2)
MLSDISPLAY (2)	MOD (2)
MODULE (2)	MODULEEXPORT (2)
MODULEIMPORT (2)	MOVESTACK (2)
MUX (2)	NAME (2)
NEQ (2)	NO (3)
NORANGECHECK (3)	NORMALSTATE (3)

NOSWAP (3)	NOT (2)
NOTDSABLE (3)	NULL (2)
NUMERIC (2)	OD (1)
OF (2)	OFFER (3)
ON (2)	OPEN (2)
OR (2)	ORDERED (2)
OUTPUTMESSAGE (1)	OVERWRITE (2)
PAGE (3)	PAGED (3)
PARITYFAIL1 (3)	PAUSE (2)
PERMANENT (3)	PICTURE (1)
POINTER (1)	PRED (2)
PRIVATELIBRARIES (3)	PRIVILEGEDINSTRUCTION (3)
PROCEDURE (1)	PROCESS (2)
PROCURE (2)	PROGRAMDUMP (2)
PROGRAMMEDOPERATOR (3)	PROTECTED (3)
PURGE (3)	RANGECHECK (3)
READ (2)	READLOCK (2)
READMEMORYCONTROL (2)	READNANDCLEAREXTERNALS (2)
READPROCESSORSTATE (2)	READXMEMORYTABLE (2)
REAL (1)	RECEIVEFROMREQUESTOR (2)
REFERENCE (1)	REGISTERS (3)
REPLACE (1)	RESETEVENT (2)
RESIZE (2)	RESUME (2)
RETAIN (3)	RETURN (2)
RETURN2 (2)	RETURNORIGINALS (2)
ROW (3)	RUN (2)
RUNNINGLIGHT (2)	SAFE (3)
SAVE (2)	SCAN (1)
SCANIN (2)	SCANOUT (2)
SCANPARITY (3)	SEEK (2)
SEGMENT (2)	SEGMENTLEVEL (3)
SELECT (2)	SENDTOREQUESTOR (2)
SEPCOMPLEVEL (3)	SET (1)
SETACTUALNAME (2)	SETEVENT (2)

Reserved Words

SETINHIBIT (2)	SETLIMITS (2)
SETTIMEOFDAY (2)	SHOW (2)
SIBS (3)	SIGNALPROCESSOR (2)
SKIP (3)	SORT (2)
SPACE (3)	STACKER (3)
STACKOVERFLOW (3)	STATION (3)
STATSUMMARY (3)	STEP (2)
STOP (2)	STOP77 (2)
STRING (1)	STRINGPROTECT (3)
STRUCTURE (1)	SUBFILE (3)
SUBTYPE (2)	SUCC (2)
SUPPLY (2)	SUSPEND (2)
SWAPNOW (3)	SYNCHRONIZE (3)
SYSTEMCONTROL (2)	SYSTEMLIB (3)
TAG (2)	TARGET (3)
TASK (2)	TEMPORARY (3)
TESTWASD (2)	THRU (1)
TIMELIMIT (3)	TIMER (2)
TITLE (3)	TO (2)
TODISK	TOPRINTER
TRANSLATETABLE (2)	TRUE (1)
TRUTHSET (2)	TYPE (2)
UBOUND (2)	UNIVERSE (1)
UNLOCK (2)	UNPACK (2)
UNSAFE (3)	UNTIL (1)
UPLEVEL (3)	URGENT (3)
VALUE (1)	VCPV (2)
VCPVA (2)	VCPVD (2)
VCPVN (2)	VCPVS (2)
VGATH (2)	VIA (1)
VSCAT (2)	VSEQ (2)
VWMV (2)	VWMVN (2)
VVOV (2)	VVPS (2)
VVPV (2)	VVPVS (2)

VSPV (2)

VTV (2)

WAIT (2)

WHILE (1)

WORD (2)

WRITE (2)

WRITEPROCESSORSTATE (2)

ZAP (2)

ZIP (2)

WTS (2)

WUV (2)

WAITANDRESET (2)

WITH (2)

WORDS (2)

WRITEMEMORYCONTROL (2)

WRITEXMEMORYTABLE (2)

ZERODIVIDE (3)

Appendix B

ALGOL Features Not Implemented in NEWP

This appendix lists the ALGOL features that are not implemented in NEWP. Some of these features have been replaced by NEWP features that perform the same function or a similar function, while others are considered inappropriate in the context of NEWP.

General Features

The following general ALGOL features are not available in NEWP:

- COMPLEX type
- STRING type
- INTERRUPT declarations
- OWN variables
- SWITCH declarations

Specific Features

The following specific items are not implemented in NEWP. Information about these items is available in the *A Series ALGOL Programming Reference Manual, Volume 1: Basic Implementation*.

The following tables include information about the NEWP features that can be used instead of the unsupported ALGOL features.

Declarations

Feature Not Available	Alternative
<array row equivalence>	<array reference declaration> and <array reference assignment>
<complex declaration>	
<direct array row equivalence>	<array reference declaration> and <array reference assignment>
<dump declaration>	TADS
<format declaration>	Format the data directly by using the various clauses of the REPLACE statement.
<forward interrupt declaration>	

ALGOL Features Not Implemented in NEWP

Feature Not Available	Alternative
<forward switch label declaration>	
<interrupt declaration>	
<list declaration>	Include the elements directly in the READ and WRITE statements.
<monitor declaration>	TADS
<picture declaration>	Format the data directly using the various clauses of the REPLACE statement.
<string declaration>	EBCDIC arrays
<string array declaration>	EBCDIC arrays
<switch file declaration>	Use a DEFINE with parameters to select the proper file instead of a switch file. For example: <pre>FILE F1, F2, F3 (KIND = DISK); DEFINE SWITCH_FL [INX] = (CASE INX OF (F1, F2, F3)) #;</pre>
<switch format declaration>	Format the data directly using the various clauses of the REPLACE statement. The switching can be simulated with a define similar to the one shown under <switch file declaration>.
<switch label declaration>	Use a DEFINE with parameters similar to the one shown under <switch file declaration>.
<switch list declaration>	Use a DEFINE with parameters similar to the one shown under <switch file declaration> to select a READ or WRITE statement with the appropriate elements included directly.

Statements

Feature Not Available	Alternative
<attach statement>	
<change file statement>	Open the file, change the TITLE attribute, and then close the file again with LOCK. Note that this process is slower than the <change file statement> because of the need to open the file.
<checkpoint statement>	
<detach statement>	
<disable statement>	
<enable statement>	
<exchange statement>	Open both files, read the appropriate information into buffers, and then write the information out to the opposite files.
<fill statement>	<assignment statement>

ALGOL Features Not Implemented in NEWP

Feature Not Available	Alternative
<merge statement>	
<multiple attribute assignment statement>	Use individual attribute assignment statements.
<on statement>	Use an <on declaration> in place of the <on statement>. NEWP provides more fault names than ALGOL does. For more information, refer to the "ON Declaration" in Section 4, "Declarations."
<removefile statement>	Use the CLOSE with PURGE statement. Note that if the file is not already open, it must be opened before the CLOSE can be done.
<rewind statement>	<close option> RETAIN statement
<space statement>	SEEK statement, for example: <pre>SEEK (<file designator>[SPACE <arithmetic expr>]);</pre>
<when statement>	WAIT statement

Expressions

Feature Not Available	Alternative
<complex case expression>	
<complex expression>	
<complex function designator>	
<complex relation>	
<conditional complex expression>	
<conditional designational expression>	
<designational case expression>	
<dmin function> and <dmax function>	These functions are unnecessary in NEWP because the MIN and MAX functions handle both single-precision and double-precision values.
<dnormalize function>	This function is unnecessary in NEWP because the NORMALIZE function accepts either a single or a double arithmetic expression as its parameter.
<string function designator>	

The following ALGOL intrinsics are not directly supported in NEWP:

ALGOL Features Not Implemented in NEWP

- CHECKSUM, COMPLEX, CONJUGATE
- DABS, DECIMAL, DELTA, DIMP, DNABS, DALPHA, DSCALELEFT, DSCALERIGHT, DSCALERIGHTT
- FIRST
- IMAG
- LNGAMMA
- SCALELEFT, SCALERIGHT, SCALERIGHTT

The following ALGOL intrinsics are not directly supported in NEWP, but their functionality can be obtained by declaring GENERALSUPPORT as a library and importing the needed entry points:

- ARCCOS, ARCSIN, ARCTAN, ARCTAN2, ATANH
- CABS, CCOS, CEXP, CLN, COS, COSH, COTAN, CSIN, CSQT
- DARCCOS, DARCSIN, DARCTAN, DARCTAN2, DCOS, DCOSH, DERF, DERFC, DEXP, DGAMMA, DLGAMMA, DLN, DLOG, DSIN, DSINH, DSQRT, DTAN, DTANH
- ERF, ERFC, EXP
- GAMMA
- LN, LOG
- RANDOM
- SIN, SINH, SQRT
- TAN, TANH

The following <string function>s are not supported by NEWP, but the same functionality can be accomplished by using EBCDIC arrays or EBCDIC pointers or both:

- DROP
- HEAD
- REPEAT
- STRING, STRING4, STRING7, STRING8
- TAIL, TAKE, TRANSLATE

Compiler Control Options

The following ALGOL compiler control options are not implemented in NEWP:

- AUTOBIND
- BCL, BEGINSEGMENT, BIND, BINDER
- CHECK
- DONTBIND, DUMPINFO
- ENDSEGMENT, EXTERNAL

- FORMAT
- GO TO
- HOST
- INITIALIZE, INTRINSICS
- LEVEL, LIBRARY, LISTDELETED, LOADINFO
- NOBCL, NOSTACKARRAYS, NOXREFLIST
- OLDRESIZE, OPTIMIZE
- PURGE
- SEGDESCABOVE, SEQERR, STOP
- USE
- WARNSUPR, WRITEAFTER
- XDECS, XREFS

Miscellaneous

The following miscellaneous ALGOL features are not supported in NEWP:

- Identifiers, numbers, and strings continued across card images.
- LB and RB as synonyms for the square brackets ([]).
- Multicharacter operators with embedded blanks. However, the Update Replacement operator (:=*) is allowed to have an arbitrary number of blanks between the equal sign (=) and the asterisk (*).
- The KIND=READER file attribute for the compiler file CARD; CARD must be file-equated to a disk file.
- Batch facility.
- Compile-time facility.

Product Interfaces

The following products can interface with ALGOL but not with NEWP:

- Advanced Data Dictionary System (ADDS)
- Communications Management System (COMS)
- Data Management System II (DMSII)
- DMSII Transaction Processing System (TPS)
- Screen Design Facility Plus (SDF Plus)
- Semantic Information Manager (SIM)

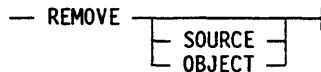
Appendix C

Understanding Railroad Diagrams

What Are Railroad Diagrams?

Railroad diagrams are diagrams that show you the rules for putting words and symbols together into commands and statements that the computer can understand. These diagrams consist of a series of paths that show the allowable structure, constants, and variables for a command or a statement. Paths show the order in which the command or statement is constructed. Paths are represented by horizontal and vertical lines. Many railroad diagrams have a number of different paths you can take to get to the end of the diagram.

Example



If you follow this railroad diagram from left to right, you will discover three acceptable commands. These commands are

REMOVE

REMOVE SOURCE

REMOVE OBJECT

If all railroad diagrams were this simple, this explanation could end here. However, because the allowed ways of communicating with the computer can be complex, railroad diagrams sometimes must also be complex.

Regardless of the level of complexity, all railroad diagrams are visual representations of commands and statements. Railroad diagrams are intended to

- Show the mandatory items
- Show the user-selected items
- Present the order in which the items must appear
- Show the number of times an item can be repeated
- Show the necessary punctuation

To familiarize you with railroad diagrams, this explanation describes the elements of the diagrams and provides examples.

Understanding Railroad Diagrams

Some of the actual railroad diagrams you will encounter might be more complex. However, all railroad diagrams, simple or complex, follow the same basic rules. They all consist of paths that represent the allowable structure, constants, and variables for commands and statements.

By following railroad diagrams, it is easy to understand the correct syntax for commands and statements. Once you become proficient in the use of railroad notation, the diagrams serve as quick references to the commands and statements.

Constants and Variables

A constant is an item that cannot be altered. You must enter the constant as it appears in the diagram, either in full or as an allowable abbreviation. If a constant is partially underlined, you can abbreviate the constant by entering only the underlined letters. In addition to the underlined letters, any of the remaining letters can be entered. If no part of the constant is underlined, the constant cannot be abbreviated. Constants can be recognized by the fact that they are never enclosed in angle brackets (< >) and are in uppercase letters.

A variable is an item that represents data. You can replace the variable with data that meets the requirements of the particular command or statement. When replacing a variable with data, you must follow the rules defined for the particular command or statement. Variables appear in railroad diagrams enclosed in angle brackets (< >).

In the following example, BEGIN and END are constants while <statement list> is a variable. The constant BEGIN can be abbreviated since it is partially underlined. Valid abbreviations for BEGIN are BE, BEG, and BEGI.

```
— BEGIN —<statement list>— END —|
```

Constraints

Constraints are used in a railroad diagram to control progression through the diagram. Constraints consist of symbols and unique railroad diagram line paths. They include

- Vertical bars
- Percent signs
- Right arrows
- Required items
- User-selected items
- Loops
- Bridges

A description of each item follows.

Vertical Bar

The vertical bar symbol (|) represents the end of a railroad diagram and indicates the command or statement can be followed by another command or statement.

— SECONDWORD — (—<arithmetic expression>—) —|

Percent Sign

The percent sign (%) represents the end of a railroad diagram and indicates the command or statement must be on a line by itself.

— STOP —%

Right Arrow

The right arrow symbol (>) is used when the railroad diagram is too long to fit on one line and must continue on the next. A right arrow appears at the end of the first line and another right arrow appears at the beginning of the next line.

— SCALERIGHT — (—<arithmetic expression>— , —————→
 →<arithmetic expression>—) —————|

Required Items

A required item can be either a constant, a variable, or punctuation. A required item appears as a single entry, by itself or with other items, on a horizontal line. Required items can also exist on horizontal lines within alternate paths or nested (lower-level) diagrams. If the path you are following contains a required item, you must enter the item in the command or statement; the required item cannot be omitted.

In the following example, the word EVENT is a required constant and < identifier > is a required variable:

— EVENT —<identifier>—|

User-Selected Items

User-selected items appear one below the other in a vertical list. You can choose any one of the items from the list. If the list also contains an empty path (solid line), none of the choices are required. A user-selected item can be either a constant, a variable, or punctuation. In the following railroad diagram, either the plus sign (+) or minus sign (-) can be entered before the required variable < arithmetic expression >, or the symbols can be disregarded because the diagram also contains an empty path.

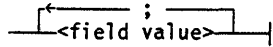
—|<arithmetic expression>—|
 [+]
 [-]

Loop

A loop represents an item or group of items that you can repeat. A loop can span all or part of a railroad diagram. It always consists of at least two horizontal lines, one below the other, connected on both sides by vertical lines. The top line is a right-to-left path that contains information about repeating the loop.

Understanding Railroad Diagrams

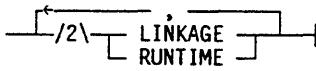
Some loops include a return character. A return character is a character (often a comma or semicolon) required before each repetition of a loop. If there is no return character, the items must be separated by one or more blank spaces.



Bridge

Sometimes a loop also includes a bridge, which is used to show the maximum number of times the loop can be repeated. The bridge can precede the contents of the loop, or it can precede the return character (if any) on the upper line of the loop.

The bridge determines the number of times you can cross that point in the diagram. The bridge is an integer enclosed in sloping lines (/ \). Not all loops have bridges. Those that do not can be repeated any number of times until all valid entries have been used.



or



In the first bridge example, you can enter LINKAGE or RUNTIME no more than two times. In the second bridge example, you can enter LINKAGE or RUNTIME no more than three times.

In some bridges an asterisk follows the number. The asterisk means that you must select one item from the group.



The following figure shows the types of constraints used in railroad diagrams.



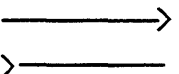
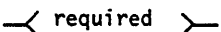
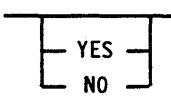
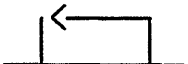
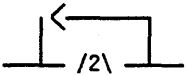
SYMBOL/PATH	EXPLANATION
	Vertical bar. Indicates that the command or statement can be followed by another command or statement.
	Percent sign. Indicates that the command or statement must be on a line by itself.
	Right arrow. Indicates that the diagram occupies more than one line.
	Required items. Indicates the constants, variables, and punctuation that must be entered in a command or statement.
	User-selected items. Indicates the items that appear one below the other in a vertical list. You select which item or items to include.
	A loop. Indicates an item or group of items that can be repeated.
	A bridge. Indicates the maximum number of times a loop can be repeated.

Figure C-1. Railroad Constraints

Following the Paths of a Railroad Diagram

The paths of a railroad diagram lead you through the command or statement from beginning to end. Some railroad diagrams have only one path, while others have several alternate paths. The following railroad diagram indicates there is only one path that requires the constant LINKAGE and the variable <linkage mnemonic> :

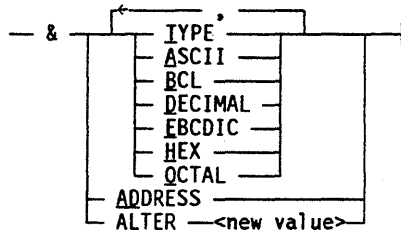
— LINKAGE —<linkage mnemonic>—|

Alternate paths provide choices in the construction of commands and statements. Alternate paths are provided by loops, user selected items, or a combination of both. More complex railroad diagrams can consist of many alternate paths, or nested (lower-level) diagrams, that show a further level of detail.

For example, the following railroad diagram consists of a top path and two alternate paths. The top path includes an ampersand (&) and the constants (that are

Understanding Railroad Diagrams

user-selected items) in the vertical list. These constants are within a loop that can be repeated any number of times until all options have been selected. The first alternate path requires the ampersand (&) and the required constant ADDRESS. The second alternate path requires the ampersand (&) followed by the required constant ALTER and the required variable <new value>.



Railroad Diagram Examples

The following examples show five railroad diagrams and possible command and statement constructions based on the paths of these diagrams.

Example 1

<lock statement>

— LOCK — (— <file identifier> —) —

Sample Input

LOCK (F1)

LOCK (FILE4)

Explanation

LOCK is a constant and cannot be altered. Because no part of the word is underlined, the entire word must be entered. The parentheses are required punctuation and F1 and FILE4 are sample <file identifier>s.

Example 2

<open statement>

— OPEN —

INQUIRY
UPDATE

 <database name> —

Sample Input

OPEN DATABASE1

OPEN INQUIRY DATABASE1

OPEN UPDATE DATABASE1

Explanation

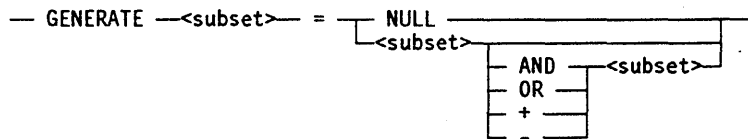
The first sample input shows the constant OPEN followed by the variable DATABASE1, which is a database name. The railroad diagram shows two user-selected items, INQUIRY and UPDATE. However, because there is an empty path (solid line), these entries are not required.

The second sample input shows the constant OPEN followed by the user-selected constant INQUIRY and the variable DATABASE1.

The third sample input shows the constant OPEN followed by the user-selected constant UPDATE and the variable DATABASE1.

Example 3

<generate statement>



Sample Input

GENERATE Z = NULL

GENERATE Z = X

GENERATE Z = X AND B

GENERATE Z = X + B

Explanation

The first sample input shows the GENERATE constant followed by the variable Z, an equal sign, and the user-selected constant NULL.

The second sample input shows the GENERATE constant followed by the variable Z, an equal sign, and the user-selected variable X.

The third sample input shows the GENERATE constant followed by the variable Z, an equal sign, the user-selected variable X, the AND command (from the list of user-selected items in the nested path), and a third variable, B.

The fourth sample input shows the GENERATE constant followed by the variable Z, an equal sign, the user-selectable variable X, the plus sign (from the list of user-selected items in the nested path), and a third variable, B.

Understanding Railroad Diagrams

Example 4

<entity reference declaration>

— ENTITY REFERENCE — $\overbrace{\text{<entity ref ID>}}^{\text{---}}$ ($\overbrace{\text{<class ID>}}^{\text{---}}$) —|

Sample Input

ENTITY REFERENCE ADVISOR1 (INSTRUCTOR)

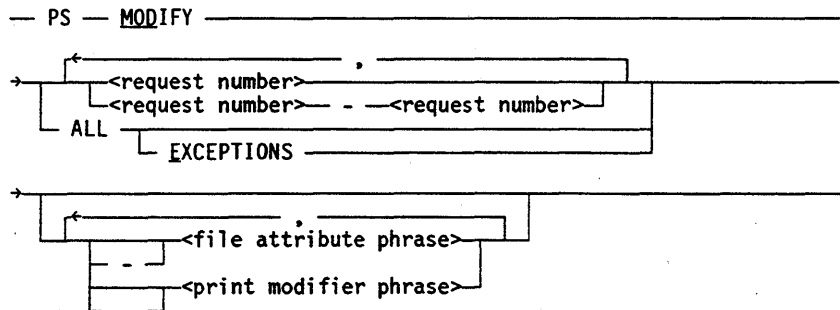
ENTITY REFERENCE ADVISOR1 (INSTRUCTOR), ADVISOR2 (ASST_INSTRUCTOR)

Explanation

The first sample input shows the required item ENTITY REFERENCE followed by the variable ADVISOR1 and the variable INSTRUCTOR. The parentheses are required.

The second sample input illustrates the use of a loop by showing the same input as in the first sample followed by a comma, the variable ADVISOR2, and the variable ASST_INSTRUCTOR. The parentheses are required.

Example 5



Sample Input

PS MODIFY 11159

PS MODIFY 11159,11160,11163

PS MODIFY 11159-11161 DESTINATION = "LP7"

PS MOD ALL EXCEPTIONS

Explanation

The first sample input shows the constants PS and MODIFY followed by the variable 11159, which is a <request number>.

The second sample input illustrates the use of a loop by showing the same input as in the first sample followed by a comma, the variable 11160, another comma, and the final variable 11163.

The third sample input shows the constants PS and MODIFY followed by the user-selected variables 11159-11161, which are <request number> s, and the user-selected variable DESTINATION = "LP7", which is a <file attribute phrase> .

The fourth sample input shows the constants PS and MODIFY followed by the user-selected constant ALL, followed by the user-selected constant EXCEPTIONS. Note that in this sample input, the constant MODIFY has been abbreviated.

Glossary

A

address couple

A representation of the address of an item in a program. An address couple consists of two numbers: the first number is a lexical level, and the second number is a displacement (offset) within that lexical level.

address equation

The process of declaring an identifier to have the same address as a previously declared identifier or a specifically supplied address.

Address Space Number (ASN)

On systems using the Master Control Program (MCP) operating system, a number that refers to a particular address space in memory.

ADDS

See Advanced Data Dictionary System.

Advanced Data Dictionary System (ADDS)

A software product that allows for the centralized definition, storage, and retrieval of data descriptions.

ASCII

American Standard Code for Information Interchange. A standard 7-bit or 8-bit information code used to represent alphanumeric characters, control characters, and graphic characters on a computer system.

ASN

See Address Space Number.

B

bad GO TO

A GO TO statement in an inner block that transfers control to a label that is global to that block. The block in which a bad GO TO statement occurs is exited abruptly and local variables are deallocated immediately.

bootstrap

A collection of data and machine instructions capable of loading another program into memory.

C

call-by-name

Pertaining to one method of passing a parameter to a procedure. The system substitutes the actual parameter wherever the formal parameter is mentioned in the procedure body. Any assignments to the actual parameter immediately change the value of the formal parameter, and vice versa.

call-by-reference

Pertaining to one method of passing a parameter to a procedure. The system evaluates the location of the actual parameter and replaces the formal parameter with a reference to that location. Any change made to the formal parameter affects the actual parameter, and vice versa.

call-by-value

Pertaining to one method of passing a parameter to a procedure. A copy of the value of the actual parameter is assigned to the formal parameter, which is thereafter handled as a variable that is local to the procedure body. Any change made to the value of a call-by-value formal parameter has no effect outside the procedure body.

Communicate with Universal I/O (CUIO) operator

An operator on A 9 systems that passes the address of an I/O Control Block (IOCB)

CUIO operator

See Communicate with Universal I/O operator

G

global identifier

Within a given block of an ALGOL, NEWP, or Pascal program, an identifier that is declared in an outer block. The value of the global identifier can be modified by any block to which it is global unless a local variable of the same name has been declared.

L

lexical level

A number that indicates the relative level of an addressing space within the stack of an executing program. Lexical levels range from 0 through either 15 or 31, depending on the computer family. A lower lexical level indicates a more global addressing space.

local identifier

An identifier that is declared within a given block of a program. The value or values associated with that identifier inside the block are not associated with that identifier outside the block.

N

nonnumeric constant

A constant identifier whose root type is an enumerated scalar type.

numeric constant

A constant identifier whose root type is REAL, INTEGER, DOUBLE, DESCRIPTOR, and WORD.

P**p-bit**

See presence bit.

PCW

See Program Control Word.

presence bit (p-bit)

A bit in a descriptor that indicates whether the address in the descriptor references a location in main memory or on a disk. If the presence bit is equal to 1, the address is in physical memory. If the presence bit is equal to 0, the address is either on a disk, or no memory or disk area is assigned for the descriptor yet. A p-bit is used in all descriptors on Address Space Number (ASN) systems, but is used only in the Actual Segment Descriptor (ASD) table on ASD systems because of the memory architecture.

Program Control Word (PCW)

A word that is used to transmit processing information from a control program to the operational programs, or between operational programs.

R**RCW**

See Return Control Word.

Return Control Word (RCW)

A tag-3 word created by the processor when the processor calls a procedure, function, or subroutine. The RCW contains program address information describing where to return in the program when the subroutine exits.

S**Screen Design Facility Plus (SDF Plus)**

A Unisys product used for creating user interface systems (UISS) for online, transaction-based application systems.

SCS

See System Control Subsystem.

SCW

See Software Control Word.

SDF Plus

See Screen Design Facility Plus.

separately compiled procedure (SEPCOMP)

A procedure that is compiled on its own, rather than as part of a program. It uses a patch, a base symbol file, and a host file from a previous host compile or SEPCOMP to perform a compilation.

SEPCOMP

See separately compiled procedure.

SIRW

See Stuffed Indirect Reference Word.

Software Control Word (SCW)

A word that is used to transmit processing information from the control program to the operational programs, or between operational programs.

Stuffed Indirect Reference Word (SIRW)

A word that references a location in an addressing environment. The form of the reference is such that the SIRW always points to the same location, no matter what the state of the current addressing environment.

System Control Subsystem (SCS)

The hardware and software on A Series Entry and Medium Systems (EMS) that control maintenance functions. For example, on an A 5 system, the User Interface Processor (UIP) has programmable read-only memory (PROM) code that enables it to act as the SCS. As the SCS, it performs a maintenance subsystem self-test and loads the maintenance subsystem software.

T

TPS

See Transaction Processing System.

Transaction Processing System (TPS)

A Unisys system that provides methods for processing a high volume of transactions, keeps track of all input transactions that access the database, enables the user to batch data for later processing, and enables transactions to be processed on a database that resides on a remote system.

U

up-level pointer assignment

In ALGOL, any construct that could result in a pointer pointing to an array declared at a higher lexical level than that at which the pointer is declared. Such a construct is disallowed by the compiler, because the array can be deallocated, leaving the pointer pointing to an invalid portion of memory.

Bibliography

A Series ALGOL Programming Reference Manual, Volume 1: Basic Implementation (form 1169844). Unisys Corporation.

A Series ALGOL Programming Reference Manual, Volume 2: Product Interfaces (form 1170099). Unisys Corporation.

A Series I/O Subsystem Programming Reference Manual (form 1169984). Unisys Corporation.

A Series System Architecture Reference Manual, Volume 2 (form 5014954). Unisys Corporation.

A Series Work Flow Administration and Programming Guide (form 1170149). Unisys Corporation.

A Series Work Flow Language (WFL) Programming Reference Manual (form 1169802). Unisys Corporation.

Index

A

abnormal exit, 4-5
Absolute Store Reference Word (ASRW), 8-3
Absolute Store Reference Word (ASRW)
operator, 9-20
ACCEPT statement, 5-1, 5-2, 5-3
<accept statement>, 5-1, 5-2, 5-3
<address couple>, 9-1
address equation
in INTRINSIC declarations, 4-13
Address Equation Declaration (UNSAFE),
9-1
<address primary>, 9-13
<addressable identifier>, 9-12
ALGOL
compared with NEWP, 1-1
<alternative identifier>, 4-19, 5-12
<alternative>, 4-19
anonymous set, 6-9
<arithmetic constant declaration>, 4-4
arithmetic expression, 6-3
arithmetic function designator, 6-1, 6-2, 6-5
<arithmetic function designator>, 6-1, 6-2,
6-5
arithmetic operator, 6-1
<arithmetic operator>, 6-1
<array class>, 4-2
ARRAY declaration, 4-2
<array declaration>, 4-2
ARRAY REFERENCE ASSIGNMENT
statement, 5-4
ARRAY REFERENCE declaration, 4-3
<array reference declaration>, 4-3
<array reference identifier>, 4-3
<array specification>, 4-25
<array type>, 4-26
ARROGATE function, 6-11
<arrogate function>, 6-11
<ASCII constant>, 3-2
<ASCII numeric character>, 3-2
<ASCII string character>, 3-4
<ASCII string constant>, 3-4

ASD compiler control option, 8-2
ASDSPACE block directive, 8-14
ASDTABLE [MACHINEOPS] intrinsic
(UNSAFE), 9-12
ASRW, (See Absolute Store Reference Word
(ASRW))
assignment compatibility, 4-41
ASSIGNMENT statement, 5-3
<assignment statement>, 5-3
AT [REFERENCE] intrinsic (UNSAFE),
9-13

B

<base type identifier>, 4-39
basic symbols, 3-1
<BCL character>, 3-4
<BCL constant>, 3-2
<BCL numeric character>, 3-2
<BCL string constant>, 3-4
<binary character>, 3-3
<binary constant>, 3-3
block directives
ASDSPACE, 8-14
CONTROLSTATE, 8-14
FIRSTFREED0CELL, 8-14
FIRSTSEGDESC, 8-15
INHERITSTATE, 8-15
INLINE, 8-15
INTERLOCKOPS, 8-15
MEMIMAGEBOUND, 8-16
NORANGECHECK, 8-16
NORMALSTATE, 8-16
PROTECTED, 8-17
RANGECHECK, 8-17
SAFE, 8-17
SEGMENT, 8-17
SEGMENTLEVEL, 8-18

- SEPCOMLEVEL, 8-18
- STATSUMMARY, 8-11, 8-18
 - < target option >, 8-18
- UNSAFE, 8-19
- BMASKSEARCH [MACHINEOPS] intrinsic (UNSAFE), 9-14
- Boolean declaration, 4-1
 - < Boolean declaration >, 4-1
- Boolean expression, 6-3
 - precedence in, 6-4
- Boolean function designator, 6-1, 6-6
 - < Boolean function designator >, 6-1, 6-6
 - < Boolean library attribute specification >, 4-14
- Boolean primary, 6-1
 - < Boolean primary >, 6-1
 - < Boolean-valued library attribute name >, 4-14
 - < bound pair >, 4-2
- BREAK function, 6-12
 - < break function >, 6-12
- BUZZ [MISC] intrinsic (UNSAFE), 9-14
- BUZZ47 [MISC] intrinsic (UNSAFE), 9-15

C

- call-by-name parameter, 4-27
- call-by-reference parameter, 4-26
- call-by-value parameter, 4-27
- CALLIO [MACHINEOPS] intrinsic (UNSAFE), 9-15
- CARD input file, 7-5
- case expression, 6-1
 - < case expression >, 6-1
 - < case head >, 5-8
- CASE statement, 5-1, 5-8
 - < case statement >, 5-1
- character representation, 3-5
- CHECKHASH [MACHINEOPS] intrinsic (UNSAFE), 9-15
- CLEAR compiler control option, 8-3
- CLOSE statement, 5-1
 - < close statement >, 5-1
- coercion, 9-3
- Communicate with Universal I/O (CUIO) operator, 9-15
- compilation order, 4-18
- compiler control options
 - ASD, 8-2
 - CLEAR, 8-3
 - INCLLIST, 8-3

- INSTALLATION, 8-3
- INTERLOCKOPS, 8-3
- LIST, 8-4
- LISTO, 8-4
- LIST1, 8-4
- MAKEHOST, 8-5
- MCP, 8-5
- MERGE, 8-5
- MODSTRICT, 8-5
- NEW, 8-6
- NOCOUNT, 8-6
 - not implemented in NEWP, B-4
- PROCREF, 8-6
- READLOCK, 8-6
- READLOCKTIMEOUT, 8-7
- SEPCOMP, 8-7
- SEPCOMPMERGE, 8-7
- SINGLE, 8-7
- STANDALONE, 8-2, 8-8, 8-9
- STATISTICS, 8-10
- TADS, 8-11
- UNDERLINE, 8-11
- VERSION, 8-12
- VOID, 8-12
- XREF, 8-12C
- XREFFILES, 8-12C
 - \$, 8-12C
- COMPILETIME (20), 8-12
- COMPILETIME (21), 8-12
- COMPILETIME (22), 8-12
- Compute Check Hash (CHEK) operator, 9-32
 - < conditional descriptor expression >, 9-9
- conditional expression, 6-1
 - < conditional expression >, 6-1
- Conditional Halt (HALT) operator, 9-28
 - < conditional operator >, 6-3
 - < conditional word expression >, 9-11
- CONSTANT declaration, 4-4
 - < constant declaration >, 4-4
 - < constant identifier >, 4-4
 - < constant scalar type expression >, 6-7
 - < constant value >, 4-4
- constants
 - numeric, 3-2
 - string, 3-4
- CONTROLSTATE block directive, 8-14

D

- DAWDLE [MISC] intrinsic (UNSAFE), 9-15
- DECIMALCONVERT function, 6-10

- < decimalconvert function >, 6-10
 - declarations
 - ARRAY, 4-2
 - ARRAY REFERENCE, 4-3
 - CONSTANT, 4-4
 - EXCEPTION PROCEDURE, 4-5
 - EXCEPTION PROCEDURE FORWARD, 4-8
 - EXPORT, 4-8
 - INTERLOCK, 4-9
 - INTERLOCK ARRAY, 4-9
 - INTRINSIC, 4-10
 - LABEL, 4-14
 - LIBRARY, 4-14
 - MODULE, 4-15
 - MODULE (old), 4-18C
 - not implemented in NEWP, B-1
 - ON, 4-22
 - OUTPUTMESSAGE ARRAY, 4-23
 - POINTER, 4-24
 - PROCEDURE, 4-24
 - PROCEDURE REFERENCE, 4-34
 - SCALAR TYPE, 4-35
 - SEGMENT, 4-34
 - SET TYPE, 4-42
 - SIMPLE VARIABLE, 4-35
 - STRUCTURE TYPE, 4-35
 - STRUCTURE TYPE VARIABLE, 4-43
 - SUPPLY, 4-44
 - VALUE ARRAY, 4-45
 - Delay (DLAY) operator, 9-16
 - DELAWSWAP option, 9-8
 - descendant types, 4-40
 - DESCRIPTOR [DESCRIPTOR] intrinsic (UNSAFE), 9-16
 - < descriptor case expression >, 9-9
 - DESCRIPTOR declaration (UNSAFE), 9-3
 - < descriptor declaration >, 9-3
 - < descriptor expression >, 9-9
 - DESCRIPTOR expressions (UNSAFE), 9-9
 - < descriptor identifier >, 9-3
 - < descriptor primary >, 9-9
 - designational expression, 6-1
 - < designational expression >, 6-1
 - < dimension specs >, 4-2
 - dinteger function, 6-1
 - < dinteger function >, 6-1
 - DINTEGERT function, 6-11
 - DIRECT ARRAY declaration, 4-1
 - < direct array declaration >, 4-1
 - < displacement >, 9-1
 - DLL [REGISTERS] intrinsic (UNSAFE), 9-16
 - DOUBLE declaration, 4-1
 - < double declaration >, 4-1
 - < double_the_cache_size >, 9-24
 - DREADMEMORYCONTROL [MACHINEOPS] intrinsic (UNSAFE), 9-16
 - DSABLE option, 9-8
 - dynamic procedure specification, 4-31
 - Dynamic Scale Right Save (DSRS) operator, 9-24
 - < D0 address couple >, 9-5
 - < D0 displacement >, 9-5
- ## E
- < EBCDIC constant >, 3-2
 - < EBCDIC numeric character >, 3-2
 - < EBCDIC string character >, 3-4
 - < EBCDIC string constant >, 3-4
 - < edit micros >, 5-11, 9-7
 - enclosing module, 4-16
 - < enumerated endpoint >, 4-39
 - < enumerated literal list >, 4-36
 - < enumerated literal >, 4-36
 - < enumerated range >, 4-39
 - < enumerated type >, 4-36
 - enumerated types, 4-36
 - ordered, 4-37
 - unordered, 4-37
 - < equation part >, 9-1
 - EVAL [MACHINEOPS] intrinsic (UNSAFE), 9-17
 - Evaluate (EVAL) operator, 9-17
 - exception procedure
 - automatic invocation, 4-7
 - restrictions, 4-6
 - EXCEPTION PROCEDURE declaration, 4-5
 - < exception procedure declaration >, 4-5
 - EXCEPTION PROCEDURE FORWARD declaration, 4-8
 - < exception procedure forward declaration >, 4-8
 - < exception procedure identifier >, 4-5
 - EXIT [MACHINEOPS] intrinsic (UNSAFE), 9-17
 - EXPORT declaration, 4-8
 - < export declaration >, 4-8
 - < export list >, 4-18C
 - < export object specification >, 4-8

- < export options >, 4-8
- < exportable identifier >, 4-16
- < expression >, 6-1
- expressions
 - arithmetic, 6-3
 - Boolean, 6-3
 - DESCRIPTOR (UNSAFE), 9-9
 - function, 6-5
 - not implemented in NEWP, B-3
 - pointer, 6-6
 - scalar type, 6-7
 - set, 6-8
 - string, 6-9
 - WORD (UNSAFE), 9-10

F

- FAILREGISTER [MACHINEOPS] intrinsic (UNSAFE), 9-17
- fault handling, 4-22
- fault names, 4-22.
- fault numbers, 4-22
- Fetch Main Memory Reference (FMMR) operator, 9-18
- Fetch Main Memory Register (FMFR) operator, 9-17
- FIRSTFREED0CELL block directive, 8-14
- FIRSTSEGDESC block directive, 8-15
 - < float character >, 9-7
- FLOAT option (UNSAFE), 9-7
 - < FLOAT parameters >, 9-7
- FMMRREADLOCK [MACHINEOPS] intrinsic (UNSAFE), 9-18
- FOR statement, 5-1, 5-8
 - < for statement >, 5-1, 5-8
- FORK statement (UNSAFE), 9-6
 - < fork statement >, 9-6
- FORKHANDLER MCP procedure, 9-6
 - < formal parameter list >, 4-25
 - < formal parameter part >, 4-24
- FREEZE statement, 5-9
 - < freeze statement >, 5-9
- FREEZE(MCP), 5-9
- full compilation, 7-1
- function expression, 6-1, 6-5
 - < function expression >, 6-1, 6-5
- functions
 - ARROGATE, 6-11
 - BREAK, 6-12
 - DECIMALCONVERT, 6-10
 - DINTEGERT, 6-11

- INTERLOCK, 6-11
- LOCK, 6-13
- LOCKSTATUS, 6-14
- LOWER BOUND, 6-16
- MAPPING, 6-16
- PACKDECIMAL, 6-16
- PREDECESSOR, 6-18
- SCALAR TYPE, 6-16
- SUCCESSOR, 6-18
- UNLOCK, 6-15
- UPPER BOUND, 6-19

G

- Gather (GATH) operator, 9-31
- GO TO statement, 5-1
 - < go to statement >, 5-1

H

- HARDWARE option, 8-4
 - < hexadecimal character >, 3-4
 - < hexadecimal constant >, 3-3
 - < hexadecimal string constant >, 3-4
- host compilation, 7-1
- HOST input file, 7-5

I

- I/O statement, 5-1
 - < I/O statement >, 5-1
- Ignore Parity (IGPR) operator, 9-18
- IGNOREPARITY [MACHINEOPS] intrinsic (UNSAFE), 9-18
- implicit string concatenation, 4-23
- implicit type transfer, 9-3
- in-line procedures, 4-28
 - restrictions on, 4-28
- INCLLIST compiler control option, 8-3
- Indirect Reference Word (IRW), 9-33
- INHERITSTATE block directive, 8-15
 - < initial part >, 5-8
 - < initialization procedure >, 4-19
- INLINE block directive, 7-6C, 8-15
 - < insert character >, 5-11
- INSERT option, 5-11
- INSTALLATION compiler control option, 8-3

< installation >, 9-1
 < integer constant identifier >, 4-39
 INTEGER declaration, 4-1
 < integer declaration >, 4-1
 integer endpoint, 4-40
 < integer endpoint >, 4-39, 4-40
 INTEGER function, 6-2
 < integer function >, 6-2
 < integer range >, 4-39
 < integer subtype endpoint >, 4-39
 < integer subtype range >, 4-39
 < interface body >, 4-16
 < interface declaration >, 4-16
 < interface identifier >, 4-16
 < interface >, 4-16
 interlock
 compared with PROCURE and
 LIBERATE statements, 4-10
 INTERLOCK ARRAY declaration, 4-9
 < interlock array declaration >, 4-9
 < interlock array identifier >, 4-9
 INTERLOCK declaration, 4-9
 < interlock declaration >, 4-9
 < interlock designator >, 6-11
 INTERLOCK function, 6-11
 < interlock identifier >, 4-9
 INTERLOCKOPS block directive, 8-15
 INTERLOCKOPS compiler control option,
 8-3
 Interrupt Channel (INCN) operator, 9-18
 INTERRUPTCHANNEL [MACHINEOPS]
 intrinsic (UNSAFE), 9-18
 INTERRUPTCOUNTZERO
 [MACHINEOPS] intrinsic
 (UNSAFE), 9-18
 interruption protection, 8-17
 INTRINSIC declaration, 4-10
 < intrinsic declaration >, 4-10
 intrinsic identifiers, 4-10
 < intrinsic >, 9-1
 intrinsics (UNSAFE), 9-11
 introduction to NEWP, 1-1
 IRW, (See Indirect Reference Word (IRW))
 < iteration part >, 5-8

L

LABEL declaration, 4-14
 < label declaration >, 4-14
 label designator, 6-2
 < label designator >, 6-2

LABELS option, 8-10
 < length >, 9-7
 < level 2 procedure heading >, 2-1
 < level 2 procedure type >, 2-1
 < lex level >, 9-1
 LEXLEVEL [MISC] intrinsic (UNSAFE),
 9-18
 LEXOFFSET [MISC] intrinsic (UNSAFE),
 9-19
 < library attribute specs >, 4-14
 LIBRARY declaration, 4-14
 library declaration limit, 4-14
 < library declaration >, 4-14
 < linkage class >, 4-8
 Linked List Lookup (LLLU) operator, 9-19
 LIST compiler control option, 8-4
 LISTLOOKUP [MACHINEOPS] intrinsic
 (UNSAFE), 9-19
 LISTLOOKUP function, 6-2
 < listlookup function >, 6-2
 LISTO compiler control option, 8-4
 LIST1 compiler control option, 8-4
 Load Protected Object Word (LPOW)
 operator, 9-19
 LOADEVENT [MACHINEOPS] intrinsic
 (UNSAFE), 9-19
 LOCK function, 6-13
 < lock function >, 6-13
 LOCKSTATUS function, 6-14
 < lockstatus function >, 6-14
 LOWER BOUND function, 6-16
 < lower bound function >, 6-16
 < lower bound list >, 4-26
 < lower bound >, 4-2
 < lower bounds >, 4-3

M

Make Program Control Word (MPCW)
 operator, 9-5
 MAKEHOST compiler control option, 8-5
 MAKEPCW [MACHINEOPS] intrinsic
 (UNSAFE), 9-19
 MAPPING function, 6-16
 < mapping function >, 6-16
 Mark Stack Control Word (MSCW), 8-9,
 9-19
 MCP compiler control option, 8-5
 MEMIMAGEBOUND block directive, 8-16
 MEMORY [MEMORY] intrinsic (UNSAFE),
 9-20

Index

MERGE compiler control option, 8-5

MLSaccept statement, 5-2

<MLSaccept statement>, 5-2

MODSTRICT compiler control option, 8-5

<module body>, 4-15

<module body> (old), 4-18C

MODULE declaration, 4-15

restrictions, 4-16

MODULE declaration (old), 4-18C

<module declaration>, 4-15

<module declaration> (old), 4-18C

module export, 4-16

<module head>, 4-15

<module head> (old), 4-18C

<module identifier>, 4-15

<module identifier> (old), 4-18C

module import, 4-17

<moduleexport declaration>, 4-16

<moduleimport declaration>, 4-16

Move Stack (MVST) operator, 9-20

MOVESTACK [MACHINEOPS] intrinsic
(UNSAFE), 9-20

MSCW, (See Mark Stack Control Word
(MSCW))

multidimensional array, 4-2

N

<name part>, 4-25

nested module, 4-16

NEW compiler control option, 8-6

NEWP program essentials, 2-1

NOBINDINFO option, 8-10

NOCOUNT compiler control option, 8-6

<nonnegative constant integer expression>,
4-36

NORANGECHECK block directive, 8-16

NORMALIZE function, 6-2

<normalize function>, 6-2

NORMALSTATE block directive, 8-16

NOSWAP option, 9-8

NOTDSABLE option, 9-8

<number list>, 5-8

<number of digits>, 6-10

numbers, 3-1

numeric constants, 3-2

O

<octal character>, 3-3, 3-4

<octal constant>, 3-3

ON declaration, 4-22

<on declaration>, 4-22

ON statement, 5-2

<on statement>, 5-2

one-dimensional array, 4-2

<open statement>, 5-2

<operand to be converted>, 6-10

<option expression>, 8-1

<option primary>, 8-1

<ordinal>, 6-8

OUTPUTMESSAGE ARRAY declaration,
4-23

<outputmessage array declaration>, 4-23

OVERWRITE option (UNSAFE), 9-7

P

PACKDECIMAL function, 6-16

<packed-decimal result>, 6-10

parameter passing, 4-26

PAUSE [MACHINEOPS] intrinsic
(UNSAFE), 9-20

Pause Until Interrupt (PAUS) operator, 9-28

PBITS option, 8-10

PCW, (See Program Control Word (PCW))

POINTER [DESCRIPTOR or WORD]
intrinsic (UNSAFE), 9-21

POINTER declaration, 4-24

<pointer declaration>, 4-24

pointer expression, 6-6

Polynomial Recurrence (POLY) operator,
9-32

POP LIST, 7-4

precedence in Boolean expressions, 6-4

PREDECESSOR function, 6-18

<predecessor function>, 6-18

Primitive Display (SHOW) operator, 9-26

<procedure body>, 4-26

PROCEDURE declaration, 4-24

PROCEDURE declaration (UNSAFE), 9-3

<procedure declaration>, 4-24, 9-3

<procedure heading>, 4-24

<procedure identifier>, 5-5

PROCEDURE INVOCATION statement, 5-2

<procedure invocation statement>, 5-2

procedure reference array assignment, 5-6

- PROCEDURE REFERENCE ARRAY
 ASSIGNMENT statement, 5-6
 < procedure reference array assignment >, 5-6
- PROCEDURE REFERENCE ARRAY
 declaration, 4-1
 < procedure reference array declaration >, 4-1
 < procedure reference array element >, 5-6
 < procedure reference array specification >, 4-25
 procedure reference assignment, 5-5
- PROCEDURE REFERENCE ASSIGNMENT
 statement, 5-4
 < procedure reference assignment >, 5-5
- PROCEDURE REFERENCE declaration,
 4-34
 < procedure reference declaration >, 4-34
 < procedure reference identifier >, 5-6
- PROCEDURE REFERENCE statement,
 5-10
 < procedure reference statement >, 5-10
 < procedure specification >, 4-25
 < procedure type >, 4-24
 procedure value, 4-30
- PRCESS statement (UNSAFE), 9-6
- PROCREF compiler control option, 8-6
- product interfaces
 not implemented in NEWP, B-5
- Program Control Word (PCW), 4-6
- program unit, 2-1
- PROTECTED block directive, 8-17
- PROTECTED clause, 4-8
- Q**
- < quaternary character >, 3-3
 < quaternary constant >, 3-3
- R**
- Railroad diagrams, explanation of, C-1
- range checking, 4-41
 < range expression >, 6-8
- RANGECHECK block directive, 8-17
- RCW, (See Return Control Word (RCW))
- Read External Interrupt Identification
 (RDID) operator, 9-21
- Read External Memory Control (REMC)
 operator, 9-16, 9-21
- Read External Memory Table (REMT)
 operator, 9-22
- Read Internal Processor State (RIPS)
 operator, 9-21
- Read Machine Identification (WATI)
 operator, 9-34
- Read Processor Identification (WHOD)
 operator, 9-34
- READ statement, 5-2
 < read statement >, 5-2
- Read Time of Day (RTOD) operator, 9-22
- Read with Lock (RDLK) operator, 9-14
- READANDCLEAREXTERNALS
 [MACHINEOPS] intrinsic
 (UNSAFE), 9-21
- READLOCK compiler control option, 8-6
 with BUZZ intrinsics, 8-6
- READLOCKTIMEOUT compiler control
 option, 8-7
- READMEMORYCONTROL
 [MACHINEOPS] intrinsic
 (UNSAFE), 9-21
- READMEMORYTABLE [MACHINEOPS]
 intrinsic (UNSAFE), 9-22
- READPROCESSORSTATE
 [MACHINEOPS] intrinsic
 (UNSAFE), 9-21
- READTIMEOFDAY [MACHINEOPS]
 intrinsic (UNSAFE), 9-22
- REAL declaration, 4-1
 < real declaration >, 4-1
- REAL function, 6-2
 < real function >, 6-2
- Receive (RECV) operator, 9-22
- RECEIVEFROMREQUESTOR
 [MACHINEOPS] intrinsic
 (UNSAFE), 9-22
- REFERENCE TO [REFERENCE and
 WORD] intrinsic (UNSAFE), 9-22
- REGISTERS [REGISTERS] intrinsic
 (UNSAFE), 9-23
 < remote module declaration >, 4-16
- REPLACE POINTER-VALUED
 ATTRIBUTE statement, 5-2
 < replace pointer-valued attribute
 statement >, 5-2
- REPLACE statement, 5-11
- REPLACE statement (UNSAFE), 9-7
 < replace statement >, 5-11
- reserved words, A-1

Index

RESET statement, 5-2
< reset statement >, 5-2
RESETEVENT statement, 5-2
< resetevent statement >, 5-2
RESIZE statement, 5-2
< resize statement >, 5-2
RETURN [MACHINEOPS] intrinsic
(UNSAFE), 9-23
Return Control Word (RCW), 8-9
RETURNORIGINALS [MACHINEOPS]
intrinsic (UNSAFE), 9-23
Running Light (RUND) operator, 9-24
RUNNINGLIGHT [MACHINEOPS] intrinsic
(UNSAFE), 9-24

S

SAFE block directive, 8-17
SAVE ARRAY declaration (UNSAFE), 9-4
< save array declaration >, 9-4
< scalar type declaration >, 4-35
SCALAR TYPE declarations, 4-35
scalar type expression, 6-7
< scalar type expression >, 6-7
SCALAR TYPE function, 6-16
scalar types
in array references, 4-3
Scale Right Save (SCRS) operator, 9-24
SCALERRIGHTS [MACHINEOPS] intrinsic
(UNSAFE), 9-24
Scan In (SCNI) operator, 9-24
Scan Out (SCNO) operator, 9-25
SCANIN [MACHINEOPS] intrinsic
(UNSAFE), 9-24
SCANOUT [MACHINEOPS] intrinsic
(UNSAFE), 9-24
Scatter (SCAT) operator, 9-31
SCW, (See Software Control Word (SCW))
SEEK statement, 5-3
< seek statement >, 5-3
SEGMENT block directive, 8-17
SEGMENT declaration, 4-34
SEGMENT declaration (UNSAFE), 9-4
< segment declaration >, 4-34, 9-4
Segment Dictionary Index (SDI) value, 9-5
< segment equate >, 9-5
< segment identifier >, 4-34
< segment MPCWSDI >, 9-5
segmentation, 2-1
SEGMENTLEVEL block directive, 8-18
SELECT statement, 5-12
< select statement >, 5-12
Send to External Processing Element
(SEND) operator, 9-25
SENDTOREQUESTOR [MACHINEOPS]
intrinsic (UNSAFE), 9-25
SEPCOMP
background, 7-3
compiling with, 7-2
guidelines, 7-6
performing a, 7-4
SEPCOMP compiler control option, 8-7
SEPCOMP MERGE, 7-8
SEPCOMPLEVEL block directive, 7-3, 8-18
SEPCOMP MERGE compiler control option,
7-2, 8-7
Sequential Occurrence (SEQ) operator, 9-32
set assignment, 5-7
< set base type >, 4-42
set expression, 6-8
< set expression >, 6-8
< set identifier >, 4-42
Set Interval Timer (SINT) operator, 9-28A
SET LIST, 7-4
Set Memory Inhibits (SINH) operator, 9-25
Set Memory Limits (SLMT) operator, 9-25
< set operator >, 6-8
< set primary >, 6-8
set relation, 6-5
< set relation >, 6-4
SET statement, 5-3
< set statement >, 5-3
SET structure type variable, 4-43
SET TYPE declaration, 4-42
< set type declaration >, 4-42
SETACTUALNAME function (UNSAFE),
9-9
< setactualname function >, 9-9
SETEVENT statement, 5-3
< setevent statement >, 5-3
SETINHIBIT [MACHINEOPS] intrinsic
(UNSAFE), 9-25
SETLIMITS [MACHINEOPS] intrinsic
(UNSAFE), 9-25
SETTIMEOFDAY [MACHINEOPS] intrinsic
(UNSAFE), 9-26
< short enumerated variable identifier >, 6-8
< short scalar type expression >, 6-8
< short subtype variable identifier >, 6-8
SHOW [MACHINEOPS] intrinsic
(UNSAFE), 9-26
< SHOW source >, 9-26
sibling module, 4-16

- Signal Processor (SPES) operator, 9-27
SIGNALPROCESSOR [MACHINEOPS]
 intrinsic (UNSAFE), 9-27
 < simple descriptor expression >, 9-9
 < simple scalar type expression >, 6-7
 < simple set constructor >, 6-8
 < simple set expression >, 6-8
SIMPLE VARIABLE declaration, 4-35
 < simple variable declaration >, 4-35
 < simple word expression >, 9-10
SINGLE compiler control option, 8-7
SIZE function, 6-2
SIZE function (UNSAFE), 9-10
 < size function >, 6-2, 9-10
Software Control Word (SCW), 4-7
SOURCE input file, 7-5
 < specification >, 4-25
 < specifier >, 4-25
 stack cell, 4-10
STACK option, 8-10
Stack Overflow interrupt, 9-18
STANDALONE compiler control option, 8-2,
 8-8
 ALPHA, 8-9
 BETA, 8-9
 BETAINTERLEAVED, 8-9
 statements
 ARRAY REFERENCE ASSIGNMENT,
 5-4
 ASSIGNMENT, 5-3
 CASE, 5-8
 FOR, 5-8
 FORK (UNSAFE), 9-6
 FREEZE, 5-9
 not implemented in NEWP, B-2
 PROCEDURE REFERENCE, 5-10
 PROCEDURE REFERENCE ARRAY
 ASSIGNMENT, 5-6
 PROCEDURE REFERENCE
 ASSIGNMENT, 5-4
 PROCESS (UNSAFE), 9-6
 REPLACE, 5-11
 SELECT, 5-12
 SET ASSIGNMENT, 5-7
 SWAP, 5-12
 WAIT (UNSAFE), 9-8
 WAITANDRESET (UNSAFE), 9-8
STATISTICS compiler control option, 8-10
STATSUMMARY block directive, 8-11, 8-18
STOP [MACHINEOPS] intrinsic (UNSAFE),
 9-28
STOP77 [MACHINEOPS] intrinsic
 (UNSAFE), 9-28
 string concatenation
 implicit, 4-23
 < string concatenation operator >, 6-9
 string constants, 3-4
 string expression, 6-9
 < string expression >, 6-9
 < string primary >, 6-9
 < structure type declaration >, 4-35
STRUCTURE TYPE declarations, 4-35
STRUCTURE TYPE VARIABLE declaration,
 4-43
 < structure type variable declaration >, 4-43
 < subtype >, 4-39
 subtypes, 4-39
SUCCESSOR function, 6-18
 < successor function >, 6-18
SUPPLY declaration, 4-44
 < supply declaration >, 4-44
SUSPEND [MACHINEOPS] intrinsic
 (UNSAFE), 9-28
SWAP statement, 5-12
 < swap statement >, 5-12
SWAPNOW option, 9-8
SYSTEMCONTROL [MACHINEOPS]
 intrinsic (UNSAFE), 9-28
SYSTEMLIB attribute
 in **LIBRARY** declarations, 4-14
- ## T
- TADS** compiler control option, 8-11
TAPE input file, 7-5
 < target option > block directive, 8-18
TESTRAD [MACHINEOPS] intrinsic
 (UNSAFE), 9-28A
TESTWASD [MACHINEOPS] intrinsic
 (UNSAFE), 9-28A
 < timeout >, 6-13
TIMER [MACHINEOPS] intrinsic
 (UNSAFE), 9-28A
Top of Stack Control Word (TOSCW), 8-9
TOSCW, (See **Top of Stack Control Word**
(TOSCW))
 < transfer part >, 9-7
 of < replace statement >, 5-11

U

UNDERLINE compiler control option, 8-11
 < unit count >, 9-7
 UNLOCK function, 6-15
 < unlock function >, 6-15
 UNSAFE block directive, 8-19
 unsafe constructs permitted, 8-19
 untyped intrinsics, 9-29
 UPPER BOUND function, 6-19
 < upper bound function >, 6-19
 < upper bound >, 4-2

V

VALUE ARRAY declaration, 4-45
 < value array declaration >, 4-45
 < variable ordinal >, 6-8
 < variable range expression >, 6-8
 < variable set constructor >, 6-8
 VCHEK intrinsic, 9-32
 VDOT intrinsic, 9-31
 VDOTX intrinsic, 9-31
 VECTOR INTRINSICS [MACHINEOPS]
 (UNSAFE), 9-29
 VERSION compiler control option, 8-12
 < version option >, 8-12
 VFMN intrinsic, 9-33
 VFMX intrinsic, 9-33
 VFMXA intrinsic, 9-33
 VGATH intrinsic, 9-31
 VIA [REFERENCE] intrinsic (UNSAFE),
 9-33
 VOID compiler control option, 8-12
 < void option >, 8-12
 VPOLY intrinsic, 9-32
 VSCAT intrinsic, 9-31
 VSEQ intrinsic, 9-32
 VSUM intrinsic, 9-31
 VSUMA intrinsic, 9-31

W

WAIT statement, 5-3
 WAIT statement (UNSAFE), 9-8
 < wait statement >, 5-3
 WAITANDRESET statement, 5-3
 WAITANDRESET statement (UNSAFE),
 9-8

< waitandreset statement >, 5-3
 WHATAMI [MACHINEOPS] intrinsic
 (UNSAFE), 9-34
 WHOAMI [MACHINEOPS] intrinsic
 (UNSAFE), 9-34
 WORD [WORD] intrinsic (UNSAFE), 9-34
 < word case expression >, 9-11
 WORD declaration (UNSAFE), 9-5
 < word expression >, 9-10
 WORD expressions (UNSAFE), 9-10
 < word primary >, 9-10
 Write External Memory Control (WEMC)
 operator, 9-34
 Write External Memory Table (WEMT)
 operator, 9-35
 Write Internal Processor State (WIPS)
 operator, 9-35
 WRITE statement, 5-3
 < write statement >, 5-3
 Write Time of Day (WTOD) operator, 9-26
 WRITEMEMORYCONTROL
 [MACHINEOPS] intrinsic
 (UNSAFE), 9-34
 WRITEPROCESSORSTATE
 [MACHINEOPS] intrinsic
 (UNSAFE), 9-34
 WRITEXMEMORYTABLE [MACHINEOPS]
 intrinsic (UNSAFE), 9-35

X

XREF compiler control option, 8-12C
 XREFFILES compiler control option, 8-12C

Z

Zero Interrupt Count (ZIC) operator, 9-18

.LBOUND value, 4-3
 .TAG clause, 9-11
 .UBOUND value, 4-3
 \$ compiler control option, 8-12C



504423300000380