

UNISYS e-@ction APPLICATION DEVELOPMENT SOLUTIONS

ALGOL **Programming Reference Manual**

Volume 1 **Basic Implementation**

UNISYS e-@ction APPLICATION DEVELOPMENT SOLUTIONS

ALGOL **Programming Reference Manual**

Volume 1 **Basic Implementation**

The UNISYS logo is rendered in a bold, black, serif typeface. The letter 'I' in 'UNISYS' is unique, featuring a solid black dot positioned directly above it, which serves as a visual separator between the 'UNI' and 'SYS' portions of the brand name.

© 2001 Unisys Corporation.
All rights reserved.

ClearPath MCP 7.0

November 2001

Printed in USA
8600 0098-505

NO WARRANTIES OF ANY NATURE ARE EXTENDED BY THIS DOCUMENT. Any product or related information described herein is only furnished pursuant and subject to the terms and conditions of a duly executed agreement to purchase or lease equipment or to license software. The only warranties made by Unisys, if any, with respect to the products described in this document are set forth in such agreement. Unisys cannot accept any financial or other responsibility that may be the result of your use of the information in this document or software material, including direct, special, or consequential damages.

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

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

Notice to Government End Users: This is commercial computer software or hardware documentation developed at private expense. Use, reproduction, or disclosure by the Government is subject to the terms of Unisys standard commercial license for the products, and where applicable, the restricted/limited rights provisions of the contract data rights clauses.

Correspondence regarding this publication can be e-mailed to **doc@unisys.com**.

Unisys e-@ction
Application Development
Solutions

ALGOL

**Programming Reference
Manual**

**Volume 1
Basic Implementation**

ClearPath MCP 7.0

Unisys e-@ction
Application
Development
Solutions

ALGOL

**Programming
Reference
Manual**

**Volume 1
Basic
Implementation**

**ClearPath
MCP 7.0**

8600 0098-505

8600 0098-505

Bend here, peel upwards and apply to spine.

Contents

Section 1. Program Structure

About This Manual	1-1
Purpose	1-1
Audience	1-1
Notation Conventions	1-1
Overview of the Language	1-2
Program Unit	1-2
Elements of an ALGOL Program	1-3
Scope of an Identifier	1-5
Local Identifiers	1-6
Global Identifiers	1-6

Section 2. Language Components

Basic Symbol	2-2
Identifier	2-5
Number	2-6
Number Ranges	2-8
Compiler Number Conversion	2-9
Exponents	2-9
Remark	2-10
String Literal	2-12
Character Size	2-15
String Code	2-15
String Length	2-16
ASCII Strings	2-16
Quotation Mark	2-16
Dollar Sign	2-16

Section 3. Declarations

ALPHA Declaration	3-2
ARRAY Declaration	3-3
PRIVATE and PUBLIC Specifiers	3-3
LONG Arrays	3-3
OWN Arrays	3-4
Identifiers	3-4
Array Class	3-5
Bound Pair List	3-6
Original and Referred Arrays	3-7

Contents

Dimensionality	3-7
Array Row Equivalence.....	3-8
Array Row	3-9
Row Selector	3-10
Examples of ARRAY Declarations	3-10
ARRAY REFERENCE Declaration.....	3-12
Identifiers	3-12
Lower Bounds	3-13
Examples of ARRAY REFERENCE Declarations	3-13
BOOLEAN Declaration	3-14
Equation Part	3-14
Boolean Simple Variable Values	3-15
Examples of BOOLEAN Declarations.....	3-15
COMPLEX Declaration	3-16
Complex Variables	3-16
Examples of COMPLEX Declarations.....	3-17
CONNECTION BLOCK REFERENCE VARIABLE Declaration.....	3-18
CONNECTION BLOCK TYPE Declaration	3-19
Examples of CONNECTION BLOCK TYPE Declaration	3-21
CONNECTION LIBRARY Declaration.....	3-24
Explanation	3-25
DEFINE Declaration	3-26
Formal Symbol Part	3-27
Define Invocation.....	3-27
Examples of DEFINE Declarations	3-32
DIRECT ARRAY Declaration.....	3-33
Declaring Direct Arrays.....	3-34
Examples of DIRECT ARRAY Declarations	3-35
DOUBLE Declaration	3-36
Declaration of Simple Variables	3-36
Examples of DOUBLE Declarations	3-36
DUMP Declaration	3-37
Control Part.....	3-38
Label Identifier	3-38
Label Identifier with Label Counter Modulus.....	3-39
Label Identifier with Dump Parameters.....	3-39
Label Identifiers with Label Counter Modulus and Dump Parameters.....	3-39
Form of Output.....	3-40
Examples of DUMP Declarations	3-41
EPILOG PROCEDURE Declaration	3-42
Restrictions on Epilog Procedures.....	3-42
Example of an Epilog Procedure.....	3-44
EVENT and EVENT ARRAY Declarations	3-45
Event Designator	3-46
Examples of EVENT and EVENT ARRAY Declarations.....	3-47
EXCEPTION PROCEDURE Declaration	3-48
Restrictions on Exception Procedures	3-49
Example of an EXCEPTION PROCEDURE Declaration	3-50

EXPORT Declaration.....	3-51
Library Entry Point Types and Parameters.....	3-53
Conditions in Which Errors Can Occur	3-54
Examples of EXPORT Declarations	3-55
EXPORTLIBRARY Declaration.....	3-56
FILE Declaration	3-57
Identifiers	3-57
Attribute Specifications.....	3-57
Examples of FILE Declarations	3-59
FORMAT Declaration	3-60
In-Out Part.....	3-60
Format Part.....	3-60
Simple String Literal.....	3-62
Repeat Part.....	3-63
Editing Phrases	3-64
Variable Editing Phrases.....	3-65
Editing Phrase Letters	3-66
A and C Editing Phrase Letters	3-66
D Editing Phrase Letter	3-68
E Editing Phrase Letter.....	3-70
F Editing Phrase Letter.....	3-71
G Editing Phrase Letter	3-71
H and K Editing Phrase Letters	3-72
I Editing Phrase Letter.....	3-75
J Editing Phrase Letter	3-76
L Editing Phrase Letter.....	3-77
O Editing Phrase Letter	3-78
R Editing Phrase Letter.....	3-79
S Editing Phrase Letter.....	3-80
T Editing Phrase Letter.....	3-82
U Editing Phrase Letter	3-82
V Editing Phrase Letter.....	3-83
X Editing Phrase Letter.....	3-84
Z Editing Phrase Letter.....	3-85
Editing Modifiers	3-86
P Editing Modifier	3-86
\$ Editing Modifier	3-86
Examples of FORMAT Declarations	3-86
FORWARD REFERENCE Declaration	3-87
Order of Referencing	3-88
Examples of FORWARD REFERENCE Declarations	3-89
IMPORTED Declaration.....	3-90
Examples of IMPORTED Declaration	3-91
INTEGER Declaration	3-92
Equation Part.....	3-92
Examples of INTEGER Declarations	3-93
INTERLOCK and INTERLOCK ARRAY Declarations	3-94
Interlock Designator.....	3-95
INTERRUPT Declaration.....	3-96
Interrupting a Program.....	3-96
Examples of INTERRUPT Declarations.....	3-97
LABEL Declaration	3-98

Contents

Using Label Identifiers	3-98
Examples of LABEL Declarations	3-98
LIBRARY Declaration	3-99
Library Attribute Specifications.....	3-100
Examples of LIBRARY Declarations	3-103
LIST Declaration	3-104
List Elements.....	3-104
Examples of LIST Declarations.....	3-105
MONITOR Declaration	3-106
Monitor Elements	3-106
Monitor Element as a Simple Variable.....	3-107
Monitor Element as a Label Identifier.....	3-108
Monitor Element as an Array Identifier	3-108
Examples of MONITOR Declarations	3-109
OUTPUTMESSAGE ARRAY Declaration.....	3-110
Output Message.....	3-111
Translators' Help Text.....	3-113
Examples of OUTPUTMESSAGE ARRAY Declarations.....	3-113
PENDING PROCEDURE Declaration	3-115
PENDING PROCEDURE ACTUAL Declaration	3-116
PICTURE Declaration	3-117
String Literals.....	3-118
Introduction.....	3-118
Introduction Codes	3-119
Characters Used by Picture Symbols	3-120
Flip-Flops Used by Picture Symbols	3-120
Character Fields	3-121
Picture Skip Characters	3-121
Control Characters.....	3-122
Single Picture Characters	3-122
Picture Characters	3-123
Examples of PICTURE Declarations	3-125
POINTER Declaration	3-128
OWN Pointers.....	3-128
Lex Level Restriction Part.....	3-129
Examples of POINTER Declarations.....	3-130
PROCEDURE Declaration	3-132
Identifiers	3-133
Formal Parameter Part.....	3-133
Specification	3-135
Procedure Reference Array Specification	3-137
Structure Block Array Specification	3-138
Procedure Body	3-138
Dynamic Procedure Specification	3-138
Library Entry Point Specification	3-139
ISOLATED Procedure Specification.....	3-139
Allowed Formal and Actual Parameters	3-141
Parameter Matching	3-141
Array Parameters	3-141
Procedure Reference Array Parameters	3-143
Procedure Parameters	3-144

Simple Variable Parameters	3-145
String Parameters.....	3-146
File Parameters	3-146
Other Types of Parameters.....	3-147
Examples of PROCEDURE Declarations	3-148
PROCEDURE REFERENCE Declaration.....	3-150
Identifiers	3-150
Example of a PROCEDURE REFERENCE Declaration.....	3-150
PROCEDURE REFERENCE ARRAY Declaration.....	3-151
Placement of Procedure Reference Arrays	3-152
Example of a PROCEDURE REFERENCE ARRAY Declaration.....	3-153
PROLOG PROCEDURE Declaration.....	3-154
REAL Declaration	3-155
Declaration of Simple Variables	3-155
Examples of REAL Declarations	3-156
SIMPLE VARIABLE Declaration	3-157
STRING Declaration.....	3-158
STRING Type	3-158
Examples of STRING Declarations	3-159
STRING ARRAY Declaration.....	3-160
String Array Type	3-160
Examples of STRING ARRAY Declarations.....	3-160
STRUCTURE BLOCK ARRAY Declaration.....	3-161
STRUCTURE BLOCK REFERENCE VARIABLE Declaration.....	3-163
STRUCTURE BLOCK TYPE Declaration	3-164
STRUCTURE BLOCK VARIABLE Declaration.....	3-167
SWITCH FILE Declaration	3-168
Switch File List.....	3-168
Example of a SWITCH FILE Declaration	3-169
SWITCH FORMAT Declaration	3-170
Switch Format List.....	3-170
Examples of SWITCH FORMAT Declarations	3-171
SWITCH LABEL Declaration.....	3-172
Switch Label List.....	3-172
Examples of SWITCH LABEL Declarations	3-173
SWITCH LIST Declaration	3-174
List Designator	3-174
Example of a SWITCH LIST Declaration	3-175
TASK and TASK ARRAY Declarations	3-176
Task and Task Array Designator	3-176
Examples of TASK and TASK ARRAY Declarations	3-177
TRANSLATETABLE Declaration	3-178
Translation Specifier.....	3-178
Translate Table Indexing	3-179
Examples of TRANSLATETABLE Declarations	3-181
TRUTHSET Declaration	3-182
Membership Expression	3-182
Truth Set Test	3-183
Examples of TRUTHSET Declarations	3-184
VALUE ARRAY Declaration	3-186

Contents

Constants.....	3-186
Example of a VALUE ARRAY Declaration	3-187

Section 4. Statements

ACCEPT Statement.....	4-2
ACCEPT Parameters.....	4-2
Examples of ACCEPT Statements.....	4-3
ASSIGNMENT Statement.....	4-4
Arithmetic Assignment.....	4-5
Arithmetic Variable.....	4-5
Arithmetic Type Transfer Variable.....	4-6
Arithmetic Attribute	4-7
Arithmetic Update Assignment.....	4-8
Examples of an Arithmetic Assignment	4-9
Array Reference Assignment	4-9
Array Reference Variable	4-10
Array Designator	4-10
Examples of Array Reference Assignments.....	4-11
Boolean Assignment	4-12
Boolean Variables.....	4-12
Boolean Attributes	4-13
Boolean Update Assignment	4-14
Examples of Boolean Assignments.....	4-14
Complex Assignment	4-15
Complex Update Assignment.....	4-15
Examples of Complex Assignments.....	4-15
Connection Block Reference Assignment.....	4-16
Mnemonic Attribute Assignment	4-16
Pointer Assignment	4-17
Pointer Variable	4-17
Examples of Pointer Assignments.....	4-17
Procedure Reference Assignment	4-18
Procedure Reference Variable	4-18
Example of a Procedure Reference Assignment.....	4-19
String Assignment.....	4-20
String Concatenation Operator	4-20
Examples of String Assignments.....	4-21
Structure Block Reference Assignment	4-21
Task Assignment	4-22
ATTACH Statement	4-23
Attachment of Interrupts	4-23
Examples of ATTACH Statements	4-23
AWAITOPEN Statement	4-24
PARTICIPATE Option.....	4-25
CONNECTTIMELIMIT Option.....	4-25
Examples of AWAITOPEN Statements.....	4-25
CALL Statement.....	4-26
Coroutines	4-26
Example of a CALL Statement	4-27

CANCEL Statement.....	4-28
Delinking a Library from a Program	4-28
Example of a CANCEL Statement	4-28
CASE Statement	4-29
Unnumbered Statement List	4-29
Numbered Statement List	4-30
Examples of CASE Statements	4-30
CAUSE Statement.....	4-31
Causes of Events	4-31
Examples of CAUSE Statements.....	4-31
CAUSEANDRESET Statement	4-32
Relationship to CAUSE Statement	4-32
Examples of CAUSEANDRESET Statements	4-32
CHANGEFILE Statement.....	4-33
Directory Element	4-34
Example of a CHANGEFILE Statement	4-35
CHECKPOINT Statement	4-36
Disposition Option	4-36
Restarting a Job	4-37
Locking.....	4-41
Rerunning Programs	4-41
Example of a CHECKPOINT Statement.....	4-41
CLOSE Statement	4-42
CLOSE Options	4-43
PORT CLOSE Option	4-44
Examples of CLOSE Statements.....	4-45
CONTINUE Statement	4-47
Coroutines.....	4-47
Examples of CONTINUE Statements	4-47
DEALLOCATE Statement.....	4-48
Deallocation with Arrays	4-48
Examples of DEALLOCATE Statements	4-48
DETACH Statement	4-49
Detaching Interrupts	4-49
Example of a DETACH Statement	4-49
DISABLE Statement.....	4-50
Disabling Interrupts.....	4-50
Examples of DISABLE Statements.....	4-50
DISPLAY Statement	4-51
Pointer and String Expressions	4-51
Examples of DISPLAY Statements.....	4-51
DO Statement	4-52
Evaluation of Boolean Expression.....	4-52
Examples of DO Statements	4-52
ENABLE Statement.....	4-53
Enabling Interrupts.....	4-53
Examples of ENABLE Statements.....	4-53
ERASE Statement	4-54
EVENT Statement	4-55
EXCHANGE Statement	4-56
Conditions for Execution of the EXCHANGE Statement.....	4-56

Contents

Examples of EXCHANGE Statements	4-57
FILL Statement	4-58
Initialization	4-58
Examples of FILL Statements	4-59
FIX Statement	4-60
FIX Statement as a Boolean Function	4-60
Examples of FIX Statements	4-60
FOR Statement	4-61
Forms of the FOR Statement	4-62
FOR-DO Loop	4-62
FOR-STEP-UNTIL Loop	4-63
FOR-STEP-WHILE Loop	4-64
FOR-WHILE Loop	4-65
Examples of FOR Statements	4-66
FREE Statement	4-67
FREE Statement as a Boolean Function	4-67
Examples of FREE Statements	4-67
FREEZE Statement	4-68
FREEZE Statements in Library Procedures	4-68
Examples of FREEZE Statement	4-69
GO TO Statement	4-70
Bad GO TO	4-70
Examples of GO TO Statements	4-70
I/O Statement	4-71
Normal I/O	4-71
Direct I/O	4-72
IF Statement	4-74
Forms of the IF Statement	4-74
Examples of IF Statements	4-75
INTERRUPT Statement	4-76
INVOCATION Statement	4-77
LIBERATE Statement	4-78
Execution of Implicit CAUSE Statement	4-78
Examples of LIBERATE Statements	4-78
LOCK File Statement	4-79
Lock Options	4-79
Examples of LOCK File Statements	4-79
LOCK Interlock Statement	4-80
Timeout Option	4-81
Examples of LOCK Interlock Statements	4-81
MERGE Statement	4-82
Merge Options	4-82
Example of a MERGE Statement	4-82
MESSAGESEARCHER Statement	4-83
Finding a Requested Message	4-84
MESSAGESEARCHER Statement as an Arithmetic Function	4-84
Example of a MESSAGESEARCHER Statement	4-85
MLSACCEPT Statement	4-86
MLSACCEPT Used for Data Input	4-86
MLSACCEPT Used as a Boolean Function	4-86
Additional MLSACCEPT Options	4-86

Example of an MLSACCEPT Statement	4-87
MLSDISPLAY Statement	4-88
MLSTRANSLATE Statement.....	4-89
MLSTRANSLATE Options.....	4-90
MLSTRANSLATE as an Arithmetic Function	4-91
Example of an MLSTRANSLATE Statement	4-92
MULTIPLE ATTRIBUTE ASSIGNMENT Statement.....	4-93
Assignment of Values	4-93
Examples of MULTIPLE ATTRIBUTE ASSIGNMENT Statements	4-93
ON Statement	4-94
Enabling ON Statements	4-94
Fault List.....	4-95
Fault Information Part	4-96
Fault Stack History	4-97
Fault Action	4-98
Disabling ON Statement	4-99
Examples of ON Statements	4-99
OPEN Statement.....	4-101
OPEN Options.....	4-101
Examples of OPEN Statements.....	4-102
POINTER Statement	4-104
POINTER Statement Options	4-104
Temporary Storage	4-104
Stack-Source-Pointer.....	4-105
Stack-Destination-Pointer.....	4-105
Stack-Auxiliary-Pointer.....	4-105
Stack-Integer-Counter	4-106
Stack-Test-Character	4-106
Stack-Source-Operand	4-106
PROCEDURE INVOCATION Statement.....	4-107
Calling Procedures with Parameters.....	4-108
Examples of PROCEDURE INVOCATION Statements	4-109
PROCEDURE REFERENCE Statement	4-110
Using Procedure References.....	4-110
Example of a PROCEDURE REFERENCE Statement.....	4-111
PROCESS Statement	4-112
Initiation of an Asynchronous Process.....	4-112
Critical Block	4-112
Examples of PROCESS Statements	4-113
PROCURE Statement.....	4-114
Testing the Available State	4-114
Sharing Resources Among Programs.....	4-114
Examples of PROCURE Statements	4-114
PROGRAMDUMP Statement.....	4-115
PROGRAMDUMP Options	4-115
Programdump Destination Options	4-118
Relation to OPTION Task Attribute.....	4-118
Retrieval of Binding Information	4-119
Examples of PROGRAMDUMP Statements	4-119

Contents

READ Statement.....	4-121
File Part.....	4-121
I/O Option or Carriage Control.....	4-122
Subfile Specification.....	4-123
Core-to-Core Part.....	4-124
Core-to-Core Blocking Part.....	4-124
Format and List Part.....	4-126
Formatted Read.....	4-126
Binary Read.....	4-127
Array Row Read.....	4-128
Action Labels or Finished Event.....	4-129
Data Format for Free-Field Input.....	4-130
Free-Field Data Format.....	4-130
Fields.....	4-131
Unquoted String.....	4-131
Number.....	4-131
Quoted String.....	4-132
Hex String.....	4-132
Slash (/).....	4-132
Asterisk (*).....	4-132
Examples of Fields.....	4-133
Examples of READ Statements.....	4-134
REMOVEFILE Statement.....	4-135
Directory Element.....	4-135
REMOVEFILE Statement as a Boolean Function.....	4-135
Family Substitution.....	4-135
Example of a REMOVEFILE Statement.....	4-136
REPLACE Statement.....	4-137
Source Part List.....	4-138
Source Part Combinations.....	4-140
String Literal Source Parts.....	4-141
<string literal>.....	4-142
<string literal> FOR <arithmetic expression>.....	4-143
<string literal> FOR <arithmetic expression> WORDS.....	4-144
Arithmetic Expression Source Parts.....	4-146
<arithmetic expression>.....	4-146
<arithmetic expression> FOR <arithmetic expression>.....	4-147
<arithmetic expression> FOR <arithmetic expression> WORDS.....	4-148
<arithmetic expression> FOR <arithmetic expression> DIGITS.....	4-149
<arithmetic expression> FOR * DIGITS.....	4-150
<arithmetic expression> FOR <arithmetic expression> SDIGITS.....	4-150
<arithmetic expression> FOR * SDIGITS.....	4-151
<arithmetic expression> FOR <count part> NUMERIC.....	4-152
<arithmetic expression> FOR * NUMERIC.....	4-153
Pointer Expression (<source>) Source Parts.....	4-153
<source> FOR <arithmetic expression>.....	4-153

<source> FOR <arithmetic expression> WORDS	4-154
<source> FOR <arithmetic expression> WITH <translate table>	4-154
<intrinsic translate table>	4-155
<translate table identifier>	4-155
<subscripted variable>	4-155
<source> WITH <picture identifier>	4-156
Source Parts with Boolean Conditions	4-156
<source> WHILE <relational operator> <arithmetic expression>	4-157
<source> UNTIL <relational operator> <arithmetic expression>	4-157
<source> WHILE IN <truth set table>	4-158
<source> UNTIL IN <truth set table>	4-158
<source> FOR <count part> WHILE <relational operator> <arithmetic expression>	4-158
<source> FOR <count part> UNTIL <relational operator> <arithmetic expression>	4-159
<source> FOR <count part> WHILE IN <truth set table>	4-159
<source> FOR <count part> UNTIL IN <truth set table>	4-160
Other Source Parts	4-160
<pointer-valued attribute>	4-160
<string expression>	4-161
Examples of REPLACE Statements	4-161
REPLACE FAMILY-CHANGE Statement	4-163
Specification of Valid Stations	4-163
Examples of REPLACE FAMILY-CHANGE Statements	4-164
REPLACE POINTER-VALUED ATTRIBUTE Statement	4-165
Specification of the Simple Source	4-166
Examples of REPLACE POINTER-VALUED ATTRIBUTE Statements	4-167
RESET Statement	4-168
WAIT and WAITANDRESET Statements	4-168
Examples of RESET Statements	4-168
RESIZE Statement	4-169
Array Row Resize Parameters	4-169
Special Array Resize Parameters	4-172
Multidimensional Array Designator	4-173
Event Array Designator	4-173
String Array Designator	4-173
Interlock Array Designator	4-173
Run-Time Error Messages	4-174
Examples of RESIZE Statements	4-175
RESPOND Statement	4-176
RESPOND Statement Options	4-176
Examples of RESPOND Statements	4-177

Contents

REWIND Statement.....	4-178
Effects on Designated Files.....	4-178
Example of a REWIND Statement.....	4-178
RUN Statement.....	4-179
Initiating Procedures.....	4-179
Examples of a RUN Statement.....	4-180
SCAN Statement.....	4-181
Scan Part Combinations.....	4-182
Scan Parts without Count Parts.....	4-182
WHILE <relational operator> <arithmetic expression>.....	4-182
UNTIL <relational operator> <arithmetic expression>.....	4-182
WHILE IN <truth set table>.....	4-183
UNTIL IN <truth set table>.....	4-183
Scan Parts with Count Parts.....	4-183
FOR <count part> WHILE <relational operator> <arithmetic expression>.....	4-183
FOR <count part> UNTIL <relational operator> <arithmetic expression>.....	4-184
FOR <count part> WHILE IN <truth set table>.....	4-184
FOR <count part> UNTIL IN <truth set table>.....	4-184
Examples of SCAN Statements.....	4-185
SEEK Statement.....	4-186
SEEK Statement as a Boolean Function.....	4-186
Example of a SEEK Statement.....	4-186
SET Statement.....	4-187
SET Statement Options.....	4-187
Examples of SET Statements.....	4-187
SETABSTRACTVALUE Statement.....	4-188
Setting the File Attribute.....	4-188
Examples of SETABSTRACTVALUE Statements.....	4-188
SORT Statement.....	4-189
Output Option.....	4-189
Input Option.....	4-190
Number of Tapes.....	4-191
Compare Procedure.....	4-191
Record Length.....	4-192
Size Specifications.....	4-193
Restart Specifications.....	4-194
Arrays in Sort Procedures.....	4-196
Examples of SORT Statements.....	4-196
SPACE Statement.....	4-197
SPACE Statement as a Boolean Function.....	4-197
Examples of SPACE Statements.....	4-197
SWAP Statement.....	4-198
Variable Type Matching.....	4-198
Example of a SWAP Statement.....	4-199
THRU Statement.....	4-200
Value of the Arithmetic Expression.....	4-200
Examples of THRU Statements.....	4-200

TRY Statement	4-201
UNLOCK Statement	4-203
Examples of UNLOCK Statements	4-203
WAIT Statement.....	4-204
Wait Option List	4-205
WAIT Parameter List.....	4-205
Start Index.....	4-206
Direct Array Row.....	4-206
Examples of WAIT Statements	4-207
WAITANDRESET Statement.....	4-209
WAITANDRESET Statement as an Arithmetic Function	4-209
Examples of WAITANDRESET Statements.....	4-210
WHEN Statement.....	4-212
Characteristics of the Time Option	4-212
Examples of WHEN Statements	4-212
WHILE Statement	4-213
Execution of the WHILE Statement	4-213
Examples of WHILE Statements	4-214
WRITE Statement	4-215
I/O Option or Carriage Control	4-216
Write Subfile Specification.....	4-217
Format and List Part.....	4-218
Formatted Write	4-219
Binary Write.....	4-219
Array Row Write.....	4-220
Free-Field Part	4-221
Example of a Free-Field Part	4-221
Action Labels or Finished Event	4-222
Examples of WRITE Statements	4-222
ZIP Statement	4-224
ZIP WITH <array row>	4-224
ZIP WITH <file designator>	4-225
Examples of ZIP Statements	4-225

Section 5. Expressions and Functions

Expressions	5-1
Arithmetic Expression.....	5-2
Precision of Arithmetic Expressions	5-3
Arithmetic Operators.....	5-3
Precedence of Arithmetic Operators.....	5-4
Types of Resulting Values	5-6
Arithmetic Primaries.....	5-7
Bit Manipulation Expression	5-8
Concatenation Expression.....	5-8
Partial Word Expression	5-12
Boolean Expression	5-13
Operators in Boolean Expressions	5-14
Logical Operators	5-15
IS and ISNT Operators.....	5-15

Contents

Relational Operators.....	5-16
Precedence in Boolean Expressions.....	5-16
Boolean Primaries	5-17
Boolean Value.....	5-17
Arithmetic Relation.....	5-18
Complex Relation	5-18
String Relation	5-19
Pointer Relation.....	5-19
String Expression Relation	5-20
Table Membership	5-21
Case Expression	5-23
Complex Expression.....	5-24
Conditional Expression	5-26
Designational Expression	5-27
Function Expression	5-28
Arithmetic Function Designator	5-29
Boolean Function Designator.....	5-29
Complex Function Designator	5-29
Pointer Function Designator	5-30
String Function Designator	5-30
NULL Value.....	5-30
Pointer Expression.....	5-31
String Expression.....	5-34
TRY Expression.....	5-37
Intrinsic Functions	5-38
Intrinsic Names by Type Returned	5-38
Arithmetic Intrinsic Names	5-38
Boolean Intrinsic Names	5-42
Complex Intrinsic Names.....	5-42
Pointer Intrinsic Names.....	5-42
String Intrinsic Names.....	5-42
Intrinsic Function Descriptions	5-43
ABS Function	5-43
ACCEPT Statement.....	5-43
ARCCOS Function.....	5-43
ARCSIN Function	5-43
ARCTAN Function	5-43
ARCTAN2 Function	5-44
ARRAYSEARCH Function	5-44
ATANH Function	5-45
AVAILABLE Function	5-45
BOOLEAN Function	5-45
CABS Function	5-46
CCOS Function	5-46
CEXP Function	5-46
CHANGEFILE Statement	5-46
CHECKPOINT Statement.....	5-46
CHECKSUM Function	5-47
CLN Function	5-47
CLOSE Statement.....	5-47
COMPILETIME Function.....	5-48
COMPLEX Function	5-48

CONJUGATE Function	5-48
COS Function	5-48
COSH Function.....	5-49
COTAN Function	5-49
CSIN Function	5-49
CSQRT Function.....	5-49
DABS Function	5-49
DALPHA Function	5-49
DAND Function	5-50
DARCCOS Function	5-50
DARCSIN Function.....	5-50
DARCTAN Function.....	5-50
DARCTAN2 Function.....	5-51
DCOS Function.....	5-51
DCOSH Function	5-51
DECIMAL Function.....	5-51
DELINKLIBRARY Function	5-52
DELTA Function	5-53
DEQV Function.....	5-54
DERF Function	5-54
DERFC Function	5-54
DEXP Function	5-54
DGAMMA Function.....	5-54
DIMP Function	5-55
DINTEGER Function	5-55
DINTEGERT Function.....	5-56
DLGAMMA Function.....	5-56
DLN Function	5-56
DLOG Function.....	5-56
DMAX Function.....	5-57
DMIN Function	5-57
DNABS Function	5-57
DNORMALIZE Function	5-57
DNOT Function.....	5-57
DOR Function.....	5-58
DOUBLE Function	5-58
DROP Function.....	5-59
DSCALELEFT Function	5-60
DSCALERIGHT Function	5-60
DSCALERIGHTT Function	5-60
DSIN Function	5-61
DSINH Function.....	5-61
DSQRT Function.....	5-61
DTAN Function	5-61
DTANH Function	5-61
ENTIER Function	5-62
ERF Function	5-62
ERFC Function.....	5-63
EXP Function	5-63
FIRST Function	5-63
FIRSTONE Function	5-63
FIRSTWORD Function	5-64

Contents

FIX Statement	5-64
FREE Statement	5-64
GAMMA Function	5-64
HAPPENED Function	5-64
HEAD Function	5-65
IMAG Function	5-65
INTEGER Function	5-66
INTEGERT Function	5-67
ISVALID Function	5-67
LENGTH Function	5-68
LINENUMBER Function	5-68
LINKLIBRARY Function	5-69
LISTLOOKUP Function	5-73
LN Function	5-73
LNGAMMA Function	5-73
LOCK Interlock Function	5-73
LOCKSTATUS Function	5-74
LOG Function	5-75
MASKSEARCH Function	5-75
MAX Function	5-75
MESSAGESEARCHER Statement	5-75
MIN Function	5-75
MLSACCEPT Statement	5-75
MLSDISPLAY Statement	5-76
MLSTRANSLATE Statement	5-76
NABS Function	5-76
NORMALIZE Function	5-76
OFFSET Function	5-76
ONES Function	5-77
OPEN Statement	5-77
POINTER Function	5-77
POT Function	5-79
PROCESSID Function	5-79
RANDOM Function	5-80
READ Statement	5-80
READLOCK Function	5-80
READYCL Function	5-81
REAL Function	5-82
REMAININGCHARS Function	5-83
REMOVEFILE Statement	5-83
REPEAT Function	5-83
SCALELEFT Function	5-84
SCALERIGHT Function	5-84
SCALERIGHTF Function	5-84
SCALERIGHTT Function	5-85
SECONDWORD Function	5-85
SEEK Statement	5-85
SETACTUALNAME Function	5-86
SIGN Function	5-87
SIN Function	5-87
SINGLE Function	5-88
SINH Function	5-88

SIZE Function	5-88
SPACE Statement	5-89
SQRT Function	5-89
STRING Function.....	5-89
TAIL Function	5-91
TAKE Function.....	5-91
TAN Function.....	5-92
TANH Function	5-92
THIS Function.....	5-92
THISCL Function	5-93
TIME Function	5-93
TRANSLATE Function	5-100
UNLOCK Statement.....	5-100
UNREADYCL Function.....	5-100
USERDATA Function.....	5-101
USERDATALOCATOR Function.....	5-102
USERDATAREBUILD Function	5-102
VALUE Function	5-103
WAIT Statement.....	5-104
WAITANDRESET Statement.....	5-104
WRITE Statement	5-104

Section 6. Compiling Programs

Files Used by the Compiler	6-1
Input Files.....	6-3
CARD File	6-3
SOURCE File	6-4
INCLUDE Files.....	6-4
HOST File	6-4
INFO File	6-4
Output Files	6-5
CODE File.....	6-5
NEWSOURCE File.....	6-5
LINE File	6-6
ERRORS File	6-6
XREFFILE File.....	6-7
INFO File	6-7
Source Record Format	6-8
Compiler Control Options.....	6-9
Compiler Control Records.....	6-10
Option Descriptions	6-16
ASCII Option.....	6-16
AUTOBIND Option	6-16
BEGINSEGMENT Option	6-18
BIND Option	6-19
BINDER Option	6-19
BINDER_MATCH Option.....	6-19
CHECK Option.....	6-20
CODE Option.....	6-20
CODE_SUFFIX Option.....	6-20

Contents

DONTBIND Option.....	6-21
DUMPINFO Option	6-21
ENDSEGMENT Option.....	6-22
ERRLIST Option	6-22
EXTERNAL Option	6-22
FORMAT Option	6-22
GO TO Option	6-23
HOST Option.....	6-23
INCLNEW Option.....	6-23
INCLSEQ Option	6-24
INCLUDE Option.....	6-24
INITIALIZE Option	6-26
INSTALLATION Option	6-26
INTRINSICS Option.....	6-27
LARGE_LINEINFO Option.....	6-27
LEVEL Option.....	6-28
LIBRARY Option	6-29
LIMIT Option or ERRORLIMIT Option	6-29
LINEINFO Option	6-29
LIST Option	6-30
LISTDELETED Option	6-30
LISTDOLLAR Option.....	6-30
LISTINCL Option	6-31
LISTOMITTED Option	6-31
LISTP Option.....	6-31
LI_SUFFIX Option	6-32
LOADINFO Option	6-32
MAKEHOST Option	6-34
MCP Option	6-36
MERGE Option	6-37
NEW Option.....	6-37
NEWSEQERR Option.....	6-38
NOBINDINFO Option.....	6-38
NOSTACKARRAYS Option.....	6-38
NOXREFLIST Option.....	6-39
OMIT Option	6-39
OPTIMIZE Option.....	6-40
PAGE Option.....	6-40
PARAMCHECK Option.....	6-41
PURGE Option	6-41
SEGDESCABOVE Option.....	6-41
SEGS Option	6-42
SEPCOMP Option.....	6-42
SEQ Option	6-44
SEQERR Option	6-44
SHARING Option	6-45
DONTCARE	6-45
PRIVATE	6-45
SHARED BY ALL	6-45
SHARED BY RUN UNIT	6-45
SINGLE Option.....	6-46
STACK Option	6-46

STATISTICS Option	6-46
STOP Option	6-47
TADS Option	6-48
TARGET Option	6-49
TIME Option	6-50
USE Option.....	6-50
USER Option	6-50
VERSION Option	6-51
VOID Option	6-52
VOIDT Option	6-52
WAITIMPORT Option.....	6-53
WARNSUPR Option	6-53
WRITEAFTER Option	6-53
XDECS Option	6-54
XREF Option.....	6-54
XREFFILES Option	6-56
XREFS Option.....	6-56

Section 7. Compile-Time Facility

Compile-Time Variable.....	7-2
Compile-Time Identifier	7-3
Compile-Time Statements.....	7-3
BEGIN Statement	7-4
DEFINE Statement.....	7-4
FOR Statement	7-5
IF Statement	7-5
INVOKE Statement	7-6
LET Statement	7-6
THRU Statement.....	7-6
WHILE Statement.....	7-6
Extension to the Define Declaration	7-7
Compile-Time Compiler Control Options	7-8
CTLIST Option.....	7-8
CTMON Option	7-8
CTTRACE Option	7-8
LISTSKIP Option	7-9

Section 8. Library Facility

Operating the Components of the Library Facility	8-2
Library Programs	8-2
Calling Programs	8-2
Library Directories and Templates	8-2
Library Initiation.....	8-3
Linkage Provisions	8-4
Discontinuing Linkage.....	8-5
Error Handling	8-5
Creating Libraries	8-6
Referencing Libraries	8-7
Library Attributes	8-8

Contents

Entry Point Type Matching	8-11
Parameter Passing	8-12
Library Examples	8-13
Library: OBJECT/FILEMANAGER/LIB	8-13
Calling Program #1	8-15
Library: OBJECT/SAMPLE/LIBRARY	8-16
Library: OBJECT/SAMPLE/DYNAMICLIB	8-17
Calling Program #2	8-19
Library: MCPSUPPORT	8-20
CONVERTDATEANDTIME Procedure	8-20
TIMEZONENAME Procedure	8-21
TIMEZONEOFFSET Procedure	8-22
Using the EVENT_STATUS Entry Point	8-22

Section 9. Internationalization

Accessing the Internationalization Features	9-2
Using the Ccsversion, Language, and Convention	
Default Settings	9-2
Understanding the Hierarchy for Default Settings	9-3
Understanding the Components of the MLS Environment	9-4
Coded Character Sets and Ccsversions	9-4
Mapping Tables	9-6
Data Classes	9-7
Text Comparisons	9-7
Providing Support for Natural Languages	9-9
Creating Messages for an Application	
Program	9-9
Creating Multilingual Messages for	
Translation	9-10
Providing Support for Business and Cultural	
Conventions	9-10
Using the Date and Time Features	9-11
Using the Numeric and Currency Features	9-12
Using the Page Size Formatting Features	9-13
Summary of CENTRALSUPPORT Library Procedures	9-13
Library Calls	9-23
Parameter Categories	9-24
Input Parameters	9-24
Input Parameters with Type Values	9-25
Output Parameters	9-25
Result	9-25
Procedure Descriptions	9-26
CCSINFO	9-26
CCSTOCCS_TRANS_TABLE	9-31
CCSTOCCS_TRANS_TABLE_ALT	9-33
CCSTOCCS_TRANS_TEXT	9-35
CCSTOCCS_TRANS_TEXT_COMPLEX	9-37
CCSVSN_NAMES_NUMS	9-40
CENTRALSTATUS	9-42
CNV_ADD	9-44

CNV_CONVERTCURRENCY_STAR	9-47
CNV_CONVERTDATE_STAR	9-49
CNV_CONVERTNUMERIC_STAR	9-52
CNV_CONVERTTIME_STAR	9-54
CNV_CURRENCYEDIT	9-57
CNV_CURRENCYEDIT_DOUBLE	9-59
CNV_CURRENCYEDITTMP	9-61
CNV_CURRENCYEDITTMP_DOUBLE	9-63
CNV_DELETE	9-65
CNV_DISPLAYMODEL	9-66
CNV_FORMATDATE	9-68
CNV_FORMATDATETMP	9-70
CNV_FORMATTIME	9-72
CNV_FORMATTIMETMP	9-74
CNV_FORMSIZE	9-76
CNV_INFO	9-78
CNV_MODIFY	9-81
CNV_NAMES	9-84
CNV_SYMBOLS	9-86
CNV_SYSTEMDATETIME	9-91
CNV_SYSTEMDATETIMETMP	9-94
CNV_TEMPLATE	9-97
CNV_VALIDATENAME	9-99
COMPARE_TEXT_USING_ORDER_INFO	9-100
GET_CS_MSG	9-102
MCP_BOUND_LANGUAGES	9-105
VALIDATE_NAME_RETURN_NUM	9-107
VALIDATE_NUM_RETURN_NAME	9-109
VSNCOMPARE_TEXT	9-111
VSNESCAPEMENT	9-114
VSNGETORDERINGFOR_ONE_TEXT	9-116
VSNINFO	9-119
VSNINSPECT_TEXT	9-123
VSNORDERING_INFO	9-126
VSNTRANSTABLE	9-129
VSNTRANS_TEXT	9-131
VSNTRUTHSET	9-134
Explanation of Error Values	9-136

Appendix A. Run-Time Format-Error Messages

Free-Field Input	A-1
Formatted Output	A-2
Formatted Input	A-3

Appendix B. Reserved Words

Reserved Words List	B-3
---------------------------	-----

Appendix C. Data Representation

Contents

Field Notation	C-1
Character Representation	C-2
Character Values and Graphics	C-4
Default Character Type	C-12
Signs of Numeric Fields	C-13
One-Word Operand	C-14
Real Operand	C-14
Integer Operand	C-15
Boolean Operand	C-16
Two-Word Operand	C-17
Double-Precision Operand	C-17
Complex Operand	C-19
Type Coercion of One-Word and Two-Word Operands	C-20
Data Descriptors and Pointer	C-20

Appendix D. Understanding Railroad Diagrams

Railroad Diagram Concepts	D-1
Paths	D-1
Constants and Variables	D-2
Constraints	D-3
Following the Paths of a Railroad Diagram	D-6
Railroad Diagram Examples with Sample Input	D-7

Appendix E. Related Product Information

Index	1
--------------------	----------

Figures

3-1.	Translate Table Indexing.....	3-180
3-2.	Truth Set Test.....	3-184
4-1.	DO-UNTIL Loop.....	4-52
4-2.	FOR-DO Loop.....	4-62
4-3.	FOR-STEP-UNTIL Loop.....	4-63
4-4.	FOR-STEP-WHILE Loop.....	4-64
4-5.	FOR-WHILE Loop.....	4-65
4-6.	THRU Loop.....	4-200
4-7.	WHILE-DO Loop.....	4-213
5-1.	Exponentiation: Meaning of $Y^{**}Z$	5-4
5-2.	Mathematical Notation.....	5-5
5-3.	Types of Values Resulting from Arithmetic Operations.....	5-6
5-4.	Results of Logical Operators.....	5-15
C-1.	Field Notation, [29:6].....	C-1
C-2.	EBCDIC Characters (8-Bit Fields).....	C-2
C-3.	ASCII Characters (8-Bit Fields).....	C-3
C-4.	Hexadecimal Characters (4-Bit Fields).....	C-3
C-5.	Real Operand.....	C-14
C-6.	Integer Operand.....	C-15
C-7.	Boolean Operand.....	C-16
C-8.	First Word, Double-Precision Operand.....	C-17
C-9.	Second Word, Double-Precision Operand.....	C-18

Figures

Tables

3-1.	Array Parameters.....	3-141
3-2.	Procedure Reference Array Parameters	3-143
3-3.	Procedure Parameters.....	3-144
3-4.	Simple Variable Parameters	3-145
3-5.	String Parameters.....	3-146
3-6.	File Parameters	3-146
3-7.	Other Types of Parameters	3-147
5-1.	Arithmetic Intrinsic Functions.....	5-38
6-1.	Compiler Input and Output Files	6-2
8-1.	Parameter Passing Rules	8-12
9-1.	Functional Grouping of CENTRALSUPPORT Library Procedures	9-14
9-2.	Error Results for Internationalization	9-137
B-1.	Reserved Words List.....	B-3
D-1.	Elements of a Railroad Diagram	D-2

Tables

Section 1

Program Structure

About This Manual

Purpose

This language reference manual provides the programmer with information on the language components, declarations, statements, expressions, and program units of Extended ALGOL.

To assist application programmers in using ALGOL, the programming reference material is divided into two volumes. Volume 1 contains the Extended ALGOL developed for general use. Volume 2 contains extensions to ALGOL that are intended for specific products.

To use this manual, a programmer should be familiar with the general concepts of ALGOL programming or of another high-level structured programming language, such as Pascal.

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

Audience

This manual is intended for the applications programmer or systems analyst who is experienced in developing, maintaining, and reading ALGOL programs.

Notation Conventions

This manual contains reference information for each ALGOL feature, which can be accessed either through the index or the table of contents. Cross references are provided within each section. Declarations, statements, expressions, functions, and compiler control options are each presented in alphabetical order within their sections.

Overview of the Language

Extended ALGOL is a high-level, structured programming language. Extended ALGOL has provisions for communication between programs and input/output (I/O) devices, the editing of data, and the implementation of diagnostic facilities for program debugging.

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

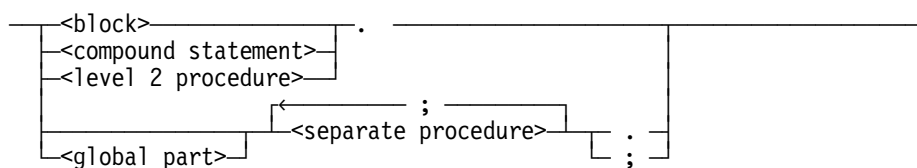
At a level of complexity higher than language components are declarations, statements, and expressions. These are the building blocks of ALGOL programs. A declaration associates identifiers with specific properties. For example, an identifier can be associated with the properties of a real number. A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

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

Program Unit

A program unit is a group of ALGOL constructs that can be compiled as a whole. The following diagram shows the elements that can be included in an ALGOL program.

<program unit>



<block>

— BEGIN —<declaration list>—; —<statement list>— END —————|

<declaration list>

—<declaration>—; —————|

<statement list>

—<statement>—; —————|

<compound statement>

— BEGIN —<statement list>— END —————|

<level 2 procedure>

—<procedure declaration>—————|

<global part>

— [—<declaration list>—] —————|

<separate procedure>

—<procedure declaration>—————|

Elements of an ALGOL Program

The simplest valid ALGOL program is 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.

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

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

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

Compound Statements

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

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

Blocks

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

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

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

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

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

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

Examples

Compound Statement

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

Block

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

Level 2 Procedure

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

Separate Procedure with Global Part

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

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

Scope of an Identifier

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

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

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

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

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

Local Identifiers

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

Global Identifiers

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

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

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

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

second WRITE statement is $A=3$. The result of the third WRITE statement is $A=0.1764$, which equals $4.2 @ -1 * 4.2 @ -1$.

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

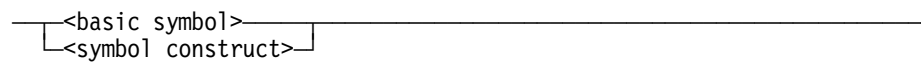
- To carry values that have been calculated in an outer block into the inner block
- To carry a value calculated inside the block to an outer block
- To preserve a value calculated within a block for use in a later entry to the same block
- To transmit a value from one block to another block that does not contain and is not contained by the first block

Section 2

Language Components

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

<language component>



<symbol construct>



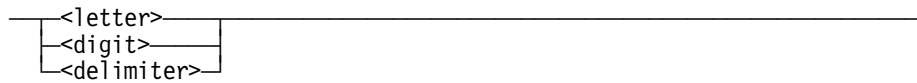
Basic symbols, identifiers, numbers, remarks, and string literals are described under separate headings in this section.

Because the define invocation is closely linked to the DEFINE declaration, the define invocation is explained under "DEFINE Declaration" in Section 3, "Declarations."

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

Basic Symbol

<basic symbol>



<letter>

Any one of the uppercase (capital) letters A through Z, or any one of the lowercase letters a through z.

<digit>

Any one of the Arabic numerals 0 through 9.

<delimiter>



<bracket>



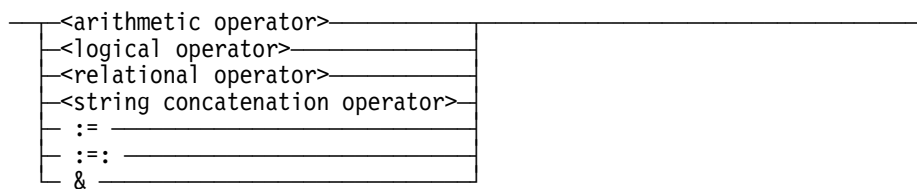
<parameter delimiter>



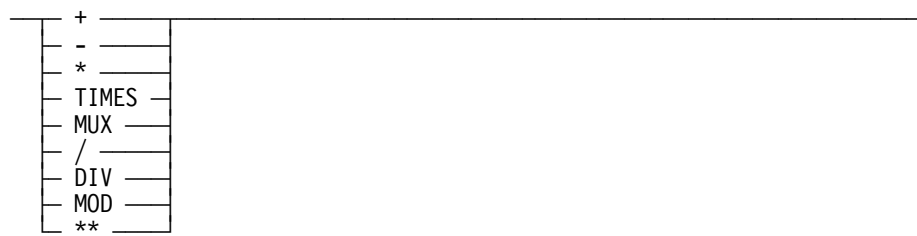
<letter string>

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

<operator>



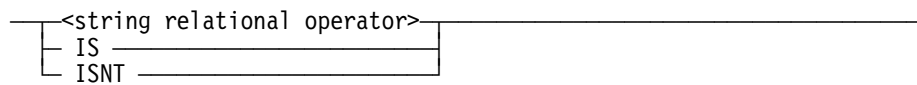
<arithmetic operator>



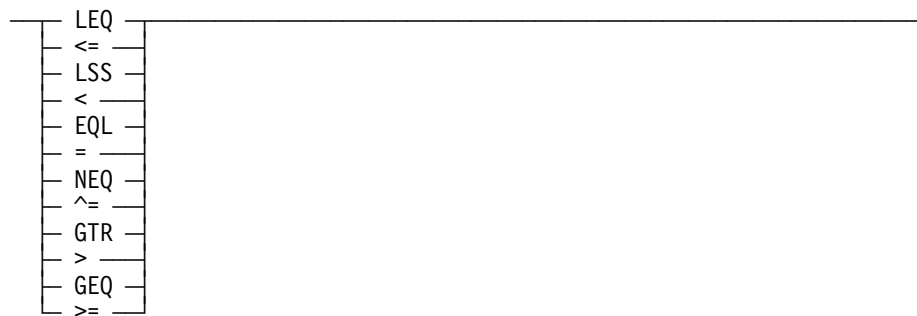
<logical operator>



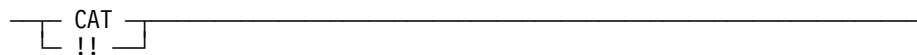
<relational operator>



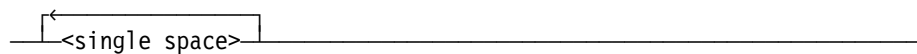
<string relational operator>



<string concatenation operator>



<space>



<single space>

One blank character.

Language Components

The compiler does not make a distinction between uppercase and lowercase letters. Individual letters do not have particular meanings except as used in pictures and formats.

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

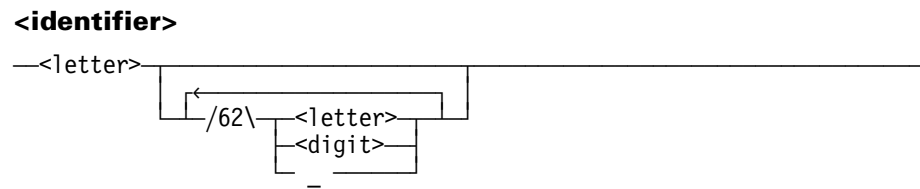
Delimiters include operators, spaces, and brackets. An important function of these elements is to delimit the various entities that make up a program. Each delimiter has a fixed meaning, which, if not obvious, is explained in this manual in the syntax of appropriate constructs. Basic symbols that are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

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

- Delimiter composed of letters
- Identifier
- Boolean value
- Unsigned number

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

Identifier



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

The scopes of identifiers are described in Section 1, “Program Structure.”

Examples

Valid Identifiers

A
I
B5
YSQUARE
EQUITY
RETURN_RATE
D2R271GL
TEST_1

Invalid Identifiers

1776
2BAD
\$
X-Y
NET GAINS
NO.
_TEST
BEGIN

Reason

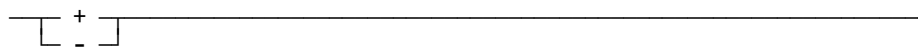
Does not begin with a letter.
Does not begin with a letter.
\$ is not an allowed character.
“-” is not an allowed character.
Blank spaces are not allowed.
“.” is not an allowed character.
Does not begin with a letter.
Reserved word.

Number

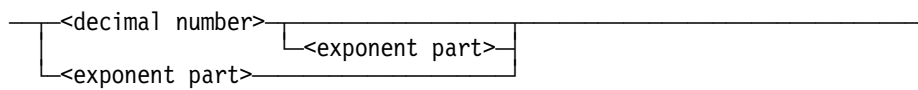
<number>



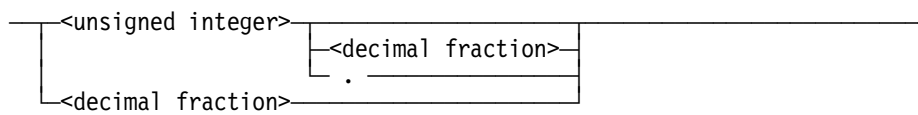
<sign>



<unsigned number>



<decimal number>



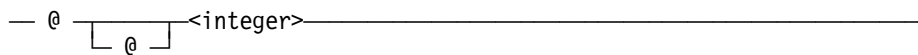
<unsigned integer>



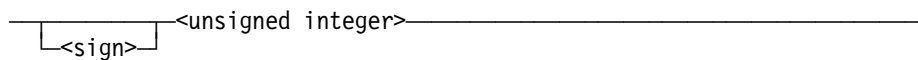
<decimal fraction>



<exponent part>



<integer>



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

Examples

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

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

Valid Numbers

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

Invalid Numbers	Reason
50 00.5@8 8	Blank spaces are not allowed.
1,505,278	Commas are not allowed.
@63.4	Exponent part must be an integer.
1.667E-01 0	E is not allowed for exponent part.

Number Ranges

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

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

- Any integer between plus and minus $549755813887 = 8^{**}13 - 1 = 4"007FFFFFFFFF"$, inclusive, can be represented in integer form.
- For single-precision numbers:
 - The largest, positive, normalized, single-precision number that can be represented is $4.31359146674@68 = (8^{**}13 - 1) * 8^{**}63 = 4"1FFFFFFFFF"$.
 - The smallest, positive, normalized, single-precision number that can be represented is $8.75811540204@-47 = 8^{**}(-51) = 4"3F9000000000"$.
- Zero and numbers with absolute values between the largest and smallest values given above can be represented as single-precision real numbers.
- For double-precision numbers:
 - The largest, positive, normalized, double-precision number that can be represented is $1.94882838205028079124467@@29603 = (8^{**}13 - 8^{**}(-13)) * 8^{**}32767 = 4"1FFFFFFFFFFFFFFFFFFFFFFFFF"$.
 - The smallest, positive, normalized, double-precision number that can be represented is $1.9385458571375858335564@@-29581 = 8^{**}(-32755) = 4"3F9000000000FF8000000000"$.
- Zero and numbers with absolute values between the largest and smallest values given above can be represented as double-precision numbers.

Compiler Number Conversion

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

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

In some internal computations, a double-precision integer might be required or enforced. A double-precision floating point with exponent equal to 13 is used as the canonical representation of a double-precision integer; an operand in this format is called a double integer. The maximum magnitude that can be represented in this format is $2^{78} - 1$, or $8^{26} - 1$, or 302,231,454,903,657,293,676,543.

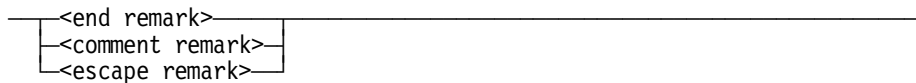
Exponents

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

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

Remark

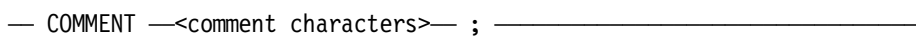
<remark>



<end remark>

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

<comment remark>



<comment characters>

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

<escape remark>



<escape text>

Any sequence of EBCDIC characters.

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

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

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

Note: An apostrophe (') within a comment remark can cause a syntax error when the compiler is within an area that the compile-time facility is skipping. You should avoid the use of an apostrophe where possible.

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

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

Examples

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

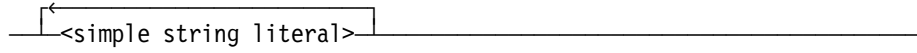
```
BEGIN
  FILE F(KIND=PRINTER COMMENT;);
  FORMAT COMMENT; FMT COMMENT; (A4,I6);
  PROCEDURE P(X,COMMENT;Y,Z);
    REAL X,Y COMMENT; ,Z;      % PERCENT SIGN CAN BE USED HERE
    X := Y + COMMENT; Z;      % HERE TOO
  IF COMMENT; 7 > 5 THEN
    WRITE(F,<"OK">);
  IF 4 COMMENT; > 2 THEN
    WRITE(F,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

The following program illustrates some invalid uses of the remark.

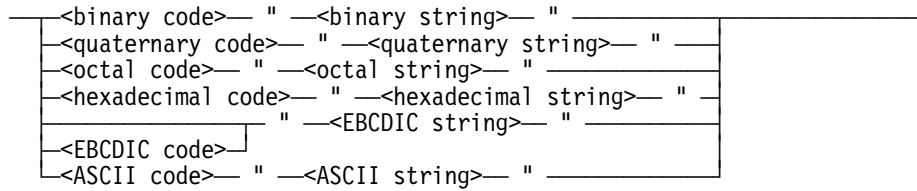
```
BEGIN
  FILE F(KIND=PRINTER);
  FORMAT FMT(13,F10.3 COMMENT; ,A4);
  ARRAY A[0:99];
  REAL X;
  FORMAT ("ABC", % CANNOT BE USED. "DEE");
  WRITE(F,<"INVALID USE" COMMENT;>);
  REPLACE POINTER(A) BY "ABCD COMMENT;EFGHIJ";
  X := "AB,COMMENT;C";
  COMMENT CANNOT BE USED HERE COMMENT; EITHER;
END.
```

String Literal

<string literal>



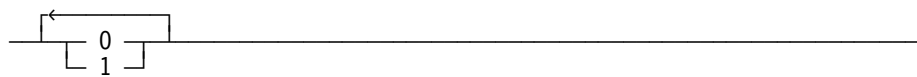
<simple string literal>



<binary code>



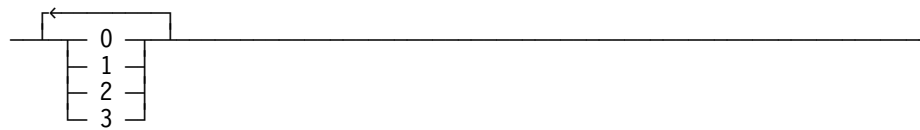
<binary string>



<quaternary code>



<quaternary string>



<octal code>



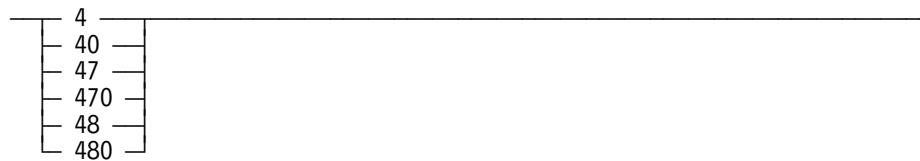
<octal string>



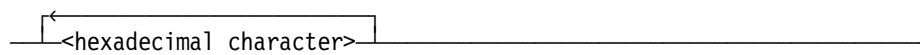
<octal character>



<hexadecimal code>



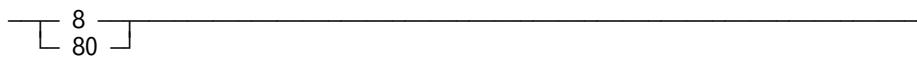
<hexadecimal string>



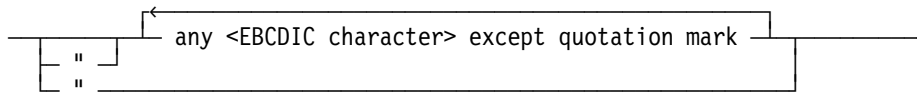
<hexadecimal character>



<EBCDIC code>



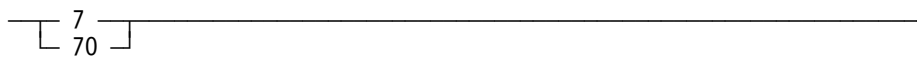
<EBCDIC string>



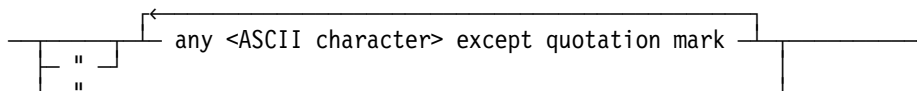
<EBCDIC character>

Any one of the 256 possible EBCDIC characters.

<ASCII code>



<ASCII string>



<ASCII character>

Any one of the 128 possible ASCII characters.

<printable character>

Any <letter>, any <digit>, any <special character>, or a blank.

<special character>

Any one of the following characters:

@ at	(left parenthesis	! exclamation point
# number) right parenthesis	? question mark
\$ dollar	[left bracket	' apostrophe (single quote)
% percent] right bracket	+ plus sign
& ampersand	{ left brace	- minus sign (hyphen)
* asterisk	} right brace	split bar
= equal sign	< less than	~ tilde
, comma	> greater than	^ circumflex (carat)
; semicolon	/ slash	` grave accent
: colon	\ backslash	" quotation mark
. period	_ underscore	

Character Size

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

String Code

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

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

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

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

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

String Length

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

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

ASCII Strings

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

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

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

Quotation Mark

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

Dollar Sign

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

Section 3

Declarations

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

In this section, the ALGOL declarations are listed and discussed in alphabetical order. In many cases, the entire syntax diagram for a declaration is divided into smaller segments, and each segment is discussed in turn. Each declaration is accompanied by examples of its use.

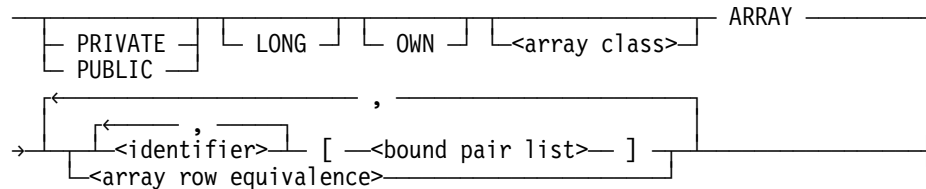
ALPHA Declaration

The ALPHA declaration is semantically identical to REAL. ALPHA is used instead of REAL when the identifier is used to hold six or fewer EBCDIC or ASCII characters. For more information, refer to the REAL declaration later in this section.

ARRAY Declaration

An ARRAY declaration declares one or more identifiers to represent arrays of specified fixed dimensions. After an array has been declared in an ARRAY declaration, values can be stored in and retrieved from the elements of the array by the use of subscripted variables, which contain the array identifier and a subscript list.

<array declaration>



PRIVATE and PUBLIC Specifiers

The PRIVATE and PUBLIC specifiers can only be used for arrays declared within a structure block or a connection block. The PRIVATE specifier limits visibility of the array to the scope of the structure block or a connection block. A PRIVATE array cannot be accessed using a structure or connection block qualifier. The PUBLIC specifier allows the array to be accessed using a structure or connection block qualifier. If neither PRIVATE nor PUBLIC is specified, the default value is PUBLIC and access to the array using a structure or connection block qualifier is allowed.

LONG Arrays

The LONG specification affects only array rows. It specifies that the array is not to be paged regardless of its length. The maximum length of a LONG array is 65,535 words. Attempting to allocate, or to use the RESIZE statement to create, an array larger than the limit causes termination of the program at run time.

Normally, an array row longer than a certain threshold is automatically subdivided, or segmented, at run time into pages. Each page is of a fixed length, except the last, which can be shorter. The page size is a property of the machine on which the program is run; it is always a power of two and is never less than 256 words. The paging threshold is maintained by the operating system on which the program is run. The operating system enforces a limit on the size of a LONG array; the maximum length of a LONG array is 65,535 words, and it is never less than *max (page_size, 1024)* words.

The array size at which an array row is automatically paged can be changed with the system command SEGARRAYSTART. For more information on the SEGARRAYSTART command, see the *System Commands Operations Reference Manual*. Arrays smaller than 1024 words are never paged.

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

OWN Arrays

If an OWN array is declared, the array and its contents are retained on exit from the block in which the array is declared and are available on subsequent reentry into the block.

OWN arrays are allocated only once, regardless of the number of times entry is made into the block in which the array is declared. If the OWN array is declared with variable bounds, these bounds are evaluated once when the array is allocated, and the affected dimension retains these bounds for the remainder of the program execution. For information on resizing the array, refer to “RESIZE Statement” in Section 4, “Statements.”

An OWN array remains unreferenced from the time the program begins execution until the first execution of a statement that references the array is encountered. Once such a statement is encountered, the array is referenced, or *touched* for the remainder of the program execution.

An array that is not an OWN array remains unreferenced from the time the program enters the block in which the array is declared until the first execution of a statement that refers to the array. Once such a statement is encountered, the array is touched until the program exits the block.

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

Identifiers

<array identifier>

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

<character array identifier>

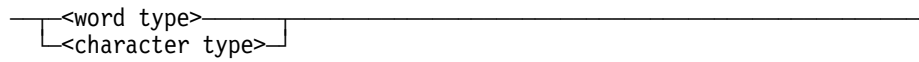
An array identifier, array reference identifier, direct array identifier, or value array identifier that is declared with a character type.

<word array identifier>

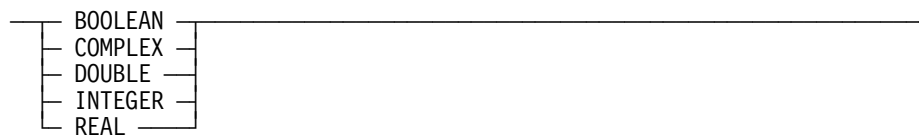
An array identifier, array reference identifier, direct array identifier, or value array identifier that is declared with a word type.

Array Class

<array class>



<word type>



<character type>



Arrays declared in the same ARRAY declaration are of the same array class. If the array class is omitted, a REAL array is assumed. Arrays not declared with a character type are called word arrays. Arrays declared with a character type are called character arrays. Word and character arrays can be passed as parameters and used as array rows. Character arrays can be used as simple pointer expressions.

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

Element Width

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

Array Class	Element Width
DOUBLE, COMPLEX	96 bits (double word)
INTEGER, REAL, BOOLEAN	48 bits (single word)
EBCDIC, ASCII	8 bits (6 characters per word)
HEX	4 bits (12 characters per word)

ARRAY Declaration

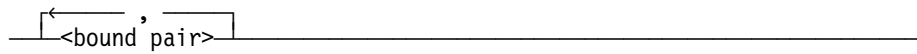
Within the operating system, arrays are manipulated by means of descriptors; each descriptor specifies an element width appropriate to the array class. Single-word and double-word descriptors are used for word arrays; 4-bit and 8-bit descriptors are used for character arrays.

Because complex and double array elements are composed of two 48-bit words, the two words are allocated contiguously. The layout of a complex array is as follows: the real part of the first element, the imaginary part of the first element, the real part of the second element, the imaginary part of the second element, and so on. Similarly, the layout of a double array is as follows: the first word of the first element, the second word of the first element, the first word of the second element, the second word of the second element, and so on.

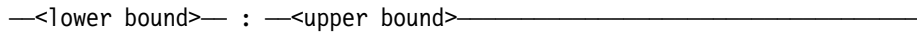
For information on the internal representation of double and complex operands, refer to "Two-Word Operand" in Appendix C, "Data Representation."

Bound Pair List

<bound pair list>



<bound pair>



<lower bound>



<upper bound>



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

For arrays declared within a STRUCTURE BLOCK TYPE declaration, evaluations of the arithmetic expressions in the bound pair list are done once for each structure block instance (structure block variable or structure block array element) when the structure block instance is first created. The structure block instance is created upon the execution of the first reference to the structure block variable or structure block array element, that is, when the activation record for the structure block instance is first made present.

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

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

Original and Referred Arrays

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

There are three other ways to associate an identifier with an array: array row equivalence, array reference assignment, and array specification in a PROCEDURE declaration. In each of these cases, the identifier refers to the same data as an original array. Such an identifier is called a referred array. An array row equivalence or array reference assignment can cause an array identifier of one array class to refer to data in an original array of another array class.

Dimensionality

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

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

$$\langle \text{upper bound} \rangle - \langle \text{lower bound} \rangle + 1$$

The maximum number of elements in an ARRAY dimension can vary and is checked by the MCP when a program is executed. The maximum size of a dimension is dependent on the system on which the program is executed. The current range of size is $2^{**}20-1$, $2^{**}24-1$, $2^{**}27-1$, and $2^{**}28-1$.

Array Row Equivalence

<array row equivalence>

—<identifier>— [—<lower bound>—] — = —<array row>—————|

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

The size of the declared array is determined by the size and element width of the array row and the element width for the array class of this declaration. For example, assume that *Sa* and *Wa* are the size and element width of the array row, and that *Wea* is the element width for the equivalent array. The size of the equivalent array, *Sea*, is then the following:

$$Sea := (Sa * Wa) \text{ DIV } Wea$$

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

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

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

  I[2] := 25.234;
  R[0] := 25.234;
END.
```


The array row equivalence part cannot appear in an ARRAY declaration that declares an OWN array. For example, the following declaration is invalid:

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

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

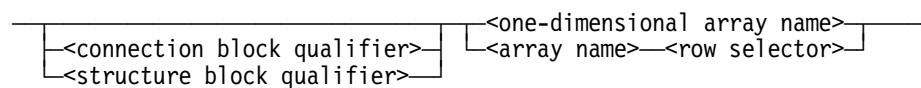
Note: *There are subtle restrictions on the correct declaration and use of an array row equivalence in which the array row of the declaration is a row of an array reference, because the default state of an array reference variable is uninitialized.*

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

If the array reference is multidimensional and/or has a different element width than the new array, the array row equivalence is established from the value of the array reference variable at the time the program enters the block containing the array row equivalence declaration. Later assignments to the array reference variable do not affect the array row equivalence. Therefore, in order for the declaration to be useful, the array reference variable must have been declared and initialized in a scope global to the block declaring the array row equivalence.

Array Row

<array row>

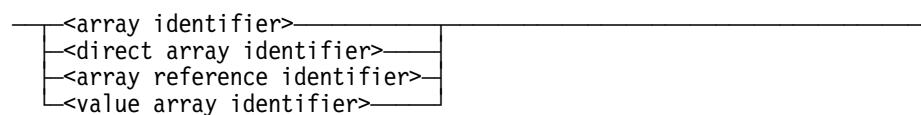


An array row is a one-dimensional array designator.

<one-dimensional array name>

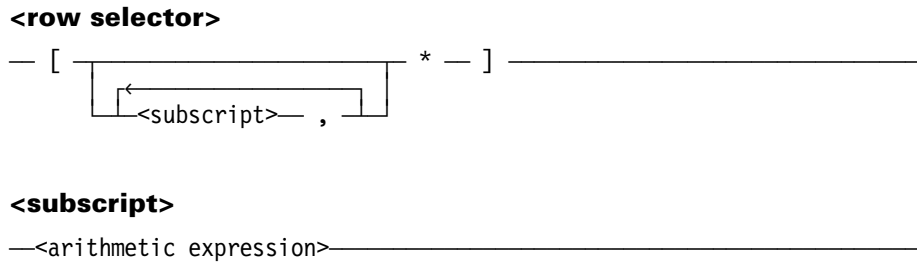
An <array name> whose identifier was declared with one dimension.

<array name>



ARRAY Declaration

Row Selector



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

Examples of ARRAY Declarations

The following example declares DOG, a four-dimensional array made up of $6 * 26 * 7 * 13 = 14196$ integer elements:

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

The following example declares STUB, a one-dimensional OWN array made up of 10 real elements:

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

The following example declares two real arrays: GROUP_REAL, which is a one-dimensional array, and CAD, which is a two-dimensional array:

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

The following example declares the EBCDIC array GROUP_EBCDIC. Array row equivalence causes GROUP_EBCDIC to refer to the same data as the previously declared real array GROUP_REAL. Note that the element width of GROUP_REAL is 48 bits, whereas the element width of GROUP_EBCDIC is 8 bits. This means that a reference to a single element in GROUP_REAL refers to 48 bits, and a reference to a single element in GROUP_EBCDIC refers to 8 bits.

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

The following example declares XRAY, a one-dimensional array. Because no array class is specified, the array class XRAY is of type REAL. The lower bound is the integerized value of $X + Y + Z$, and the upper bound is the integerized value of $3 * A + B$.

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

The following example declares BIG_ARRAY, a one-dimensional array made up of 10,000 Boolean elements. Because BIG_ARRAY is declared a LONG array, the array is not paged (segmented). Because it is not paged, the array occupies 10,000 contiguous words in memory.

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

The following example declares SEGARRAY, a one-dimensional array made up of 50,001 real elements. Because SEGARRAY is not declared a LONG array and the array row is longer than 1024 words, SEGARRAY is automatically divided at run time into segments that are 256 words long.

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

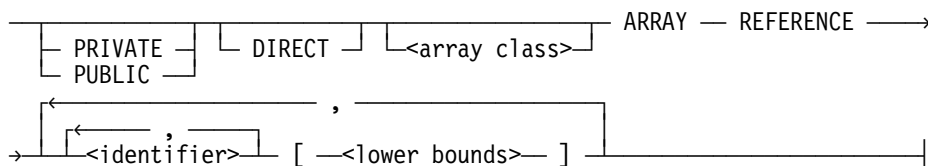
The following example declares C, a two-dimensional array made up of $3 * 61 = 183$ complex elements. Note that the element width of a complex array is 96 bits (two words).

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

ARRAY REFERENCE Declaration

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

<array reference declaration>



Following an array reference assignment, any subsequent use of the array reference identifier acts as a reference to the array assigned to it. For more information on array reference assignment, see “Array Reference Assignment” in Section 4, “Statements.”

The PRIVATE and PUBLIC specifiers can only be used for array reference variables that are declared within a structure block or a connection block. The PRIVATE specifier limits visibility of the array reference variable to the scope of the structure block or a connection block. A PRIVATE array reference variable cannot be accessed using a structure or connection block qualifier. The PUBLIC specifier allows the array reference variable to be accessed using a structure or connection block qualifier. If neither PRIVATE nor PUBLIC is specified, the default value is PUBLIC and access to the array reference variable using a structure or connection block qualifier is allowed.

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

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

Identifiers

<array reference identifier>


An identifier that is associated with an array reference in an ARRAY REFERENCE declaration. A formal array parameter to a procedure is also treated as an array reference identifier when referenced within the scope of the procedure.

<direct array reference identifier>

An identifier that is associated with an array reference that is declared as DIRECT in an ARRAY REFERENCE declaration. A formal direct array parameter to a procedure is also treated as a direct array reference identifier when referenced within the scope of the procedure.

Lower Bounds

<lower bounds>



The number of dimensions of the array reference variable is determined by the number of lower bounds in its declaration. No more than 16 dimensions are allowed. For more information on lower bounds, see “ARRAY Declaration” earlier in this section.

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

Examples of ARRAY REFERENCE Declarations

The following example declares REFARRAY, an array reference variable with a lower bound of 3. Because an array class is not specified, REFARRAY is a real array reference variable.

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

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

```
DIRECT ARRAY REFERENCE DIRREFARRAY[N]
```

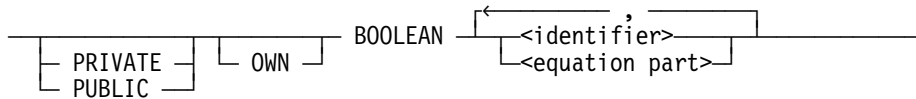
The following example declares two complex array reference variables. CREF1 is a one-dimensional array reference variable with a lower bound of 0 (zero), and CREF2 is three-dimensional with lower bounds of 0, 10, and 10.

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

BOOLEAN Declaration

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

<Boolean declaration>



The PRIVATE and PUBLIC specifiers can only be used for simple variables that are declared within a structure block or a connection block. The PRIVATE specifier limits visibility of the simple variable to the scope of the structure block or the connection block. A PRIVATE simple variable cannot be accessed using a structure or connection block qualifier. The PUBLIC specifier allows the simple variable to be accessed using a structure or connection block qualifier. If neither PRIVATE nor PUBLIC is specified, the default value is PUBLIC and access to the simple variable using a structure or connection block qualifier is allowed.

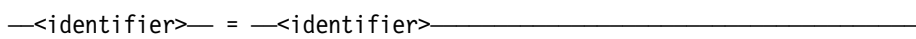
A simple variable declared with the OWN specification retains its value when the program exits the block in which the variable is declared. The value of that variable is again available when the program reenters the block in which the variable is declared.

<Boolean identifier>

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

Equation Part

<equation part>



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

Address equation is allowed only among INTEGER, REAL, and BOOLEAN variables. Because both identifiers of the equation part have the same address, altering the value of either variable affects the value of both variables. For more information, see “Type Coercion of One-Word and Two-Word Operands” in Appendix C, “Data Representation.”

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

Boolean Simple Variable Values

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

When a Boolean simple variable is allocated, it is initialized to FALSE, a 48-bit word with all bits equal to 0 (zero).

Refer to Appendix C, "Data Representation," for additional information on the internal structure of a Boolean operand.

Examples of BOOLEAN Declarations

The following example declares BOOL as a Boolean simple variable.

```
BOOLEAN BOOL
```

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

```
OWN BOOLEAN DONE, ENDOFIT
```

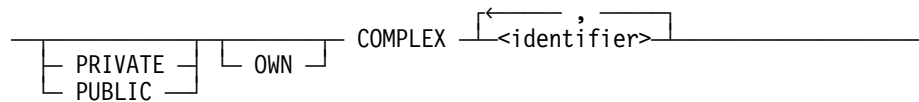
The following example declares FLAG and BINT as Boolean simple variables, and address-equates BINT to the previously declared simple variable INTGR. The variables BINT and INTGR share the same address.

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

COMPLEX Declaration

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

<complex declaration>



<complex identifier>

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

Complex Variables

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

Because a real value is a complex value with an imaginary part equal to 0 (zero), the set of real values is a subset of the set of complex values. Therefore, arithmetic values can be assigned to complex variables, but complex values cannot be assigned to arithmetic variables.

The PRIVATE and PUBLIC specifiers can only be used for simple variables that are declared within a structure block or a connection block. The PRIVATE specifier limits visibility of the simple variable to the scope of the structure or connection block. A PRIVATE simple variable cannot be accessed using a structure block or the connection block qualifier. The PUBLIC specifier allows the simple variable to be accessed using a structure or connection block qualifier. If neither PRIVATE nor PUBLIC is specified, the default value is PUBLIC and access to the simple variable using a structure or connection block qualifier is allowed.

A simple variable declared to be OWN retains its value when the program exits the block in which it is declared. The value of that variable is again available when the program reenters the block in which the variable is declared.

Refer to Appendix C, "Data Representation," for additional information on the internal structure of a complex operand.

Examples of COMPLEX Declarations

The following example declares C1 and C2 as complex simple variables.

```
COMPLEX C1, C2
```

The following example declares CURRENT, VOLTAGE, and AMP as complex simple variables. Because they are declared as OWN, these simple variables retain their values when the program exits the block in which the simple variables are declared.

```
OWN COMPLEX CURRENT, VOLTAGE, AMP
```

CONNECTION BLOCK REFERENCE VARIABLE Declaration

<connection block reference variable declaration>

`—<connection block type identifier> . REFERENCE —<identifier>—`

A CONNECTION BLOCK REFERENCE VARIABLE declaration declares a connection reference variable with the type of the connection block type identifier. A CONNECTION BLOCK REFERENCE ASSIGNMENT statement can then be used to assign an instance of the connection block of the declared type to this variable.

A reference to an embedded connection block is allowed by the following syntax:

```
outerSBtype.embeddedCBtype REFERENCE <id>
```

<connection block reference variable>

An identifier that is associated with a connection block reference in a CONNECTION BLOCK REFERENCE VARIABLE declaration.

The following example shows the declaration of a connection block reference variable:

```
TYPE STRUCTURE BLOCK SB;  
  BEGIN  
    TYPE CONNECTION BLOCK CBINNER;  
      BEGIN  
        REAL X;  
        .  
        .  
      END;  
    CBINNER LIBRARY CLINNER(CONNECTIONS = 8);  
  END;  
  
SB SBVAR;  
SB.CBINNER REFERENCE CBINNER_REF;  
CBINNER_REF:= SBVAR.CLINNER [5];
```

CONNECTION BLOCK TYPE Declaration

Connection blocks are similar to structure blocks but provide a means of specifying procedures for export and import. From the CONNECTION BLOCK TYPE declaration you can declare a connection library.

<connection block type declaration>

```
— <local connection block type declaration> —————|
  | <global connection block type declaration> |
```

<local connection block type declaration>

```
— TYPE CONNECTION BLOCK —<identifier>— ; — BEGIN —<declaration list>— END —|
```

<global connection block type declaration>

A global connection block can appear only in the global part of a program unit and cannot be nested. Global connection blocks enable the replacement binding of connection block procedures.

```
— <forward global connection block type declaration> —————|
  | <external global connection block type declaration> |
```

<forward global connection block type declaration>

```
— TYPE CONNECTION BLOCK —<identifier>— ; — FORWARD —————|
```

Connection blocks can be declared FORWARD within the global part of a subprogram. The global connection block then can be used for global connection library declarations within the global part. The global connection block must be fully specified before it is referenced. This syntax is used when the subprogram references a connection block item from outside the connection block.

<external global connection block type declaration>

```
— TYPE CONNECTION BLOCK —<identifier>— ; — EXTERNAL —————|
```

The EXTERNAL connection block sets up the environment for connection block pending procedures in the subprogram to be bound to the host. Only those connection block items that are referenced by the pending procedures need to be specified in the subprogram connection block declaration. The local declarations of the connection block items in the host that are not referenced by the pending procedures do not need to be specified in the subprogram. This enables pending procedures in the subprogram to be bound without having to fully declare the entire connection block. EXTERNAL connection blocks cannot be bound. Connection libraries cannot be derived from EXTERNAL connection blocks. EXTERNAL connection blocks must be declared at level 3.

All rules and restrictions that apply to the STRUCTURE BLOCK TYPE declaration also apply to the CONNECTION BLOCK TYPE declaration.

Connection blocks can be declared within connection or structure blocks, but a connection library cannot be declared with the type of the inner connection block outside the outer connection or structure block.

CONNECTION BLOCK TYPE Declaration

<connection block type identifier>

An identifier that is associated with a connection block type in a CONNECTION BLOCK TYPE declaration.

Referencing Connection Block Items Outside the Connection Block

<connection block item designator>

—<connection block qualifier>—<connection block item>—

<connection block qualifier>

—<connection library instance designator> . <connection block reference variable>—

<connection block item>

Any identifier that was declared inside a CONNECTION BLOCK TYPE declaration.

When items of a connection block are referenced outside the connection block, the connection block qualifier must be used to identify those items. The restrictions about what types of items can be referenced outside a structure block also apply to connection blocks.

A connection block procedure can be declared as NULL in an EXTERNAL connection block in a subprogram. The procedure serves as a placeholder for a procedure that is referenced by the pending procedure that is to be bound. A procedure in an EXTERNAL connection block can be declared only as PENDING, NULL, or IMPORTED.

For more information about structure block binding, refer to the *Binder Programming Reference Manual*.

Examples of CONNECTION BLOCK TYPE Declaration

The following example includes the ALGOL host program and an ALGOL subprogram that references connection block items from outside the connection block.

ALGOL Host Program

```
BEGIN
TYPE CONNECTION BLOCK CB;
  BEGIN
    REAL PROCEDURE R1;
      BEGIN
        REAL R;
        R:= 5.5
        R1:= R;
      END R1;
    END CB;

  CB SINGLE LIBRARY CL;

  PROCEDURE EXTERNAL_PROC;
    EXTERNAL;

  EXTERNAL_PROC;
END.
```

ALGOL Subprogram

```
[
  TYPE CONNECTION BLOCK CB; FORWARD;
  CB SINGLE LIBRARY CL;
]

TYPE CONNECTION BLOCK CB; % CB is fully specified here
  BEGIN
    REAL PROCEDURE R1;
      BEGIN
        REAL R;
        R:=5.5;
        R1:=R;
      END R1;
    END CB;

  $ SET LEVEL 3
  PROCEDURE EXTERNAL_PROC;
    BEGIN
      REAL REEL;
      REEL := CL.R1;
    END.
```

CONNECTION BLOCK TYPE Declaration

The following example includes an ALGOL host program that declares a connection block with imported library objects and an external procedure. Also included is an example of a subprogram containing the procedure to be bound and, for completeness, an example of the library file.

ALGOL Library Program

```
BEGIN
EVENT EV;
TYPE CONNECTION BLOCK CB;
  BEGIN
  EBCDIC ARRAY OUTPUTMSG[0:30];
  PROCEDURE SENDOUTPUT;
    BEGIN
    DISPLAY (STRING (OUTPUTMSG[0],10));
    END;

  EXPORT
    OUTPUTMSG (READWRITE),
    SENDOUTPUT;
  END CB;

CB SINGLE LIBRARY CL_LIB;

EXPORT EV;

READYCL (CL_LIB);

FREEZE (TEMPORARY);
END.
```

ALGOL Host Program

```
BEGIN
TYPE CONNECTION BLOCK CB;
  BEGIN
  REAL R;
  IMPORTED EBCDIC ARRAY      % Imported library object
    OUTPUTMSG[0] (READWRITE);
  PROCEDURE SENDOUTPUT;     % Imported library object
    IMPORTED;
  PROCEDURE P1;
    EXTERNAL;
  END CB;

CB SINGLE LIBRARY CL
  (AUTOLINK=TRUE,
  INTERFACENAME="CL_LIB.",
  TITLE="OBJECT/IMPORTED/CL/LIB."
  );

CL.P1;
END.
```

ALGOL Subprogram

```
$ set level 4
[TYPE CONNECTION BLOCK CB; EXTERNAL; ]

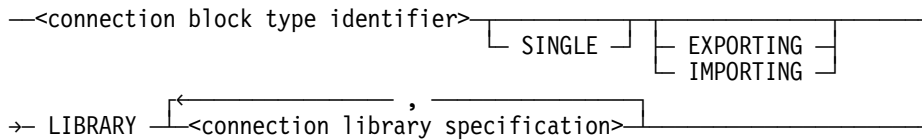
TYPE CONNECTION BLOCK CB;
BEGIN
  IMPORTED EBCDIC ARRAY          % Imported library object
    OUTPUTMSG[0] (READWRITE);
  PROCEDURE SENDOUTPUT;          % Imported library object
    IMPORTED;
  PROCEDURE P1;
    PENDING;
  END CB;

PROCEDURE CB.P1;
BEGIN
  REPLACE OUTPUTMSG[0] BY "IN P1";
  SENDOUTPUT;
  END P1.
```

CONNECTION LIBRARY Declaration

Connection libraries provide the ability to establish asynchronous two-way connections between libraries that provide access to the procedures of the connected library.

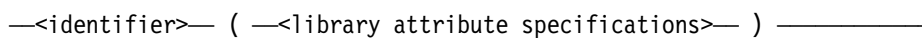
<connection library declaration>



When you declare a connection library, if you specify SINGLE, then only a single connection is allowed for this connection library and a connection index must not be specified when referencing an item of a connection block. An attempt to modify the CONNECTIONS attribute of a SINGLE library to a value other than 1 results in an attribute error.

If you specify EXPORTING, the compiler gives a syntax error if the specified connection block contains imports. If you specify IMPORTING, the compiler gives a syntax error if the specified connection block contains exports. EXPORTING and IMPORTING cannot appear in FORWARD declarations of connection libraries.

<connection library specification>

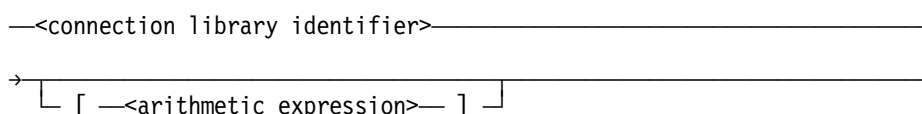


The <procedure identifier> specified for the CHANGE <procedure-valued library attribute name> must be declared inside the specified connection block. The <procedure identifier> specified for the APPROVAL <procedure-valued library attribute name> must be declared outside of the specified connection block if it is declared at all.

<connection library identifier>

An identifier that is associated with a connection library specification in a CONNECTION LIBRARY declaration.

<connection library instance designator>



A <connection library instance designator> specifies a particular instance of a connection library. If the connection library was declared without specifying SINGLE, the brackets and <arithmetic expression> must be included in order to specify a particular instance. The arithmetic expression must evaluate to a nonnegative integer.

For more information about procedure-valued library attributes, see the *Task Management Manual*.

Explanation

At library linkage time, if either half of a linkage operation is an EXPORTING or IMPORTING connection library, the MCP checks that only a one-way import/export relationship is being established. If the linkage operation would produce a two-way import/export relationship, in conjunction with existing library and IPC relationships, the linkage fails.

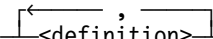
Also, if either half of a linkage operation is an EXPORTING or IMPORTING connection library, the importing side of the connection must not be a server library, which is a library established by the FREEZE statement.

You can avoid creating two-way import/export relationships by linking EXPORTING or IMPORTING connection libraries first and then closing the loop with a server library linkage or with a connection library linkage not specifying EXPORTING or IMPORTING. This concern can be addressed by using only EXPORTING and IMPORTING connection libraries for the library linkages within an application.

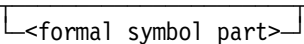
DEFINE Declaration

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

<define declaration>

— DEFINE —  —> —<definition> —————|

<definition>

—<identifier> —————|  = —<text>— # —————|

<define identifier>

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

<text>

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

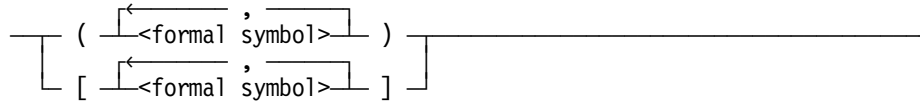
Text is bracketed on the left by the equal sign (=) and on the right by the number sign (#). The equal sign is said to be matched with the number sign. The text can be any sequence of characters not containing a free number sign. A free number sign is one that is not in a string literal, not in a remark,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

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

Note: *When declaring defines using included files, you should have the <text> and terminating free pound sign (#) within the same file. These two parts should either be in the same symbol file or the same include file. Failure to follow this guideline can produce unexpected results. This is known to be true for number defines, for example `DEFINE A = 50#`. For more information about include files, see the “INCLUDE Option” in Section 6, “Compiling Programs.”*

Formal Symbol Part

<formal symbol part>



<formal symbol>



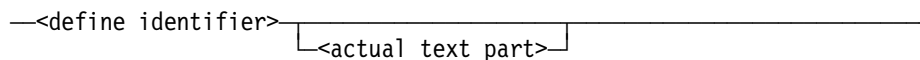
A define has two forms: simple and parametric. These forms are readily differentiated because parametric defines have a series of parameters (called formal symbols) enclosed in matching parentheses or brackets. The parentheses and the brackets have identical meanings.

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

Define Invocation

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

<define invocation>



<actual text part>



The parentheses and the brackets have identical meanings. These symbols are used when a parametric define is invoked.

DEFINE Declaration

<actual text>

Program text that cannot contain mismatched or unmatched parentheses, brackets, or quotation marks, or any comma outside of these bracketing symbols.

The invocation of a parametric define causes the actual text to be substituted into the positions in the text designated by the proper formal symbol.

The actual text need not be simple. As an example, assume you are using the following DEFINE declaration:

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

For this declaration, the following applies:

Define Invocation	Expands to
FORJ(0,B*3,MAX(X,Y,Z))	FOR J:= 0 STEP B*3 UNTIL MAX(X,Y,Z)

The actual text can be empty in a define invocation. In this case, all occurrences of the corresponding formal symbol in the text are replaced by no text. For example, assume you are using the following DEFINE declaration:

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

For this declaration, the following applies:

Define Invocation	Expands to	Syntactically Correct?
R := F(, 1);	R := +1;	Yes
R :=F(2,);	R :=2+;	No

A define identifier cannot be invoked as a part, rather than the whole, of a language component such as a string literal or a number. For example, assume you are using the following declarations:

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

For these declarations, the following two statements are not interpreted by the compiler to be equivalent:

Statement	Syntactically Correct?
S := EBCDIC_STR"ABC";	No
S := 8"ABC";	Yes

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

As a further example, assume you are using the following declarations:

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

For these declarations, the following applies:

Statement	Legal?
R := ITEM;	Yes
R := ITEM.30;	No, because it is equal to R := (15).30;, which is illegal.

In the following instances, the appearance of a define identifier does not cause the define to be expanded:

- Defines are not expanded in an end remark, a comment remark, or an escape remark.
- Defines are not expanded within quoted strings. For example, assume you are using the following declaration:

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

For this declaration, the string ONE WEEK is not equivalent to the string THE FIRST WEEK.

- Defines are not expanded within identifiers. For example, assume you are using the following declaration:

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

For this declaration, the identifiers A_B and ABC are not expanded to PREFIX_B and PREFIXBC.

DEFINE Declaration

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

```
DEFINE A = ARRAY #;
A B[0:10];           % A is expanded.
REAL A B[0:10];     % A can be interpreted as an identifier
                   % in a REAL declaration. A is not
                   % expanded. A syntax error results.
EBCDIC A B[0:10];  % A is expanded.

DEFINE THE_PARAMS = A PROCEDURE, B, C#, %THE_PARAMS is not expanded
    THE_CALL (THE_PARAMS, D) =         %when a reference to THE_CALL
    BEGIN                               %is made; thus, THE_CALL only
    A PROCEDURE (D);                   %has two formal parameters, not
    END#;                               %four. The following receives a
PROCEDURE A_PROCEDURE (Z); ...         %syntax error indicating that the
END; % of A_PROCEDURE                 %compiler expected a right
                                       %parenthesis after the second
                                       %parameter:
                                       % THE_CALL (P, Q, R, S);
```

- A define identifier is not expanded either in the format part of a FORMAT declaration or in the editing specifications of a READ statement or WRITE statement. Furthermore, if a FORMAT declaration or editing specifications are located within the text of a parametric define, they cannot reference the formal symbols of that define.
- A define identifier is not expanded when used in place of a file or task attribute mnemonic. Refer to the *File Attributes Programming Reference Manual* for file attribute mnemonics and the *Work Flow Language (WFL) Programming Reference Manual* for task attribute mnemonics.

In the following example, the define identifiers are not expanded in the FILE declaration or in the VALUE function:

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

FILE F(KIND = PRINTER);           % INTERPRETED AS PRINTER,
                                % NOT REMOTE

T.STATUS := VALUE(NEVERUSED); % INTERPRETED AS NEVERUSED,
                                % NOT NEWTASK
```

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

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

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

The assignment of a value to BOOL differs, depending on whether you invoke LOGIC1 or LOGIC2, as shown in the following table:

Invocation	Evaluates As	Value Assigned to BOOL
LOGIC1	BOOL := TRUE OR (TRUE AND FALSE);	TRUE
LOGIC2	BOOL := (TRUE OR TRUE) AND (FALSE);	FALSE

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

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

For this declaration, the following applies:

Define Invocation	Expands to
Q (X := X + 1)	X := X + 1 + 2 * X := X + 1

Examples of DEFINE Declarations

The following example declares BLANKIT as a define identifier:

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

Where BLANKIT appears as an allowable define invocation, it is expanded to the following when the program is compiled:

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

The following example declares SEC as a define identifier with a formal symbol X:

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

If SEC(N) appears as an allowable define invocation, it is expanded to the following when the program is compiled:

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

The following example declares LENGTH as a define identifier with two formal symbols, X and Y:

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

If LENGTH(3,4) appears as an allowable define invocation, it is expanded to the following when the program is compiled:

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

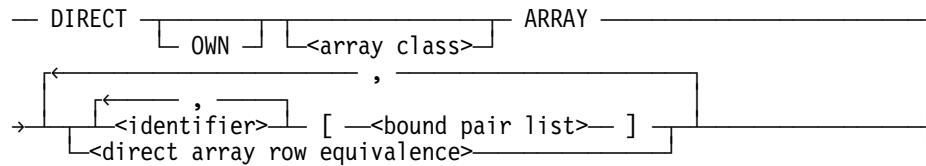
The following define results in a syntax error when it is invoked by passing a semicolon as its parameter. The syntax error occurs because the parameter is not expanded when viewed as part of an <end commentary>:

```
DEFINE X_STATEMENT (SEP) = BEGIN
    BEGIN
    Z := -5
    END SEP
    A := 5;
END #;
```


DIRECT ARRAY Declaration

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

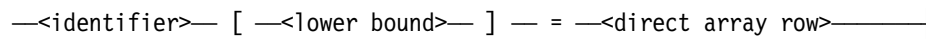
<direct array declaration>



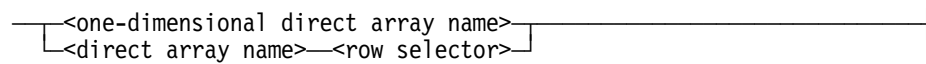
<direct array identifier>

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

<direct array row equivalence>



<direct array row>



<one-dimensional direct array name>

A direct array name whose identifier is declared with one dimension.

<direct array name>



Declaring Direct Arrays

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

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

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

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

For more information on the OWN specification, array class, bound pair list, lower bound, and row selector, see "ARRAY Declaration" earlier in this section. For information on the direct array reference identifier, see "ARRAY REFERENCE Declaration" earlier in this section.

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

Because a direct array can be used in performing direct I/O operations, a direct array is automatically unpagged (nonsegmented). For more information on direct I/O operations, see "I/O Statement" in Section 4, "Statements."

Caution

A DIRECT ARRAY declaration can also be used, instead of a LONG ARRAY declaration, to declare an array that is longer than the 65,535-word maximum limit for a long array. Doing so, however, disables normal system safeguards for array sizing and addressing, and could cause incorrect addressing of that array and data corruption on some system configurations.

The maximum size of a long array is determined by the configuration of the system. Those limitations are imposed as safeguards to prevent data corruption. Declaring a direct array explicitly defeats those safeguards.

For example, a long array cannot be declared so large that a valid character pointer cannot be developed for any location in that array. The operating system discontinues the program when it encounters such a declaration during program or procedure initiation. Similarly, a long array cannot be the target of a RESIZE command that would cause it to exceed the limits and endanger the contents of the array.

Because these limits are explicitly disabled for a direct array, data in a very large direct array could be corrupted without warning by an invalid pointer. Even if the initial size of the array is within the system limitations, a direct array could be resized beyond those limitations.

Always make sure that your program accesses to a direct array do not cause data corruption.

Examples of DIRECT ARRAY Declarations

The following example declares DIRARY, a one-dimensional direct array. Because no array class is specified, the array class of DIRARY is of type REAL.

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

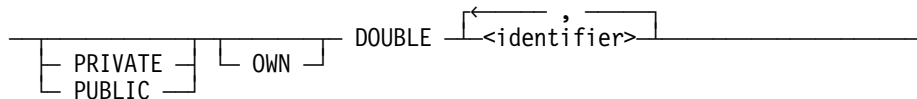
The following example declares the direct integer array DIREQVARAY. Array row equivalence causes the array DIREQVARAY to refer to the same data as the previously declared direct real array DIRARY.

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

DOUBLE Declaration

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

<double declaration>



<double identifier>

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

Declaration of Simple Variables

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

When a double-precision simple variable is allocated, it is initialized to a double-precision 0 (zero), which is two 48-bit words with all bits equal to zero. Refer to Appendix C, "Data Representation," for additional information on the internal structure of a double-precision operand.

The PRIVATE and PUBLIC specifiers can only be used for simple variables that are declared within a structure block or a connection block. The PRIVATE specifier limits visibility of the simple variable to the scope of the structure block or the connection block. A PRIVATE simple variable cannot be accessed using a structure or connection block qualifier. The PUBLIC specifier allows the simple variable to be accessed using a structure or connection block qualifier. If neither PRIVATE nor PUBLIC is specified, the default value is PUBLIC and access to the simple variable using a structure or connection block qualifier is allowed.

Examples of DOUBLE Declarations

The following example declares DUBL, a double-precision simple variable.

```
DOUBLE DUBL
```

The following example declares three double-precision variables: BIGNUMBER, GIGUNDOUS, and DUBLPRECISION.

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

DUMP Declaration

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

<dump declaration>

```
— DUMP —<file identifier>— ( —<dump list>— ) —<control part>—
```

<dump list>

```
—<simple variable>— , —<array identifier>—
  |
  |<label identifier>—
```

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

- Arrays with multiple dimensions
- Character arrays
- String variables
- String arrays

Control Part

<control part>

—<label identifier> —┬──┘
└─<label counter modulus>┘ └─<dump parameters>┘

<label counter modulus>

— : —<unsigned integer> —┬──┘

<dump parameters>

— (┬──┘) —┬──┘
└─<label counter>┘ └─<bounds part>┘

<label counter>

—<simple variable> —┬──┘

<bounds part>

┬──┘
└─ , —<lower limit> ┬──┘
└─ , — , —<upper limit> ┬──┘

<lower limit>

—<arithmetic expression> —┬──┘

<upper limit>

—<arithmetic expression> —┬──┘

The control part determines when the items are to be displayed. The control part can be just a label identifier or it can have a combination of components.

Label Identifier

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

Label Identifier with Label Counter Modulus

If a label counter modulus appears, the items in the dump list are dumped every <label counter modulus> times that the statement labeled by the label identifier is encountered. For example, if N is the label counter modulus and E is the number of times that the labeled statement has been encountered, then the items in the dump list are dumped whenever $E \text{ MOD } N$ is equal to 0 (zero).

Label Identifier with Dump Parameters

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

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

The items in the dump list are dumped when the number of times the labeled statement is encountered (or the value of the label counter variable, if specified) is greater than or equal to the lower limit and less than or equal to the upper limit. If the lower limit is not specified, it has a default value of 0 (zero). If the upper limit is not specified, it has a default value of infinity (no limit).

Label Identifiers with Label Counter Modulus and Dump Parameters

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

- The number is greater than or equal to the lower limit and less than or equal to the upper limit.
- The number is evenly divisible by the label counter modulus.

Form of Output

The information produced when a dump occurs depends on the declared types of the items to be dumped. When a dump occurs, the 1-character to 6-character symbolic name of each item in the dump list is produced, along with the following information:

For dumped simple variables,

- If the simple variable is of type REAL or DOUBLE, a real value is printed—for example, `R = .10000000000` or `DUBL = 0.0`.
- If the simple variable is of type INTEGER, an integer value is printed—for example, `I = 2`.
- If the simple variable is of type BOOLEAN, the Boolean value is printed—for example, `BOOL = .FALSE.`
- If the simple variable is of type COMPLEX, it is printed as a pair of numbers. The format consists of a left parenthesis, the real part in REAL format, a comma, the imaginary part in REAL format, and a right parenthesis—for example, `COMP = (3.0000000000, 5.0000000000)`.

For dumped arrays,

- If the array is of type REAL, each element is printed as if the value were operated on by an R editing phrase. For more information, see “FORMAT Declaration.”
- If the array is of type BOOLEAN, the value of each element is shown as `.TRUE.` or `.FALSE.`
- If the array is of type INTEGER, each element is printed as an integer value.
- If the array is of type COMPLEX, each element is printed in the form used for complex variables for example, `CA = (2.0000000000, 3.0000000000), (5.0000000000, 7.0000000000)`.

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

Examples of DUMP Declarations

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

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

The following example dumps the values of I, INFO, and INDX to a file named PRNTR when the statement labeled NEXT is encountered. A label counter, DMPCOUNT, counts the number of times the statement labeled NEXT is encountered. Dumps occur until the value of DMPCOUNT exceeds DPHIGH. Note that when a label counter is specified, the counter can also be altered elsewhere in the program.

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

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

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

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

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

EPILOG PROCEDURE Declaration

The EPILOG PROCEDURE declaration enables you to designate a procedure that must be executed before exiting the block in which the EPILOG PROCEDURE declaration is contained. If an EPILOG PROCEDURE declaration exists for a block, the epilog procedure is automatically executed before exiting the containing block. The epilog procedure is not required to be invoked before exiting the containing block. However, the program can explicitly invoke an epilog procedure during execution, if you desire.

The epilog procedure functions differently if it is declared in a structure or connection block. Instead of being executed upon a block exit of the structure or connection block, epilog procedures in structure or connection blocks are executed when the block declaring the structure block variable or structure block array or connection library associated with that structure or connection block is exited. The epilog procedure of a structure block variable or array or connection library is called only if that structure block variable or array or connection library has been created. If the block declaring the structure block variable or structure block array or connection library contains an epilog procedure, the epilog for the structure or connection block is executed first.

<epilog procedure declaration>

```
— EPILOG PROCEDURE —<epilog procedure identifier>— ; —————→  
→<unlabeled statement>—————|
```

<epilog procedure identifier>

```
—<identifier>—————|
```

Restrictions on Epilog Procedures

The following restrictions apply to epilog procedures:

- No parameters are allowed.
- An invalid GO TO statement cannot be used to exit from an epilog procedure to an outer block. A run-time error occurs if an attempt is made to perform an invalid GO TO statement.
- An epilog procedure cannot return a value.
- An epilog procedure cannot contain an EPILOG PROCEDURE declaration. A block or procedure cannot have more than one EPILOG PROCEDURE declaration (or an EPILOG PROCEDURE declaration and an EXCEPTION PROCEDURE declaration).
- An epilog procedure cannot be declared as a formal parameter.
- Certain restrictions are placed on programs that contain EPILOG PROCEDURE declarations. Epilog procedures cannot be declared as EXTERNAL. Only replacement binding can be used. A block or procedure with a EPILOG PROCEDURE declaration in its outer block cannot be used as the host code file when running BINDER. No procedure in a block that has an EPILOG PROCEDURE can be replaced by the BINDER.

- If a program that contains one or more EPILOG PROCEDURE declarations fails due to a fatal stack overflow, the epilog procedures will not be executed.
- If the outer block (or procedure) of a program contains an EPILOG PROCEDURE declaration and the statistics option is TRUE, the epilog procedure is executed at block exit time before the statistics wrap-up code.
- If certain Data Management System (DMS) functions such as DATABASE OPEN or DATABASE CLOSE are called, it might not be possible to return to the epilog procedure if the executing task is discontinued.
- Structure or connection block epilog procedures can declare and invoke additional structure types and structure blocks. However, if a new structure or connection block instance is created as a result of calling the epilog procedure of another structure or connection block in the block, its own epilog procedure is not invoked if its declaration occurs after the point in the block that the declaration occurs for the structure or connection block instance whose epilog procedure is running.

In the following example, the only structure block that has been created (allocated) when the exit of the block occurs is SBA[1], which is at a lower lex offset than SBG.

```
BEGIN
TYPE STRUCTURE BLOCK SBT; FORWARD;
SBT ARRAY SBGA [0:9];
SBT SBG;

TYPE STRUCTURE BLOCK SBT;
BEGIN
EPILOG PROCEDURE EPI;
    BEGIN
    SBG.P;
    SBGA[9].P;
    END;    %EPI
PROCEDURE P;
    BEGIN
    END;    %P
END;    % SBT

SBGA[1].P;
END;    %BLOCK
```

In this example, the first call to EPI of SBT occurs for SBGA[1] which is after the point at which epilogs would have been called for SBG and SBGA[9], yet the epilog call for SBGA[1] causes both of those to become active.

The epilogs of elements of structure block arrays are called in reverse index order for the structure block array upon exit of the block that declared the structure block array. Only epilogs of structure block array elements that have been allocated are invoked.

EPILOG PROCEDURE Declaration

Every procedure with critical locking code or critical block exit code of some type should have an EPILOG PROCEDURE declaration in it. All critical block exit code must be contained in the epilog procedure. Whenever the procedure is exited (either normally or abnormally) the epilog procedure is executed.

The order for cleanup of blocks containing structure block variables or structure block arrays or connection libraries is the following:

1. If an epilog procedure for this block exists, it is run first.
2. Epilog procedures for existing structure or connection blocks emanating from this block run and any libraries linked to those structure or connection blocks are delinked.
3. Libraries linked to the block are delinked. Libraries delinked by BLOCKEXIT cannot be relinked.
4. Critical block action is taken for process family members.
5. The structure or connection block instances are cleaned up and destroyed.

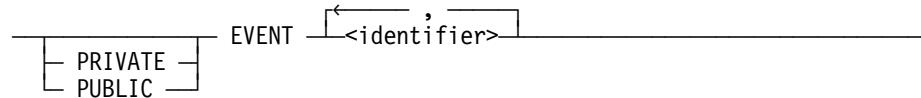
Example of an Epilog Procedure

```
BEGIN
FILE OUT(KIND=DISK,MAXRECSIZE=14,AREASIZE=420,AREAS=5);
ARRAY A[0:13];
REAL I;
EPILOG PROCEDURE CLEANUP;
  BEGIN
  %IF I=100 THEN PROGRAM TERMINATED NORMALLY
  %IF I<100 THEN PROGRAM TERMINATED ABNORMALLY
  REPLACE POINTER(A) BY "LAST RECORD, I=",I FOR 3 DIGITS;
  WRITE(OUT),14,A);
  LOCK(OUT,CRUNCH);
  END CLEANUP;
  .
  .
  .
```

EVENT and EVENT ARRAY Declarations

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

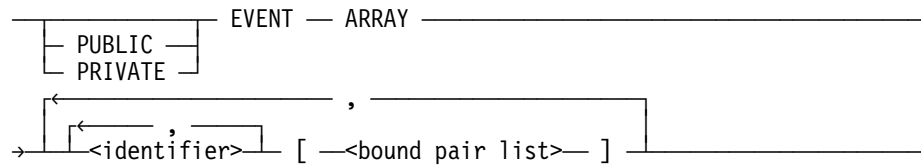
<event declaration>



<event identifier>

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

<event array declaration>



<event array identifier>

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

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

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

The PRIVATE and PUBLIC specifiers can only be used for events and event arrays that are declared within a structure block or a connection block. The PRIVATE specifier limits visibility of the event or event array to the scope of the structure block or the connection block. A PRIVATE event or event array element cannot be accessed using a structure or connection block qualifier. The PUBLIC specifier allows the event or event array element to be accessed using a structure or connection block qualifier. If neither PRIVATE nor PUBLIC is specified, the default value is PUBLIC and access to the event or event array element using a structure or connection block qualifier is allowed.

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

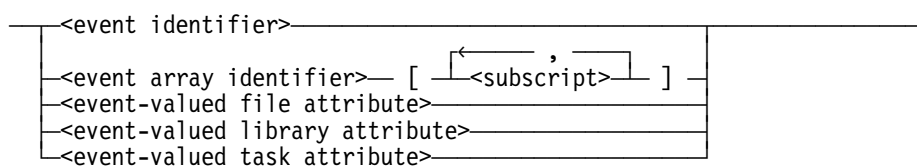
EVENT and EVENT ARRAY Declarations

The initial available state of an event is TRUE (available), and the initial happened state of an event is FALSE (not happened). For more information on events, refer to “Event Statement” in Section 4, “Statements.” For more information on interrupts, refer to “INTERRUPT Declaration” later in this section. For more information on the AVAILABLE function and the HAPPENED function, see Section 5, “Expressions and Functions.”

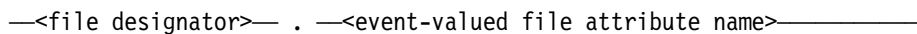
Event Designator

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

<event designator>



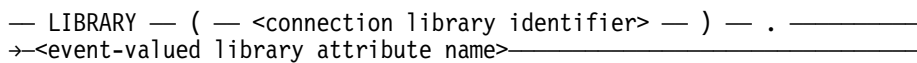
<event-valued file attribute>



<event-valued file attribute name>

ALGOL supports all file attributes and file attribute values described in the *File Attributes Programming Reference Manual*.

<event-valued library attribute>

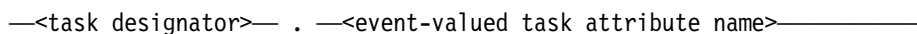


<event-valued library attribute name>

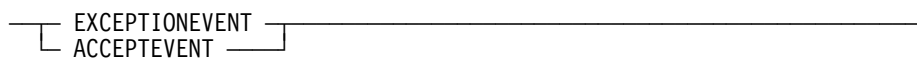


For more information on library attributes, see the *Task Management Programming Guide*.

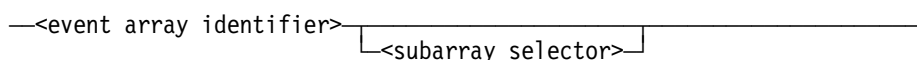
<event-valued task attribute>



<event-valued task attribute name>



<event array designator>



Examples of EVENT and EVENT ARRAY Declarations

The following example declares an event, FILEA.

```
EVENT FILEA
```

The following example declares an event array, SWAPPEE, which can store up to six events.

```
EVENT ARRAY SWAPPEE[0:5]
```

EXCEPTION PROCEDURE Declaration

The EXCEPTION PROCEDURE declaration enables you to designate a procedure that must be executed automatically by the system whenever an abnormal exit occurs for the block in which the EXCEPTION PROCEDURE declaration is contained.

<exception procedure declaration>

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

<exception procedure identifier>

```
—<identifier>—————|
```

An exception procedure is invoked when a block containing it terminates in any way other than a normal exit. It is not automatically invoked on a normal exit of the block. Abnormal exits include the following:

- A discontinue (DS) command
- A bad branch (GO TO) leading out of the block
- A branch to a global label in the block
- Any unhandled fault

An exception procedure can be invoked directly like any other procedure; it can be passed as a parameter (with limitations), and so on. The use of a procedure as an exception procedure is allowed only to the block that contains the procedure. A block or procedure cannot contain more than one EXCEPTION PROCEDURE declaration (or an EXCEPTION PROCEDURE declaration and an EPILOG PROCEDURE declaration).

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.

Restrictions on Exception Procedures

The following restrictions apply to exception procedures:

- An exception procedure must be a named, untyped procedure without parameters, and therefore cannot return a value.
- An exception procedure cannot contain parameters.
- An exception procedure cannot be specified as a formal parameter. However, an exception procedure can be passed as an actual parameter, if the formal parameter is an untyped procedure without parameters.
- An invalid GO TO statement cannot be used to exit from the exception procedure to an outer block. A run-time error occurs if an attempt is made to perform an invalid GO TO statement.
- An exception procedure cannot contain an EXCEPTION PROCEDURE declaration or an EPILOG PROCEDURE declaration.
- An exception procedure cannot be declared as EXTERNAL. Only replacement binding can be used. A block or procedure with an EXCEPTION PROCEDURE declaration in its outer block cannot be bound to. The exception procedure body cannot be a library entry point specification or a dynamic procedure specification.
- For other than the outer block of a program, certain string handling declarations, statements, expressions, and functions cause a string pool epilog procedure to be generated for that block to return the string space that was allocated. If a string pool epilog procedure is required for the same block in which an exception procedure is declared, a warning message is generated at compile time because the exception procedure will not be executed at run time for a nonfatal stack overflow fault.
- 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;
```

Note: *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.*

EXCEPTION PROCEDURE Declaration

Example of an EXCEPTION PROCEDURE Declaration

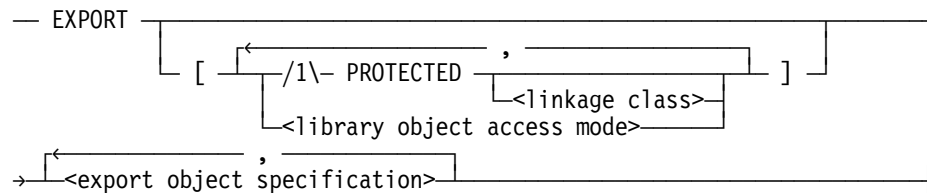
```
PROCEDURE P1;
  BEGIN
    REAL A, B;
    FILE MYFILE (KIND=DISK);
    EXCEPTION PROCEDURE CLEANUP;
      BEGIN
        CLOSE (MYFILE, LOCK);
      END; %OF EXCEPTION PROCEDURE CLEANUP

    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 WE EXIT P1 ABNORMALLY.
```

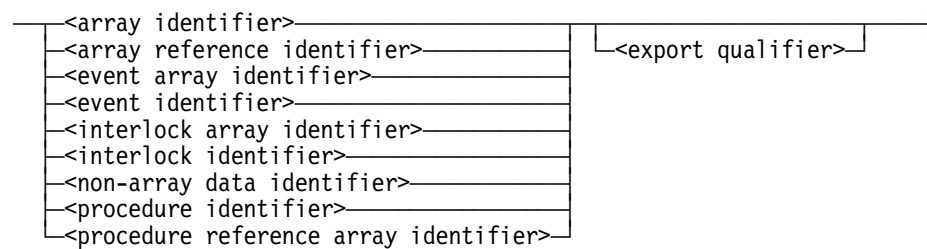
EXPORT Declaration

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

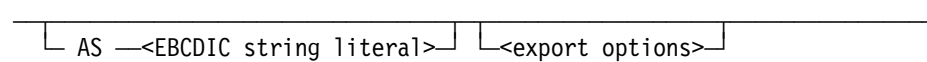
<export declaration>



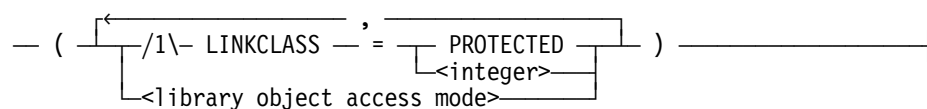
<export object specification>



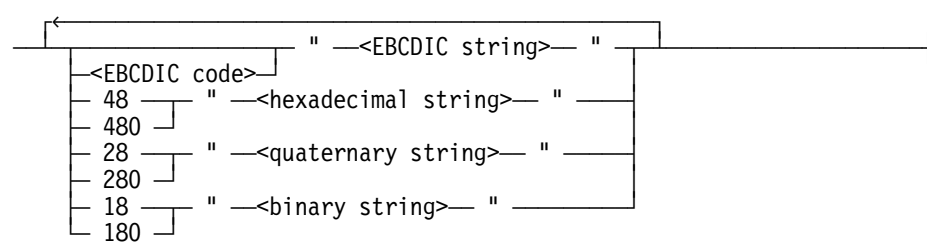
<export qualifier>



<export options>



<EBCDIC string literal>



<non-array data identifier>



<library object access mode>

—/1\ — READONLY —
└ READWRITE ┘

All procedures to be exported must be declared before the appearance of the EXPORT declaration and must be declared in the same block as the EXPORT declaration. All data to be exported must be declared no later than the appearance of the EXPORT declaration and must be declared in the same block as the EXPORT declaration.

A program can give an exported access mode to an exported data library object by specifying a <library object access mode>. A <library object access mode> can be specified only for data library objects. Data library objects include <non-array data identifier>s and <data array identifier>s. The READONLY option indicates that the exporter is allowing read-only access to the data. The READWRITE option indicates that the exporter is allowing both read and write access to the data. The access mode, if not specified, defaults to READONLY. An exported access mode is not allowed for an EVENT, an EVENT ARRAY, an INTERLOCK, or an INTERLOCK ARRAY. If all the library objects in the EXPORT declaration are data library objects and they all have the same access mode, the access mode can be specified in the square brackets just after the EXPORT keyword.

Multidimensional arrays can be included in an EXPORT declaration if they are specified with a READWRITE access mode.

A program can export an array row by specifying the name of the array, which was declared with one dimension, as an <array identifier> in the EXPORT declaration. No bounds should be specified. A multi-dimensioned array cannot be exported, but array row equivalence can be used to export one of its rows. Value arrays cannot be exported, but array references can be exported, and the array reference can be assigned the value array in an array reference assignment.

To provide a library object with a security level, the object can be exported with a linkage class assigned to it. The linkage class of the user program, which is assigned by the system, is matched to the linkage class of the exported object on a per object basis to determine if visibility is allowed to the calling program. The PROTECTED linkage class provides the highest level of security. The values 0 through 15 can be assigned. The default linkage class is 0, which provides the lowest level of security.

A procedure reference array can be exported. Any type or parameters allowed for an exported procedure can be reference by an exported procedure reference array.

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

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

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

The EBCDIC string literal in the AS clause can contain only valid <printable character>s, but must not contain any leading, trailing, or embedded blanks or periods. The entry point name indicated by this string literal will be case sensitive.

Library Entry Point Types and Parameters

A library entry point can be any of the following:

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

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

- ASCII character array
- ASCII string variable or array
- Boolean variable or array
- Complex variable or array
- Double variable or array
- EBCDIC character array
- EBCDIC string variable or array
- Event variable or array
- File
- Hexadecimal character array
- Hexadecimal string variable or array
- Integer variable or array

EXPORT Declaration

- Pointer
- Real variable or array
- Task variable or array

A parameter to a library entry point can also be a formal procedure with the above restrictions on its type and parameters. The formal procedure must be fully specified, that is, the <formal parameter specifier> construct of the PROCEDURE declaration must be used. A fully-specified formal procedure cannot take a by-reference pointer as a parameter.

Conditions in Which Errors Can Occur

A library can export a procedure that is declared to be an entry point in yet another library. When a program calls this entry point, the template of the library to which the procedure is declared to belong is searched for an entry point with the same name as that of the called entry point in the directory for this library.

For example, assume the following declarations have been compiled:

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

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

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

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

Another program calls entry point P of this library, and the template for library L is searched for an entry point named P. If it is found, that entry point is called. If that entry point is not found, a run-time error is given and the calling program is terminated. In either case, procedure LIBPROC of library L is not executed. For more information on libraries, refer to Section 8, "Library Facility."

A library entry point must not declare any OWN arrays. An attempt to execute a library entry point that declares an OWN array results in a run-time error.

If a library exports a procedure reference array, a program importing that procedure reference array can access the procedures in that library. A program cannot assign into an element of an imported procedure reference array. If such an assignment is attempted, a compile-time or run-time error occurs.

Programs that export procedure reference arrays cannot be used for binding.

Examples of EXPORT Declarations

The following example declares the procedure EXPROC as an entry point in a library program.

```
EXPORT EXPROC
```

The following example declares the procedure PROC1 as an entry point in a library program.

The name exported for this procedure is LIBPROC3; consequently, a program calls PROC1 in this library by using the name LIBPROC3.

```
EXPORT PROC1 AS "LIBPROC3"
```

The following example declares the procedure reference arrays PRA1 and PRA2 and the procedure PROCID as entry points in a library program. The procedure reference array PRA2 is exported with the name PROCREF; consequently, a program must use the name PROCREF to call PRA2.

```
EXPORT PRA1, PROCID, PRA2 AS "PROCREF"
```

EXPORTLIBRARY Declaration

The EXPORTLIBRARY declaration allows a CHANGE procedure to be associated with a regular server library, that is, a library that contains an EXPORT list and a FREEZE statement. The CHANGE procedure is useful for providing linkage and delinkage notification about callers to the library.

<exportlibrary declaration>

— EXPORTLIBRARY — (—<procedure library attribute specification>—→
→) —————|

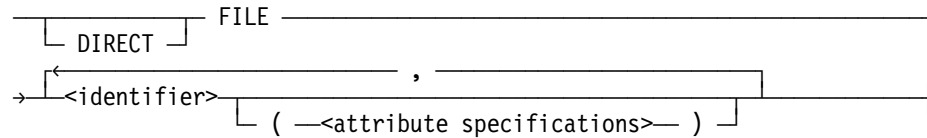
The EXPORTLIBRARY declaration must appear in the same block as the EXPORT list and FREEZE statement for the library.

For EXPORTLIBRARY, the only kind of procedure-valued library attribute name that can be specified is the CHANGE attribute.

FILE Declaration

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

<file declaration>



Identifiers

<file identifier>

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

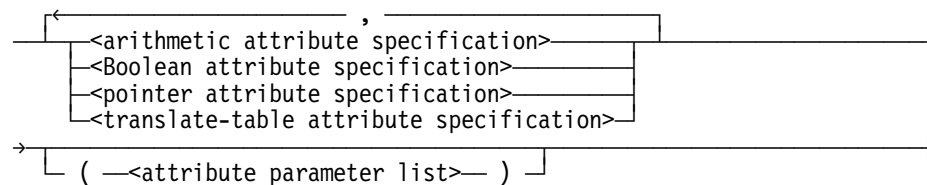
<direct file identifier>

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

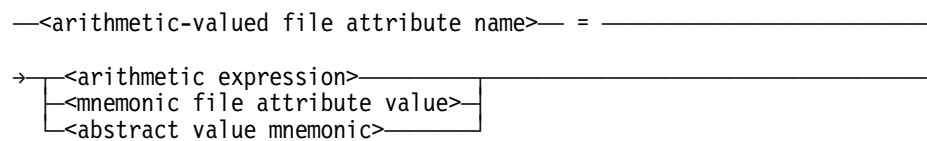
If DIRECT is specified, the file is declared as a direct file to be used for direct I/O.

Attribute Specifications

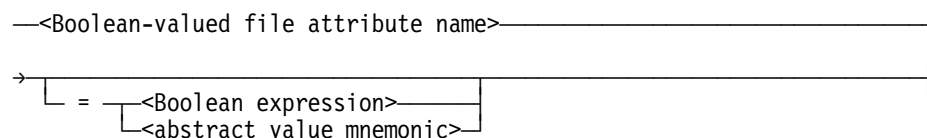
<attribute specifications>



<arithmetic attribute specification>

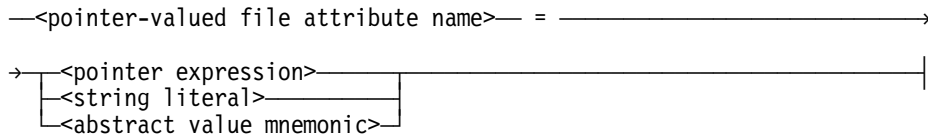


<Boolean attribute specification>

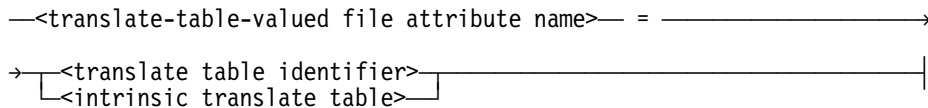


FILE Declaration

<pointer attribute specification>



<translate-table attribute specification>



<attribute parameter list>



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

ALGOL supports all file attributes and file attribute values described in the *File Attributes Programming Reference Manual*.

File declaration can be used to set a file attribute to one of the abstract value mnemonics allowed for that file attribute.

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

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

For example, the result of the following declaration is that the KIND attribute of file C is assigned the value DISK, and the KIND attributes of files A and B are assigned the default value for the KIND attribute, which might or might not be the value DISK.

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

For more information on file attributes and their default values, refer to the *File Attributes Programming Reference Manual*.

Valid mnemonics for file attributes are also type 3 reserved words in ALGOL. When it is not certain whether an identifier is a variable or a mnemonic, the compiler always assumes that the identifier is a variable or other local identifier if it is enclosed in parentheses. If the identifier is not enclosed in parentheses, the compiler always assumes it is a mnemonic. This rule can be used to resolve ambiguities when new attributes with names that conflict with local variables are added to the system.

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

When the value of a *<pointer-valued file attribute name>* is a *<string literal>*, the last character of the *<string literal>* must be a period (".").

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

Attribute parameters are allowed in FILE declarations and MULTIPLE ATTRIBUTEASSIGNMENT statements. In a FILE declaration, the attribute specifications cannot reference the file identifier of the file being declared. For example, the following is not valid:

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

Examples of FILE Declarations

The following example declares a file named F.

```
FILE F
```

The following example declares a file named NEWFILE. This FILE declaration is the first step in creating a new disk file with the title DATA on a pack named PACK. Synchronized output, which is useful for auditing and recovery, will be performed.

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

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

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

The following FILE declaration sets the arithmetic-valued file attribute AREAS to its maximum value:

```
FILE F (AREAS = MAXIMUM)
```

FORMAT Declaration

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

<format declaration>

— FORMAT —<in-out part> [<format part>]

In-Out Part

The in-out part affects the processing of simple string literals appearing in the editing specifications. If the in-out part of a FORMAT declaration is designated as OUT or unspecified (in which case OUT is assumed), simple string literals appearing in the editing specifications of the format are read-only. If the in-out part is designated as IN, such simple string literals are read-write. For more information, refer to “Simple String Literal” later in this section.

<in-out part>

[IN]
[OUT]

Format Part

<format part>

—<identifier> [(—<editing specifications>—)]
[< —<editing specifications>— >]

<format identifier>

An identifier associated with a set of editing specifications in a FORMAT declaration.

<editing specifications>

[/] [<simple string literal> [/]]
[/] [<repeat part> [(—<editing specifications>—)]]

<repeat part>

—<unsigned integer>—
[*]

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

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

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

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

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

Example of Editing Specifications

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

FORMAT Declaration

Assume that the following two input records are used:

```
1234
5678
```

Using these two input records, this program produces the following output:

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

56
```

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

Simple String Literal

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

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

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

Example of Simple String Literal

Assume you are using the following statement:

```
WRITE(LINE,<4"C1C2",8"ABC">);
```

This statement produces the following output:

```
ABABC
```

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

Simple string literals appearing in editing specifications can be read-only or read/write, depending on the in-out part specified in the FORMAT declaration. If the in-out part is IN, simple string literals appearing in the editing specifications are read-write, and the format can be used in both READ statements and WRITE statements. When a format used in a READ statement is declared with an in-out part of IN and contains a simple string literal in the editing specifications, then data is read into the memory location of the simple string literal over the original value.

The number of characters read always equals the length of the simple string literal as it is defined in the FORMAT declaration. When the format is used in a subsequent WRITE statement, the new data is written to the output record. If the in-out part is OUT or unspecified (in which case OUT is assumed), any simple string literals appearing in the editing specifications are read-only. Any attempt to change the value of a read-only simple string literal by using that format in a READ statement results in a run-time error.

Repeat Part

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

Editing specifications and their corresponding repeat parts can be nested. For example, assume you are using the following WRITE statement:

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

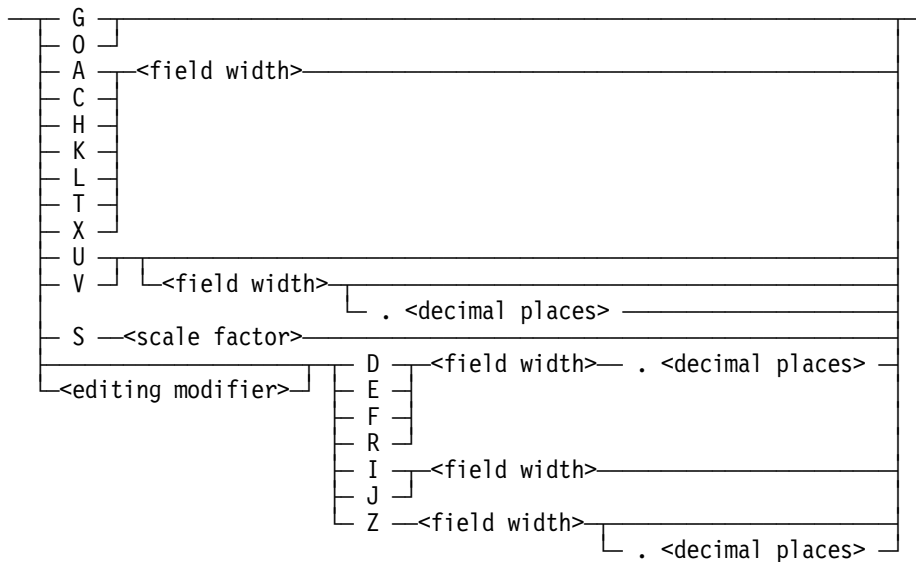
The first repeat part indicates that the editing specifications (2(2I3)) are to be repeated twice, the second repeat part indicates that the editing specifications (2I3) are to be repeated twice, and the third repeat part specifies that the editing phrase I3 is to be repeated twice, causing the editing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

The following examples show the correct syntax of repeat parts:

```
3F10.4  
3(A6/)  
3(3A6,3(/I12)/)
```

Editing Phrases

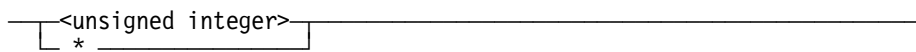
<editing phrase>



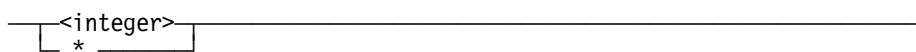
<field width>



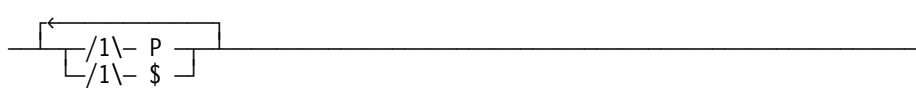
<decimal places>



<scale factor>



<editing modifier>



Field Width

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

Decimal Places

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

Variable Editing Phrases

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

Examples of Variable Editing Phrases

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

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

When an asterisk is used as the repeat part, the number of repetitions performed depends on the value supplied by the list element. If the value of the list element is greater than 0 (zero), that number of repetitions is performed; if the value is equal to 0, an unlimited number of repetitions are performed. If the value is less than 0, no repetitions are performed, and control passes to the next editing phrase.

When an asterisk is used for the field width of an editing phrase, the actual width of the field depends on the value supplied by the list element. If the value of the list element is greater than 0 (zero), that value is used as the width of the field. If the value of the list element is less than or equal to 0, no editing is performed, the list elements corresponding to the editing phrase are skipped, and control passes to the next editing phrase.

Editing Phrase Letters

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

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

Letter	Meaning
r	<repeat part>
w	<field width>
d	<decimal places>

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

In the examples in the following sections, the lowercase letter *b* is used to denote a blank character.

A and C Editing Phrase Letters

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

When A is used, characters are read from, or written to, the word starting at the rightmost position. If C is used, the starting character position is the leftmost one.

In the explanations of the editing phrase letters A and C, the letter Q is used.

The value of Q is 6 for single-precision and 12 for double-precision, and if the list element is of the following form, then the value of the arithmetic expression is used as the value of Q:

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

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

Input

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

For the editing phrase letter A, if w is greater than or equal to Q , the rightmost Q characters of the input field are transferred to the list element. If w is less than Q , then w characters of the input field are transferred right-justified to the list element. The unused high-order bits of the list element are set to 0 (zero).

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

The following are input examples for the A and C editing phrase letters. The default character type is 8-bit in all cases.

External String	Editing Phrase	Internal Value
ABCDEFGHijkl	A9	8"DEFGHI"
AbCbEbGblbK	A4	4"0000"8"AbCb"
ABCDEFGHijkl	A12	ABCDEFGHijkl (pointer as list element)
ABCDEFGHijkl	A12	4"0000"8"ABCDEFGHijkl" (8-bit pointer FOR 14)
ABCDEFGHijkl	C9	8"DEFGHI"
ABCD	C4	8"ABCD"4"0000"
ABCDEFGHijkl	C12	8"ABCDEFGHijkl"4"0000" (8-bit pointer FOR 14)

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

Output

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

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

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

FORMAT Declaration

The following are output examples for the A and C editing phrase letters:

External String	Editing Phrase	Internal Value
8"DEFGHI"	A9	bbbDEFGHI
4"0000000000"8"A"	A4	??A
8"ABCDEFGH" (8-bit pointer FOR 7)	A11	bbbbABCDEFGH
8"DEFGHI"	C9	bbbDEFGHI
8"ABCD"4"0000"	C5	ABCD?
8"ABCDEFGH" (8-bit pointer FOR 7)	C11	bbbbABCDEFGH

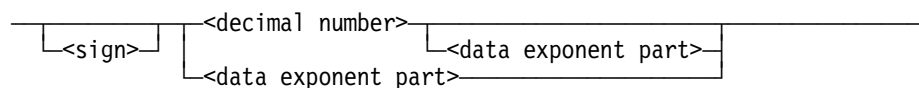
D Editing Phrase Letter

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

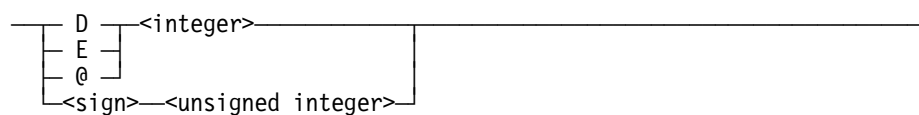
Input

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

<data number>



<data exponent part>



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

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

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

The following are input examples for the D editing phrase letter:

External String	Editing Phrase	Internal Value
bbbbbb25046	D11.4	+2.5046
bbbb25.046	D11.4	+25.046
-bb25046E-3	D11.4	-0.0025046
-bbb25046-3	D11.4	-0.0025046
bb250.46D-3	D11.4	+0.25046
bbb250.46-3	D11.4	+0.25046
b-b25.04678	D11.4	-25.04678

Output

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

The value of *w* must be greater than or equal to $d + 7$. This width allows for a 4-character exponent part, a decimal point, a digit preceding the decimal point, and a sign. If *w* is less than $d + 7$, the field is filled with asterisks (*).

FORMAT Declaration

The editing phrase letter D always uses four or seven characters to represent the exponent of the list element being written. The magnitude of the exponent determines in which syntactic form the exponent is expressed:

Magnitude of Exponent	Form
4-character	D+xx or D-XX (where ABS(XX) <= 99)
4-character	+XXX or -XXX (where 100 <= ABS(XXX) <= 999)
7-character	D+XXXXX or D-XXXXX (where 1000 <= ABS(XXXXX) <= 99999)

The following are output examples for the D editing phrase letter:

Internal Value	Editing Phrase	External String
+36.7929	D13.5	bb3.67929D+01
-36.7929	D12.5	-3.67929D+01
-36.7929	D11.5	*****
+36.7929	D10.5	*****
1.234@@-73	D14.5	bbb1.23400D-73
-789@@1234	D15.3	bb-7.890D+01236
6.54@@321	D9.2	b6.54+321

E Editing Phrase Letter

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

The following are output examples for the E editing phrase letter:

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

F Editing Phrase Letter

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

Input

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

Output

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

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

The following are output examples for the F editing phrase letter:

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

G Editing Phrase Letter

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

H and K Editing Phrase Letters

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

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

Editing Phrase Letter		
	H	K
Single-precision	12	16
Double-precision	24	32

Input

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

If w is less than or equal to Q , the value is stored right-justified in the storage location. Both words of a double-precision variable are included. Unused high-order bits are set to 0 (zero). If w is greater than Q , the leftmost $w - Q$ characters must be blanks, zeros, or minus signs; otherwise, a data error is returned.

The following are input examples for the H and K editing phrase letters:

External String	Editing Phrase	Internal Value
6F	H2	4"00000000006F"
1FFFFFFFFF	H12	4"1FFFFFFFFF"
-16	H3	4"40000000016"
1234b568	H8	4"000012340568"
FFCb	H4	4"0000000FFC0"
00C1C2C3C4C5C6	H14	4"C1C2C3C4C5C6"
-ABCD	H5	4"40000000000000000000ABCD" (double-precision)
123456789ABCDEF	H15	4"00000000123456789ABCDEF" (double-precision)
16	K2	3"000000000000016"
1777777777777777	K16	3"1777777777777777"
-16	K3	3"200000000000016"
1234b56	K7	3"000000001234056"
77b	K3	3"000000000000770"
-567	K4	3"200000000000000000000000000000567" (double-precision)
1234567654321234567	K19	3"00000000000001234567654321234567" (double-precision)

FORMAT Declaration

Output

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

The following are output examples for the H and K editing phrase letters:

Internal Value	Editing Phrase	External Value
4"0000E5551010"	H5	51010
4"0000E5551010"	H12	0000E5551010
4"0000E5551010"	H16	bbbb0000E5551010
8"123456"	H12	F1F2F3F4F5F6
4"000000000000000012345678 (double-precision)	H4	5678
8"123456789bbb" (double-precision)	H24	F1F2F3F4F5F6F7F8F9404040
3"0005677701234445"	K5	34445
3"0005677701234445"	K16	0005677701234445
3"0005677701234445"	K18	bb0005677701234445
3"00000000000000000000000000001234567" (double-precision)	K4	4567

I Editing Phrase Letter

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

Input

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

The following are input examples for the I editing phrase letter:

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

Output

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

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

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

The following are output examples for the I editing phrase letter:

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

J Editing Phrase Letter

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

Input

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

Output

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

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

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

The following are output examples for the J editing phrase letter:

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

L Editing Phrase Letter

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

Input

The editing phrase letter L specifies that *w* characters of input data are to be read, converted to one of the Boolean values TRUE or FALSE, and assigned to the corresponding list element. If the first nonblank character of the input data is the letter T, then bit 0 (zero) of the list element is assigned the value 1; otherwise, bit 0 is assigned the value 0 (zero). All other bits in the list element are assigned the value 0 (zero). An all-blank field yields the value FALSE. If the list element is a double-precision variable, the first word is assigned a value according to the rules just described, and the second word is set to 0 (zero).

The following are input examples for the L editing phrase letter:

External String	Editing Phrase	Internal Value
T	L1	TRUE (4"000000000001")
bbF	L3	FALSE (4"000000000000")
bbbTRU	L6	TRUE (4"000000000001")
b	L1	FALSE (4"000000000000")
T	L1	TRUE (4"0000000000010000000000") (double-precision)

Output

On output, the editing phrase letter L specifies that TRUE is to be written to the output record if bit 0 (zero) of the corresponding list element equals 1, and the letter L specifies that FALSE is to be written if bit 0 (zero) of the corresponding list element equals the number 0. If *w* is less than 5, the first *w* characters of TRUE or FALSE are written. If *w* is greater than 4, TRUE or FALSE is written right-justified with blank fill.

The following are output examples for the L editing phrase letter:

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

O Editing Phrase Letter

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

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

	Precision		Pointers	
	Single	Double	4-bit	8-bit
EBCDIC	6	12	12	6

Input

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

`<pointer expression> FOR <arithmetic expression>`

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

Output

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

`<pointer expression> FOR <arithmetic expression>`

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

Example of Input and Output

The following example shows the use of the editing phrase letter O:

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

Input and output data for this example are as follows:

Input	Output
A	A
ABCDEFGH	ABCDEF

R Editing Phrase Letter

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

Input

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

Output

On output, the editing phrase letter R specifies that the value of the corresponding list element is to be converted to a string that expresses the value in either simple decimal or exponential notation.

In general, if *w* is greater than or equal to the number of characters required to express the value of the list element using simple decimal notation, then simple decimal notation is used. If *w* is less than the number of characters required to express the value using simple decimal notation and greater than or equal to the number of characters required to express the value using exponential notation, then exponential notation is used. If *w* is less than the number of characters required to express the value using exponential notation, the field is filled with asterisks (*).

FORMAT Declaration

Examples of Input and Output

External Input String	List Element Type	Editing Phrase	External Output String
-.333333bb	REAL	R10.4	bbb-0.3333
-.333333bb	DOUBLE	R10.4	bbb-0.3333
-.333333bb	INTEGER	R10.4	bbbb0.0000
3333.333E2	DOUBLE	R10.4	3.3333D+05
3333.333E2	INTEGER	R10.4	3.3333E+05
-.333bbbbb	REAL	R10.9	*****
-.333bbbbb	INTEGER	R10.9	.000000000
333.333E2b	DOUBLE	R10.4	3.3333D+22
bbbbbbbbbbbbbb1.23D12	REAL	R20.4	bb123000000000.0000
bbbbbbbbbb1.23D12345	DOUBLE	R20.4	bbbbbb1.2300D+12345
bbbb4.3@68	REAL	R10.4	4.3000E+68

S Editing Phrase Letter

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

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

Input

On input, the value of the input data corresponding to the subsequent R editing phrase is divided by the following number before the input data value is assigned to the list element:

10 ** <scale factor>

The following are input examples for the S editing phrase letter:

External String	Editing Specifications	Internal Value
bbbb10000.	S2,R10.2	100.0
bbbbbb5.41	S1,R10.2	0.541
bbbbbb05.5	S1,R10.2	0.55
bb5.01521	S-1,R10.2	50.1521
bbbbbb541	S1,R10.2	0.541

Output

On output, the value of the list element corresponding to the subsequent R editing phrase is multiplied by the following number before the list element value is written to the output field:

10 ** <scale factor>

The following are output examples for the S editing phrase letter:

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

FORMAT Declaration

T Editing Phrase Letter

The editing phrase letter T specifies that the buffer pointer is to be moved to character position *w* of the input or output record. The value of *w* must be greater than 0 (zero); if *w* is equal to 0, the buffer pointer is moved to the first character position in the record. No list element corresponds to this editing phrase letter.

Input Examples

External String	Editing Specifications	Internal Value
012345678910111213	T13,I3	111
012345678910111213	T1,I4	123
012345678910111213	T15,I4	1213
ABCDEFGHIJKLMNPOQR	T8,A6	HIJKLM

Output Example

```
BEGIN
  FILE DCOM(KIND=REMOTE,MYUSE=IO);
  ARRAY A[0:9];
  WRITE(DCOM, <T11,I3,T2,I3>, 123, 456); % WRITE STATEMENT 1
  WRITE(DCOM, <T4,A3,T1,A2>, "ABC", "DE"); % WRITE STATEMENT 2
END.
```

This program produces the following output:

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

U Editing Phrase Letter

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

Input

The editing phrase letter U is not implemented for input.

Output

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

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

If d is specified and d is greater than w , then $d - w$ leading blanks are inserted in the value that is written to the list element before the field is written.

Thus, when the editing phrase letter U is used, the number of characters actually written cannot be less than d and can be greater than w .

The following are output examples for the U editing phrase letter:

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

V Editing Phrase Letter

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

The editing phrase letter extracted from the list element is an 8-bit character.

Example of V Editing Phrase Letter

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

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

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

X Editing Phrase Letter

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

Z Editing Phrase Letter

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

Input

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

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

Output

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

The following are output examples for the Z editing phrase letter:

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

Editing Modifiers

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

P Editing Modifier

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

\$ Editing Modifier

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

Examples of P and \$ Editing Modifiers

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

Examples of FORMAT Declarations

The following examples illustrate the FORMAT declaration syntax:

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

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

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

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

```
FORMAT FMT1(*I*)
```

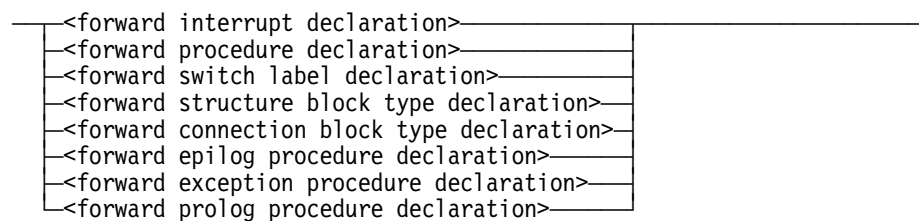
```
FORMAT FMT2(*V*.*)
```

FORWARD REFERENCE Declaration

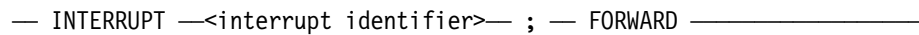
The FORWARD REFERENCE declaration enables the ALGOL compiler to handle situations in which two procedures, two interrupts, two switch labels, two connection blocks, or two structure blocks make references to each other. Normally, a procedure, interrupt, switch label, connection block, or structure block must be declared before it can be used in a program. However, if two such entities make reference to each other, regardless of which procedure, interrupt, switch label, connection block, or structure block is declared first, then the body of the procedure, interrupt, switch label, connection block, or structure block contains a reference to an undeclared entity.

The FORWARD REFERENCE declaration enables the compiler to recognize such entities before they have been declared in full. When a procedure is declared in full, the declaration must match the FORWARD REFERENCE declaration in its type. Also, if there are parameters, these must also match the FORWARD REFERENCE declaration in number and type.

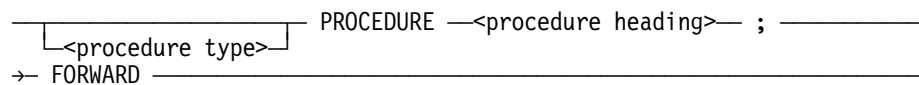
<forward reference declaration>



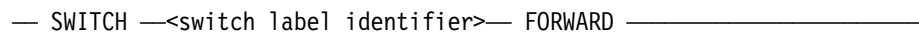
<forward interrupt declaration>



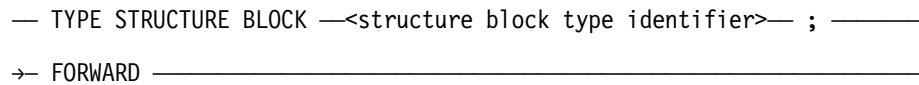
<forward procedure declaration>



<forward switch label declaration>



<forward structure block type declaration>



FORWARD REFERENCE Declaration

<forward connection block type declaration>

```
— TYPE CONNECTION BLOCK —<connection block type identifier>— ; —————→  
→ FORWARD ——————|
```

<forward epilog procedure declaration>

```
— EPILOG PROCEDURE —<epilog procedure identifier>— ; — FORWARD ———|
```

<forward exception procedure declaration>

```
— EXCEPTION PROCEDURE —<exception procedure identifier>— ; —————→  
→ FORWARD ——————|
```

<forward prolog procedure declaration>

```
— PROLOG PROCEDURE —<prolog procedure identifier>— ; — FORWARD ———|
```

Order of Referencing

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

```
PROCEDURE PROC_TWO; FORWARD
```

When PROC_ONE calls PROC_TWO, the compiler recognizes the second procedure. Later in the program, the second procedure, PROC_TWO, is declared in full.

Similar methods are used for mutually referencing interrupts, switch labels, structure block types, and connection block types.

Examples of FORWARD REFERENCE Declarations

The following example declares a forward reference to a switch label named SELECT. Later in the program, SELECT must be declared in full.

```
SWITCH SELECT FORWARD
```

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

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

The following example declares a forward reference to a connection block type named CB. Later in the program, CB must be declared in full.

```
TYPE CONNECTION BLOCK CB; FORWARD
```

IMPORTED Declaration

An IMPORTED declaration can be specified only within a CONNECTION BLOCK declaration.

<imported declaration>

— IMPORTED —<imported data specification>—————|

<imported data specification>

—<data type>—|<identifier>—, —————|
 | |
 |<imported array specification>—|
 |
 |<library object attributes>—|

<imported array specification>

—<data type>—|<character type>—| ARRAY —————>
 | |
 →|<identifier>— [—<lower bound>—] —————|
 |
 |<library object attributes>—|

<library object attributes>

— (—|</1\— ACTUALNAME — = —<EBCDIC string literal>—|) —————|
 | |
 |<library object access mode>—|

<data type>

—| BOOLEAN —————|
 —| COMPLEX —————|
 —| DOUBLE —————|
 —| EVENT —————|
 —| INTEGER —————|
 —| INTERLOCK —————|
 —| REAL —————|

A program can give an imported access mode to a data library object by specifying a <library object access mode>. The READONLY option indicates that the importer wants read-only access to the data. The READWRITE option indicates that the importer wants both read and write access to the data. The access mode, if not specified, defaults to READONLY. An imported access mode is not allowed for an EVENT, an EVENT ARRAY, an INTERLOCK, or an INTERLOCK ARRAY.

The imported access mode, as well as the exported access mode, is used to determine whether two data library objects match each other. A library object exported READONLY matches only a library object imported READONLY. A library object exported READWRITE can match a library object imported either READONLY or READWRITE. If the library objects match, the actual access mode used for accessing the data is determined by the imported access mode. If the imported access mode is READONLY, any attempt to write to the data object results in a memory protection error.

Multidimensional arrays can be imported inside of Connection Blocks if they are specified with a READWRITE access mode.

Note: *IMPORTED EVENTS and EVENT ARRAYS cannot be used as:*

- *An <event designator> in an <event list> for the WAIT and WAITANDRESET statements*

However, if the compiler control option WAITIMPORT is set, an IMPORTED EVENT that is declared in a CONNECTION BLOCK can be used as an <event designator> in an <event list> for a WAIT or WAITANDRESET statement.

- *An <event designator> in a direct I/O READ or WRITE statement*
- *An <event designator> in an ATTACH statement*
- *An <event designator> for the LOCK Interlock statement*
- *Actual parameters in a PROCEDURE INVOCATION statement or a PROCEDURE REFERENCE statement*

Examples of IMPORTED Declaration

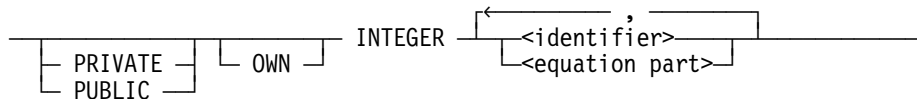
```
IMPORTED ARRAY D[0] (ACTUALNAME = "AA"),  
                B[3];
```

```
IMPORTED INTEGER I (READWRITE), J, K;
```

INTEGER Declaration

An INTEGER declaration declares simple variables that can have integer values, that is, arithmetic values that have exponents of 0 (zero) and no fractional parts.

<integer declaration>



<integer identifier>

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

The PRIVATE and PUBLIC specifiers can only be used for simple variables that are declared within a structure block or a connection block. The PRIVATE specifier limits visibility of the simple variable to the scope of the structure block or the connection block. A PRIVATE simple variable cannot be accessed using a structure or connection block qualifier. The PUBLIC specifier allows the simple variable to be accessed using a structure or connection block qualifier. If neither PRIVATE nor PUBLIC is specified, the default value is PUBLIC and access to the simple variable using a structure or connection block qualifier is allowed.

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

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

When an integer simple variable is allocated, it is initialized to 0 (zero) (a 48-bit word with all bits equal to zero).

See Appendix C, "Data Representation," for additional information on the internal structure of an integer operand.

Equation Part

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

Address equation is allowed only between INTEGER, REAL, and BOOLEAN variables. Because both identifiers of the equation part have the same address, altering the value of either variable affects the value of both variables. For more information, see "Type Coercion of One-Word and Two-Word Operands" in Appendix C, "Data Representation."

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

Examples of INTEGER Declarations

The following example declares INDEX as an integer simple variable.

```
INTEGER INDEX
```

The following example declares COUNT, VAL, and NOEXPONENT as integer simple variables.

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

The following example declares SAVEVALUE and MAX as integer simple variables. Because they are declared to be OWN, these simple variables retain their values when the program exits the block in which the simple variables are declared.

```
OWN INTEGER SAVEVALUE,MAX
```

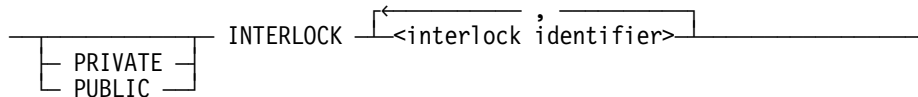
The following example declares INT and CAL as integer simple variables, and address-equates INT to the previously declared simple variable BOOL. The variables INT and BOOL share the same address.

```
INTEGER INT = BOOL,
```

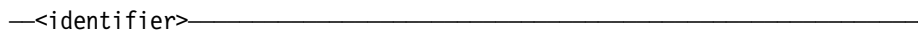
INTERLOCK and INTERLOCK ARRAY Declarations

An interlock, or an interlock array element, protects a resource that is shared by several participating processes. Using an interlock with the LOCK and UNLOCK functions is similar to using an event with the PROCURE and LIBERATE statements. Using interlocks, however, can significantly improve the run-time performance, compared with using the PROCURE and LIBERATE statements on events.

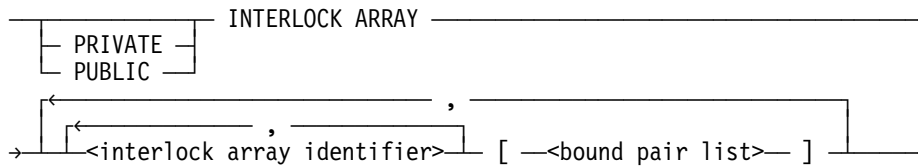
<interlock declaration>



<interlock identifier>



<interlock array declaration>



<interlock array identifier>



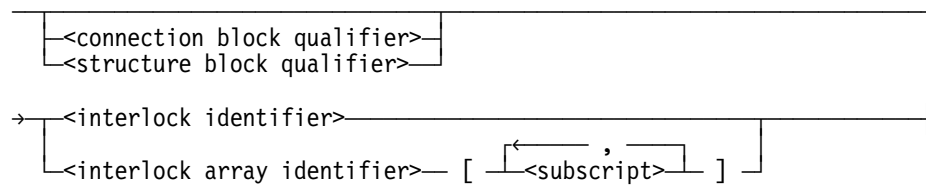
The PRIVATE and PUBLIC specifiers can only be used for interlocks and interlock arrays that are declared within a structure block or a connection block. The PRIVATE specifier limits visibility of the interlock or interlock array to the scope of the structure block or the connection block. A PRIVATE interlock or interlock array element cannot be accessed using a structure or connection block qualifier. The PUBLIC specifier allows the interlock or interlock array element to be accessed using a structure or connection block qualifier. If neither PRIVATE nor PUBLIC is specified, the default value is PUBLIC and access to the interlock or interlock array element using a structure or connection block qualifier is allowed.

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 5.

Interlock Designator

An interlock designator represents a single interlock.

<interlock designator>



When an interlock or interlock array is declared within a connection or structure block, you must use a connection or structure block qualifier to reference the interlock identifier or interlock array identifier from outside of the connection or structure block.

INTERRUPT Declaration

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

<interrupt declaration>

— INTERRUPT —<identifier>— ; —<unlabeled statement>—————|

<interrupt identifier>

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

Interrupting a Program

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

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

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

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

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

For more information, refer to “ATTACH Statement,” “DETACH Statement,” “DISABLE Statement,” and “ENABLE Statement” in Section 4, “Statements.”

Examples of INTERRUPT Declarations

The following example declares ERR to be an interrupt and associates with it the statement GO TO ABORT.

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


The following example declares BLOCK1 to be an interrupt. When BLOCK1 is invoked, two messages are displayed. Because a GO TO statement does not occur within the declaration, after the interrupt code is executed, the program continues from the point at which the interrupt occurred.

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

LABEL Declaration

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

<label declaration>

— LABEL  

<label identifier>

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

Using Label Identifiers

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

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

Examples of LABEL Declarations

The following example declares START as a label.

```
LABEL START
```

The following example declares ENTER, EXIT, START, and LOOP as labels.

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

LIBRARY Declaration

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

<library declaration>

— LIBRARY [<library specification>]

<library specification>

— <identifier> [(<library attribute specifications>)]
 → [<library object declaration list>]

<library object declaration list>

— [[<library object declaration>]]

<library object declaration>

— [<procedure reference array specification>]
 [<imported data specification>]

<procedure reference array specification>

— <procedure reference array declaration>
 → [<library object attributes>]

<library identifier>

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

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

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

The <imported data specification> portion of the LIBRARY declaration is identical to the <imported data specification> portion of the IMPORTED declaration.

An imported access mode can be given to a data library object by specifying a <library object access mode> as part of the <library object attributes> variable item. For more information on library object attributes, including compile-time restrictions, see the IMPORTED declaration earlier in this section.

LIBRARY Declaration

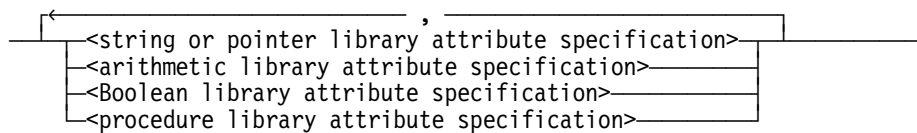
The ALGOL compiler does not allow an access mode to be specified for a procedure, a procedure reference array, an event, an event array, an interlock, or an interlock array.

Section 8, "Library Facility," contains extended examples of libraries and programs that use libraries, as well as information about library attributes, library linkage, and library usage in general.

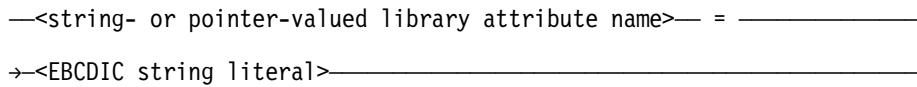
Programs that declare procedure reference arrays as library objects cannot be used for binding.

Library Attribute Specifications

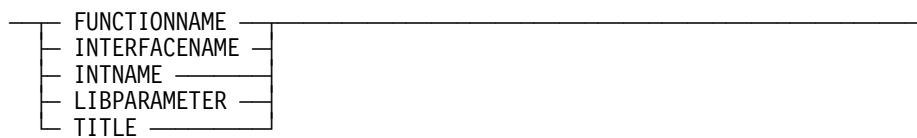
<library attribute specifications>



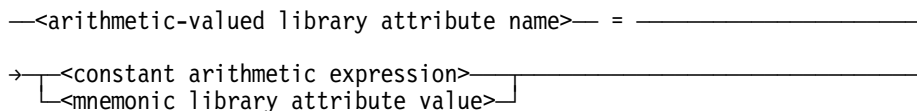
<string or pointer library attribute specification>



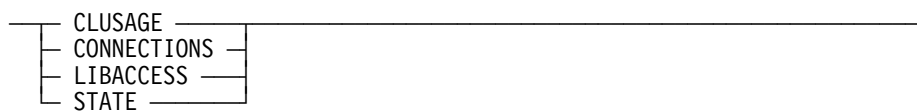
<string- or pointer-valued library attribute name>



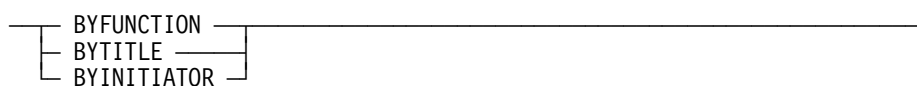
<arithmetic library attribute specification>



<arithmetic-valued library attribute name>



<mnemonic library attribute value>



<Boolean library attribute specification>

—<Boolean-valued library attribute name>—————→
 ↳ = —<constant Boolean expression>—|

<Boolean-valued library attribute name>

— AUTOLINK —————|

<procedure library attribute specification>

—<procedure-valued library attribute name>— = —————→
 ↳<procedure identifier>—————|

<procedure-valued library attribute name>

↳ CHANGE —————|
 ↳ APPROVAL —————|

<library object declaration list>

— [↳<library object declaration> ; —————→
 ↳<library object attributes>—|
 ↳] —————|

<library object declaration>

—<procedure reference array declaration>—————|

<library object attributes>

— (— ACTUALNAME — = —<EBCDIC string literal>—) —————|

The FUNCTIONNAME attribute specifies the system function name used to find the object code file for the library. For example, GENERALSUPPORT is a system library function name.

The INTERFACENAME attribute is used to identify a particular connection library within a program.

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

LIBRARY Declaration

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

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

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

For information on LIBACCESS, see Section 8, "Library Facility."

For LIBRARY declarations, only the CHANGE <procedure-valued library attribute name> can be specified.

A PROCEDURE REFERENCE ARRAY declaration that appears in a LIBRARY declaration can be either the local or the global form of the declaration; that is, either NULL or EXTERNAL can appear at the end of the declaration. However, only the lower bound is required. If an upper bound is given, it is ignored. The procedure reference array is said to be imported from the library.

The handling of a procedure reference array that is declared to be a library object is comparable to the handling of a procedure that is declared to be a library entry point. For more information, see "PROCEDURE Declaration" later in this section.

For more information on library attributes, see the *Task Management Programming Guide*.

Examples of LIBRARY Declarations

The following example declares a reference to the library LIB that is to be referenced by the title OBJECT/LIBRARY.

```
LIBRARY LIB(TITLE="OBJECT/LIBRARY.");
```

The following example declares a reference to the library L, from which the procedure reference array REFID is to be imported. The procedure reference array is exported from the library as PROCREF.

```
LIBRARY L [ PROCEDURE REFERENCE ARRAY REFID [0] (I);  
            VALUE I; INTEGER I;  
            EXTERNAL (ACTUALNAME="PROCREF")];
```

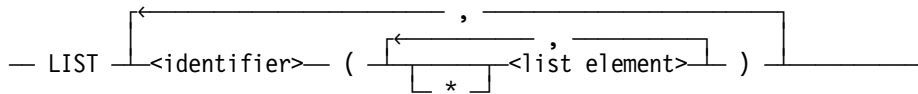
The following example declares a reference to the library LIB, from which the procedure reference arrays PA1 and PA2 are imported.

```
LIBRARY LIB [ REAL PROCEDURE REFERENCE ARRAY PA1[0:10];  
              NULL;  
              PROCEDURE REFERENCE ARRAY PA2[0:3,0:10] (R,B);  
              REAL R;  
              BOOLEAN B;  
              NULL ];
```

LIST Declaration

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

<list declaration>



<list identifier>

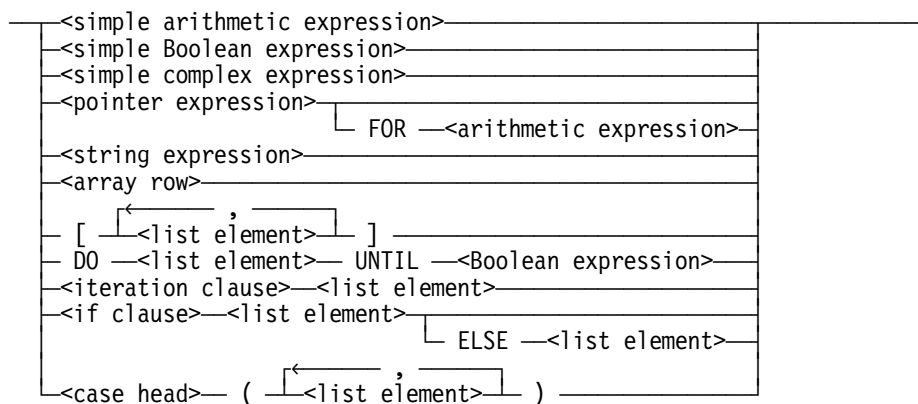
An identifier that is associated with a set of list elements in a LIST declaration.

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

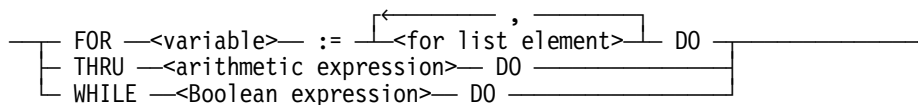
A simple complex expression or complex value appearing in a list is considered to be a pair of real values: the first value is the real part of the complex value, and the second is the imaginary part.

List Elements

<list element>



<iteration clause>



List elements of the following form enable the user to specify the number of characters to be read to or written from the pointer-specified location:

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

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

A string variable is a valid list element for editing phrase letters A, C, and U and for free-field formatting. For more information on free-field formatting, see "READ Statement" in Section 4, "Statements."

A string variable acts in the same manner as *<pointer expression> FOR <arithmetic expression>* when used with the A, C, and U editing phrases. For more information about the A, C, and U editing phrases, refer to "FORMAT Declaration" earlier in this section.

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

Examples of LIST Declarations

The following example declares L1 as a list identifier for the list consisting of X, Y, the array row A[4,*], and B[2], B[3], B[4], and B[5].

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

This list identifier might appear in a WRITE statement such as the following:

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

The following example declares ANSWERS and RESULTS as two list identifiers with associated list elements.

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

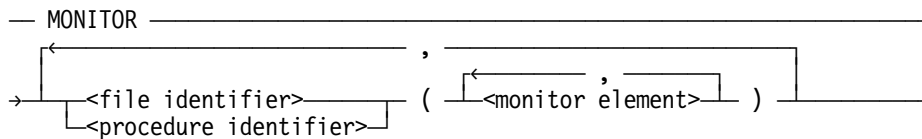
The following example declares LIST3 as a list identifier with an associated list consisting of nested FOR clauses indexing array A. This list identifier can be used in a READ statement to read the specified elements of array A.

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

MONITOR Declaration

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

<monitor declaration>



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

The monitor action does not occur within procedures that are declared before the MONITOR declaration is encountered. Monitoring of a variable in the monitor list does not occur if this identifier is passed as an actual parameter to a call-by-name formal parameter that is modified within the procedure. In addition, the control variable in a FOR statement cannot be monitored. The monitor action does not occur when a value changes as the result of a READ statement or a REPLACE statement.

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

For a debugging feature, refer to "TADS Option" in Section 6, "Compiling Programs."

Monitor Elements

<monitor element>



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

Diagnostic information is given for the specified monitor elements as follows:

- If the monitor element is a simple variable or a subscripted variable, the symbolic name and the previous and new values of the variable are printed (for example, $B = 0 := 13$).
- If the monitor element is a label identifier, the symbolic name of the label is shown (for example, *LABEL L*).
- If the monitor element is an array identifier, the symbolic name of the array, the subscript of the element, and the previous and new values of the changed array element are printed (for example, $A[12] = 0 := 12$).

If the monitor element is to be assigned a value, this assignment must be done by the monitoring procedure. This value also can be assigned to the procedure value to be used, for example, in evaluating the remainder of an expression in which the assignment is embedded. In the example under "Monitor Element as an Array Identifier," the assignment statement $NAME := MON := VAL$; allows the subsequent use of the value assigned to the monitor element.

Monitor Element as a Simple Variable

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

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

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

- The first parameter, *NAME*, is a call-by-name parameter of the same type as the monitor element. The parameter *NAME* is passed a reference to the monitor element, and it is normally used to store the value of the second parameter, *VAL*.
- The second parameter, *VAL*, is also of the same type as the monitor element, but it is a call-by-value parameter and is passed the new value to be assigned to the monitor element.
- The third parameter, *SPELL*, must be a call-by-value real variable that is passed the name of the monitor element as a string of characters. Only the first six characters of the symbolic name are passed to this formal parameter. If the symbolic name is less than six characters long, it is left-justified, and trailing blanks are added, up to six characters.

Monitor Element as a Label Identifier

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

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

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

Monitor Element as an Array Identifier

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

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

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

The value being assigned to the array element also can be assigned to the procedure value to be used, for example, in evaluating the remainder of an expression that contains the array element.

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

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

The following statements are equivalent to each other, where A is monitored by MON. A is a two-dimensional array declared in the same program where the monitoring procedure MON is declared. The first assignment statement assigns 4 to A[I,J], as does the second statement because inside the procedure MON the fourth parameter (VAL) is assigned to the 3rd parameter (NAME).

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

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

Examples of MONITOR Declarations

The following example declares the simple variable A to be a monitor element. When A is used, monitoring information on A is written to file FYLE.

```
MONITOR FYLE (A)
```

In the following program, simple variable I, array MON1, subscripted variable MON2[1], and label FINISH are monitored.

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

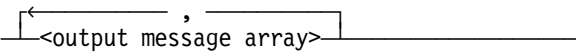
When the program is executed, the following output is written to the terminal:

```
0148 @ 003:000E:4 (00001000) I      =0:=27      (4"000000000001B")
0148 @ 003:0013:4 (00001100) MON1  [0]=0:=27    (4"000000000001B")
0148 @ 003:0020:4 (00001300) MON2  [1]=0:=54    (4"0000000000036")
0148 @ 003:0024:4 (00001500) LABEL FINISH
```

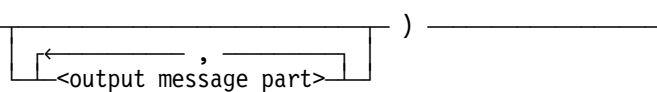
OUTPUTMESSAGE ARRAY Declaration

An OUTPUTMESSAGE ARRAY declaration declares output message arrays. An output message array contains output messages to be used by the MultiLingual System (MLS). For a description of how to use these arrays, refer to "MESSAGESEARCHER Statement" in Section 4, "Statements."

<output message array declaration>

— OUTPUTMESSAGE — ARRAY —  —

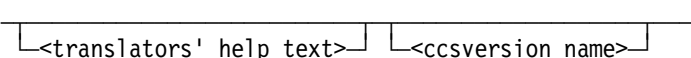
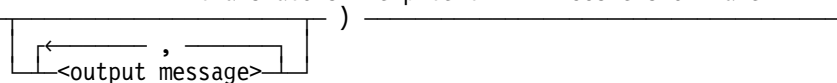
<output message array>

—<identifier>— () —

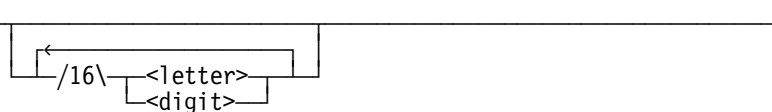
<output message array identifier>

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

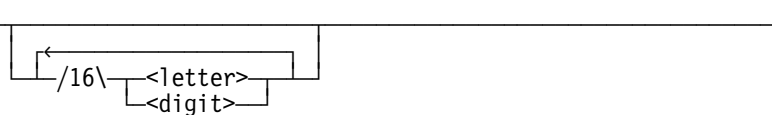
<output message part>

—<language name>—  →
 → () —

<language name>

—<letter>—  —

<ccsversion name>

—<letter>—  —

<translator's help text>

— < —<EBCDIC string constant>— > —

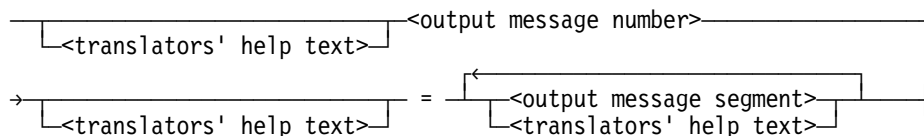
The OUTPUTMESSAGE ARRAY declaration is part of the ALGOL interface to the MultiLingual System (MLS), which enables the user to access system messages in various natural languages, that is languages used by humans rather than machines.

Each output message array identifier must be unique throughout the entire program. This requirement is an exception to the description of the scope of identifiers given in Section 1, "Program Structure."

The ccsversion name identifies the ccsversion to be associated with the messages contained in the output message array. This information is used during translation of the declared output messages by the MLSTRANSULATE statement to provide case-insensitivity. If unspecified, the associated ccsversion becomes the internationalized system default collating sequence.

Output Message

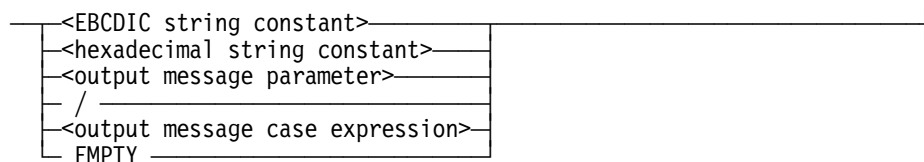
<output message>



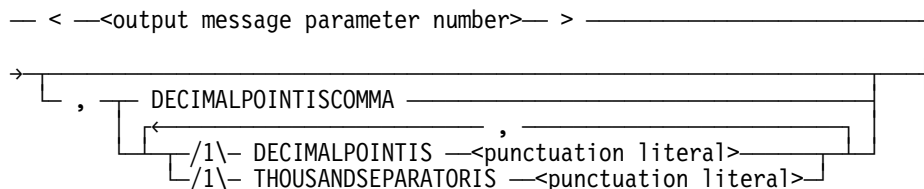
<output message number>



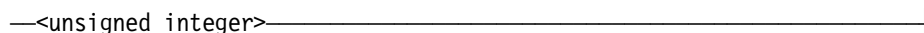
<output message segment>



<output message parameter>



<output message parameter number>



OUTPUTMESSAGE ARRAY Declaration

<output message case expression>

— CASE — < —<output message parameter number>— > — OF — BEGIN —>
→ {<output message case part> , } END —————|

<output message case part>

{<output message parameter value> : —————>
 /1\— ELSE —————>
→ {<output message segment> , }
 <translators' help text> —————|

<output message parameter value>

{<EBCDIC string constant> —————>
 <hexadecimal string constant> —————>
 EMPTY —————|

An output message number must be less than 8 digits long. For each output message part, the output message number must uniquely identify an output message. For example, a number is assigned to one and only one output message segment, and each output message segment has only one number assigned to it.

An output message parameter number represents a parameter to be substituted into the message when the MESSAGESEARCHER statement is executed. The number identifies which parameter is to be substituted. The output message parameters are numbered consecutively from 1 through n, where n is the number of parameters in the output message. The maximum number of parameters in an output message is 255.

DECIMALPOINTISCOMMA indicates that any decimal point (.) appearing in the preceding output message parameter number is changed to a decimal comma (,). In addition, all commas are changed to decimal points. DECIMALPOINTIS <punctuation literal> causes any decimal points appearing in the parameter value to be changed to the specified character. THOUSANDSEPARATORIS <punctuation literal> causes any commas appearing in the parameter value to be changed to the specified character.

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

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

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

Defines are expanded within an OUTPUTMESSAGE ARRAY declaration.

Translators' Help Text

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

Only one set of beginning and ending angle brackets are needed for each segment of translators' help text. However, beginning and ending quotation marks are required for each line if the help text continues over more than one line. The following is a multilined translators' help text:

```
<"This message should not be translated into another language."  
"However, it can be uppercased.">
```

Examples of OUTPUTMESSAGE ARRAY Declarations

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

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

OUTPUTMESSAGE ARRAY Declaration

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

```
OUTPUTMESSAGE ARRAY SUMMARY (
  ENGLISH (
    100 =
      "THIS PROGRAM IS TO BE EXECUTED WITH "
      CASE <1> OF
        BEGIN
          "1": "MAX PROCESSING TIME " <2> " SEC.",
          "2": "MAX I/O TIME " <3> " SEC.",
          "3": "MAX PROCESSING TIME " <2> " SEC., MAX "
              "I/O TIME " <3> " SEC."
        END
      ),
  FRANCAIS (
    100 =
      "CE PROGRAMME DOIT S'EXECUTER EN MOINS DE "
      CASE <1> OF
        BEGIN
          "1": <2, DECIMALPOINTISCOMMA>
              " SEC. DE CALCUL.",
          "2": <3, DECIMALPOINTISCOMMA> " SEC. D'E/S.",
          "3": <2, DECIMALPOINTISCOMMA>
              " SEC. DE CALCUL OU "
              <3, DECIMALPOINTISCOMMA> " SEC. D'E/S."
        END
      ));
```

PENDING PROCEDURE Declaration

<pending procedure declaration>

┌──────────────────┐ PROCEDURE —<procedure heading>— ; ───────────>
└─<procedure type>┘
→ PENDING ───┘

The PENDING PROCEDURE declaration enables procedures declared within a structure or connection block to have their bodies specified later outside the original STRUCTURE or CONNECTION BLOCK TYPE declaration. It is similar to the FORWARD reference of a PROCEDURE declaration.

A syntax error results if the PENDING procedure is not fully resolved in the block containing the STRUCTURE or CONNECTION BLOCK TYPE declaration with the PENDING PROCEDURE ACTUAL declaration.

PENDING PROCEDURE ACTUAL Declaration

<pending procedure actual declaration>

┌──────────────────────────┐ PROCEDURE ─<structure block type identifier>→
└<procedure type>┘
→ . ─<procedure heading>─ ; ─<unlabeled statement>──────────────────┘

The PENDING PROCEDURE ACTUAL declaration provides the body of the procedure that was previously declared as PENDING in the STRUCTURE or CONNECTION BLOCK TYPE declaration. This declaration must appear at the same level as, and outside the STRUCTURE or CONNECTION BLOCK TYPE declaration in which, the pending procedure is contained.

All variables declared inside the STRUCTURE or CONNECTION BLOCK TYPE declaration become the outer scope for the procedure. Variables declared outside the STRUCTURE or CONNECTION BLOCK TYPE declaration become the next most outer scope.

The following is an example of a PENDING PROCEDURE ACTUAL declaration:

```
REAL X;  
TYPE STRUCTURE BLOCK SB;  
  BEGIN  
    PROCEDURE P; PENDING;  
    REAL X;  
    REAL R;  
    REAL Z;  
  END;  
  
REAL R;  
REAL Z;  
PROCEDURE SB.P;  
  BEGIN  
    REAL Z;  
    X := 100;    %Updates X contained in SB  
    R := 100;    %Updates R contained in SB  
    Z := 100;    %Updates Z contained in procedure P  
  END;
```

PICTURE Declaration

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

<picture declaration>

— PICTURE <identifier> (<picture>)

<picture identifier>

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

<picture>

<picture symbol>

<picture symbol>

<string literal>
<introduction>
<picture skip> <repeat part value>
<control character>
<single picture character>
<picture character> <repeat part value>

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

- Unconditional character moves
- Moves of characters with leading 0 (zero) editing
- Moves of characters with leading 0 (zero) editing and floating character insertion
- Moves of characters with conditional character insertion
- Moves of characters with unconditional character insertion
- Moves of only the numeric parts of characters
- Forward and reverse skips of source characters
- Forward skips of destination characters
- Insertion of a zone field on the previous character

PICTURE Declaration

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

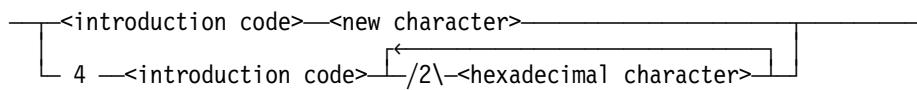
One value array, also called an edit table, is generated for each PICTURE declaration; therefore, for run-time efficiency, all pictures should be collected under a single PICTURE declaration.

String Literals

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

Introduction

<introduction>



<introduction code>



<new character>



<special new character>

Any of the following special characters:

. , [] () + - /
> < = % & *; @:
\$ "

Introduction Codes

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

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

Characters Used by Picture Symbols

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

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

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

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

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

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

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

Flip-Flops Used by Picture Symbols

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

The value of FLTF affects the function performed by the picture symbols D, E, F, J, R, and Z. FLTF is set to 0 (zero) at the beginning of every picture. The picture symbols E, F, and Z can change the value of FLTF to 1, and the picture symbols J, R, and D unconditionally assign the number 0 to FLTF.

The value of EXTF affects the function performed by the picture symbols E, F, J, Q, R, and S. EXTF is not assigned a value by the REPLACE statement that is using the picture; EXTF remains in the state in which it was left after the most recent operation that affected it. For example, a REPLACE statement of the following form sets EXTF to reflect the sign of the first arithmetic expression: the number 0 if the arithmetic expression is positive, and the number 1 if it is negative.

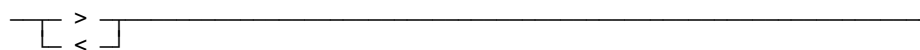
```
REPLACE <destination> BY <arithmetic expression>  
FOR <arithmetic expression> DIGITS
```


Character Fields

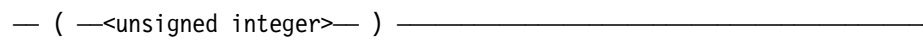
Pictures can act on both EBCDIC and hexadecimal characters. In the descriptions of the picture symbols, the term *numeric field* is used to mean either an entire hexadecimal character or the rightmost four bits of an EBCDIC character. The term *zone field* is used to mean the leftmost four bits of an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

Picture Skip Characters

<picture skip characters>



<repeat part value>

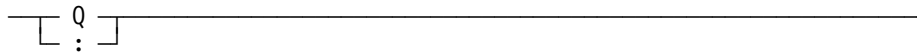


The picture skip characters are described in the following table. If a repeat part value is given with the picture symbol, then this unsigned integer indicates the number of characters that are to be skipped in the source. If no repeat part value is given, one character is skipped in the source.

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

Control Characters

<control character>



The control characters are described in the following table.

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

Single Picture Characters

<single picture character>



The single picture characters are described in the following table.

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

Picture Characters

<picture character>



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

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

Character	Action
------------------	---------------

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

Examples of PICTURE Declarations

The following picture transfers five characters from the source to the destination:

```
PICTURE NUM (ZZZZ9)
```

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

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

The following picture transfers nine characters from the source to the destination and inserts one character into the destination, yielding 10 characters in the destination:

```
PICTURE USECS (ZZZI999999)
```

PICTURE Declaration

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

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

The following picture transfers six characters from the source:

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

The introduction code N causes the insert character to be the colon (:). The string literal " " causes the blank character to be inserted into the destination. The first and second source characters are transferred to the destination without leading zero replacement, the insert character is inserted into the destination, the third and fourth source characters are transferred to the destination, the insert character is inserted, and the fifth and sixth source characters are transferred to the destination. The destination receives a total of nine characters.

The following table gives some sample results of this picture.

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

The following picture transfers 11 characters from the source to the destination, formatting the information into a table:

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

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

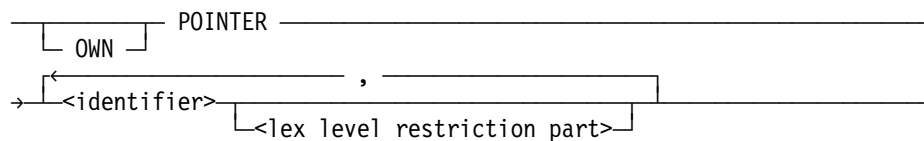
The following table gives some sample results of this picture. In the table, it is assumed that the destination area was filled with blanks before the picture was used, and that EXTF was properly set up to reflect the sign of the change value.

Source	Destination
8"00035000420020"	8"1983 = \$35 1984 = \$42 CHANGE = +20%"
8"00110003680235"	8"1983 = \$110 1984 = \$368 CHANGE = +235%"
8"02246021060006"	8"1983 = \$2246 1984 = \$2106 CHANGE = -6%"
8"00089000350061"	8"1983 = \$89 1984 = \$35 CHANGE = -61%"

POINTER Declaration

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

<pointer declaration>



<pointer identifier>

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

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

The following declaration, for example, declares PTS, PTD, SOURCE, and DEST to be pointers:

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

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

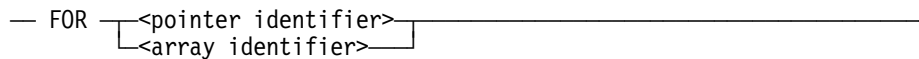
OWN Pointers

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

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

Lex Level Restriction Part

<lex level restriction part>

— FOR — 

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

An explicit up-level pointer assignment such as the following results in a syntax error, because the locally declared array LOCALARRAY might be deallocated, leaving the global pointer GLOBALPOINTER pointing at an invalid memory location:

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

A potential up-level pointer assignment such as the following also results in a syntax error, because the local pointer LOCALPOINTER can point to a locally declared array:

```
GLOBALPOINTER := LOCALPOINTER
```

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

The lex level restriction part causes assignments to the declared pointer to be restricted so that the pointer can be used to assign values to pointers declared at lower lex levels. The lex level restriction part specifies that, for up-level pointer assignment checking, the compiler is to treat the pointer being declared as if it were declared at the same lex level as the pointer or array whose identifier follows the *FOR*. For example, the following declaration declares a pointer LOCALPOINTER that can point only to arrays declared at lex levels equal to or less than the lex level at which GLOBALPOINTER is declared:

```
POINTER LOCALPOINTER FOR GLOBALPOINTER
```

Because assignments to LOCALPOINTER are restricted by the lex level restriction part in the preceding declaration, an assignment such as the following one cannot result in an up-level pointer assignment, and therefore is allowed by the compiler:

```
GLOBALPOINTER := LOCALPOINTER
```

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

Examples of POINTER Declarations

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

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

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

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

```

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

```

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

```

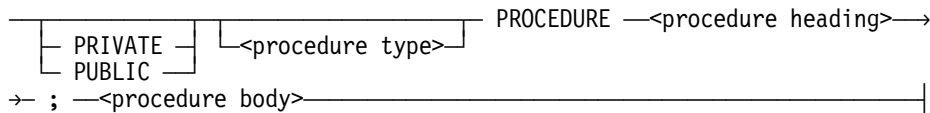
[POINTER PTRA,PTRB;]
PROCEDURE P;
  BEGIN
    ARRAY A[0:9];
    PTRA := PTRA + 2;      % OK
    PTRA := POINTER(A);   % SYNTAX ERROR -- THIS IS AN
                          % UP-LEVEL POINTER ASSIGNMENT.
    PTRA := PTRB;         % SYNTAX ERROR -- THE LEX LEVELS
                          % OF PTRA AND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

```

PROCEDURE Declaration

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

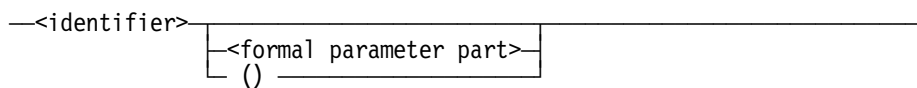
<procedure declaration>



<procedure type>



<procedure heading>



The PRIVATE and PUBLIC specifiers can only be used for procedures that are declared within a structure block or a connection block. The PRIVATE specifier limits visibility of the procedure to the scope of the structure block or the connection block. A PRIVATE procedure cannot be accessed using a structure or connection block qualifier. The PUBLIC specifier allows the procedure to be accessed using a structure or connection block qualifier. If neither PRIVATE nor PUBLIC is specified, the default value is PUBLIC and access to the procedure using a structure or connection block qualifier is allowed.

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

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

If a program is a procedure, parameters can be passed to it. If the procedure is initiated through CANDE (which passes only one parameter, a quoted string), then the formal parameter must be declared as a real array with an asterisk lower bound. If the procedure is initiated through Work Flow Language (WFL), a formal parameter for a string actual parameter must be declared as a real array with an asterisk lower bound. Both CANDE and WFL pass strings as arrays. For more information, refer to the

EXECUTE command in the *CANDE Operations Reference Manual* and the RUN statement in the *Work Flow Language (WFL) Programming Reference Manual*. When the program is initiated, the array is allocated the minimum number of words needed to contain the string plus at least one null character (48"00"), which is appended to the end of the string.

Identifiers

<procedure identifier>

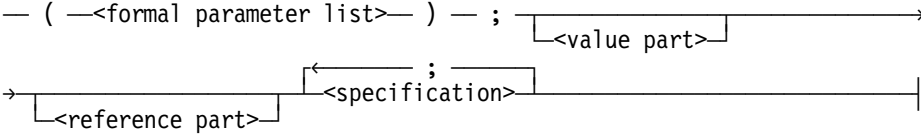
An identifier that is associated with a procedure in a PROCEDURE declaration.

<string procedure identifier>

An identifier that is associated with a procedure that is declared a string procedure in a PROCEDURE declaration.

Formal Parameter Part

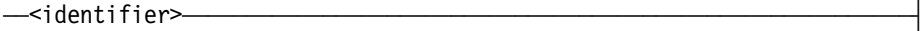
<formal parameter part>



<formal parameter list>



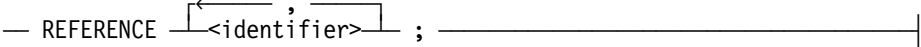
<formal parameter>



<value part>



<reference part>



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

PROCEDURE Declaration

If the identifier of a formal parameter is declared again as a local variable, the identifier gets local significance. This makes the actual parameter that corresponds to it inaccessible throughout this inner scope. Any reference to the parameter identifier actually is a reference to the local variable.

For maximum efficiency, as many formal parameters as possible should be call-by-value, and each specified lower bound should have a value of 0 (zero).

To ensure that a parameter is passed call-by-reference, the parameter name must appear in the <reference part> of the parameter description. Constants and arithmetic expressions cannot be passed to parameters whose name appears in the reference part.

If a formal parameter is call-by-reference and the actual parameter being passed to it is itself a parameter and is call-by-name, then evaluation of the call-by-name parameter is done in order to generate the call-by-reference parameter. This ensures that any expressions evaluated due to an accidental entry generated for the call-by-name parameter are evaluated only once for the call-by-reference parameter.

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

For more information on <parameter delimiter>, see Section 2, "Language Components."

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

Formal parameters not listed in the value part are call-by-reference, except for Boolean, complex, double, integer, and real parameters, which are call-by-name, unless they are listed in the reference part. Wherever a call-by-name formal parameter appears in the procedure body, the formal parameter is, in effect, replaced by the actual parameter itself and not by the value of the actual parameter. A call-by-name formal parameter is essentially global to the procedure body, because any change made to its value within the procedure body also changes the value of the corresponding actual parameter outside the procedure body. If the formal parameter is a complex call-by-name parameter and the actual parameter is not of type COMPLEX, an assignment within the procedure body to the formal parameter causes the program to discontinue with a fault.

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

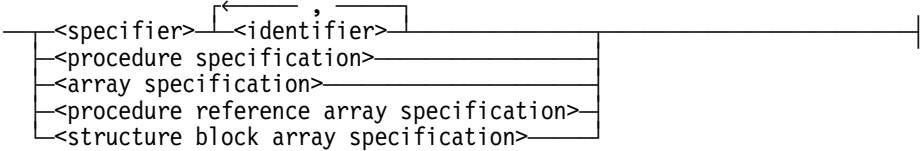
The default mode of passing a string is call-by-reference instead of call-by-name. Any string expression can be passed to a call-by-reference string formal parameter. When a

string variable or a subscripted string variable is passed as an actual parameter to a call-by-reference string formal parameter, a reference to the actual string is passed. If the value of the formal parameter is changed within the procedure body, the actual string is also changed.

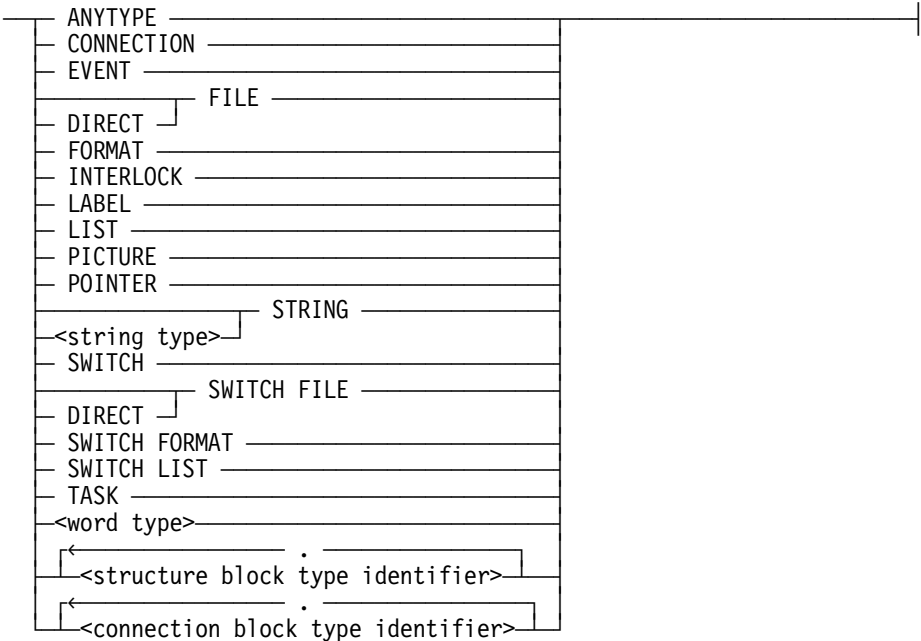
If any other form of string expression is passed as an actual parameter to a call-by-reference string formal parameter, the string expression is evaluated once at the time the expression is passed, and a reference to the value of the expression is passed to the called procedure. This value can be altered by the called procedure. However, any change in the value of the formal parameter within the procedure body has no effect outside the procedure body. A string expression cannot be passed as an actual parameter to a call-by-name parameter of a procedure in a PROCESS or CALL statement.

Specification

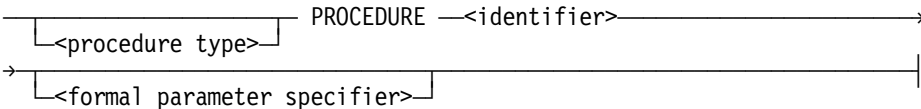
<specification>



<specifier>



<procedure specification>



PROCEDURE Declaration

<formal parameter specifier>

() ; — FORMAL —

└─<formal parameter part>┘

<array specification>

└─<array type>┘ ARRAY └─<identifier>┘

→ [└─<lower bound list>┘]

<array type>

└─<array class>┘

DIRECT	└─<array class>┘
EVENT	
INTERLOCK	
└─<string type>┘	STRING
TASK	

<lower bound list>

└─<specified lower bound>┘

<specified lower bound>

└─<integer>┘

└─ * ┘

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

If the specified lower bound is an integer, then the corresponding dimension of the formal array equals that integer. An asterisk (*) used as a specified lower bound indicates that the corresponding dimension of the formal array has a lower bound that is passed to the procedure with the actual array.

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

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

If a formal parameter is a structure block variable, the actual parameter passed to that formal structure block variable must be a structure block array element, a structure block variable, or a structure block reference variable of the same structure block type as the formal parameter.

When a structure block variable is a formal parameter of an imported or exported library procedure, there are restrictions on the items that can be declared within the parent structure block type. The PRIVATE specifier cannot be used for the items declared. Only items allowed in the <export object specification> list of the EXPORT declaration can be declared within the structure block type: plus two additional items. The additional items are an embedded structure block variable and a structure block reference variable. The structure block type of the embedded structure block variable or structure block reference variable has the same restrictions as the parent structure block type.

The use of ANYTYPE as the formal parameter specifier allows for a flexible interface between an exported unsafe procedure in a NEWP library and the invocation of that entry point in an ALGOL program. The actual parameter passed is matched at run time to a WORD variable in the NEWP library procedure. Therefore, the actual parameter can be of any type that ALGOL allows to be passed to a library. The compiler does not perform parameter type checking. If ANYTYPE is specified as a formal parameter, the procedure must be an imported library entry point.

If CONNECTION or <connection block type identifier> is used as the specifier for the formal parameter, an individual connection library must be passed as the actual parameter. The actual parameter must be an indexed library connection variable unless the connection library has been declared with the SINGLE modifier. If the connection library has been declared with the SINGLE modifier, the actual parameter must be an unindexed connection library variable. In addition, the connection parameter must be a call-by-reference parameter. An exported or imported library procedure cannot have a <connection block type identifier> as the specifier for a formal parameter.

When CONNECTION is used as a specifier, the procedure cannot see the elements of the connection library associated with the connection parameter. The procedure can pass the connection parameter to MCP procedures and can access the connection attributes. The procedure cannot use the parameter either to call procedures within the connection library or to access nonconnection attributes.

Nonlibrary procedures—procedures that are neither imported nor exported from a library—can specify either CONNECTION or a <connection block type identifier> for connection parameters. Using CONNECTION restricts the use of the parameter. Library procedures can specify CONNECTION only for a connection parameter.

Procedure Reference Array Specification

<procedure reference array specification>
 _____ PROCEDURE — REFERENCE — ARRAY —<identifier>→
 |
 |<procedure type>|
 → [—<lower bound list>—] —<formal parameter specifier>—————|

A procedure reference array specification must have a formal parameter specifier.

Structure Block Array Specification

<structure block array specification>

— [<structure block type identifier>] ARRAY [<identifier>] [→
→ <lower bound list>]

Structure Block Array Parameters

If a formal parameter is a structure block array, the actual parameter passed to that formal structure block array must be a structure block array designator of the same structure block type as the formal parameter. A structure block array cannot be a formal parameter for an exported or imported library procedure.

Procedure Body

<procedure body>

— [<block>
— [<unlabeled statement>
— EXTERNAL
— [<dynamic procedure specification>
— [<library entry point specification>
— [<isolated procedure specification>]]]

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

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

Dynamic Procedure Specification

<dynamic procedure specification>

— BY CALLING — <selection procedure identifier>

<selection procedure identifier>

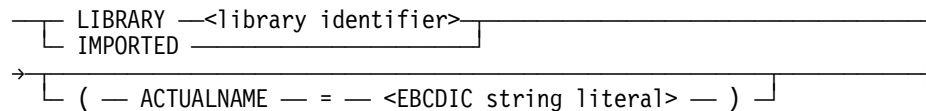
— <procedure identifier>

A dynamic procedure specification is used in a library program to declare a procedure that is to be exported dynamically. Such a procedure is also called a by-calling procedure. For more information on by-calling procedures, refer to Section 8, "Library Facility." The by-calling procedure cannot be declared FORWARD or PENDING and cannot be a separately compiled procedure. Also, the by-calling procedure cannot be referenced directly in the library program that declares it.

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

Library Entry Point Specification

<library entry point specification>



A library entry point specification declares a procedure to be an entry point in the library known to this program by the library identifier. The procedure cannot be declared FORWARD, EXTERNAL, or PENDING. The IMPORTED clause is allowed only inside a connection block type.

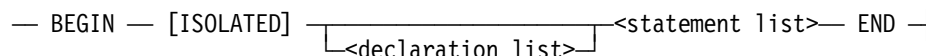
If a program declares a library and entry points in that library, the object code file for the program contains a structure called a library template, which describes the library and its declared entry points. Each declared library has one template. The template description of an entry point includes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

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

The name given for an entry point in a library template is the procedure identifier in the entry point declaration, unless an ACTUALNAME clause appears, in which case the name is given by the EBCDIC string literal. The EBCDIC string literal in the ACTUALNAME clause can contain only valid <printable character>s, but must not contain any leading, trailing, or embedded blanks or periods. The entry point name indicated by this string literal is case sensitive.

ISOLATED Procedure Specification

<isolated procedure specification>



An isolated procedure is primarily used in the context of shared libraries that have global data protected by locking mechanisms. However, an isolated procedure can be used in any situation where a procedure from one stack runs on another stack. In the following

PROCEDURE Declaration

text, the stack in which the isolated procedure is declared is called the parent stack and the other stack is called the client stack. The ISOLATED procedure attribute is used to protect the parent stack of an isolated procedure from being stopped or discontinued when the client stack on which the isolated procedure is running is stopped or discontinued.

If the parent stack of the isolated procedure is discontinued, then the isolated procedure is discontinued.

If the client stack is stopped by an ST command, the isolated procedure is allowed to run for an additional 5 seconds of CPU time. After 5 seconds of CPU time, the isolated procedure is also stopped.

If the client stack is discontinued, the isolated procedure is allowed to continue to run. After 5 seconds of CPU time, a message is displayed to allow the operator to take any necessary action.

Using the ISOLATED procedure attribute, a library or other IPC program can assume that an isolated procedure will not be interrupted by a DS command while running on a client stack. Also, an isolated procedure will not be delayed by an ST command, provided the procedure does not use more than 5 seconds of CPU time before exiting.

When an isolated procedure fails to exit cleanly, the parent stack of the isolated procedure is discontinued. Failure to exit cleanly includes: a fatal error in the isolated procedure, a fatal error in a procedure invoked above the isolated procedure, or a GO TO statement that cuts back the stack environment of the isolated procedure.

When an isolated procedure is running on a client stack, asynchronous actions such as software interrupts, signals, approval procedures, and change procedures for the client stack are delayed until the isolated procedure exits. Software interrupts declared within an isolated procedure or within procedures invoked above the isolated procedure will never run.

If an isolated procedure uses the DSABLE option of a WAIT or LOCK Interlock function, code in the isolated procedure should be written to exit cleanly if the value returned by the WAIT or LOCK Interlock function indicates that the client stack has been discontinued.

When an isolated procedure is running on a client stack, the isolated semantics apply to the client stack until the isolated procedure exits or the parent stack of the isolated procedure is discontinued. The isolated semantics apply to procedures invoked above the isolated procedure. When multiple isolated procedures are running in a client stack, the isolated semantics apply down to the first isolated procedure of the client stack in which the parent stack has not been discontinued.

An isolated procedure can cause some system operations to be delayed such as a DS command, or an ST command. Therefore, the code for an isolated procedure should be written to run quickly and efficiently, not consuming excessive system resources.

Allowed Formal and Actual Parameters

All parameters can be declared to be call-by-name or, in the case of strings, call-by-reference. A parameter of type ANYTYPE is declared as call-by-reference. The following types of parameters also can be declared to be call-by-value:

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

Parameter Matching

Array Parameters

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

The types of actual arrays that can be passed to formal arrays are listed in Table 3–1. The <null value> is allowed as an actual parameter for all formal parameters that are arrays.

Table 3–1. Array Parameters

Formal Parameters	Allowed Actual Parameters
ASCII array	ASCII array ASCII value array
ASCII string array	ASCII string array
Boolean array	Boolean array Direct Boolean array Boolean value array
Complex array	Complex array Complex value array
Direct ASCII array	Direct ASCII array
Direct Boolean array	Direct Boolean array

Table 3-1. Array Parameters

Formal Parameters	Allowed Actual Parameters
Direct double array	Direct double array
Direct EBCDIC array	Direct EBCDIC array
Direct hexadecimal array	Direct hexadecimal array
Direct integer array	Direct integer array
Direct real array	Direct real array
Double array	Double array Direct double array Double value array
EBCDIC array	EBCDIC array EBCDIC value array
EBCDIC string array	EBCDIC string array
Event array	Event array
Hexadecimal array	Hexadecimal array Hexadecimal value array
Hexadecimal string array	Hexadecimal string array
Interlock array	Interlock array
Integer array Real array	Direct integer array Direct real array Integer array Integer value array Real array Real value array
Task array	Task array

Procedure Reference Array Parameters

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

The following must also be true:

- The actual procedure reference array designator must have the same number of parameters as the formal procedure reference array.
- Each parameter of the actual procedure reference array designator must have the same type as the corresponding parameter in the formal procedure reference array.
- Each parameter of the actual procedure reference array designator must be passed in the same manner (call-by-name or call-by-value) as the corresponding parameter in the formal procedure reference array.

The types of procedure reference array designators that can be passed to formal procedure reference arrays are listed in Table 3–2.

Table 3–2. Procedure Reference Array Parameters

Formal Parameter	Allowed Actual Parameters
ASCII string procedure reference array	ASCII string procedure reference array designator
Boolean procedure reference array	Boolean procedure reference array designator
Complex procedure reference array	Complex procedure reference array designator
Double procedure reference array	Double procedure reference array designator
EBCDIC procedure reference array	EBCDIC procedure reference array designator
Hexadecimal procedure reference array	Hexadecimal procedure reference array designator
Integer procedure reference array	Integer procedure reference array designator
Real procedure reference array	Real procedure reference array designator
Untyped procedure reference array	Untyped procedure reference array designator

Procedure Parameters

If a formal parameter is a procedure and the actual parameter is not a <null value>, the actual parameter passed to the formal procedure must be the identifier of a procedure or procedure reference for which the following is true:

- The actual procedure has the same number of parameters as the formal procedure.
- Each parameter of the actual procedure must have the same type as the corresponding parameter in the formal procedure.
- Each parameter of the actual procedure must be passed in the same manner (call-by-name or call-by-value) as the corresponding parameter in the formal procedure.

The types of procedures that can be passed to formal procedures are listed in Table 3–3. The <null value> and procedure reference identifiers are allowed as actual parameters for all formal procedures.

Table 3–3. Procedure Parameters

Formal Parameter	Allowed Actual Parameters
ASCII string procedure	ASCII string procedure ASCII string procedure reference array element
Boolean procedure	Boolean procedure Boolean procedure reference array element
Complex procedure	Complex procedure Complex procedure reference array element
Double procedure	Double procedure Double procedure reference array element
EBCDIC string procedure	EBCDIC string procedure EBCDIC string procedure reference array element
Hexadecimal string procedure	Hexadecimal string procedure Hexadecimal string procedure reference array element
Integer procedure	Integer procedure Integer procedure reference array element
Real procedure	Real procedure Real procedure reference array element
Untyped procedure	Untyped procedure Untyped procedure reference array element

Simple Variable Parameters

The types of actual parameters that can be passed to formal parameters that are simple variables are listed in Table 3–4.

Table 3–4. Simple Variable Parameters

Formal Parameter	Allowed Actual Parameters
Boolean simple variable (call-by-name or call-by-value)	Boolean identifier Boolean procedure identifier Boolean expression
Complex simple variable (call-by-name or call-by-value)	Complex identifier Double identifier Integer identifier Real identifier Complex procedure identifier Double procedure identifier Integer procedure identifier Real procedure identifier Arithmetic expression (single or double-precision) Complex expression
Double simple variable (call-by-name)	Double identifier Double procedure identifier Arithmetic expression (double-precision only)
Double simple variable (call-by-value)	Double identifier Integer identifier Real identifier Double procedure identifier Integer procedure identifier Real procedure identifier Arithmetic expression (single or double-precision)
Integer simple variable Real simple variable (call-by-name)	Integer identifier Real identifier Integer procedure identifier Real procedure identifier Arithmetic expression (single-precision only)
Integer simple variable Real simple variable (call-by-value)	Double identifier Integer identifier Real identifier Double procedure identifier Integer procedure identifier Real procedure identifier Arithmetic expression (single or double-precision)

String Parameters

The types of actual parameters that can be passed to formal parameters that are strings are listed in Table 3–5.

Table 3–5. String Parameters

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

File Parameters

The types of actual parameters that can be passed to formal parameters that are files are listed in Table 3–6.

Table 3–6. File Parameters

Formal Parameter	Allowed Actual Parameters
Direct file	Direct file identifier Subscripted direct switch file identifier
Direct switch file	Direct switch file identifier
File	Nondirect file identifier Subscripted switch file identifier
Switch file	Switch file identifier

Other Types of Parameters

The types of actual parameters that can be passed to formal parameters that are not arrays, procedures, simple variables, strings, or files are listed in Table 3–7.

Table 3–7. Other Types of Parameters

Formal Parameter	Allowed Actual Parameters
Anytype	Any allowable library parameter type; must match the corresponding type in the library procedure
Connection block	Connection library instance designator Connection block reference variable null value
Event	Event identifier An element of an event array File identifier.event-valued file attribute name Subscripted switch file identifier.event-valued file attribute name
Format	Format identifier Subscripted switch format identifier
Interlock	Interlock identifier An element of an interlock array
Label (call-by-name) (call-by-value)	Label identifier Subscripted switch identifier Designational expression
List	List identifier Subscripted switch list identifier
Picture	Picture identifier
Pointer (call-by-name)	Pointer identifier
Pointer (call-by-value)	Pointer identifier Pointer expression null value
Structure block	Structure block array element Structure block reference variable Structure block variable null value
Structure block array	Structure block array designator
Switch label	Switch label identifier
Switch format	Switch format identifier
Switch list	Switch list identifier
Task	Any task designator

Examples of PROCEDURE Declarations

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

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

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

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

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

In the following example, procedure RESULT is a typed procedure that returns a real value. The value to be returned is assigned to the procedure identifier by the following assignment:

```
RESULT := X + PARAM;
```

RESULT has two parameters, a call-by-name real variable and a file.

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

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

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

The following example declares MATCH to be a typed procedure that returns a Boolean value. MATCH has three parameters that are all call-by-value integer variables.

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

The following example is a FORWARD PROCEDURE declaration for the procedure FURTHERON. For more information, refer to “FORWARD REFERENCE Declaration” earlier in this section.

```
PROCEDURE FURTHERON;
FORWARD
```

The following example declares MUCHO to be a double-precision procedure with three parameters. DBL1 is a call-by-name double-precision variable, DBL2 is a call-by-value double-precision variable, and BOOL is a call-by-value Boolean variable. The body of MUCHO is a block.

```
DOUBLE PROCEDURE MUCHO(DBL1,DBL2,BOOL);
VALUE DBL2,BOOL;
DOUBLE DBL1,DBL2;
BOOLEAN BOOL;
BEGIN
REAL LOCALX,LOCALY;
.
.
.
MUCHO := DOUBLE(LOCALX,LOCALY);
END OF MUCHO
```

The following example declares GETDATA to be a by-calling procedure. The selection procedure is SELECTDATASOURCE. GETDATA has one parameter, a one-dimensional real array, A, with an asterisk lower bound, meaning that the lower bound is to be passed as a parameter.

```
PROCEDURE GETDATA(A); % BY-CALLING PROCEDURE
ARRAY A[*];
BY CALLING SELECTDATASOURCE
```

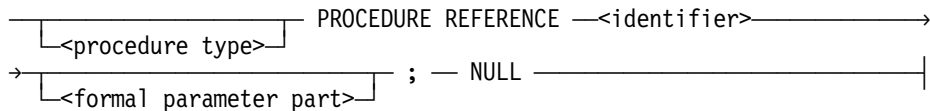
The following example declares NUMRECORDS to be an entry point in the library DATAHANDLER. The entry point is exported from DATAHANDLER with the name COUNTRECS, but will be called NUMRECORDS in this program.

```
INTEGER PROCEDURE NUMRECORDS(TYPE); % LIBRARY ENTRY POINT
VALUE TYPE;
INTEGER TYPE;
LIBRARY DATAHANDLER (ACTUALNAME="COUNTRECS")
```

PROCEDURE REFERENCE Declaration

The PROCEDURE REFERENCE declaration declares a procedure reference, which is a structure that allows a procedure to be invoked by referencing the procedure reference.

<procedure reference declaration>



Identifiers

<procedure reference identifier>

An identifier associated with a procedure reference in a PROCEDURE REFERENCE declaration.

The PROCEDURE REFERENCE declaration does not initialize the procedure reference during declaration. The procedure reference must be assigned to a value before it can be invoked. A run time error occurs if the procedure reference identifier is invoked before it is initialized.

A PROCEDURE REFERENCE declaration cannot appear in the formal parameter part of a PROCEDURE declaration, PROCEDURE REFERENCE ARRAY declaration, or of another PROCEDURE REFERENCE declaration. A procedure reference can be passed as an actual parameter to a formal procedure that is of the same type and that has the same parameter descriptions.

Example of a PROCEDURE REFERENCE Declaration

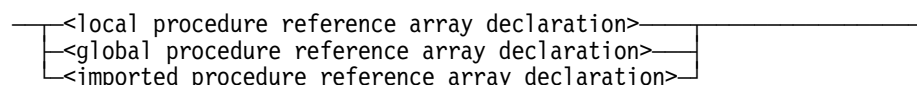
The following example declares a procedure reference for a real procedure with two parameters of type integer. The second parameter is passed by value.

```
REAL PROCEDURE REFERENCE RPREF(A,B);  
    VALUE B;  
    INTEGER A,B;  
    NULL;
```

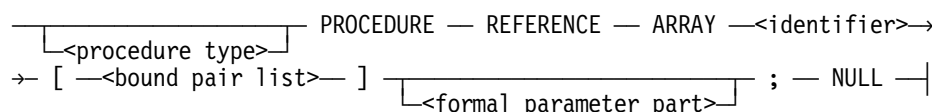
PROCEDURE REFERENCE ARRAY Declaration

The PROCEDURE REFERENCE ARRAY declaration declares a procedure reference array, which is a structure that allows a group of like procedures to be treated as a single entity. A procedure in the group can be invoked by referencing an element of the procedure reference array. An imported procedure reference array declaration is allowed only inside a connection block type.

<procedure reference array declaration>



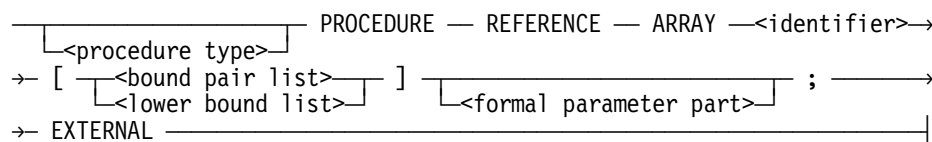
<local procedure reference array declaration>



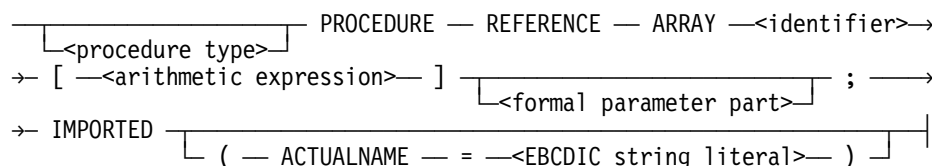
<procedure reference array identifier>

An identifier that is associated with a procedure reference array in a PROCEDURE REFERENCE ARRAY declaration.

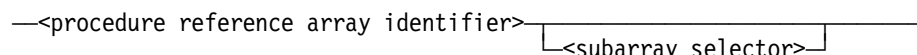
<global procedure reference array declaration>



<imported procedure reference array declaration>



<procedure reference array designator>



PROCEDURE REFERENCE ARRAY Declaration

<procedure reference array row>

—<procedure reference array identifier> —————
 └─<row selector>┘

<procedure reference array element>

—<procedure reference array identifier> — [┌─<subscript>┐] ————

Placement of Procedure Reference Arrays

A procedure reference array is an array of references to procedures of identical type and parameters. An element of a procedure reference array can appear in the following places:

- On either side of a procedure reference array assignment
- As a primary in an expression, if the procedure reference array has a type associated with it
- In a PROCEDURE REFERENCE ARRAY statement
- As a formal or actual parameter
- As an object exported by, or imported from, a library in a LIBRARY declaration

For more information, see “LIBRARY Declaration” earlier in this section.

Before an element of a procedure reference array can be used as a parameter, as a primary, or in a PROCEDURE REFERENCE ARRAY statement, it must be initialized in a procedure reference array assignment. A procedure assigned to the element must have had its parameters declared explicitly.

A procedure reference array can appear in the formal parameter part of a PROCEDURE declaration or of another PROCEDURE REFERENCE ARRAY declaration. A formal parameter that is a procedure reference array must be declared FORMAL so that all of its parameters are checked at compilation time.

A procedure reference array element can be passed as an actual parameter to a formal procedure that is of the same type and that has the same parameter descriptions.

A local PROCEDURE REFERENCE ARRAY declaration cannot appear in the global part of a program unit. A global PROCEDURE REFERENCE ARRAY declaration can appear in the global part of a program unit or in a LIBRARY declaration.

Example of a PROCEDURE REFERENCE ARRAY Declaration

The following example declares a 10-element procedure reference array, each element of which references a procedure of type INTEGER with two parameters. The first parameter is a call-by-value integer simple variable, and the second is an untyped procedure reference array with a lower bound of 0 (zero) and no parameters.

```
INTEGER PROCEDURE REFERENCE ARRAY REFARRAY[1:10] (Q,R);  
  
        VALUE                Q;  
        INTEGER              Q;  
        PROCEDURE REFERENCE ARRAY R[0] ();  
  
        FORMAL;  
  
        NULL
```

PROLOG PROCEDURE Declaration

<prolog procedure declaration>

— PROLOG PROCEDURE —<identifier>— ; —<unlabeled statement>—————|

The PROLOG PROCEDURE declaration can only be declared inside a STRUCTURE or CONNECTION BLOCK TYPE declaration. The PROLOG PROCEDURE declaration allows you to designate a procedure that must be executed when the structure or connection block instance is first created. If a PROLOG PROCEDURE declaration exists for a structure or connection block, the prolog procedure is executed automatically by the operating system when the structure or connection block instance (structure block variable, structure block array element, or connection) is first touched. You can also invoke the prolog procedure directly.

The following restrictions apply to PROLOG PROCEDURE declarations:

- No parameters are allowed.
- A prolog procedure cannot return a value.
- The body of a prolog procedure can contain a compound statement only. The prolog procedure body cannot be declared as external, as a library entry point specification, or as a dynamic procedure specification.
- A prolog procedure cannot be declared as a formal parameter. However, a prolog procedure can be passed as an actual parameter if the formal parameter is an untyped procedure without parameters.

Identifiers

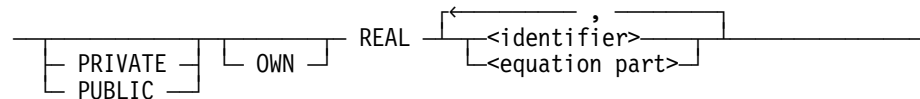
<prolog procedure identifier>

An identifier that is associated with a prolog procedure in a PROLOG PROCEDURE declaration.

REAL Declaration

A REAL declaration declares simple variables that can have real values, that is, arithmetic values that have exponents and fractional parts.

<real declaration>



<real identifier>

An identifier that is associated with the REAL data type in a REAL declaration.

Declaration of Simple Variables

The PRIVATE and PUBLIC specifiers can only be used for simple variables that are declared within a structure block or a connection block. The PRIVATE specifier limits visibility of the simple variable to the scope of the structure block or the connection block. A PRIVATE simple variable cannot be accessed using a structure or connection block qualifier. The PUBLIC specifier allows the simple variable to be accessed using a structure or connection block qualifier. If neither PRIVATE nor PUBLIC is specified, the default value is PUBLIC and access to the simple variable using a structure or connection block qualifier is allowed.

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

The equation part causes the simple variable being declared to have the same address as the simple variable associated with the second identifier.

For more information on <equation part>, see "BOOLEAN Declaration" earlier in this section.

This action is called address equation. An identifier can be address-equated only to a previously declared local identifier or to a global identifier. The first identifier must not have been previously declared within the block of the equation part. An equation part is not allowed in the global part of a program unit.

Address equation is allowed only between integer, real, and Boolean variables. Because both identifiers of the equation part have the same address, altering the value of either variable affects the value of both variables. For more information, see "Type Coercion of One-Word and Two-Word Operands" in Appendix C, "Data Representation."

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

REAL Declaration

If a real or integer value is assigned to a real variable, it is stored as is into the variable. If a double-precision value is assigned to a real variable, it is rounded to single-precision before it is stored in the variable.

When a real simple variable is allocated, it is initialized to 0 (zero), which is a 48-bit word with all bits equal to 0.

See Appendix C, "Data Representation," for additional information on the internal structure of a real operand.

Examples of REAL Declarations

The following example declares INDX, X, Y, and TOTAL as real variables.

```
REAL INDX,X,Y,TOTAL
```

The following example declares CALC, INDEX, and VALU as real variables. CALC is address-equated to the simple variable BOOL, and VALU is address-equated to the simple variable INTR. According to this declaration, CALC and BOOL share the same address, and VALU and INTR share the same address.

```
REAL CALC = BOOL, INDEX, VALU = INTR
```

The following example declares DISTANCE and REALINDEX as real variables. Because these variables are declared to be OWN, the variables retain their values when the program exits the block in which they are declared.

```
OWN REAL DISTANCE, REALINDEX
```

SIMPLE VARIABLE Declaration

A SIMPLE VARIABLE declaration declares simple variables that can be used in a manner appropriate to the specified type.

<simple variable declaration>



Type-transfer functions can be used, as can the equation part construct, to perform operations on a variable other than those that are valid for the type of the variable.

Each type of simple variable is used as follows:

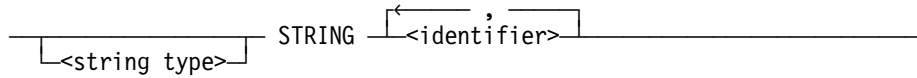
Type	Meaning/Description
BOOLEAN	Boolean values. A Boolean variable is a one-word variable in which the Boolean value (TRUE or FALSE) depends on the low-order bit (bit zero) of the word. The use of partial word parts and concatenation enables all 48 bits to be tested or manipulated as needed.
COMPLEX	Complex values. A complex variable consists of two real variables in which the first variable contains the real part and the second variable contains the imaginary part.
DOUBLE	Double-precision arithmetic values. A double-precision variable is a two-word variable.
INTEGER	Integer arithmetic values. An integer value is one that has an exponent of 0 (zero) and no fractional part. Integer variables are one-word variables.
REAL	Real arithmetic values. A real value is one that can have an exponent and a fractional part. Real variables are one-word variables.

See Appendix C, "Data Representation," for more information regarding the internal structure of each type of simple variable.

STRING Declaration

A STRING declaration declares simple variables to be strings. Strings allow storage and manipulation of character strings in a program.

<string declaration>



<string type>



<string identifier>

An identifier that is associated with the STRING data type in a STRING declaration.

STRING Type

The type STRING is a structured data type that contains characters of only one character type.

A string has two components: contents and length. No trailing blanks or null characters are added to a string; therefore the length of a string is exactly the number of characters stored in the string. The maximum string length allowed is $2^{16}-2$ characters.

All strings declared in a STRING declaration are of the same string type. If no string type is specified in the STRING declaration, then the default character type is used. The default character type can be designated by the compiler control option ASCII. If no such compiler control option is designated, the default character type is EBCDIC. For more information, refer to "Default Character Type" in Appendix C, "Data Representation."

The number of strings that can be declared in a program is limited by the operating system to 500. If this limit is exceeded, the message *STRING POOL EXCEEDED* is given.

Examples of STRING Declarations

The following example declares S1, S2, and S3 as string simple variables of string type ASCII. S1, S2, and S3 contain ASCII characters.

```
ASCII STRING S1,S2,S3
```

The following example declares S5, S6, S7, and S8 as string simple variables of string type EBCDIC. These strings contain EBCDIC characters.

```
EBCDIC STRING S5,S6,S7,S8
```

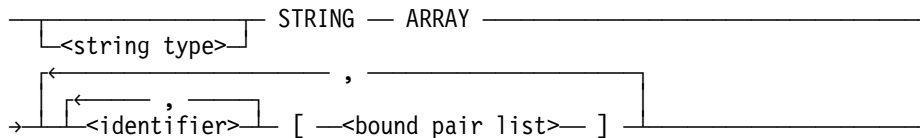
The following example declares S9 as a string simple variable. Because no string type is specified, the default character type is used. This character type is EBCDIC unless the compiler control option ASCII is TRUE, in which case the string type is ASCII.

```
STRING S9
```

STRING ARRAY Declaration

A STRING ARRAY declaration declares string arrays. A string array is an array that has string elements.

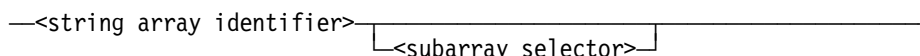
<string array declaration>



<string array identifier>

An identifier that is associated with a string array in a STRING ARRAY declaration.

<string array designator>



String Array Type

All string arrays declared in a STRING ARRAY declaration are of the same string type. If no string type is specified, the default character type is used. The default character type can be designated by the compiler control option ASCII. If no such compiler control option is used, the default character type is EBCDIC. For more information, refer to “Default Character Type” in Appendix C, “Data Representation.”

The restrictions that apply to arrays also apply to string arrays. For more information on bound pair lists, subarray selectors, and arrays, see “ARRAY Declaration” earlier in this section.

Examples of STRING ARRAY Declarations

The following example declares SA, SB, and SC as one-dimensional arrays of strings, each with a lower bound of 0 and an upper bound of 10. Because no string type is specified, the default character type is used. This character type is EBCDIC unless the compiler control option ASCII is TRUE, in which case the string type is ASCII.

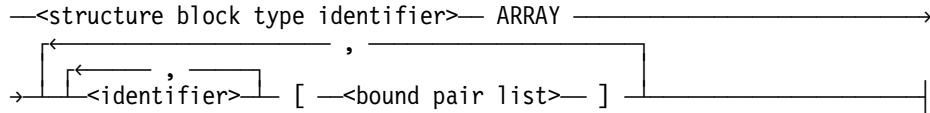
```
STRING ARRAY SA,SB,SC[0:10]
```

The following example declares ESA, ESB, and ESC as arrays of strings. The string type is EBCDIC, so each is an array of EBCDIC strings. ESA is one-dimensional and has a lower bound of 1 and an upper bound of 15. Arrays ESB and ESC are two-dimensional arrays with lower bounds of 0 and upper bounds of 10 for both dimensions.

```
EBCDIC STRING ARRAY ESA[1:15], ESB, ESC[0:10, 0:10]
```


STRUCTURE BLOCK ARRAY Declaration

<structure block array declaration>



A STRUCTURE BLOCK ARRAY declaration declares an array whose elements are structure block instances. Each lower bound and upper bound in the bound pair list must resolve to a constant arithmetic expression.

Structure block types can be declared within structure blocks, but a structure block variable or array cannot be declared with the type of an inner structure block outside the outer structure block. The following example illustrates the declaration of structure blocks:

```

BEGIN
TYPE STRUCTURE BLOCK STRBLOCKTYPE1;
  BEGIN
    REAL R;
    TYPE STRUCTURE BLOCK STRBLOCKTYPE2;
      BEGIN
        REAL RR;
        * * *
        END; %STRBLOCKTYPE2
      STRBLOCKTYPE2 ARRAY STR2ID [0:10];          %ALLOWED
      * * *
    END; %STRBLOCKTYPE1

    STRBLOCKTYPE1 ARRAY STRID1 [0:10];           %ALLOWED
    STRID1.STRBLOCKTYPE2 ARRAY STR2ID [0:10];   %NOT ALLOWED
    STRBLOCKTYPE1.STRBLOCKTYPE2 ARRAY STR2ID2 [0:10]; %NOT ALLOWED
    * * *
  END.

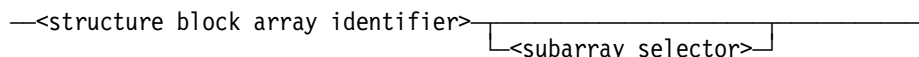
```

Identifiers

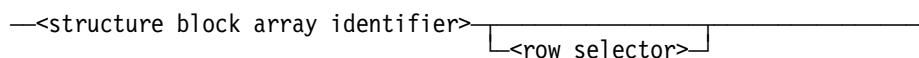
<structure block array identifier>

An identifier associated with a structure block array in a STRUCTURE BLOCK ARRAY declaration.

<structure block array designator>



<structure block array row>



STRUCTURE BLOCK ARRAY Declaration

<structure block array element>

—<structure block array identifier> [$\overbrace{\text{<subscript>}}^{\leftarrow, \rightarrow}$] —————|

STRUCTURE BLOCK REFERENCE VARIABLE Declaration

<structure block reference variable declaration>

`—<structure block type identifier> . REFERENCE —<identifier>—|`

A STRUCTURE BLOCK REFERENCE VARIABLE declaration declares a structure reference variable with the type of the structure block type identifier. A STRUCTURE BLOCK REFERENCE ASSIGNMENT statement can then be used to assign an instance of the structure block of the declared type to this variable.

A reference to an embedded structure block is allowed by the following syntax:

```
outerSBtype.embeddedSBtype REFERENCE <id>
```

Identifiers

<structure block reference variable>

An identifier that is associated with a structure block reference in a STRUCTURE BLOCK REFERENCE VARIABLE declaration.

The following example shows the declaration of a structure block reference variable:

```
TYPE STRUCTURE BLOCK SB;
  BEGIN
    TYPE STRUCTURE BLOCK SBINNER;
      BEGIN
        REAL X;
        .
        .
      END;
    SBINNER SBINNERVAR;
  END;

SB SBVAR;
SB.SBINNER REFERENCE SBINNER_REF;
SBINNER_REF := SBVAR.SBINNERVAR;
```

STRUCTURE BLOCK TYPE Declaration

A STRUCTURE BLOCK TYPE declaration provides a method of grouping together data and procedures that act upon that data into a logical unit, and have the data persist past block exit of those procedures. Structure block variables and structure block arrays can be declared from the STRUCTURE BLOCK TYPE declaration.

<structure block type declaration>

—<local structure block type declaration>—
—<global structure block type declaration>—

<local structure block type declaration>

— TYPE STRUCTURE BLOCK —<identifier>— ; — BEGIN —————→
→<declaration list>— END —————|

There are restrictions on what can be declared in the declaration list. The following cannot be declared within a structure block: <database declaration>, <label declaration>, <switch label declaration>, <dictionary option>, <dump declaration>, <exception procedure declaration>, <exportlibrary declaration>, <transaction base declaration>, <transaction record declaration>, <transaction record array declaration>, and <transaction declaration>.

PROLOG PROCEDURE declarations and PENDING PROCEDURE declarations can only be located within a structure block or connection block.

A simple variable declared with the OWN specification in a structure block has one value for all structure block variables and arrays with the type of that structure block. This enables all structure block variables and arrays with the same structure block type to share a variable.

A structure block variable or a structure block array can be declared within structure or connection blocks, but a structure block variable or array cannot be declared with the type of the inner structure block outside the outer structure or connection block.

Note: An *EVENT* or *EVENT ARRAY* element that is declared in the <declaration list> of a STRUCTURE BLOCK TYPE declaration cannot be used as the <event part> of a direct I/O statement unless the direct array involved is declared in the same structure block and both are items of the same instance of that type.

<global structure block type declaration>

—<forward global structure block type declaration>—
—<external global structure block type declaration>—

A global structure block can appear only in the global part of a program unit and cannot be nested. Global structure blocks enable the replacement binding of structure block procedures.

<forward global structure block type declaration>

— TYPE STRUCTURE BLOCK —<identifier>— ; — FORWARD —————|

Structure blocks can be declared FORWARD within the global part of a subprogram. The global structure block then can be used for structure block variable declarations within the global part. The global structure block must be fully specified before it is referenced. This syntax is used when the subprogram procedure references a structure block item from outside the structure block.

<external global structure block type declaration>

— TYPE STRUCTURE BLOCK —<identifier>— ; — EXTERNAL —————|

The EXTERNAL structure block sets up the environment for structure block pending procedures in the subprogram to be bound to a host. Only those structure block items that are referenced by pending procedures need to be specified in the subprogram structure block declaration. The local declarations of the structure block items in the host that are not referenced by pending procedures do not need to be specified in the subprogram. This enables pending procedures in the subprogram to be bound without having to fully declare the entire structure block. EXTERNAL structure blocks cannot be bound. EXTERNAL structure blocks must be declared at level 3.

For more information about structure block binding, refer to the *Binder Programming Reference Manual*.

Identifiers

<structure block type identifier>

An identifier that is associated with a structured block type in a STRUCTURE BLOCK TYPE declaration.

Referencing Structure Block Items Outside the Structure Block

<structure block item designator>

—<structure block qualifier>—<structure block item>—————|

<structure block qualifier>

—| —<structure block variable identifier> . —————|
| —<structure block reference variable> ————|
| —<structure block array element> —————|

<structure block item>

Any identifier that was declared inside a STRUCTURE BLOCK TYPE declaration.

When items of a structure block are referenced outside the structure block, the structure block qualifier must be used to identify those items. The types of structure block items that can be referenced outside the structure block are procedures, structure block variables, structure block arrays, structure block reference variables, connection libraries, connection block reference variables, events, event arrays, interlocks, interlock arrays, data, and data arrays of type BOOLEAN, COMPLEX, DOUBLE, INTEGER, and REAL. Array references and character array types of ASCII, EBCDIC, and HEX can also be referenced.

Procedures declared within an EXTERNAL structure block can be declared only as NULL or PENDING. A structure block procedure can be declared as NULL in an EXTERNAL structure block in a subprogram. The procedure serves as a placeholder for a procedure that is referenced by the pending procedure that is to be bound.

STRUCTURE BLOCK VARIABLE Declaration

<structure block variable declaration>

—<structure block type identifier> —<identifier> —

A STRUCTURE BLOCK VARIABLE declaration declares a single instance of a structure block with the type of the structure block type identifier.

Structure block types and instances can be declared within structure blocks, but a structure block variable or array cannot be declared with the type of an inner structure block outside the outer structure block. The following example illustrates the declaration of structure blocks:

```

BEGIN
TYPE STRUCTURE BLOCK STRBLOCKTYPE1;
  BEGIN
    REAL R;
    TYPE STRUCTURE BLOCK STRBLOCKTYPE2;
      BEGIN
        REAL RR;
        * * *
      END;  %STRBLOCKTYPE2
    STRBLOCKTYPE2 STR2ID;          %ALLOWED
    * * *
  END;  %STRBLOCKTYPE1

STRBLOCKTYPE1 STRID1;          %ALLOWED

STRID1.STRBLOCKTYPE2 STR2ID    %NOT ALLOWED
STRBLOCKTYPE1.STRBLOCKTYPE2 STR2ID2; %NOT ALLOWED
* * *
END.

```

Identifiers

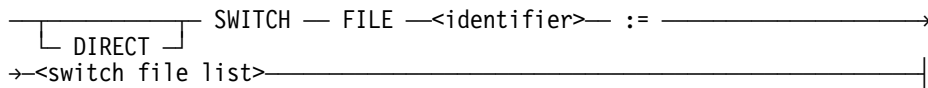
<structure block variable identifier>

An identifier associated with a structure block in a STRUCTURE BLOCK VARIABLE declaration.

SWITCH FILE Declaration

A SWITCH FILE declaration associates an identifier with a list of file designators. Any of these file designators can later be referenced by using the identifier and a number corresponding to the position of the file designator in the list.

<switch file declaration>



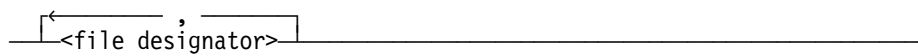
<switch file identifier>

An identifier that is associated with a switch file list in a SWITCH FILE declaration.

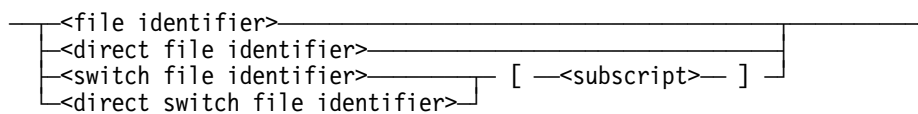
<direct switch file identifier>

An identifier that is associated with a switch file list in a DIRECT SWITCH FILE declaration.

<switch file list>



<file designator>



Switch File List

An integer index is associated with each file designator in the switch file list. The indexes are 0, 1, 2, and so on through N-1, where N is the number of file designators in the list. These indexes are obtained by counting the file designators in order of their appearance in the list. A file designator in the list can be referenced by subscripting the switch file identifier with a subscript whose value is equal to the index of the file designator.

If a subscript to a switch file identifier yields a value outside the range of the switch file list (that is, less than 0 or greater than N-1), a fault occurs at run time.

Any subscripts in the switch file list are evaluated at the time of the SWITCH FILE declaration.

A switch file can reference itself in the switch file list, in which case a stack overflow might occur when the program is executed. For example, assume a switch file is declared as the following:

```
SWITCH FILE SF := F1, F2, SF[N]
```

If N equals 2, the subscripted switch file identifier SF[N] references itself indefinitely.

The switch file list of a switch file that is not designated as DIRECT can contain only file designators that are not DIRECT, and the switch file list of a switch file that is designated DIRECT can contain only file designators that are DIRECT.

Example of a SWITCH FILE Declaration

The following example declares CHOOSEUNIT to be a switch file identifier with a list of three file designators. CHOOSEUNIT[0] evaluates to file CARDOUT, CHOOSEUNIT[1] evaluates to file TAPEOUT, and CHOOSEUNIT[2] evaluates to file PRINTOUT.

```
SWITCH FILE CHOOSEUNIT :=  
    CARDOUT,  
    TAPEOUT,  
    PRINTOUT;  
.  
.  
.  
WRITE(CHOOSEUNIT[0], 14, A[*]); % WRITES TO CARDOUT  
WRITE(CHOOSEUNIT[1], 14, A[*]); % WRITES TO TAPEOUT  
WRITE(CHOOSEUNIT[2], 14, A[*]); % WRITES TO PRINTOUT
```

SWITCH FORMAT Declaration

A SWITCH FORMAT declaration associates an identifier with a list of items representing editing specifications. Any of these items and the associated editing specifications can later be referenced by using the identifier and a number corresponding to the position of the item in the list.

<switch format declaration>

— SWITCH — FORMAT —<identifier>— := —<switch format list>—

<switch format identifier>

An identifier that is associated with a switch format list in a SWITCH FORMAT declaration.

<switch format list>

— [—<switch format segment>— , —<switch format segment>—] —

<switch format segment>

—<format designator>— [(—<editing specifications>—) —]
—<editing specifications>—

<format designator>

—<format identifier>— [—<switch format identifier>— [—<subscript>—] —]

Switch Format List

An integer index is associated with each switch format segment in the switch format list. The indexes are 0, 1, 2, and so on through N-1, where N is the number of switch format segments in the list. These indexes are obtained by counting the switch format segments in order of their appearance in the list. A switch format segment in the list can be referenced by subscripting the switch format identifier with a subscript whose value is equal to the index of the switch format segment.

If a subscript to a switch format identifier yields a value outside the range of the switch format list (that is, less than 0 or greater than N-1), a fault occurs at run time.

Any subscripts in the switch format list are evaluated at the time the subscripted switch format identifier is encountered.

A switch format can reference itself in the switch format list, in which case a stack overflow might occur when the program is executed. For example, assume a switch format is declared as the following:

```
SWITCH FORMAT SF := FMT1, FMT2, SF[N]
```

If N equals 2, the subscripted switch format identifier SF[N] references itself indefinitely.

A simple string literal in a SWITCH FORMAT declaration is always read-only if the switch format segment in which it appears consists of editing specifications rather than a format designator.

Examples of SWITCH FORMAT Declarations

The following example declares SF to be a switch format identifier with a switch format list of four sets of editing specifications. The editing specifications (X78, I2), for example, can be referenced as SF[2].

```
SWITCH FORMAT SF := (A6, 3I4, I2, X60), % 0
                   (I4, X2, 2I4, 3I2), % 1
                   (X78, I2),         % 2
                   (X2)                % 3
```

The following example declares SWHFT to be a switch format identifier with a switch format list of three format designators. SWHFT[0] evaluates to format FMT1, SWHFT[1] to FMT2, and SWHFT[2] to FMT3.

```
SWITCH FORMAT SWHFT := FMT1,FMT2,FMT3
```

SWITCH LABEL Declaration

A SWITCH LABEL declaration associates an identifier with a list of designational expressions, which are expressions that evaluate to labels. Any of these designational expressions can later be referenced by using the identifier and a number corresponding to the position of the designational expression in the list.

<switch label declaration>

— SWITCH —<identifier>— := —<switch label list>—————|

<switch label identifier>

An identifier that is associated with a switch label list in a SWITCH LABEL declaration.

<switch label list>

—|<designational expression>—————|

Switch Label List

An integer index is associated with each designational expression in the switch label list. The indexes are 1, 2, 3, and so on through N, where N is the number of designational expressions in the list. These indexes are obtained by counting the designational expressions in order of their appearance in the list. A designational expression in the list can be referenced by subscripting the switch label identifier with a subscript whose value is equal to the index of the designational expression.

Note that the indexing of a switch label list begins at 1.

If a subscript to a switch label identifier yields a value outside the range of the switch label list (that is, less than 1 or greater than N), the statement using the switch label is not executed, and control proceeds to the next statement. Typically, the next statement is a specification of some form of error handling.

The designational expressions in a switch label list are evaluated at the time the subscripted switch label identifier is encountered.

A switch label can reference itself in the switch label list, in which case a stack overflow might occur when the program is executed. For example, assume a switch label is declared as the following:

```
SWITCH SW := L1, L2, L3, SW[N]
```

If N equals 4, the designational expression SW[N] references itself indefinitely.

Examples of SWITCH LABEL Declarations

The following example declares CHOOSEPATH to be a switch label identifier with labels L1, L2, L3, and L4 in the switch label list. CHOOSEPATH[1] evaluates to label L1, CHOOSEPATH[2] to L2, and so on.

```
SWITCH CHOOSEPATH := L1,L2,L3,L4
```

The following example declares SELECT to be a switch label identifier with labels START and ERROR1 and designational expression CHOOSEPATH[2] in the switch label list. Note that from the previous SWITCH LABEL declaration, CHOOSEPATH[2] evaluates to L2; therefore, SELECT[3] evaluates to L2.

```
SWITCH SELECT := START,           % 1  
                ERROR1,          % 2  
                CHOOSEPATH[2]    % 3
```

SWITCH LIST Declaration

A SWITCH LIST declaration associates an identifier with a list of list designators. Any of these list designators can later be referenced by using the identifier and a number corresponding to the position of the list designator in the list.

<switch list declaration>

— SWITCH — LIST —<identifier>— := —<list designator>—

<switch list identifier>

An identifier that is associated with a list of list designators in a SWITCH LIST declaration.

<list designator>

—<list identifier>— [—<switch list identifier>— [—<subscript>—]]

List Designator

An integer index is associated with each list designator in the declaration. The indexes are 0, 1, 2, and so on through N-1, where N is the number of list designators in the declaration. These indexes are obtained by counting the list designators in order of their appearance in the declaration. Any of these list designators can be referenced by subscripting the switch list identifier with a subscript whose value is equal to the index of the list designator.

If a subscript to a switch list identifier yields a value outside the range of the list of list designators (that is, less than 0 or greater than N-1), a fault occurs at run time.

Any subscripts in the list of list designators are evaluated at the time the subscripted switch list identifier is encountered.

A switch list can reference itself in the list of list designators, in which case a stack overflow might occur when the program is executed. For example, assume a switch list is declared as the following:

```
SWITCH LIST SL := L1, L2, SL[N]
```

If N equals 2, the subscripted switch list identifier SL[N] references itself indefinitely.

Example of a SWITCH LIST Declaration


The following example declares NUMVARIABLES to be a switch list identifier and associates four list designators with it. NUMVARIABLES[0] evaluates to the list NOVARS, NUMVARIABLES[1] evaluates to ONEVAR, and so on.

```
SWITCH LIST NUMVARIABLES := NOVARS,    % 0
                           ONEVAR,     % 1
                           TWOVARS,    % 2
                           THREEVARS  % 3
```

TASK and TASK ARRAY Declarations

The TASK and TASK ARRAY declarations are used to declare tasks and task arrays, which can then be associated with a process or coroutine. Task attributes can be used to control or to contain information about the process or coroutine.

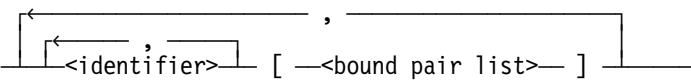
<task declaration>

— TASK — 

<task identifier>

An identifier that is associated with a task in a TASK declaration.

<task array declaration>

— TASK — ARRAY 

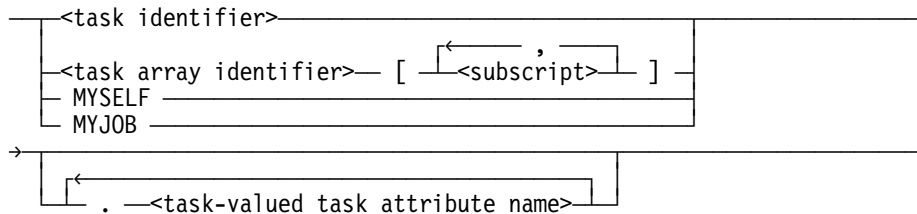
<task array identifier>

An identifier that is associated with a task array in a TASK ARRAY declaration.

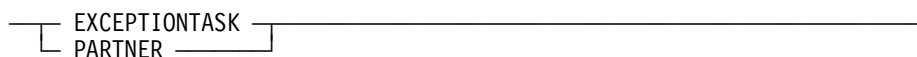
A task array is an array whose elements are tasks. A task array can have no more than 15 dimensions.

Task and Task Array Designator

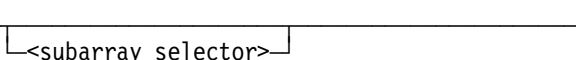
<task designator>



<task-valued task attribute name>



<task array designator>

—<task array identifier> 

A task designator represents a single task. A task array designator represents an array of tasks. MYSELF is the task designator for the currently running program. MYJOB is the task designator for the currently running job.

When a process or coroutine is invoked, a task can be associated with it. For example, a task designator can appear in a CALL statement, PROCESS statement, or RUN statement. Task attributes can be assigned values by the program to control the process or coroutine, and the program can interrogate the values of task attributes as the process or coroutine executes.

The EXCEPTIONEVENT attribute of the EXCEPTIONTASK of a program is caused whenever the status of that program changes (for example, if the program is suspended or terminated).

Attributes associated with a task designator can be assigned values or interrogated in a program by specifying the task designator and the appropriate task attribute names in assignment statements.

For information on processes and coroutines, see "CALL Statement," "PROCESS Statement," and "RUN Statement" in Section 4, "Statements." For more information on assigning and interrogating task attributes, see the <arithmetic task attribute> construct under "Arithmetic Assignment," the <Boolean task attribute> construct under "Boolean Assignment" and "Task Assignment" in Section 4, "Statements."

Examples of TASK and TASK ARRAY Declarations

The following example declares PROCESSTASK to be a task identifier.

```
TASK PROCESSTASK
```

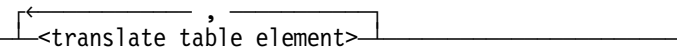
The following example declares CHILDREN as a one-dimensional task array with a lower bound of 0 (zero) and an upper bound of LIM. The CHILDREN task array might be used to store the tasks associated with a group of processes and coroutines initiated by a program.

```
TASK ARRAY CHILDREN[0:LIM]
```

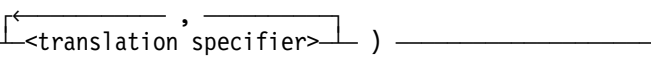
TRANSLATETABLE Declaration

The TRANSLATETABLE declaration defines one or more translate tables. Used in a REPLACE statement, a translate table indicates translations to be performed from one group of characters to another group of characters.

<translate table declaration>

— TRANSLATETABLE —  —

<translate table element>

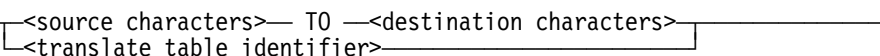
— <identifier> — () —

<translate table identifier>

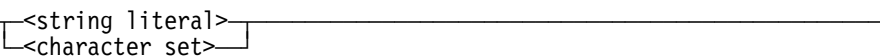
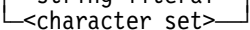
An identifier that is associated with a group of one or more translation specifiers in a TRANSLATETABLE declaration.

Translation Specifier


<translation specifier>

 — <source characters> — TO — <destination characters> —

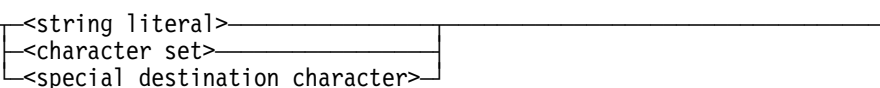
<source characters>

 —  —

<character set>

 —

<destination characters>

 —

<special destination character>

A string literal that is 1 character long.

Specifying a character set is equivalent to specifying all the characters in that set, in ascending binary sequence. The length of a character set is equal to the total number of characters in the set.

A string literal specifies all the characters in the string literal. The length of a string literal is equal to the number of characters in the string literal in terms of the largest character size specified by the string literal.

A translation specifier is enclosed in parentheses, and each succeeding translation specifier overrides the previous translation specifiers.

Within a single translate table, all source character sizes and all destination character sizes must be the same, although the character sizes of the source and destination parts need not be the same.

The number of destination characters must equal the number of source characters, unless the special destination character is used or unless a character set is used for both the source characters and the destination characters. If the special destination character is used, all the source characters are translated to the special destination character.

Every translate table has a default base in which all source characters are translated to characters with all bits equal to 0 (zero). This means that all source characters that do not appear in the TRANSLATETABLE declaration are translated to the character whose binary representation had all bits equal to 0 (zero).

The use of a character set for both the source and destination parts invokes a standard table from the operating system and provides a way of obtaining a legitimate base on which additional translation specifiers can be used, if desired, to override certain parts of the standard table. The use of a translate table identifier as a translation specifier can also be used to provide a base.

When string literals of equal length are used for the source and destination parts, translation is based on the corresponding positions of the source and destination characters, from left to right.

Translate Table Indexing

The size of a translate table is determined by the size of the source characters (the characters to be translated): 4-bit characters require a 4-word table; 7-bit and 8-bit characters require a 64-word table. A translate table is a one-dimensional read-only array.

Each word in a translate table (Figure 3–1) has its low-order 32 bits divided into four 8-bit fields, numbered 0 to 3 from left to right. The high-order 16 bits are all zeros.

When a character is to be translated, the binary representation of the character is divided into two parts: a word index and a field index. The field index consists of the two low-order bits; the word index consists of the remaining high-order bits. The word index designates the word in the translate table in which the field index designates the character into which the source character is to be translated.

TRANSLATETABLE Declaration

Figure 3-1 below shows indexing for the translation of *a* to *A* that would result from the following declaration:

```
TRANSLATETABLE UPCASE (EBCDIC TO EBCDIC,
                        "abcdefghijklmnopqrstuvwxyz" TO
                        "ABCDEFGHIJKLMNOPQRSTUVWXYZ")
```

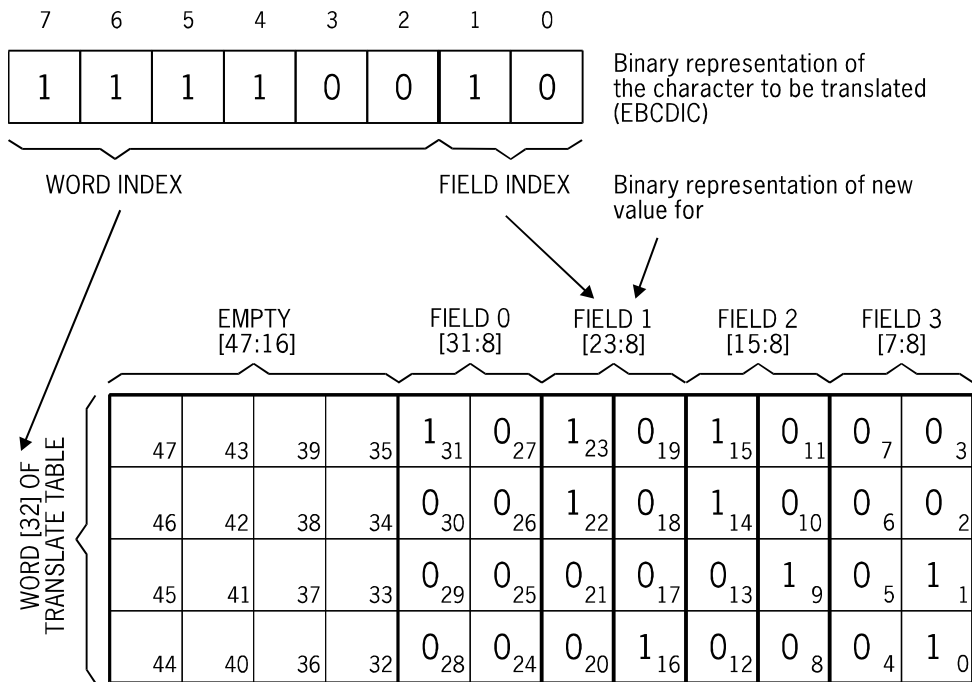


Figure 3-1. Translate Table Indexing

Examples of TRANSLATETABLE Declarations

The following example translates the letters *L* to *G*, *E* to *O*, *A* to *L*, and *D* to *D*. All other characters are translated to the character whose binary representation has all bits equal to 0 (zero). Both the source and the destination characters are of the default character type.

```
TRANSLATETABLE ALCHEMY ("LEAD" TO "GOLD")
```

The following example translates all EBCDIC characters to themselves except for the lowercase letters, which it translates to uppercase letters.

```
TRANSLATETABLE UPCASE (EBCDIC TO EBCDIC,  
                        "abcdefghijklmnopqrstuvwxyz" TO  
                        "ABCDEFGHIJKLMNOPQRSTUVWXYZ")
```

The following example translates all EBCDIC characters to themselves except for the left parenthesis (*()*), which is translated to the left square bracket (*[]*).

```
TRANSLATETABLE PAREN_TO_BRACKET (EBCDIC TO EBCDIC, 8(" TO 8[")
```

The following example translates all EBCDIC characters to themselves except for the digits, which it translates to periods (*.*).

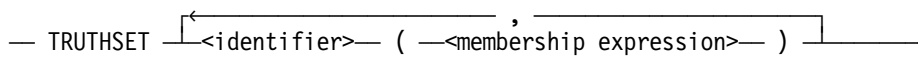
```
TRANSLATETABLE NUMBERS_TO_PERIODS (EBCDIC TO EBCDIC,  
                                    "0123456789" TO ".")
```

TRUTHSET Declaration

The TRUTHSET declaration associates an identifier with a set of characters. From the characters in a TRUTHSET declaration, the compiler builds a truth set table, which is used in a truth set test to determine whether a given character is a member of that group of characters. The identifier can then be used in a SCAN statement to scan while or until any character in the truth set occurs.

The identifier also can appear as a condition in a REPLACE statement, so that replacement takes place while or until any character in the truth set occurs.

<truth set declaration>

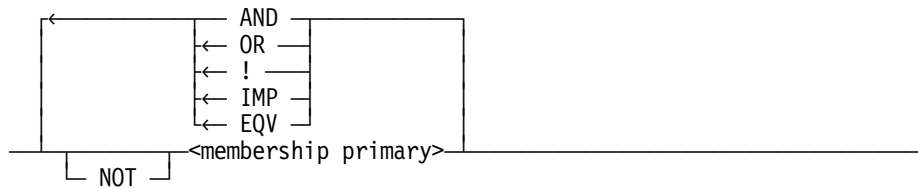


<truth set identifier>

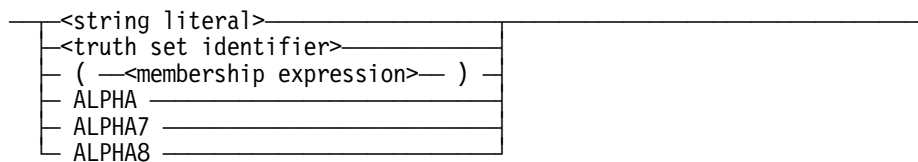
An identifier that is associated with a membership expression in a TRUTHSET declaration.

Membership Expression

<membership expression>



<membership primary>



All membership primaries of a membership expression must be of the same character size (4-bit, 7-bit, or 8-bit); this character size determines the type of the truth set. The character size of a string literal is determined by the maximum character size indicated by its component string codes. For more information, refer to “String Literal” in Section 2, “Language Components.”

A membership expression is evaluated according to the normal rules of precedence for Boolean operators. This precedence is described under “Boolean Expression” in Section 5, “Expressions and Functions.”

ALPHA, ALPHA7, and ALPHA8 are intrinsic truth sets defined as follows:

Truth Set	Definition
ALPHA7	A truth set that contains the ASCII digits and uppercase letters
ALPHA8	A truth set that contains the EBCDIC digits and uppercase letters
ALPHA	A truth set that contains the digits and uppercase letters of the default character type

If a default character type is not explicitly specified by the compiler control option ASCII, then the default character type is EBCDIC, and ALPHA is the same as ALPHA8. If the ASCII compiler control option is TRUE, then ALPHA is the same as ALPHA7.

Truth Set Test

From the characters in a TRUTHSET declaration, the compiler builds a truth set table, which is used in a truth set test to determine whether a given character is a member of that group of characters.

All truth sets declared by a single TRUTHSET declaration are stored in a single read-only array. Separate TRUTHSET declarations produce separate read-only arrays.

A truth set test references a bit in the read-only array containing the truth set by dividing the binary representation of the character being tested into two parts: the low-order five bits are used as a bit index, and the three high-order bits are used as a word index. If the size of the source character is smaller than eight bits, high-order zero bits are inserted to make an 8-bit character before the indexing algorithm is used.

The word index selects a particular word in the truth set table. The bit index is then subtracted from 31, and the result is used to reference one of the low-order 32 bits in the selected word. If the bit selected by the following expression is equal to 1, the character is a member of the truth set:

```
TABLE[CHAR.[7:3]].[(31-CHAR.[4:5]):1]
```

TRUTHSET Declaration

Figure 3–2 shows an example of a truth set test. In this example, the referenced bit (13) is equal to 1; therefore, the test character is a member of the truth set.

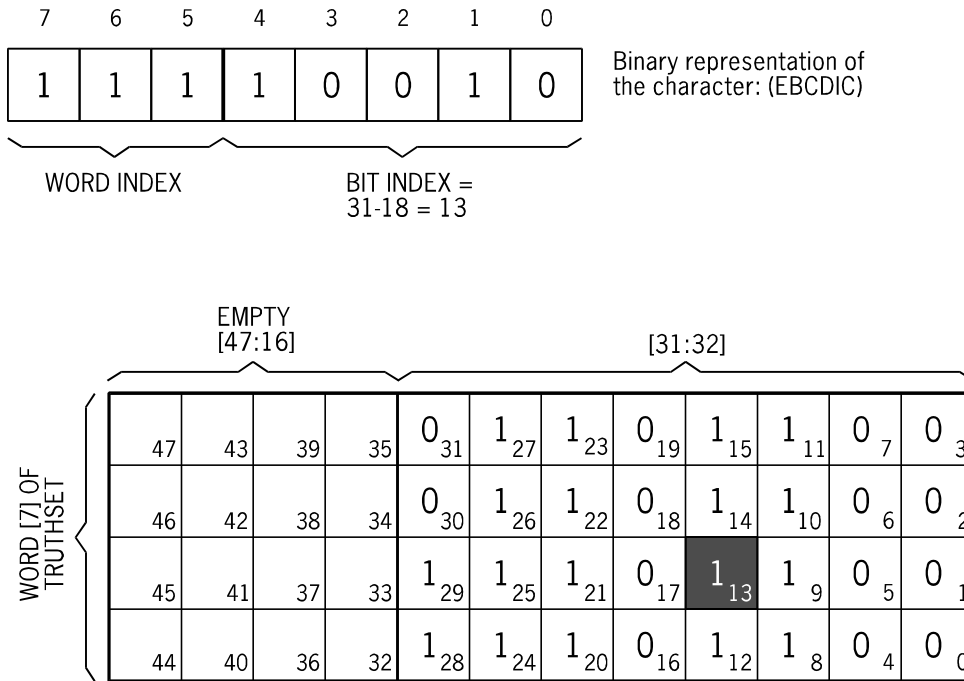


Figure 3–2. Truth Set Test

Examples of TRUTHSET Declarations

The following example declares T to be a truth set with membership equal to that of ALPHA. ALPHA consists of all uppercase letters and the digits 0 through 9, in the default character set.

```
TRUTHSET T(ALPHA)
```

The following example declares Z to be a truth set with membership of ALPHA8 and the hyphen (-).

```
TRUTHSET Z(ALPHA8 OR "-")
```

The following example declares NUMBERS to be a truth set with a membership of the digits 0 through 9 in the default character set.

```
TRUTHSET NUMBERS("0123478956")
```


The following example declares LETTERS to be a truth set with a membership of ALPHA but not the digits 0 through 9; that is, consisting of the uppercase letters in the default character set.

```
TRUTHSET LETTERS(ALPHA AND NOT NUMBERS)
```

The following example declares two truth sets:

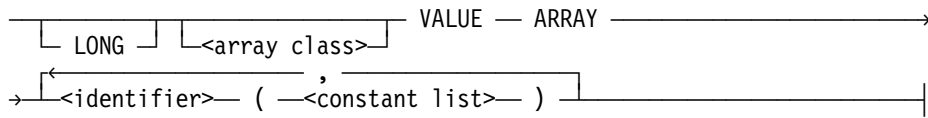
- HEXN, with a membership of the hexadecimal characters 1, 2, and 3
- ASCN, with a membership of the ASCII characters 1, 2, and 3

```
TRUTHSET HEXN(4"123"), ASCN(7"123")
```

VALUE ARRAY Declaration

A VALUE ARRAY declaration declares a read-only, one-dimensional array of constants.

<value array declaration>



<value array identifier>

An identifier that is associated with a value array in a VALUE ARRAY declaration.

A value array is a one-dimensional, read-only array. An element of a value array is referenced in the same manner as for any other array; that is, through a subscripted variable or by using a pointer. However, an attempt to store a value into a value array is flagged with a compile-time or run-time error.

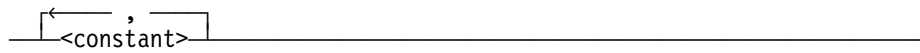
The lower bound of a value array is 0 (zero).

Normally, a value array longer than 1024 words is automatically paged (segmented) at run time into segments 256 words long. LONG specifies that the value array is not to be paged, regardless of its length.

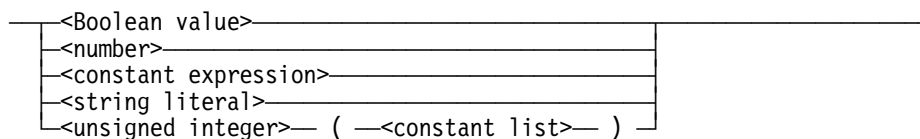
If no array class appears in a VALUE ARRAY declaration, a REAL array is assumed.

Constants

<constant list>



<constant>



<constant expression>

An arithmetic, Boolean, or complex expression that can be entirely evaluated at compilation time.

Each constant initializes an integral number of words. The number of words initialized depends on the type of the array and the kind of constant.

Single-precision numbers, single-precision expressions, Boolean values, and Boolean expressions initialize one word in value arrays other than double or complex value arrays. In double value arrays, this word is extended with a second word of 0 (zero). In complex value arrays, this word is normalized and then extended with an imaginary part of 0 (zero).

Double-precision numbers and expressions are stored unchanged in two words in double value arrays. In complex value arrays, the value is rounded and normalized to single-precision and then extended with an imaginary part of 0 (zero). For other types of value arrays, the second word of the double-precision value is dropped and the first word initializes one word of the array.

Complex expressions can appear only in complex value arrays, and they initialize two words of the array.

String literals more than 48 bits long initialize as many words as are needed to contain the string and are left-justified with trailing zeros inserted in the last word, if necessary. In complex and double value arrays, long string literals can initialize an odd number of words, causing the following constant to start in the middle of a two-word element of the array.

String literals less than or equal to 48 bits long are right-justified within one word with leading zeros, if necessary. This word initializes one word in value arrays other than double or complex value arrays. In double value arrays, this word is extended with a second word of 0 (zero). In complex value arrays, this word is normalized and then extended with an imaginary part of 0 (zero).

The <unsigned integer> (<constant list>) form of constant causes the values within the parentheses to be repeated the number of times specified by the unsigned integer.

The operating system overlays value arrays more efficiently than other arrays because value arrays need not be written to disk when their space in memory is relinquished.

The maximum size of an unpagged value array is 4095 words; the maximum size of a pagged value array is 32,767 words.

Example of a VALUE ARRAY Declaration

The following example declares DAYS to be a value array of real elements. DAYS stores the names of the days of the week, one day name in each two words. The string *FRIDAY*, for example, is stored in DAYS[8] and DAYS[9], and can be retrieved by assigning a pointer to DAYS[8] and using the pointer.

```
VALUE ARRAY DAYS ("MONDAY    ", "TUESDAY    ",
                  "WEDNESDAY  ", "THURSDAY   ",
                  "FRIDAY     ", "SATURDAY   ",
                  "SUNDAY     ")
```

VALUE ARRAY Declaration

Section 4

Statements

Statements are the active elements of an ALGOL program. They indicate an operation to be performed. Statements are normally executed in the order in which they appear in the program. A statement that transfers control to another program location can alter this sequential flow of execution. Note that a statement can be null or empty.

In this section, the ALGOL statements are listed and discussed in alphabetical order. In many cases, portions of the syntax of a statement are discussed before moving on to the next syntax segment.

The syntax for any statement is recursive: a statement can be a block or a compound statement, each of which, in turn, can include statements. For a description of the syntax of <block> and <compound statement>, see Section 1, "Program Structure."

Statements can be labeled or unlabeled. A <labeled statement> is of the following form:

—<label identifier>— : —<statement>—————|

An <unlabeled statement> is any statement that does not contain a label identifier.

ACCEPT Statement

The ACCEPT statement causes the display of a specified message on the Operator Display Terminal (ODT).

<accept statement>

— ACCEPT — (—

<pointer expression>
<string variable>
<subscripted string variable>

) ————— |

ACCEPT Parameters

The message displayed on the ODT is designated by the parameter to the ACCEPT statement. If the parameter is a pointer expression, then the characters to which the pointer expression points are displayed on the ODT. The pointer expression must point to EBCDIC characters, and the message to be displayed must be terminated by the EBCDIC null character (48"00"). Following display of the characters, the program is suspended until a response is entered at the ODT. The response is placed, left-justified, with leading blanks discarded and with an EBCDIC null character added at the end, into the location to which the pointer expression points, and the program continues execution with the statement following the ACCEPT statement.

If the parameter to the ACCEPT statement is a string variable or subscripted string variable, then the contents of the specified string are displayed on the ODT. The string variable or subscripted string variable must be of type EBCDIC. Following the display of the characters, the program is suspended until a response is entered at the ODT. The response is placed in the string variable or subscripted string variable, and the program continues execution with the statement following the ACCEPT statement.

The ACCEPT statement can be used as a Boolean function. If a response is not available, the value of the ACCEPT statement is FALSE. If a response is available, the value of the ACCEPT statement is TRUE, and the response is placed in the specified location. The program continues execution regardless of the value returned by the ACCEPT statement.

No more than 430 characters can be displayed by the ACCEPT statement. No more than 960 characters can be accepted as a response.

The response to the ACCEPT statement can be entered before the actual execution of that statement. The response can be entered using the AX (Accept) system command. For more information, refer to the *System Commands Operations Reference Manual*.

Examples of ACCEPT Statements

The following example displays the string of EBCDIC characters in the array Z, from the beginning of the array to the EBCDIC null character (48"00).

```
ACCEPT(POINTER(Z,8))
```

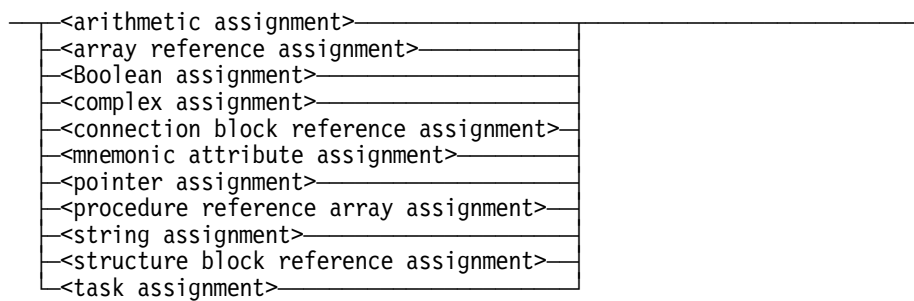
The following example displays the contents of string STR on the ODT. If a response is available, the string "THANK YOU." is displayed. If no response is available, the string "PLEASE REENTER." is displayed.

```
IF ACCEPT(STR) THEN  
  DISPLAY("THANK YOU.")  
ELSE  
  DISPLAY("PLEASE REENTER.")
```

ASSIGNMENT Statement

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

<assignment statement>



The action of an ASSIGNMENT statement is as follows:

- The location of the target is determined.
- The item following the assignment operator (:=) is evaluated.
- The resulting value is assigned to the target.

The various forms of the ASSIGNMENT statement are called assignments instead of statements because they can appear both as statements and in expressions. For example, the following is a statement when it stands alone:

A := A + 1

However, the same construct can be used in an expression, such as in the following:

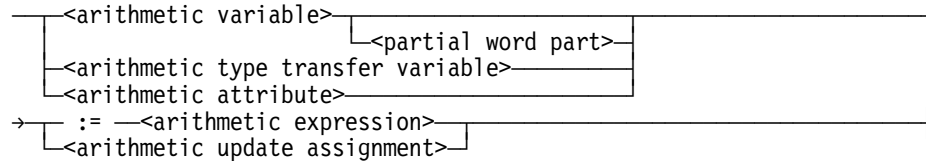
IF (A := A + 1) > 100 THEN <statement>

Too many arithmetic, Boolean, complex, pointer, or string assignments in one statement can cause a stack overflow fault in the compiler. The fault can be avoided by breaking the statement into several separate statements, each containing fewer assignments, or by increasing the maximum stack size for the program by using the task attribute STACKLIMIT.

Arithmetic Assignment

An arithmetic assignment assigns the value of the arithmetic expression on the right side of the assignment operator (:=) to the arithmetic target on the left side.

<arithmetic assignment>



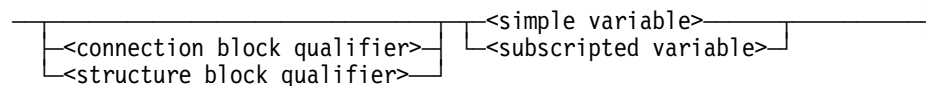
Arithmetic Variable

The attribute error number returned from the operating system can be captured in the arithmetic variable.

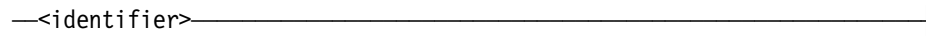
<arithmetic variable>



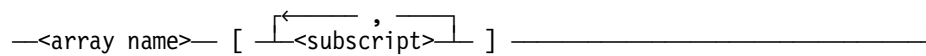
<variable>



<simple variable>



<subscripted variable>



If the <arithmetic variable> <partial word part> syntax or the <arithmetic attribute> syntax appears in a statement with multiple assignments, then it must appear as the leftmost target in the statement. The following examples illustrate this rule.

Allowed

X.[7:8] := Y := 1

F1.MAXRECSIZE:= RECLNGTH:= 30

Not Allowed

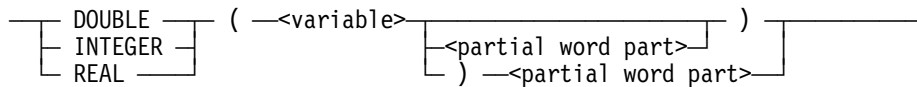
Y := X.[7:8] := 1

RECLNGTH:= F1.MAXRECSIZE:= 30

An <arithmetic variable> <partial word part> assignment leaves the remainder of the arithmetic variable unchanged, despite any possible side effects, such as embedded assignments, in the arithmetic expression.

Arithmetic Type Transfer Variable

<arithmetic type transfer variable>



If the declared type of the target item to the left of the assignment operator (:=) and the type of the value to be assigned to it are different, then the appropriate implicit type conversion is performed according to the following rules:

- If the left side is of type INTEGER and the expression value is of type REAL, then the value is rounded to an integer before it is stored.
- If the left side is of type INTEGER and the expression value is of type DOUBLE, then the value is rounded to a single-precision integer before it is stored.
- If the left side is of type REAL and the expression value is of type INTEGER, then the value is stored unchanged.
- If the left side is of type REAL and the expression value is of type DOUBLE, then the value is rounded to single-precision before it is stored.
- If the left side is of type DOUBLE and the expression value is of type INTEGER or REAL, then the value is converted to double-precision by appending a second word of zero (all bits equal to zero) before it is stored.

The use of an arithmetic type transfer variable causes the value on the right side of the assignment operator to be stored unchanged into the variable on the left side, regardless of type. However, if an attempt is made to assign a double-precision value into a single-precision variable by using the DOUBLE form of the construct, only the first word of the double-precision value is stored unchanged into the single-precision variable. For more information, see "Type Coercion of One-Word and Two-Word Operands" in Appendix C, "Data Representation."

If more than one assignment operator appears in a single assignment (for example, *A := B := C := 1.414*), assignment of values is executed from right to left. If, during this process, a value is converted to another type so that it can be assigned, then it remains in that converted form following that assignment; that is, the value does not resume its original form. For example, assume you are executing the following program:

```
BEGIN
  DOUBLE DBL1, DBL2;
  REAL REL1, REL2;
  INTEGER INT1;
  DBL2 := REL2 := INT1 := REL1 := DBL1 := 1.414213562373095048801@@0;
END.
```

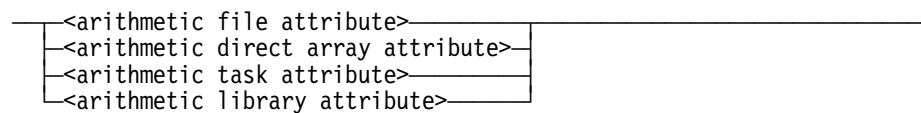
In this program, the variables are assigned the following values:

```

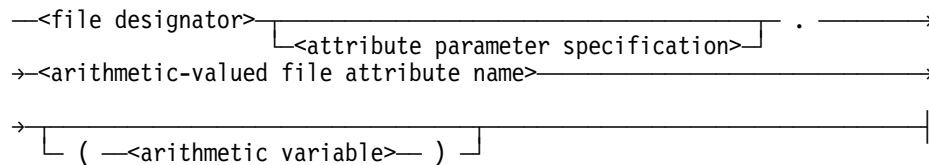
DBL1 = 1.414213562373095048801
REL1 = 1.41421356237
INT1 = 1
REL2 = 1.0
DBL2 = 1.0
    
```

Arithmetic Attribute

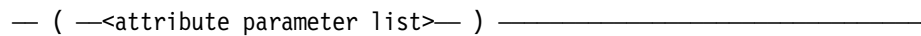
<arithmetic attribute>



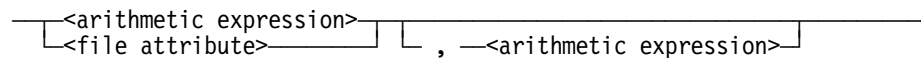
<arithmetic file attribute>



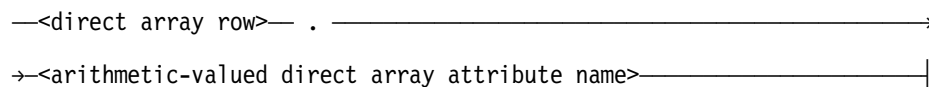
<attribute parameter specification>



<attribute parameter list>



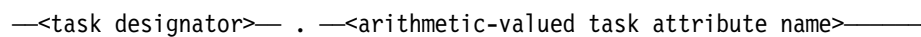
<arithmetic direct array attribute>



<arithmetic-valued direct array attribute name>

ALGOL supports all direct array attributes and direct array attribute values described in the *File Attributes Programming Reference Manual*.

<arithmetic task attribute>



<arithmetic-valued task attribute name>

ALGOL supports all task attributes of type real and integer described in the *Task Attributes Programming Reference Manual*.

ASSIGNMENT Statement

<arithmetic library attribute>

—<library attribute designator>— . —————→
→<arithmetic-valued library attribute name>—————|

<library attribute designator>

—<library identifier>—————|
| —<thiscl intrinsic>—————|
| LIBRARY — (—<connection lib id>————— [—<arith expression>—]) —|

Arithmetic Update Assignment

<arithmetic update assignment>

—<update symbols>— [—<arithmetic operator>—<arithmetic expression>—] —|

<update symbols>

— := — * —————|

The arithmetic update assignment is a shorthand form of assignment that can be used when the arithmetic target on the left side of the assignment operator also appears in the arithmetic expression on the right side of the operator. The arithmetic update assignment form can be specified only following an arithmetic target that does not contain a partial word part. The asterisk (*) represents a duplication of the item to the left of the assignment operator. For example, the same results are produced by the following two assignments:

A := * + 1

A := A + 1

The target item is not reevaluated at the appearance of the asterisk. Hence, if I equals zero initially, the following applies:

Assignment

B[I := I + 1] := * + 1

Equivalent

B[1] := B[1] + 1

Not Equivalent

B[1] := B[2] + 1

If the item to the left of the assignment operator is a subscripted variable, it cannot reference a value array.

If an expression is used as a subscript to a variable, the subscript is evaluated first. In the following example, the expression used as a subscript [I+2] is evaluated first:

A[I+2] := I := 10

Examples of an Arithmetic Assignment

```

VAL := 7

A[4,5].[30:4] := X

FYLE.AREAS := 50

FYLE(5).AREAS := 10

FYLE.SYNCHRONIZE := VALUE(OUT)

DIRARRAY.IOCW := 4"1030"

TSK.COREESTIMATE := 10000

NEWARRAY[I] := * + OLDARRAY[I]

ONE := SIN(X := 3)**2 + COS(X)**2

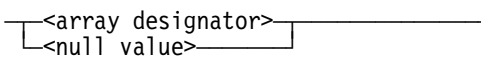
DISTANCE := SQRT(X**2 + Y**2 + 2**2)

CONNECTION_LIBRARY_INT.R := CONNECTION_LIBRARY_EXT[4].T;

```

Array Reference Assignment

<array reference assignment>

—<array reference variable>— := 

An array reference assignment associates a variable, called an array reference variable, with an array or a portion of an array. The array reference variable can then be used to reference the array or array portion.

An array reference assignment generates a copy descriptor of an array or array row.

Typical uses of an array reference assignment include the following:

- To perform more efficiently arithmetic operations on multidimensional arrays (for example, by extracting a particular row to avoid repeated indexing to the same row)
- For concurrent, but different, uses of the same array (for example, for storing values of type REAL into an array that is originally declared as Boolean)

Array Reference Variable

<array reference variable>

—<array reference identifier>—————|

The array reference variable cannot be global to the array designator.

If the array reference variable is declared as DIRECT, then only an array designator for a direct array can be assigned to it. However, a nondirect array reference variable can be assigned an array designator for either a direct or a nondirect array.

The dimensionality of the array reference variable and the array designator must be the same. If both are multidimensional, then the array classes must be compatible. INTEGER, REAL, and BOOLEAN types are compatible with each other. Other array classes are compatible only with themselves. If the array reference identifier and the array designator are both one-dimensional, then they can have any array class.

The size of each dimension of a multidimensional array reference variable is the same as the size of the corresponding dimension of the array designator. The size of a one-dimensional array reference variable is determined by the size and element width of the array designator and the element width for the array class with which the array reference variable was declared. Let S_a and W_a be the size and element width, respectively, of the array designator, and let W_r be the element width for the array reference variable. The size of the array reference variable, S_r , is then the following:

$$S_r := (S_a * W_a) \text{ DIV } W_r$$

Because of the truncation implicit in the DIV operation, $S_r * W_r$ might be less than $S_a * W_a$. In this case, indexing the array reference variable by $S+LB$, where LB is the lower bound in the ARRAY REFERENCE declaration, causes an invalid index fault. Nevertheless, pointer operations using the array reference variable can access the entire area of memory allocated to the original array to which the array designator ultimately refers; the memory area may hold more than S_r elements of width W_r .

Array Designator

<array designator>

—<array name>—————|
 └─<subarray selector>┘

<subarray selector>

— [————— |
 └─<subscript>— , ┘
 └─ * ┘

The array designator indicates the array or array portion to be associated with the array reference variable. Following an array reference assignment, the array reference variable becomes a referred array, describing the same data as the array designator, which can itself be an original array or another referred array.

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

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

For example, assume you are using the following declarations:

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

For these declarations, the following applies:

A and A[*,*,*]	Denote the entire three-dimensional array.
A[I,*,*]	Denotes one of the 10 two-dimensional arrays that constitute A.
A[I,J,*]	Denotes one of the 40 one-dimensional arrays (array rows) that constitute A[I,*,*], and one of the 400 one-dimensional arrays that constitute A.

If the array designator is an uninitialized array reference variable, the array reference assignment causes the target array reference variable to become uninitialized.

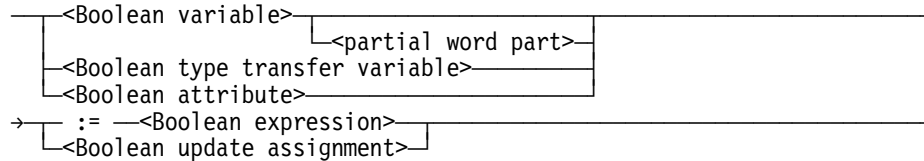
Examples of Array Reference Assignments

```
BOOLARRAY := REELARRAY  
EBCDICARAY := INPUTARAY[*]  
SUBARRAY :=BIGARRAY[N,*,*]  
ARAYROW := MULTIDIMARAY[I,J,K,*]
```

Boolean Assignment

A Boolean assignment assigns the value of the Boolean expression on the right side of the assignment operator (:=) to the Boolean target on the left side.

<Boolean assignment>

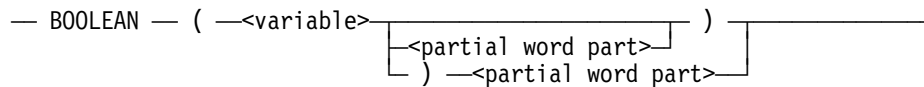


Boolean Variables

<Boolean variable>

A <variable> of type Boolean.

<Boolean type transfer variable>



If the <Boolean variable> <partial word part> syntax or the <Boolean attribute> syntax appears in a statement with multiple assignments, then it must appear as the leftmost target in the statement. The following examples illustrate this rule.

Allowed

X.[7:8] := Y := FALSE

F1.OPEN := OPENED := FALSE

Not Allowed

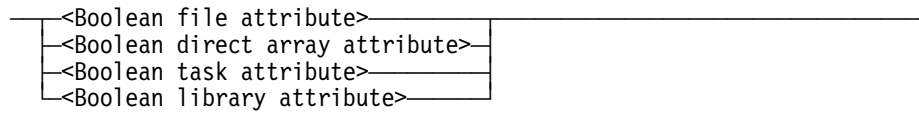
Y := X.[7:8] := FALSE

OPENED := F1.OPEN := FALSE

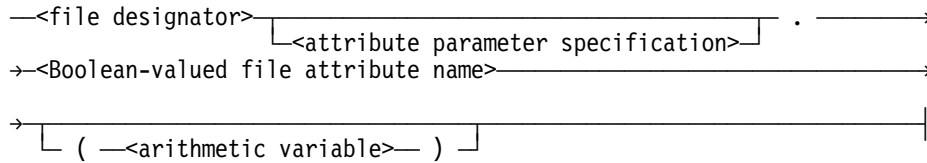
A <Boolean variable> <partial word part> assignment leaves the remainder of the Boolean variable unchanged, despite any possible side effects, such as embedded assignments, in the Boolean expression.

Boolean Attributes

<Boolean attribute>

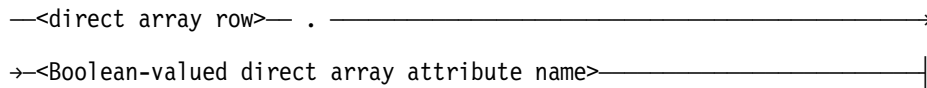


<Boolean file attribute>



The attribute error number returned from the operating system can be captured in the <arithmetic variable>.

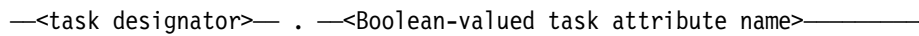
<Boolean direct array attribute>



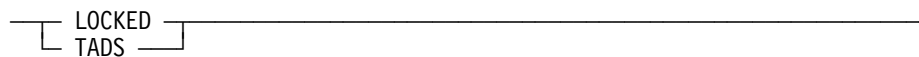
<Boolean-valued direct array attribute name>

ALGOL supports all direct array attributes and direct array attribute values described in the *File Attributes Programming Reference Manual*.

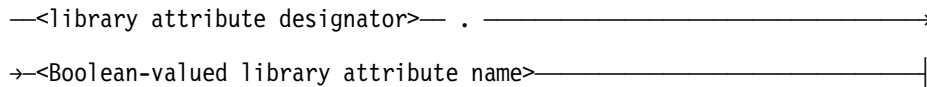
<Boolean task attribute>



<Boolean-valued task attribute name>



<Boolean library attribute>



Boolean Update Assignment

<Boolean update assignment>

—<update symbols>—┌──┐
└──────────<Boolean operator>──────────<simple Boolean expression>──────────┘

The Boolean update assignment is a shorthand form of assignment that can be used when the Boolean target on the left side of the assignment operator (:=) also appears in the Boolean expression on the right side of the operator. The Boolean update assignment form can be specified only following a Boolean target that does not contain a partial word part. The asterisk (*) represents a duplication of the item to the left of the assignment operator. For example, the following two assignments produce the same results:

```
B := * AND BOOL
```

```
B := B AND BOOL
```

The target item is not reevaluated at the appearance of the asterisk.

If the item to the left of the assignment operator is a subscripted variable, it cannot reference a value array.

Examples of Boolean Assignments

```
BOOL := TRUE
```

```
BOOLARRAY[N].[30:1] := Q < VAL
```

```
HIGHER := PTR > PTS FOR 6
```

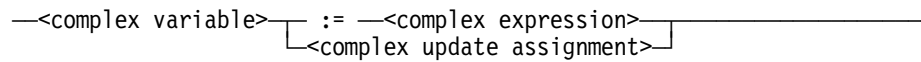
```
TAUTOLOGY := * OR TRUE
```

```
STRUCTURE_BLOCK_VARIABLE.B := STRUCTURE_BLOCK_ARRAY[2].B;
```

Complex Assignment

A complex assignment assigns the value of the complex expression on the right side of the assignment operator (:=) to the complex variable on the left side.

<complex assignment>

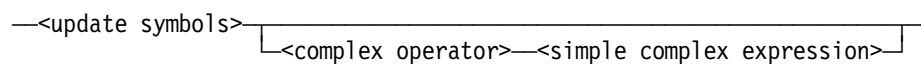


<complex variable>



Complex Update Assignment

<complex update assignment>



The complex update assignment is a shorthand form of assignment that can be used when the complex variable on the left side of the assignment operator (:=) also appears in the complex expression on the right side of the operator. The asterisk (*) represents a duplication of the variable to the left of the assignment operator. For example, the following two assignments produce the same results:

`C := * + COMPLEX(3,4)`

`C := C + COMPLEX(3,4)`

The target variable is not reevaluated at the appearance of the asterisk.

If the item to the left of the assignment operator is a subscripted variable, it cannot reference a value array.

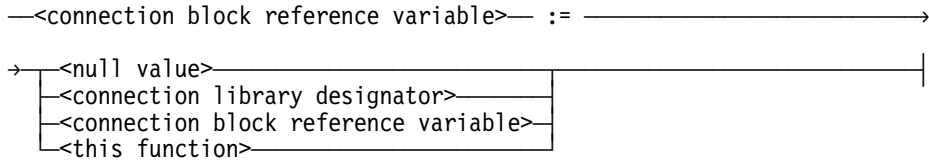
Examples of Complex Assignments

`C1 := COMPLEX(8,1.5)`

`C2 := * + C1/2`

Connection Block Reference Assignment

<connection block reference assignment>



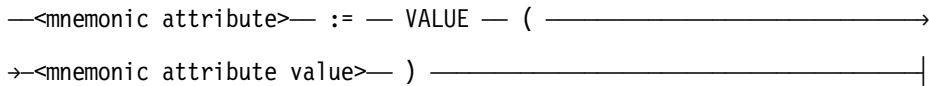
A connection block reference assignment associates a connection block reference variable with a particular connection block instance.

Mnemonic Attribute Assignment

A mnemonic attribute assignment assigns a value to the mnemonic-valued library attribute LIBACCESS.

Refer to “Library Attributes” in Section 8, “Library Facility,” for a description of the library attribute LIBACCESS.

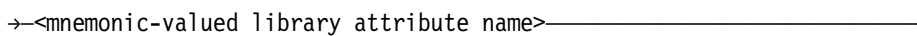
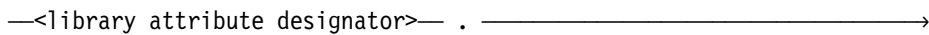
<mnemonic attribute assignment>



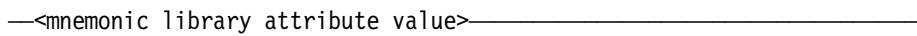
<mnemonic attribute>



<mnemonic library attribute>



<mnemonic attribute value>



The following are examples of Mnemonic Attribute Assignment:

L.LIBACCESS := VALUE(BYTITLE)

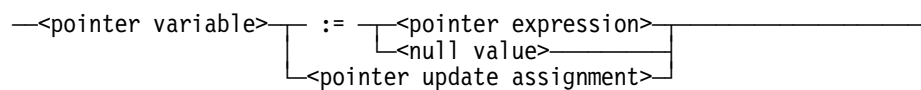
L.LIBACCESS :=VALUE(BYFUNCTION)

Pointer Assignment

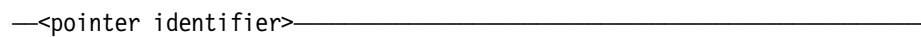
A pointer assignment assigns the pointer on the left side of the assignment operator (:=) to point to the location in an array indicated by the expression on the right side of the assignment operator. Such a pointer is then considered initialized and can be used in the REPLACE and SCAN statements for character manipulation.

A pointer cannot be assigned to an array that is at a higher lex level than the pointer. This would create an up level pointer condition, a situation in which the pointer might reference deallocated memory when the program exits from the higher lex level.

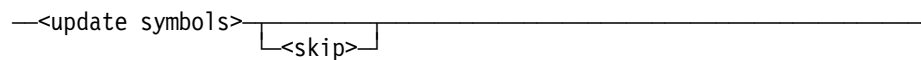
<pointer assignment>



<pointer variable>



<pointer update assignment>



Pointer Variable

A pointer assignment causes the creation of a pointer variable, or copy descriptor, to an array. The pointer variable can be set up with the needed character size by using the POINTER function syntax. For more information, see "POINTER Function" in Section 5, "Expressions and Functions."

Examples of Pointer Assignments

The following example assigns a pointer named PTS to point to the EBCDIC character in the EBCDIC array EBCDICARAY identified by the subscripted variable EBCDICARAY[5].

```
PTS := EBCDICARAY[5]
```

The following example assigns a pointer named PTR to point to the leftmost character position in the first element of the real array REALARAY.

```
PTR := POINTER(REALARAY)
```

The following example assigns the pointer PINFO to point to the seventeenth character position after the character position pointed to by the pointer PTR.

```
PINFO := PTR + 17
```

ASSIGNMENT Statement

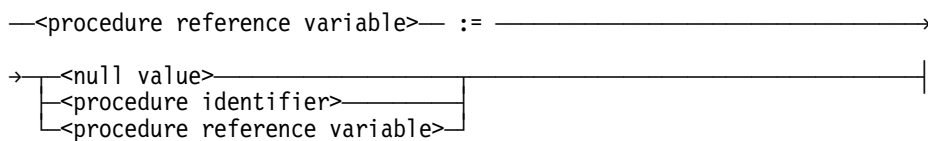
The following example assigns the pointer POUT to point to the leftmost character position in the array element identified by INSTUFF[N]. The 4 following the comma indicates that POUT is a hexadecimal pointer and thus points to hexadecimal characters.

```
POUT := POINTER(INSTUFF[N],4)
```

Procedure Reference Assignment

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

<procedure reference assignment>



Procedure Reference Variable

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

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

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

If the procedure reference variable on the right side of the assignment operator is uninitialized or has been assigned NULL, an attempt to invoke the procedure reference variable that was on the left side results in a run-time error.

Example of a Procedure Reference Assignment

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

```

BEGIN
  REAL PROCEDURE P;
  BEGIN
    REAL A;
    A := T * T;
    P := A;
  END;
  REAL PROCEDURE Q;
  BEGIN
    INTEGER A;
    A := T * T * T;
    IF A > 0 THEN
      Q := A
    ELSE
      Q := -1;
    END;
  END;

  REAL PROCEDURE REFERENCE ARRAY RA[1:10]; NULL;
  REAL PROCEDURE REFERENCE PREF1; NULL;

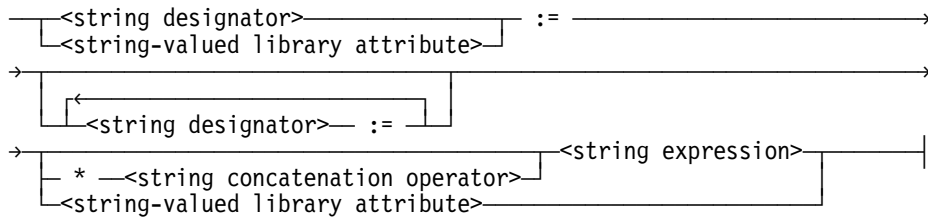
  RA[2] := Q;    %RA[2] CONTAINS A REFERENCE TO PROCEDURE Q
  RA[3] := NULL; %RA[3] CONTAINS A NULL VALUE
  RA[4] := RA[3]; %RA[4] CONTAINS A NULL VALUE
  PREF1 := P;    %PREF1 CONTAINS A REFERENCE TO PROCEDURE P
  PREF1 := NULL; %PREF1 CONTAINS A NULL VALUE
END.

```

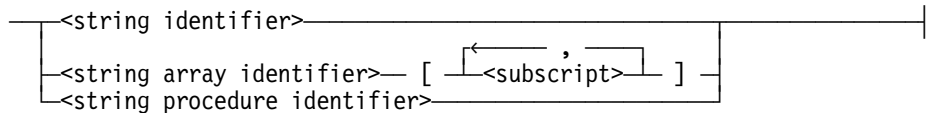
String Assignment

A string assignment assigns the string that results from evaluation of the string expression on the right side of the assignment operator (:=) to the string target on the left side.

<string assignment>



<string designator>



The result of the expression on the right side of the assignment operator (:=) must be a string of the same character type as the declared type of the string designator on the left side.

Embedded assignment is not allowed. For example, the following is not allowed:

```
S1 := DROP(S2 := "ABC", 2)
```

Assignment can be made to a string procedure identifier only within the body of that string procedure.

String Concatenation Operator

The * *<string concatenation operator>* form is a shorthand form of assignment that can be used when the string designator on the left side of the assignment operator also appears in the expression on the right side of the operator. The asterisk (*) represents a duplication of the item to the left of the assignment operator. For example, the following two assignments produce the same results:

```
S := * CAT "ABC"
```

```
S := S CAT "ABC"
```


Examples of String Assignments

The following example assigns the EBCDIC string ABCD123 to the string variable STR1.

```
STR1 := 8"ABCD123"
```

The following example assigns the string 1234 (of the default character type) to both of the string variables S2 and S1.

```
S1 := S2 := "1234"
```

The following example concatenates the string INPUT onto the end of the string stored in SOUT1, and then assigns the result to both of the string variables SOUT1 and SOUT.

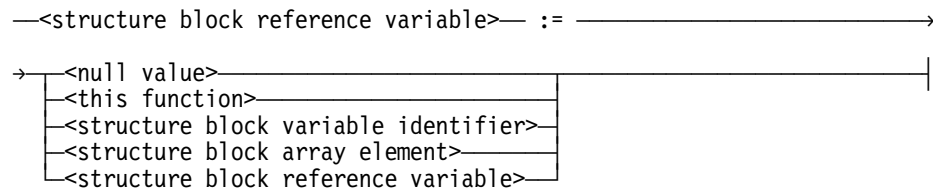
```
SOUT := SOUT1 := * CAT "INPUT"
```

The following example concatenates the string ABC onto the end of the string stored in SOUT, and this string is then concatenated onto the end of the string stored in SOUT. The resulting string is assigned to the string variable SOUT.

```
SOUT := * || SOUT || "ABC"
```

Structure Block Reference Assignment

<structure block reference assignment>



A structure block reference assignment associates a structure block reference variable with a particular structure block instance.

Task Assignment

A task assignment associates the task designator on the right side of the assignment operator (:=) with the task indicated by the expression on the left side.

<task assignment>

—<task designator>— . —<task-valued task attribute name>— := —————
→<task designator>—————|

For information on task designator and task-valued task attribute name, see “TASK and TASK ARRAY Declarations” in Section 3, “Declarations.”

The PARTNER task attribute is used in conjunction with the CONTINUE statement.

The following are examples of Task Assignment:

The following example assigns the task TASKIT to the EXCEPTIONTASK attribute of TISKIT.

```
TISKIT.EXCEPTIONTASK := TASKIT
```

The following example assigns the task identified by the task array element TASKARRAY[N] to the EXCEPTIONTASK attribute of TSK.

```
TSK.EXCEPTIONTASK := TASKARRAY[N]
```

The following example assigns the task COHORT to the PARTNER attribute of TASKVARB.

```
TASKVARB.PARTNER := COHORT
```

The following example assigns the task identified by the task array element COWORKERS[INDX] to the PARTNER attribute of MYSELF.

```
MYSELF.PARTNER := COWORKERS[INDX]
```

The following example assigns the task that is the PARTNER attribute of the task MYSELF.PARTNER to the task that is the EXCEPTIONTASK attribute of the task MYSELF.PARTNER.

```
MYSELF.PARTNER.EXCEPTIONTASK := MYSELF.PARTNER.PARTNER
```

ATTACH Statement

The ATTACH statement associates an interrupt with an event so that when the event is caused, the program is interrupted, and the interrupt code is placed in execution, provided that the interrupt is enabled.

<attach statement>

```
— ATTACH —<interrupt identifier>— TO —<event designator>—————|
```

Attachment of Interrupts

Although different interrupts can be simultaneously attached to the same event, a particular interrupt can be attached to only a single event at any one time. For this reason, if, at attach time, the interrupt is found to be already attached to an event, then it is automatically detached from the old event and attached to the new event. Any pending invocations of the interrupt are lost.

An interrupt can be attached to an event that is declared in a different block. For example, a local interrupt can be attached to a formal event. Such an attachment can cause compile-time or run-time up-level attach errors if the block containing the event can be exited before the block that contains the interrupt is exited.

Event-valued file attributes are allowed. If the file is declared (or specified as a formal parameter) at least as global as the interrupt, then run-time checking can be bypassed.

However, the operating system can prevent some attachments at run time. For example, the INPUTEVENT of a remote file is available only after the file has been opened with an OPEN statement. In the operating system, run-time verification that an interrupt is not declared more global than the event always fails for attribute events. This causes a task fatal UP LEVEL ATTACH error. Therefore, a formal parameter event whose actual parameter is a file attribute cannot be attached, nor can an attribute of a formal parameter file be attached, to a global interrupt.

Imported events and event arrays cannot be used as <event designator>s in an ATTACH statement.

Examples of ATTACH Statements

The following example attaches the interrupt THEPHONE to the event THEBELL. When THEBELL is caused, the code associated with THEPHONE begins executing.

```
ATTACH THEPHONE TO THEBELL
```

The following example attaches the interrupt ANSWERHI to the event MYSELF.EXCEPTIONEVENT. Whenever the task MYSELF undergoes a change in status, the EXCEPTIONEVENT attribute is caused, and the code associated with ANSWERHI begins executing.

```
ATTACH ANSWERHI TO MYSELF.EXCEPTIONEVENT
```

AWAITOPEN Statement

The AWAITOPEN statement is used to await a request for dialog establishment. Information on networks that support this function can be found in the *I/O Subsystem Programming Guide*.

<awaitopen statement>

— AWAITOPEN — (—<awaitopen file part> — [—<awaitopen options>]) —

<awaitopen file part>

—<file designator> — [[SUBFILE —<subfile index>—]] —

<awaitopen options>

— [—<awaitopen control option> —] —
 → [— PARTICIPATE —] —
 = [TRUE] —
 [FALSE] —
 → [—<connecttimelimit option> —] —

<awaitopen control option>

— [AVAILABLE] —
 — [DONTWAIT] —
 — [WAIT] —

<connecttimelimit option>

— CONNECTTIMELIMIT — = —<arithmetic expression> —

The AWAITOPEN statement can be used only when the kind of the file designator is PORT and only when the SERVICE attribute for the port is set to a network type that supports this feature.

The subfile index, if present, specifies the subfile that is waiting for dialog establishment.

The AWAITOPEN statement can be used as an arithmetic function. It returns the same values as the file attribute AVAILABLE. For a description of these values refer to the *File Attributes Programming Reference Manual*. If the result of this statement is not interrogated by the program, the program terminates when the awaitopen action fails.

The control options AVAILABLE, DONTWAIT, and WAIT are described in the *I/O Subsystem Programming Guide*. The control option is used to indicate when control should be returned to the program. If a control option is not specified, WAIT is assumed.

PARTICIPATE Option

The PARTICIPATE option is used to indicate that the program specifies the option of accepting or rejecting offers, through the RESPOND statement, when the subfile is matched to an incoming dialog request. Specifying PARTICIPATE is equivalent to specifying PARTICIPATE = TRUE. Upon notification of a matching dialog request, through CHANGEVENT and FILESTATE attributes, the program can interrogate the value of attributes, read open user data, and negotiate the value of negotiable attributes. The program can then reject or accept an incoming dialog request. For more information on FILESTATE and CHANGEVENT attributes, see the *File Attributes Programming Reference Manual*. If the PARTICIPATE option is not included, a default value of FALSE is assumed, and any offer is unconditionally accepted.

CONNECTTIMELIMIT Option

The CONNECTTIMELIMIT option can be used to specify the maximum amount of time, in minutes, that the system will allow for a successful match with a corresponding endpoint. The default for this option is an unlimited wait. If the amount specified is negative, an error result is returned. If the value is zero, there is no time limit on the wait. If the value is not a single-precision integer, it is integerized. If the FILESTATE of the port file does not change to an OPENED or OPENRESPONSEPLEASE file state within the time specified, the AWAITOPEN fails and an implicit CLOSE ABORT is performed on the subfile.

Examples of AWAITOPEN Statements

The following example indicates that the program is being used to await dialog establishment on all subfiles of the port file FILEID.

```
AWAITOPEN (FILEID [SUBFILE 0])
```

The following example indicates that the program is being used to await dialog establishment on subfile 1 of the port file FILEID.

```
AWAITOPEN (FILEID [SUBFILE 1])
```

The following example indicates that the program is being used to await dialog establishment on subfile 1 of port file FILEID. Control is not returned to the program until the subfile is matched. In addition, the participate option is used to indicate that the program can accept or reject any offers through the RESPOND statement.

```
AWAITOPEN (FILEID [SUBFILE 1], WAIT, PARTICIPATE=TRUE)
```

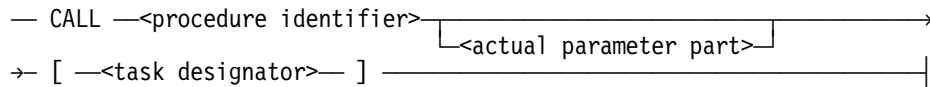
The following example indicates that the program is being used to await dialog establishment on subfile N of port file FILEID. The maximum amount of time to wait for a successful match is set to the result of the arithmetic expression $(X * 60 + 3)$.

```
AWAITOPEN (FILEID [SUBFILE N], CONNECTTIMELIMIT=(X * 60 + 3))
```

CALL Statement

The CALL statement initiates a procedure as a coroutine.

<call statement>



Coroutines

Initiation of a coroutine consists of setting up a separate stack, passing any parameters (call-by-name or call-by-value), and beginning the execution of the procedure.

Processing of the initiating program, called the *initiator* or the *primary coroutine*, is suspended.

If the called procedure, referred to as the *secondary coroutine*, is a typed procedure, the return value is discarded. If the procedure identifier is a system supplied process, such as an intrinsic, the library GENERALSUPPORT must be declared using a library entry point specification. The procedure identifier must be declared in the program or the syntax error, *PROCEDURE MUST BE USER DECLARED*, results. The actual parameter part must agree in number and type with the formal parameter part in the declaration of the procedure; otherwise, a run-time error occurs.

The task designator associates a task with the coroutine at initiation; the values of the task attributes of that task, such as COREESTIMATE, STACKSIZE, and DECLARED PRIORITY, can be used to control the execution of the coroutine. For more information about assigning values to task attributes, refer to <arithmetic task attribute> under "Arithmetic Assignment," <Boolean task attribute> under "Boolean Assignment," and "Task Assignment" earlier in this section.

Every coroutine has a partner task to which control can be passed by using the CONTINUE statement. The partner task of the secondary coroutine is the initiator by default but can be changed by assignment to the task-valued task attribute PARTNER of the task designator. Local variables and call-by-value parameters of the secondary coroutine retain their values as control is passed to or from the coroutine.

The *critical block*, described in "PROCESS Statement" later in this section, in the initiator cannot be exited until the secondary coroutine is terminated. Any attempt by the initiator to exit that block before the secondary coroutine is terminated causes the initiator and all tasks it has initiated through CALL or PROCESS statements to be terminated.

A secondary coroutine is terminated by exiting its own outermost block or by execution in the initiator of the following statement, where the task designator specifies the task associated with the secondary coroutine to be terminated:

```
<task designator>.STATUS:= VALUE(TERMINATED)
```

Note: *The CALL statement causes the initiation of a separate stack as a coroutine. Because of the cost involved, a coroutine should be established once and then used through CONTINUE statements. If a CALL statement is used to invoke a procedure, overall system efficiency is severely degraded. A string expression cannot be passed as an actual parameter to a call-by-name parameter of a procedure in a CALL statement.*

Example of a CALL Statement

The following example initiates as a coroutine the procedure COROOTEEN, and passes the parameters X, Y, 7, and X + Y + Z. COROOTEEN has the task designator T associated with it.

```
CALL COROOTEEN(X, Y, 7, X + Y + Z) [T]
```

CANCEL Statement

The CANCEL statement can be used to delink a library from a program and cause the library program to thaw (or unfreeze) and resume running as a regular program.

<cancel statement>

```
— CANCEL — ( —<library identifier>————— ) —————|
                |—————<connection library instance designator>—————|
                |—————<this intrinsic>—————|
                |—————<connection parameter>—————|
```

Delinking a Library from a Program

Normally, a library is linked to a program when the program calls one of the library entry points or the LINKLIBRARY intrinsic, and the library is delinked from the program when the block in which the library is declared is exited. The CANCEL statement can be used to delink a library before it would normally be delinked.

When a library is canceled, all users of the library are delinked from the library, and the library thaws and resumes running as a regular program regardless of whether it is temporary or permanent. Refer to “FREEZE Statement” later in this section for a discussion of temporary and permanent libraries.

After a program has canceled a library, the program can again link to a new instance of the library as if for the first time.

Only libraries whose SHARING compiler control option is specified as PRIVATE or SHARED BY RUNUNIT can be canceled. If an attempt is made to cancel a library that is not PRIVATE or SHARED BY RUNUNIT, a run-time message is given and the library is delinked as if DELINKLIBRARY was called.

To delink a program from a library without affecting any other users of the library, use the DELINKLIBRARY function. For more information, see “DELINKLIBRARY Function” in Section 5, “Expressions and Functions.”

For more information on libraries, refer to Section 8, “Library Facility.”

Example of a CANCEL Statement

The following example delinks the library LIB from the program.

```
CANCEL (LIB)
```


CASE Statement

The CASE statement provides a means of dynamically selecting one of many alternative statements.

<case statement>

—<case head>—<case body>—

<case head>

— CASE —<arithmetic expression>— OF —

<case body>

— BEGIN —<statement list>— END —
 └<numbered statement list>┘

<numbered statement list>

└<numbered statement group>┘ ; └<numbered statement group>┘

<numbered statement group>

—<number list>—<statement list>—

<number list>

└<constant arithmetic expression>┘ : └<constant arithmetic expression>┘
 └ELSE └<constant arithmetic expression>┘┘

Unnumbered Statement List

If the case body contains an unnumbered statement list, then the statement to be executed is selected in the following manner:

- The arithmetic expression in the case head is evaluated. If the resulting value is not an integer, it is integerized by rounding.
- The integer value is used as an index into the list of statements in the case body. The N statements in the case body are numbered 0 to N-1. The statement corresponding to the index value is the statement executed. If the index value is less than zero or greater than N-1, the program is discontinued with a fault.

Numbered Statement List

If the case body contains a numbered statement list, then the statement list to be executed is selected in the following manner:

- The arithmetic expression in the case head is evaluated. If the resulting value is not an integer, it is integerized by rounding.
- If the integer value is equal to one of the statement numbers, the statement list associated with the number is executed.

If the integer value is not equal to any of the statement numbers, then an invalid index fault occurs unless the word ELSE appears in a number list in the CASE statement, in which case control is transferred to the statement list following ELSE.

The statement numbers given by the constant arithmetic expressions in the number list must lie in the range 0 to 1023, inclusive. The word ELSE can appear only once in a CASE statement.

Examples of CASE Statements

```
CASE I OF
  BEGIN
    J := 1;  % STATEMENT 0
    J := 20; % STATEMENT 1
    BEGIN   % STATEMENT 2
      J := 3;
      K := 0;
    END;
    J := 4;  % STATEMENT 3
  END;
```

```
CASE I OF
  BEGIN
    1:
    2:
    5:
    7:
      J := 3;
      Q := J-1;
    3:
    4:
    20:
      J := 4;
    ELSE:
      GO TO BADCASEVALUE;
  END;
```

CAUSE Statement

The CAUSE statement activates all tasks that are waiting on the specified event.

<cause statement>

— CAUSE — (—<event designator>—) —————|

Causes of Events

Normally, the CAUSE statement also sets the happened state of the event to TRUE (happened). For an explanation of exceptions to this condition, see “WAITANDRESET Statement” later in this section.

If an enabled interrupt is attached to the event, each cause of the event results in one execution of the interrupt code.

Activating a task does not necessarily place the task into immediate execution. Activating a task consists of delinking the task from an event queue (each event has its own queue) and linking that task in priority order into a system queue called the ready queue.

The ready queue is a queue of all tasks that are capable of running. Tasks are taken out of the ready queue either when a processor is assigned to the task or when the task must wait for an operation (such as an I/O operation) to complete or for an event to be caused. A task is placed in execution only when it is the top item in the ready queue and a processor is available.

When a program causes a happened event, the CAUSE statement is ignored (a *no-op* is caused); the system does not remember every cause unless an interrupt is attached to the event. For more information on events, see “EVENT Statement” later in this section.

Examples of CAUSE Statements

The following example activates the tasks waiting for the event EVNT.

```
CAUSE (EVNT)
```

The following example activates the tasks waiting for the event identified by EVNTARAY[INDX].

```
CAUSE (EVNTARAY [INDX])
```

The following example activates the tasks waiting for a change in the status of the task TSK.

```
CAUSE (TSK.EXCEPTIONEVENT)
```

CAUSEANDRESET Statement

The CAUSEANDRESET statement activates all tasks that are waiting on the specified event and sets the happened state of the event to FALSE (not happened).

<causeandreset statement>

— CAUSEANDRESET — (—<event designator>—) —————|

Relationship to CAUSE Statement

This statement differs from the CAUSE statement in that the happened state of the event is set to FALSE (not happened).

For further information on the relationship between the CAUSEANDRESET statement and events see the discussion in “CAUSE Statement” earlier in this section.

Examples of CAUSEANDRESET Statements

The following example activates the tasks waiting for the event EVNT, and sets the happened state of EVNT to FALSE (not happened).

```
CAUSEANDRESET(EVNT)
```

The following example activates the tasks waiting for the event identified by EVNTARRAY[INDX], and sets the happened state of that event to FALSE (not happened).

```
CAUSEANDRESET(EVNTARRAY[INDX])
```

The following example activates the tasks waiting for a change in the status of the task TSK, and sets the happened state of TSK.EXCEPTIONEVENT to FALSE (not happened).

```
CAUSEANDRESET(TSK.EXCEPTIONEVENT)
```

CHANGEFILE Statement

The CHANGEFILE statement changes the names of files without opening them.

<change file statement>

```
— CHANGEFILE — ( —<directory element>— , —<directory element>— )
→ ) —————|
```

The CHANGEFILE statement returns a value of TRUE if an error occurs. Error numbers, stored in field [39:20] of the result, correspond to the causes of failure as follows:

Value	Meaning
10	The first directory element is in error.
20	The second directory element is in error.
30	File names have not been changed.

File names and directory names must be specified in EBCDIC and must be followed by a period. All errors in the names are detected at run time.

If a family substitution specification is in effect, the CHANGEFILE statement affects only the substitute family, not the alternate family.

If a directory name is specified as the source, the names of the files in that directory are changed according to the following rules:

- If the specified target directory is a new directory, then the names of all the files in the source directory are changed.
- If the specified target directory is not a new directory, then only files that do not have corresponding names in the target directory are changed. For example, the first column in the following table shows file names that exist before the statement *CHANGEFILE("A.", "B.")* is executed, and the second column shows the file names resulting from execution of the statement.

Existing Files	Resulting Files
A/B/C	B/B/C
A/B/D	A/B/D
A/C/C	B/C/C
B/B/D	B/B/D
B/C/D	B/C/D

Note that because the file name B/B/D already exists, the file name A/B/D is not changed.

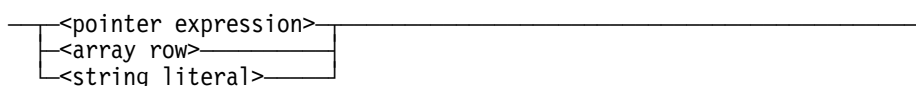
- A directory element of the form <file name>/= affects only files in that directory. It does not affect a file named <file name>.

Conflicting CHANGE requests

Two simultaneous change requests that affect the same set of files can produce unpredictable results because directory change requests are processed in groups of files. For example, two simultaneous CHANGEFILE ("A.", "B.") and CHANGEFILE ("B.", "A.") statements, where both directories A/= and B/= exist with nonconflicting file names, can result in all of the files being held in directory A/=, all of the files being held in directory B/=, or all of the files being split between the two directories.

Directory Element

<directory element>



A directory element is a file name, a directory name, or both a file name and a directory name. A directory name references a group of files. For example, the following files are all in the directory named JAMES. The first six files are in the directory named (JAMES)OBJECT, and the first five files are in the directory named (JAMES)OBJECT/TEST. Note that (JAMES)OBJECT/TEST/PRIMES is both a file name and a directory name.

```
(JAMES)OBJECT/TEST/COMM
(JAMES)OBJECT/TEST/SORT
(JAMES)OBJECT/TEST/PRIMES
(JAMES)OBJECT/TEST/PRIMES/1
(JAMES)OBJECT/TEST/PRIMES/2
(JAMES)OBJECT/LIBRARY1
(JAMES)MEMO
```

In the CHANGEFILE statement, the second directory element, the *target*, designates the name to which the first directory element, the *source*, is to be changed. If the change applies to files on pack, and a family substitution specification is not in effect (either by default through the USERDATA file or by specification in either CANDE or WFL), the target must include *ON <family name>*, and the source must not include a family name. If a family substitution specification is in effect, *ON <family name>* is not required; if *ON <family name>* does not appear, the family substitution specification is used to determine the family on which the files reside.

Example of a CHANGEFILE Statement

The following program changes A/B to C/D and then removes C/D.

```
BEGIN
  ARRAY OLD, NEW[0:44];
  BOOLEAN B;
  REPLACE POINTER(OLD) BY 8"A/B.";
  REPLACE POINTER(NEW) BY 8"C/D.";
  IF B := CHANGEFILE(OLD,NEW) THEN
    DISPLAY("CHANGEFILE ERROR");
  IF B := REMOVEFILE(8"C/D.") THEN
    DISPLAY("REMOVEFILE ERROR");
END.
```

CHECKPOINT Statement

The CHECKPOINT statement writes to a disk file the complete state of the job at a specified point. Using the disk file, the job can later be restarted from this point.

<checkpoint statement>

— CHECKPOINT — (—<device>— , —<disposition>—) —————|

<device>

— DISK —————|
— DISKPACK —|
— PACK —————|

The checkpoint/restart facility can protect a program against the disruptive effects of unexpected interruptions during the program's execution. If a halt/load or other system interruption occurs, a job is restarted either before the initiation of the task that was interrupted or, if the operator permits, at the last checkpoint, whichever is more recent. Checkpoint information can also be retained after successful runs to permit restarting jobs to correct bad data situations.

The <device> specification determines the family name of the checkpoint file. If the device is DISK, the family name is DISK. If the device is PACK or DISKPACK, the family name is PACK.

The CHECKPOINT statement can be used as a Boolean function. An attempted checkpoint returns a value with the following information:

[0: 1] = Exception bit
[10:10] = Completion code
[25:12] = Checkpoint number
[46: 1] = Restart flag (1 = restart)

In response to the request for a completion code, a program can receive a variety of messages. See "Restarting a Job" later in this section for a list of the completion codes and messages.

Disposition Option

<disposition>

— LOCK —————|
— PURGE —————|

The disposition option PURGE causes all checkpoint files to be removed at the successful termination of the job and protects the job against system failures. The LOCK option causes all checkpoint files to be saved indefinitely and can be used to restart a job even if it has terminated normally.

When a checkpoint is invoked, the following files are created:

- The checkpoint file, CP/<JN>/<CPN>, where <JN> is a four- or five-digit job number and <CPN> is a three-digit checkpoint number

If the PURGE option has been specified, the checkpoint number is always zero, and each succeeding checkpoint with PURGE removes the previous file. If the LOCK option is used, the checkpoint number starts with a value of 1 for the first checkpoint and is incremented by 1 for each succeeding checkpoint with LOCK. If the two types are mixed within a job, the LOCK checkpoints use the ascending numbers and the PURGE checkpoints use 0 (zero), leaving files 0 through N at the completion of the job.

- Temporary files, CP/<JN>/T<FN>, where <FN> is a three-digit file number beginning with 1 and incremented by 1 for each temporary disk file
- The job file, CP/<JN>/JOBFILE

This file is created under the LOCK option only.

The LOCK and PURGE options are also effective when the task terminates. If the task terminates abnormally and the last checkpoint has used the PURGE option, then the checkpoint file (numbered zero) is changed to have the next sequential checkpoint number, and the job file is created (if necessary). If the job terminates normally and only PURGE checkpoints have been taken, the CP/<JN> directory is removed.

Restarting a Job

A job can be restarted in two ways:

- After a halt/load

The system automatically attempts to restart any job that was active at the time of a halt/load. If a checkpoint has been invoked during the execution of the interrupted task, then the operator is given a message requiring a response to determine whether the job should be restarted. The operator can respond with the system command OK (to restart at the last checkpoint), DS (to prevent a restart), or QT (to prevent a restart but save the files for later restart if the job was a checkpoint with PURGE).

- By a Work Flow Language (WFL) RERUN statement

A WFL job can be restarted programmatically by use of the WFL RERUN statement.

The following conditions can inhibit a successful restart:

- An invalid usercode
- Recompilation of the program since the checkpoint
- Change of the operating system, since the checkpoint

The restart fails if the creation time stamp of the operating system that created the checkpoint file does not match the creation time stamp of the current operating system.

CHECKPOINT Statement

- Intrinsic after the checkpoint that are different from the intrinsic before the checkpoint

The messages in the following list can appear as the result of an attempt to restart.

- RESTART PENDING (RSVP)
- MISSING CHECKPOINT FILE
- IO ERROR DURING RESTART
- USERCODE NO LONGER VALID
- OPERATOR DSED RESTART
- OPERATOR QTED RESTART
- MISSING CODE FILE
- NOT ABLE TO RESTART
- INVALID JOB FILE
- RESTART AS CP/nnnn
- MISSING JOB FILE
- FILE POSITIONING ERROR
- WRONG JOB FILE
- WRONG CODE FILE
- BAD CHECKPOINT FILE
- BAD STACK NUMBER
- WRONG MCP

The following can inhibit a successful checkpoint/restart:

- Direct I/O (direct arrays or files)
- Datacomm I/O (open datacomm files)
- Open Data Management System II (DMSII) sets
- ODT files
- Output directly to a printer (backup files are acceptable)
- Checkpoints taken inside sort input or output procedures. The sort intrinsic provides its own restart capability; for more information, see "SORT Statement" later in this section.
- Checkpoints taken in a compile-and-go program

If a job that produces printer backup files is restarted, the backup files can already have been printed and removed, and on restart, the job requests the missing backup files. In this situation, when the backup files are requested, the operator must respond with the system command OF (Optional File). A new backup file is created. Output preceding the checkpoint is not re-created.

The messages in the following table can appear as the result of a checkpoint/restart. Error conditions can be handled in a program by checking for them by completion code number and instructing the program to handle the result.

Checkpoint Message	Completion Code
CHECKPOINT#nn/yy TAKEN	0
INVALID AREA IN STACK	1
SYSTEM ERROR	2
BAD IPC ENVIRONMENT	3
NO USER DISK FOR CP FILE	4
IO ERROR DURING CHECKPOINT	5
# ROWS IN CP FILE > 1024	6
DIRECT FILE NOT ALLOWED	7
TOO MANY TEMPORARY DISK FILES	8
ILLEGAL FILEKIND	9
ILLEGAL FILE ORGANIZATION	11
INSUFFICIENT MEMORY TO CHECKPOINT	12
OPEN REVERSED TAPE FILE NOT ALLOWED	13
ICM AREA IN STACK	14
DMS AREA IN STACK	15
DIRECT ARRAY IN STACK	16
SECURITY ERROR SAVING TEMPORARY DISK FILE	17
SUBSPACE IN STACK	18
STACKMARK	19
SORT AREA IN STACK	20
IN USE ROUTINE NOT ALLOWED	21
ILLEGAL CONSTRUCT	22
BDBASE ILLEGAL	23
ILLEGAL FILE STRUCTURE	24
MULTI-REEL UNLABELED TAPE NOT ALLOWED	25
SURROGATE TASK NOT ALLOWED	26
NON-EVENT PO IN STACK	27
PROGRAM USES LIBRARIES	28
JOB STACK NOT VISIBLE	29
ROW SIZE TOO SMALL FOR CP FILE	30
VISIBILITY ATTRIBUTE SHOULD BE USED	31

CHECKPOINT Statement

Checkpoint Message	Completion Code
OPERATOR CHECKPOINT REQUEST CANCELLED	32
CHECKPOINT REQUEST DENIED	33
BR REQUEST REJECTED	34
OPEN BACKUP FILE WITH PRINTDISPOSITION = EOT NOT ALLOWED	36
OPEN BACKUP FILE WITH PRINTDISPOSITION = CLOSE NOT ALLOWED	37
OPEN BACKUP FILE WITH PRINTDISPOSITION = DIRECT NOT ALLOWED	38
ERROR OCCURRED DURING CHECKPOINT IN FILE	39
ATTEMPT TO EXCEED TEMPORARY FILE LIMIT ON CP FILE	40
ATTEMPT TO EXCEED FAMILY LIMIT ON CP FILE	41
FAMILY INTEGRAL LIMIT EXCEEDED ON CP FILE	42
INVALID ENVIRONMENT IN STACK	43
DISK TYPE MUST BE DISK OR PACK	44
CHECKPOINT TYPE MUST BE ZERO OR ONE	45
SHARED BUFFERS NOT ALLOWED	46
STACK CONTAINS STRUCTURE TYPE VARIABLES	47
LIBRARY IS LINKED TO A CONNECTION LIBRARY ENVIRONMENT	48
PROGRAM USES CONNECTION LIBRARIES	49
PROGRAM USES SIGNALS	50
PROGRAM USES FILE DESCRIPTORS	51
FIFO NOT ALLOWED	52
POSIX SPECIAL FILES NOT ALLOWED	53
CHECKPOINT ABORTED: PRINTDISPOSITION OF OPEN BACKUP FILE MUST BE EOJ OR DONTPRINT	54
CHECKPOINT ABORTED: PRINTER FILES MUST BE BACKUP	55

Locking

For jobs that take a large number of checkpoints with LOCK, the checkpoint number counts up to 999 and then recycles to 1 (leaving zero undisturbed). When this recycling occurs, previous checkpoint files are lost as new ones using the same numbers are created.

If a temporary disk file is open at a checkpoint, it is locked under the CP directory. If it is subsequently locked by the program, the name is changed to the current file title. At restart time, the file is sought only under the CP directory, resulting in a no-file condition. To avoid this condition, all files that are to be locked eventually should be opened with the file attribute PROTECTION assigned the value SAVE. To remove the file, it must be closed with PURGE. True temporary files, which are never locked, do not have this problem. All data files must be on the same medium as at the checkpoint, but need not be on the same units or the same locations on disk or disk pack. They must retain the same characteristics, such as blocking. The checkpoint/restart system makes no attempt to restore the contents of a file to their state at the time of the checkpoint; the file is merely repositioned. At this time, volume numbers are not verified.

Note: *CANDE and remote job entry (RJE) cannot be used to run a program with checkpoints. The checkpoints are ignored if used.*

Rerunning Programs

If a rerun is initiated and the job number is in use by another job, a new job number is supplied, and the CP/<JN> directory node is changed to reflect the new job number.

If a rerun is initiated and the PROCESSID function is used, the value returned by the function can be different for the restarted job. Refer to "PROCESSID Function" in Section 5, "Expressions and Functions," for more information.

When a job is restarted at some checkpoint before the last, subsequent checkpoints taken from the restarted job continue in numerical sequence from the checkpoint used for the restart. Previous higher numbered checkpoints are lost.

Example of a CHECKPOINT Statement

```
BOOL := CHECKPOINT(DISK,PURGE)
```

CLOSE Statement

The CLOSE statement breaks the link between a logical file declared in the program and its associated physical file, which is the actual file data is sent to or from. For port files, it is used to close dialogs between processes.

<close statement>

— CLOSE — (—<close file part> [, —<close options>]) —————

<close file part>

—<file designator> [— SUBFILE —<subfile index>—] —
—<task designator> . —<file-valued task attribute name>—

<subfile index>

—<arithmetic expression>—————

The CLOSE statement can be used as an arithmetic function. For information about results returned, see the *File Attributes Programming Reference Manual*.

When no CLOSE option is specified, the CLOSE statement closes the file, depending on the kind of file, as follows:

Line Printer File

The printer is skipped to channel 1, an ending label is printed, and the printer is again skipped to channel 1. The file must be labeled.

Unlabeled Tape Output File

A double tape mark is written after the last block on the tape, and the tape is rewound.

Labeled Tape Output File

A tape mark is written after the last block on the tape; then an ending label is written followed by a double tape mark, and the tape is rewound.

Disk File

If the file is a temporary file, the disk space is returned to the system.

For all types of files, the I/O unit and the buffer areas are released to the system.

The <subfile index> syntax is used to specify the subfile to be closed.

CLOSE Options

<close options>



If the asterisk (*) is used and the file is a tape file, the I/O unit remains under program control, and the tape is not rewound. This construct is used to create multifile reels.

When the asterisk is used on multifile input tapes and the value of the LABEL file attribute is STANDARD, the CLOSE statement closes the file as follows:

- If the value of the DIRECTION file attribute is FORWARD, the tape is positioned forward to a point just following the ending label of the file.
- If the value of the DIRECTION file attribute is REVERSE, the tape is positioned to a point just in front of the beginning label for the file.
- If the end-of-file branch of a READ statement or WRITE statement has been taken, the CLOSE statement does not position the file.

The close action performed on a single-file reel is the same as that performed on a multifile reel. The next I/O operation performed on the file must be done in the direction opposite to that of the prior I/O operations; otherwise, an end-of-file error is returned.

When the asterisk is used and the LABEL file attribute does not have the value STANDARD, the tape is spaced beyond the tape mark (on input), or a tape mark is written going forward (on output). The essential difference is that if LABEL is OMITTEDEOF, labels are not spaced over, but if LABEL is STANDARD, labels are spaced over.

The CRUNCH option is meaningful only for disk files. It causes the unused portion of the last row of disk space, beyond the end-of-file indicator, to be returned to the system. The file cannot be expanded but can be written inside of the end-of-file limit.

If the LOCK option is used, the file is closed. If the file is a tape file, it is rewound, and a system message is displayed that notifies the operator that the reel will be saved. The tape unit is made inaccessible to the system until the operator readies it manually. If the file is a disk file, it is kept as a permanent file on disk. The file buffer areas are returned to the system.

The DOWNSIZEAREA and DOWNSIZEAREALOCK options are meaningful only for disk files. If the file uses a small portion of the first area only, both options cause the area size of the file to be reduced and the remaining portion to be returned to the system. Otherwise, the DOWNSIZEAREA option is like the REWIND option, and the DOWNSIZEAREALOCK option is like the LOCK option. The DOWNSIZEAREA and

CLOSE Statement

DOWNSIZEAREALOCK options do not return as much space as the CRUNCH option, but they do enable the file to be extended.

If the PURGE option is used, the file is closed, purged, and released to the system. If the file is a permanent disk file, it is removed from the disk directory, and the disk space is returned to the system.

If the REEL option is used, the file must be a multireel tape file. The current reel is closed, and the next reel is opened. This option is provided primarily for use with direct tape files, for which the system does not automatically perform reel switching.

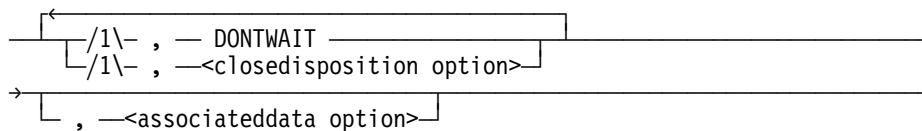
If the REWIND option is used, the file is closed. If the file is a tape file, it is rewound. For disk files, the record pointer is reset to the first record of the file. The file buffer areas are returned to the system, and the I/O unit (or disk file) remains under program control.

All forms of the CLOSE statement that are not appropriate for the type of unit assigned to the file are equivalent to using the REWIND option. For example, when the asterisk or the REEL option is specified for a disk file, the result is the same as when the REWIND option is specified for a disk file.

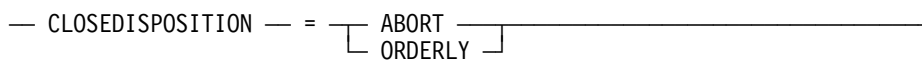
A CLOSE statement that leaves the disk file under program control is referred to as a *close with retention*. For example, a CLOSE statement that designates a disk file and the asterisk option or the REWIND option is a close with retention.

PORT CLOSE Option

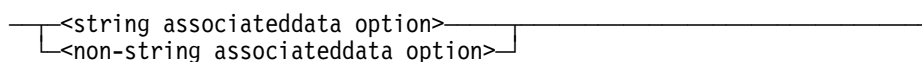
<port close option>



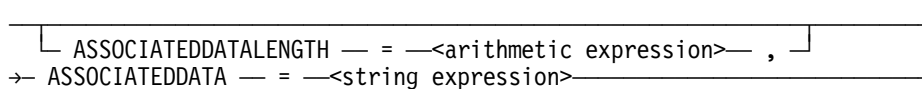
<closedisposition option>



<associateddata option>



<string associateddata option>



<non-string associateddata option>

— ASSOCIATEDDATALENGTH — = —<arithmetic expression>— , —————→
 → ASSOCIATEDDATA — = —————→
 |
 |—<array row>—|
 |—<subscripted variable>—|
 |—<pointer expression>—|

The PORT CLOSE option is meaningful only for files for which the KIND file attribute has the value PORT.

The control option DONTWAIT is used to indicate that control should be returned to the program as soon as possible. If DONTWAIT is not specified, WAIT is assumed and control is returned when processing of the CLOSE statement is complete. Refer to the *I/O Subsystem Programming Guide* for more information on control options.

The CLOSEDISPOSITION = ORDERLY option and ASSOCIATEDDATA option are meaningful only for certain port file services. The CLOSEDISPOSITION = ABORT terminates a dialog immediately. A CLOSEDISPOSITION = ORDERLY steps the subfile through an orderly termination procedure that involves handshaking between the two programs. If the file is a port file and the CLOSEDISPOSITION option is not specified, the ABORT operation is assumed. With an ABORT termination, the processes must go through their own handshake procedure to ensure no data loss. The ASSOCIATEDDATA option can be used to send associated data with the subfile close. If a string expression is specified, the length is calculated automatically and used as the ASSOCIATEDDATALENGTH value. Otherwise, the ASSOCIATEDDATALENGTH option specifies how many characters are to be sent. If the ASSOCIATEDDATA value is of type HEX, the ASSOCIATEDDATALENGTH option indicates the number of HEX characters, otherwise the number of EBCDIC characters. If the ASSOCIATEDDATALENGTH value is not a single-precision integer it is integerized.

Examples of CLOSE Statements

In the following example, if FILEID is a temporary disk file, this statement closes the file and returns the disk space to the system.

```
CLOSE(FILEID)
```

The following example closes FILEID and, assuming FILEID is a tape file, positions the tape according to the description under "CLOSE Options" earlier in this section.

```
CLOSE(FILEID,*)
```

The following example closes, purges, and releases FILEID to the system. If FILEID is a permanent disk file, it is removed from the disk directory and the disk space is returned to the system.

```
CLOSE(FILEID,PURGE)
```

CLOSE Statement

In the following example, assuming FILEID is a multireel tape file, the current reel is closed, and the next reel is opened.

```
CLOSE(FILEID,REEL)
```

The following example closes FILEID and, assuming FILEID is a disk file, returns to the system the unused portion of the last row of FILEID.

```
CLOSE(FILEID,CRUNCH)
```

The following example requests an orderly close on subfile 1 of port file FILEID. Control is returned to the program as soon as the close has been checked for semantic consistency, because the DONTWAIT port close option is included.

```
CLOSE (FILEID [SUBFILE 1], CLOSEDISPOSITION = ORDERLY, DONTWAIT)
```

The following example requests a close of subfile 1 of port file FILEID. Since the <closedisposition option> is not specified, a CLOSEDISPOSITION ABORT operation is performed. Since the <control option> is not specified, WAIT is assumed, and control is not returned to the file until the close is complete. The information specified in the <associateddata option> is sent to the correspondent program during the close process. The length need not be specified because a string expression is being used.

```
CLOSE (FILEID [SUBFILE 1], ASSOCIATEDDATA = STRNG)
```

The following example requests a close of subfile 1 of port file FILEID. During the close process, 14 characters of data are taken, beginning at the location pointed to by PTR, and are sent to the correspondent process as associated data.

```
CLOSE (FILEID [SUBFILE 1], ASSOCIATEDDATALENGTH = 14,  
      ASSOCIATEDDATA = PTR)
```

CONTINUE Statement

The CONTINUE statement causes control to pass from the program in which the statement appears to a coroutine.

<continue statement>

```
— CONTINUE [ ( —<task designator>— ) ]
```

Coroutines

A coroutine is a procedure that is initiated as a separate task by using a CALL statement. The caller is referred to as the primary coroutine and the called procedure as the secondary coroutine.

Because the execution of CONTINUE statements causes control to alternate between primary and secondary coroutines, processing always continues at the point where it last terminated.

The secondary coroutine uses the CONTINUE statement form without the task designator to pass control back to its partner task, which is the primary coroutine by default. The task designator is used by the primary coroutine to pass control to the secondary coroutine associated with that task designator by the CALL statement. For more information, refer to “CALL Statement” earlier in this section.

Examples of CONTINUE Statements

The following example passes control from this program, a secondary coroutine, to its partner task, which is, by default, the primary coroutine.

```
CONTINUE
```

The following example passes control to the coroutine associated with the task TSK.

```
CONTINUE(TSK)
```

DEALLOCATE Statement

The DEALLOCATE statement causes the contents of the specified array row or procedure reference array row to be discarded and the memory area to be returned to the system.

Notes:

- The DEALLOCATE statement cannot be used for task arrays or structure block arrays.
- When a procedure reference array is imported from a library, it cannot be deallocated using the DEALLOCATE statement. An attempt to deallocate an imported procedure reference array results in a compile-time or run-time error.

<deallocate statement>

```
— DEALLOCATE — ( [ <array row> ] ) —————  
                  [ <procedure reference array row> ]
```

Deallocation with Arrays

When an array row or procedure reference array row is deallocated, it is made not present (all data is lost). When the array row or procedure reference array row is used again, it is made present, and each element is reinitialized to 0 (zero) if it is an array row and to the uninitialized state if it is a procedure reference array row.

Event arrays cannot be deallocated by using the DEALLOCATE statement.

When a procedure reference array is imported from a library, it cannot be deallocated using the DEALLOCATE Statement. An attempt to deallocate an imported procedure reference array results in a compile-time or run-time error.

Examples of DEALLOCATE Statements

The following example discards the contents of ARAY and returns the memory area to the system. Note that ARAY must be a one-dimensional array or a syntax error results.

```
DEALLOCATE(ARAY)
```

The following example discards the contents of the row MATRIXARY[INDX,*] and returns the memory area to the system.

```
DEALLOCATE(MATRIXARY[INDX,*])
```

The following example discards the contents of the procedure reference array row PROCARRAY[1,*] and returns the memory area to the system.

```
DEALLOCATE(PROCARRAY[1,*])
```

DETACH Statement

The DETACH statement severs the association of an interrupt with an event.

<detach statement>

— DETACH —<interrupt identifier>—————|

Detaching Interrupts

Any pending invocations of a detached interrupt are discarded. Detaching an interrupt that is not attached to an event is essentially a *no-operation*; no error occurs.

The enabled/disabled condition of an interrupt is not changed by a DETACH statement. When an interrupt is attached after it has been detached, the enabled/disabled condition of the interrupt is the same as it was before it was detached. For more information, see “ATTACH Statement,” “DISABLE Statement,” and “ENABLE Statement” in this section and “INTERRUPT Declaration” in Section 3, “Declarations.”

Example of a DETACH Statement

The following example severs the association between the interrupt THEPHONE and the event to which it is attached.

```
DETACH THEPHONE
```

DISABLE Statement

The DISABLE statement prevents interrupt code from being executed.

<disable statement>

```
— DISABLE —┬──────────────────────────────────────────────────────────────────────────────────┘  
              └<interrupt identifier>┘
```

Disabling Interrupts

A DISABLE statement that does not specify an interrupt identifier is referred to as a *general disable*. A general disable has the effect of disabling all the interrupts for the task. The interrupts whose associated events are caused are placed in an interrupt queue for the task.

If the DISABLE statement specifies an interrupt identifier, only that interrupt is disabled. The system queues these interrupts until the interrupt is enabled.

Interrupts are queued to ensure that none are lost during the time they are attached. Queuing continues until the appropriate ENABLE statement is executed.

Disabling or enabling an interrupt is not affected by whether or not the interrupt is attached to an event.

For more information, see “ATTACH Statement,” “DETACH Statement,” and “ENABLE Statement” in this section and “INTERRUPT Declaration” in Section 3, “Declarations.”

Examples of DISABLE Statements

The following example is a general disable; it disables all interrupts.

```
DISABLE
```

The following example disables the interrupt named THEPHONE.

```
DISABLE THEPHONE
```

DISPLAY Statement

The DISPLAY statement causes the specified message to be displayed on the Operator Display Terminal (ODT) and to be printed in the job summary of the program.

<display statement>

— DISPLAY — (<pointer expression> | <string expression>) —————

Pointer and String Expressions

The message to be displayed is specified by the pointer expression or the string expression. If the parameter to the DISPLAY statement is a pointer expression, execution of the DISPLAY statement causes the characters to which the pointer expression points to be displayed on the ODT. The pointer expression must point to EBCDIC characters and the message to be displayed must be terminated by a null character (48"00").

If the parameter to the DISPLAY statement is a string expression, execution of the DISPLAY statement causes the contents of the string specified by the string expression to be displayed on the ODT. The string expression must be of type EBCDIC.

Display messages from programs run in CANDE appear on the user's terminal if the MESSAGES option of the CANDE SO command has been specified.

A maximum of 430 characters can be displayed.

Examples of DISPLAY Statements

The following example displays the EBCDIC characters stored in array Q, from the beginning of the array to the EBCDIC null character (48"00) or to the end of the array.

```
DISPLAY(POINTER(Q,8))
```

The following example displays the string created by concatenating VALUE IS and the string STR.

```
DISPLAY("VALUE IS " CAT STR)
```

The following example displays the string stored in the string variable MESSAGESTRING.

```
DISPLAY(MESSAGESTRING)
```

DO Statement

The DO statement causes a statement to be executed until a specified condition is met.

<do statement>

— DO —<statement>— UNTIL —<Boolean expression>—

Evaluation of Boolean Expression

The statement following DO is executed. The Boolean expression is evaluated, and if it is FALSE, the statement is executed again and the Boolean expression is reevaluated. This sequence of operations continues until the value of the Boolean expression is TRUE. At that time, control passes to the statement following the DO statement.

Note that both <block> and <compound statement> are statements and can be substituted for <statement>.

Figure 4–1 illustrates the DO-UNTIL loop.

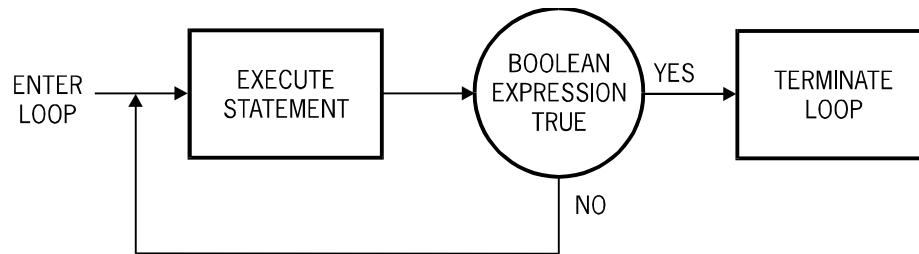


Figure 4–1. DO-UNTIL Loop

Examples of DO Statements

```
DO
  BEGIN
    PTR := *-4;
    CTR := *+4;
  END
UNTIL PTR IN LOOKEDFOR
```

```
DO
  J := J/2
UNTIL BUF[J] < JOB
```


ENABLE Statement

The ENABLE statement allows interrupt code to be executed.

<enable statement>

```
— ENABLE [ <interrupt identifier> ]
```

Enabling Interrupts

Previously disabled interrupts can be enabled with the ENABLE statement. If the event associated with the interrupt is caused after an interrupt has been enabled, then the interrupt code is executed.

An ENABLE statement that does not specify an interrupt identifier is referred to as a general enable and causes the system to look for, and place in execution, all interrupts that are in the interrupt queue of the task.

If the ENABLE statement specifies an interrupt identifier, only that interrupt is enabled. The system executes all occurrences of the interrupt in the interrupt queue.

Disabling or enabling an interrupt is not affected by whether or not the interrupt is attached to an event.

For more information, refer to “ATTACH Statement,” “DETACH Statement,” and “DISABLE Statement” earlier in this section and to “INTERRUPT Declaration” in Section 3, “Declarations.”

Examples of ENABLE Statements

The following example is a general enable: it enables all previously disabled interrupts.

```
ENABLE
```

The following example enables the interrupt named THEPHONE.

```
ENABLE THEPHONE
```

ERASE Statement

The ERASE statement removes all records from a file, leaving the attributes of the file unchanged where possible. The LASTRECORD attribute is set to -1 and all data are lost. For more information on file attributes, see the *File Attributes Programming Reference Manual*.

<erase statement>

— ERASE — (—<file designator>—) —————|

The ERASE statement can be used as a Boolean primary. When it is used in this way, it returns one of the following enumerated results if an error has occurred:

NOTCLOSERETAINED
VALIDONLYFORDISK
NODISKHEADER
SECURITYERROR
NOTONLYUSEROFFILE
INTERCHANGEFILENOTALLOWED

In using the ERASE statement, the following requirements must be met:

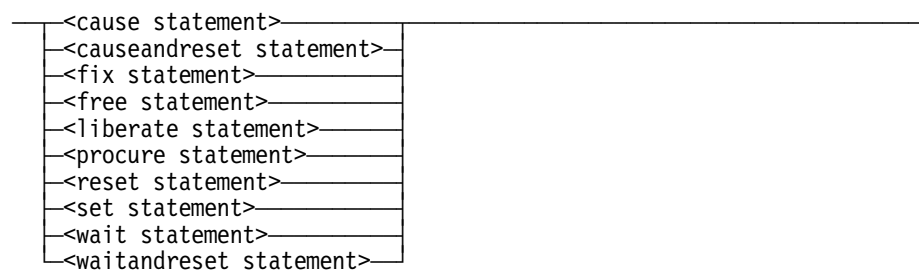
- The file specified in the ERASE statement must be a local file or a File Transfer, Access, and Management (FTAM) foreign file and the KIND attribute must be disk or pack. For more information on FTAM, see the *I/O Subsystem Programming Guide*.
- The file must be closed with retention.
- The open count of the file must be 1.
- The PERMITTEDACTIONS attribute must permit the erase.
- The user must have access to the file.

The user process is discontinued if an error occurs when the ERASE statement is not used as a Boolean primary.

EVENT Statement

Events have two Boolean characteristics, happened and available. Each characteristic can be in one of two states: TRUE or FALSE. These states can be changed using event statements.

<event statement>



The happened and available states of an event can be interrogated using the HAPPENED function and the AVAILABLE function. For more information, see "AVAILABLE Function" and "HAPPENED Function" in Section 5, "Expressions and Functions."

EXCHANGE Statement

The EXCHANGE statement is used to exchange rows between two disk files.

<exchange statement>

— EXCHANGE — (—<file designator>— [—<row/copy numbers>—] —————→
→ , —<file designator>— [—<row/copy numbers>—] —) —————|

<row/copy numbers>

—<row number>—|
└ , —<copy number>—|

<row number>

—<arithmetic expression>—|

<copy number>

—<arithmetic expression>—|

Conditions for Execution of the EXCHANGE Statement

The two disk files must be closed when the EXCHANGE statement is executed, the two rows must be the same size, the specified row numbers and the specified copy numbers must be valid, and the two files cannot be code files of any kind.

Row numbers begin with zero and copy numbers begin with 1. If there are copies of the file and a copy number is specified, then only the rows of that copy are exchanged.

For the exchange to take place, the referenced files must be closed with retention. For more information, see "CLOSE Statement" earlier in this section.

If the system detects an error, the exchange is not performed and the program resumes execution with the next statement. After the program uses the EXCHANGE statement, the row addresses should be checked by using file attributes to ensure that the exchange was successfully completed.

Caution

The EXCHANGE statement creates an unaudited directory transaction. An unaudited directory transaction is not subject to recovery in the event of halt/load disk family verification. Thus, the disk family directory can be left in an inconsistent state with an area within the family allocated to both of the files involved in the exchange operation.

Always make sure that an EXCHANGE statement finishes successfully. You can do this by programmatically capturing the row addresses that were involved in the exchange, and comparing them with the respective rows of the files that were involved in the exchange. If the comparison yields an inconsistency, then take corrective measures to recover the affected files to their respective states before the EXCHANGE statement was executed.

Refer the *System Messages Support Reference Manual* for information about recovery techniques.

Examples of EXCHANGE Statements

The following example exchanges the contents of row ROW6 of FILE1 with the contents of row ROW0 of FILE2.

```
EXCHANGE(FILE1[ROW6],FILE2[ROW0])
```

The following example exchanges row I of MASTERFYLE with row J of REBUILTFYLE.

```
EXCHANGE(MASTERFYLE[I],REBUILTFYLE[J])
```

FILL Statement

The FILL statement fills an array row with specified values. The FILL statement cannot be used with character arrays.

<fill statement>

— FILL —<array row>— WITH —<value list>—

<value list>

—<initial value>—

<initial value>

—<number>
—<string literal>
—<unsigned integer>— (—<value list>—) —

Initialization

Each initial value initializes an integral number of words. The number of words initialized depends on the type of the array and the kind of initial value.

Single-precision numbers initialize one word in arrays other than double or complex arrays. In double arrays, this word is extended with a second word of 0 (zero). In complex arrays, this word is normalized and then extended with an imaginary part of 0 (zero).

Double-precision numbers are stored unchanged in two words in double arrays. In complex arrays, the value is rounded and normalized to single-precision and then extended with an imaginary part of 0 (zero). For other types of arrays, the second word of the double-precision value is dropped and the first word initializes one word of the array.

String literals more than 48 bits long initialize as many words as are needed to contain the string and are left-justified with trailing zeros inserted in the last word, if necessary. In complex and double arrays, long string literals can initialize an odd number of words, causing the following initial value to start in the middle of a two-word element of the array.

String literals less than or equal to 48 bits long are right-justified within one word with leading zeros, if necessary. This word initializes one word in arrays other than double or complex arrays. In double arrays, this word is extended with a second word of 0 (zero). In complex arrays, this word is normalized and then extended with an imaginary part of 0 (zero).

An initial value of the form *<unsigned integer>* (*<value list>*) causes the values in the value list to be repeated the number of times specified by the unsigned integer.

If the value list contains more values than will fit in the array row, filling stops when the array row is full.

If the value list contains fewer values than the array row can hold, the remainder of the array row is left unchanged.

The length of the value list cannot exceed 4095 48-bit words.

Examples of FILL Statements

The following example fills the first 250 words of the one-dimensional array MATRIX with zeros.

```
FILL MATRIX[*] WITH 250(0)
```

The following example fills the designated row of array GROUP with the value .25, the string ALGOL right-justified with leading zeroes, the character " right-justified with leading zeros, and with the string LONGER STRING, which fills two words and part of a third word. Trailing zeros fill the rest of the third word.

```
FILL GROUP[1 ,*] WITH .25, "ALGOL", "", "LONGER STRING"
```

FIX Statement

The FIX statement examines the available state of an event. After the FIX statement executes, the available state of the designated event is always FALSE (not available).

<fix statement>

— FIX — (—<event designator>—) —————|

FIX Statement as a Boolean Function

The FIX statement can be used as a Boolean function. If the available state of the specified event is TRUE (available), the event is procured, the state is set to FALSE (not available), and FALSE is returned as the value of the function. If the available state of the specified event is FALSE (not available), the FIX statement returns TRUE, and the available state is left unchanged.

The FIX statement is sometimes referred to as the conditional procure function.

When the FIX statement has finished execution, the available state of the event is FALSE (not available).

Examples of FIX Statements

The following example examines the available state of the event EVNT.

```
FIX(EVNT)
```

The following example examines the available state of the event designated by EVENTARRAY[INDEX].

```
FIX(EVENTARRAY[INDEX])
```

The following example examines the available state of event FILELOCK and stores in GOTIT a value indicating this state.

```
IF GOTIT:= FIX(FILELOCK) THEN...
```

The following example examines the available state of the task's EXCEPTIONEVENT.

```
FIX(MYSELF.EXCEPTIONEVENT)
```


FOR Statement

The FOR statement constructs a loop consisting of one or more statements that are executed a specified number of times.

<for statement>

— FOR —<variable>— := —<for list, element>— DO —<statement>—

<for list element>

—<initial part>—
 —<iteration part>—

<initial part>

—<arithmetic expression>—

<iteration part>

— STEP <arithmetic expression> — UNTIL <arithmetic expression> —
 — WHILE <Boolean expression> — WHILE <Boolean expression> —

The number of times a FOR loop is traversed is determined by a variable, called the control variable, which is initialized when the FOR statement is first entered, and which can be updated during each iteration of the loop.

The action of a FOR statement can be described by isolating the following three distinct steps:

- Assignment of a value to the control variable
- Test of the limiting condition
- Execution of the statement following DO

Each type of <for list element> syntax specifies a different process. However, all of these processes have one property in common: the initial value assigned to the control variable is that of the arithmetic expression in the <initial part> construct.

The <for list element> construct establishes which values are assigned to the control variable and which test to make of the control variable to determine whether or not the statement following DO is executed. When a for-list element is exhausted, the next for-list element, if any, is evaluated, progressing from left to right. When all for-list elements have been used, the FOR statement is considered completed, and execution continues with the statement following the FOR statement. The statement following DO can transfer control outside the FOR statement, in which case some for-list elements might not have been exhausted before the FOR statement is exited.

Forms of the FOR Statement

In the following discussion of the various forms of the FOR statement, the letter V stands for the control variable; AEXP1, AEXP2,...are arithmetic expressions; BEXP is a Boolean expression; and S1 is a statement.

FOR-DO Loop

Assume that a for-list element consists of only an initial part, such as the following:

```
FOR V:= AEXP1, AEXP2,... DO
```

In this case that for-list element designates only one value to be assigned to the control variable. Because no limiting condition is present, no test is made. After assignment of the arithmetic expression to the control variable, the statement following DO is executed, and the for-list element is considered exhausted.

Figure 4-2 illustrates the FOR-DO loop.

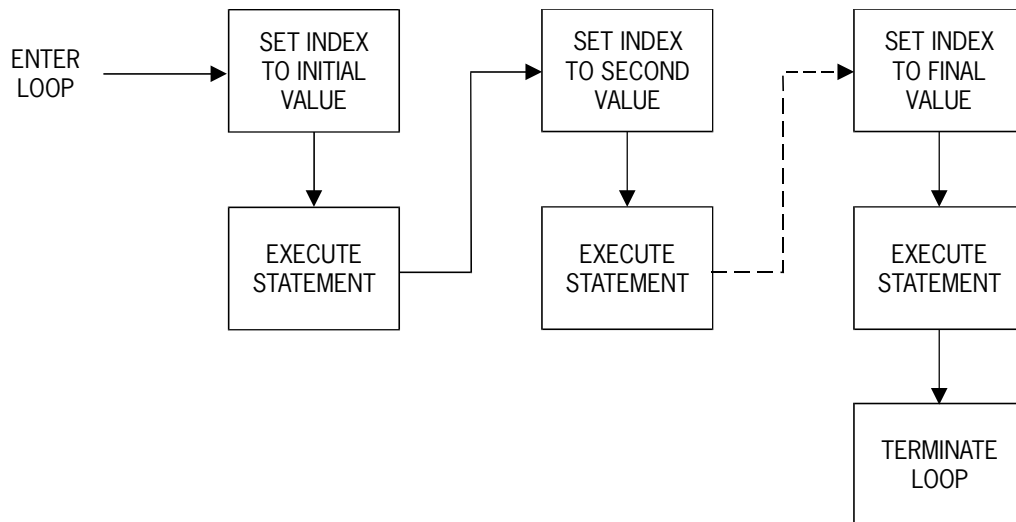


Figure 4-2. FOR-DO Loop

FOR-STEP-UNTIL Loop

Assume a for-list element is of the form *<initial part> STEP <arithmetic expression> UNTIL <arithmetic expression>* such as the following:

```
FOR V:= AEXP1 STEP AEXP2 UNTIL AEXP3 DO <statement>
```

In this case, a new value is assigned to the control variable V before each execution of the statement following DO. First, the initial value, that of AEXP1, is assigned to the control variable. After each execution of the statement following DO, the assignment $V := V + AEXP2$ is performed. Both AEXP2 and AEXP3 are reevaluated each time through the loop.

A test is made immediately after each assignment of a value to V to determine whether or not the value of V has passed the value of AEXP3. Whether AEXP3 is an upper or a lower limit depends on the sign of AEXP2; AEXP3 is an upper limit if AEXP2 is positive and a lower limit if AEXP2 is negative. If AEXP3 is an upper limit, then V has passed AEXP3 when the expression $V \leq AEXP3$ is no longer TRUE. If AEXP3 is a lower limit, then V has passed AEXP3 when the expression $V \geq AEXP3$ is no longer TRUE. If V has not passed AEXP3, the statement following DO is executed; otherwise, the for-list element is exhausted. Figure 4-3 illustrates the FOR-STEP-UNTIL loop.

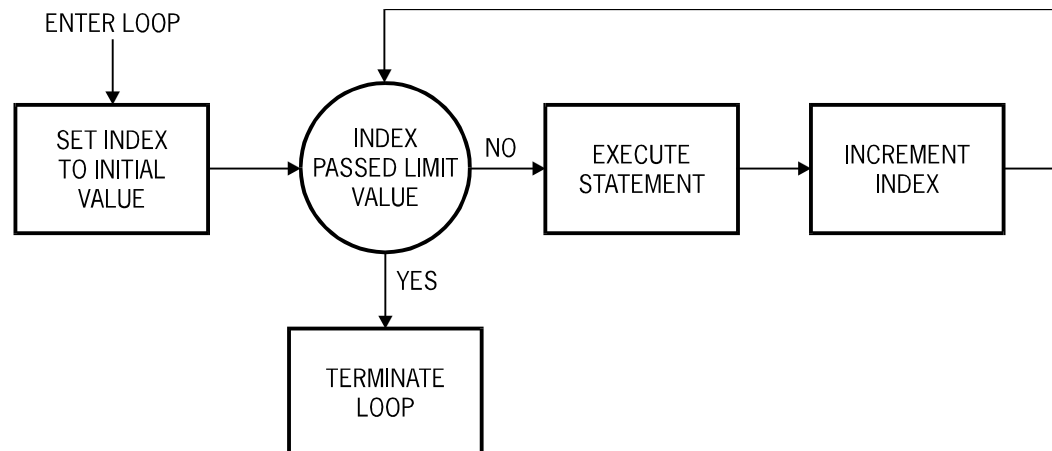


Figure 4-3. FOR-STEP-UNTIL Loop

FOR-STEP-WHILE Loop

Assume a for-list element is of the form *<initial part> STEP <arithmetic expression> WHILE <Boolean expression>* such as the following:

```
FOR V := AEXP1 STEP AEXP2 WHILE BEXP DO <statement>
```

In this case a new value is assigned to the control variable V before each execution of the statement following DO. First, the initial value, that of AEXP1, is assigned to the control variable. After each execution of the statement following DO, the assignment $V := V + AEXP2$ is performed. AEXP2 is reevaluated each time through the loop. After each assignment to V, the Boolean expression BEXP is evaluated and, if BEXP is TRUE, the statement following DO is executed. If BEXP is FALSE, this for-list element is exhausted. Figure 4-4 illustrates the FOR-STEP-WHILE loop.

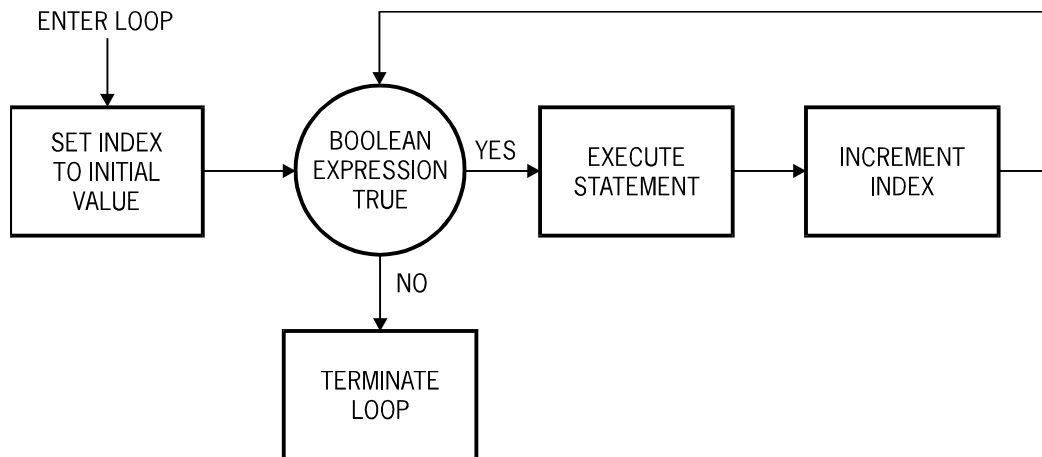


Figure 4-4. FOR-STEP-WHILE Loop

FOR-WHILE Loop

Assume the for-list element is of the form *<initial part> WHILE <Boolean expression>*, such as the following:

```
FOR V:= AEXP1 WHILE BEXP DO
```

In this case the control variable V is assigned the value of AEXP1 before each execution of the statement following DO. AEXP1 is reevaluated for each assignment to V. After each assignment to V, the Boolean expression BEXP is evaluated. If the value of BEXP is TRUE, the statement following DO is executed. If the value of BEXP is FALSE, this for-list element is exhausted. For example, in the following FOR statement if V had the value zero before execution of this statement, S1 would be executed five times:

```
FOR V:= V + 1 WHILE V LEQ 5 DO
  S1;
```

Figure 4–5 illustrates the FOR-WHILE loop.

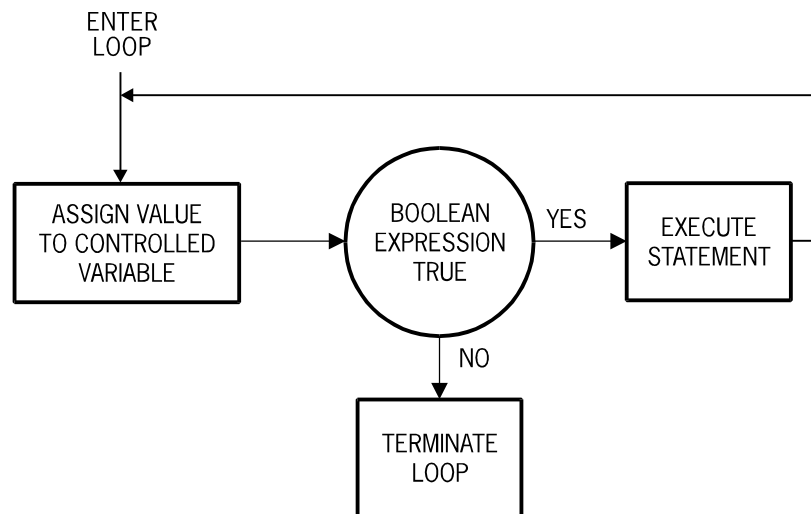


Figure 4–5. FOR-WHILE Loop

Examples of FOR Statements

The following example executes the statement following DO just once, with I assigned zero.

```
FOR I:= 0 DO
```

The following example assigns 1 to elements 0 through 255 of array LOOKEDFOR.

```
FOR J:= 0 STEP 1 UNTIL 255 DO  
  LOOKEDFOR[J]:= 1
```

The following example assigns ITEM to elements 0, 1, 2, 5, 10, 15, 16, and 37 of array BUF.

```
FOR INDEX:= 0, 1, 2, 10, 15, 37, 5, 16 DO  
  BUF[INDEX]:= ITEM
```

The following example calls FETCH repeatedly, passing the values 0, 1, 2, 3, 4, 5, 29, and the values of $(47 + 3 * X)$ where $X = 0, 1, 2,$ and so on, as long as $(47 + 3 * X)$ is less than LIM.

```
FOR X := 0 STEP 1 UNTIL 5, 29, 47 STEP 3 UNTIL LIM DO  
  FETCH(X)
```

The following example calls PANHANDLE and assigns to NEXT values equal to BEG, BEG + AMT, BEG + 2*AMT, and so on, as long as DONE is FALSE.

```
FOR NEXT := BEG STEP AMT WHILE NOT DONE DO  
  PANHANDLE
```

The following example increments TARGET by the value IX + 7 as long as TARGET is less than or equal to RANGE.

```
FOR N := IX + 7 WHILE TARGET LEQ RANGE DO  
  TARGET := * + N
```

FREE Statement

The FREE statement sets the available state of the specified event to TRUE (available).

<free statement>

— FREE — (—<event designator>—) —————|

FREE Statement as a Boolean Function

The FREE statement can be used as a Boolean function that returns FALSE if the available state of the event is already TRUE (available) and TRUE if the available state of the event is FALSE (not available). In either case, the available state of the event is unconditionally set to TRUE (available).

The FREE statement does not activate any task attempting to procure the event, nor does it activate any task waiting on the event.

Examples of FREE Statements

The following example sets the available state of the event EVNT to TRUE (available).

```
FREE(EVNT)
```

The following example sets the available state of the event designated by EVNTARRAY[INDX] to TRUE (available).

```
FREE(EVNTARRAY[INDX])
```

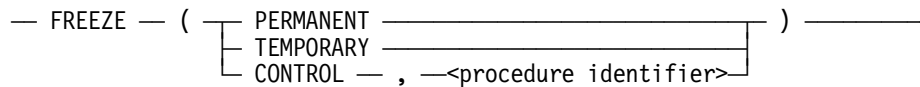
The following example assigns to WASPROCURED a value indicating the available state of the event FYLELOCK, and sets the available state of FYLELOCK to TRUE (available).

```
IF WASPROCURED := FREE(FYLELOCK) THEN...
```

FREEZE Statement

The FREEZE statement changes the running program into a library.

<freeze statement>



For more information on <procedure identifier>, see “PROCEDURE Declaration” in Section 3, “Declarations.”

FREEZE Statements in Library Procedures

At least one EXPORT declaration must appear in the same block as the FREEZE statement. The procedures affected by a FREEZE statement are the procedures that appear in EXPORT declarations in the same block as the FREEZE statement. After the FREEZE statement is executed, these procedures are library entry points.

The PERMANENT and TEMPORARY specifications of the FREEZE statement control the permanence of the library. A permanent library remains available until it is discontinued. A temporary library remains available as long as there are users of the library. A temporary library that is no longer in use unfreezes (thaws) and resumes running as a regular program. However, a temporary library does not unfreeze until it has been referenced at least once. When a library unfreezes, it cannot execute another FREEZE statement in an attempt to become a library again.

The CONTROL specification of the FREEZE statement controls the nature of the freeze. The program is set up as a permanent library, but after the freeze operation has been performed, control is transferred to the specified procedure, known as the control procedure. The procedure must be untyped and must have no parameters.

Once the control procedure is in control, the library can keep track of the number of its users through the task attribute LIBRARYUSERS.

When the control procedure is exited, the library unfreezes if there are no users. If there are users, the library becomes an ordinary library, and a warning message is issued. A control frozen library can prevent the warning message from being issued and prevent the linkage of new users by changing the value of the task attribute STATUS to VALUE(GOINGAWAY). This status change causes the library to unfreeze and resume execution as a normal program as soon as possible. For further information on the value setting for the STATUS attribute, refer to the *Task Attributes Programming Reference Manual*.

Because a library program initially runs as a regular program, the flow of execution can be such that the execution of a FREEZE statement is conditional and can occur anywhere in the outer block of the program.

If a calling program causes a library to be initiated and this library does not execute a FREEZE statement (if, for example, it was not a library program and thus had no FREEZE statement), then the attempted linkage to the library entry points cannot be made, and the calling program is discontinued. For more information on libraries, refer to Section 8, "Library Facility."

Examples of FREEZE Statement

In the following example, control is transferred to procedure Z after executing the FREEZE statement. Procedure Z is untyped and has no parameters. The value of the STATUS is changed to GOINGAWAY, which causes the library to become temporary. New users cannot link to this library. If there are users, the library does not resume execution until the number of users becomes zero.

```
BEGIN
PROCEDURE X(A,B);
...                               %Procedure to be exported
PROCEDURE Y(P,Q);
...                               %Procedure to be exported
PROCEDURE Z;
BEGIN      %Control procedure - untyped and no parameters
...
MYSELF.STATUS:= VALUE(GOINGAWAY);
WHILE MYSELF.LIBRARYUSERS GTR 0 DO
WAITANDRESET (MYSELF.EXCEPTIONEVENT);
...
END;
EXPORT X, Y;
...
FREEZE(CONTROL,Z);
...
END.
```

The following example transfers control to the procedure CTRL_PROCEDURE after the freeze operation is completed.

```
FREEZE(CONTROL,CTRL_PROCEDURE)
```

GO TO Statement

The GO TO statement transfers control to the statement in the program with the specified label.

<go to statement>

— GO TO <designational expression>—————|

The value of the designational expression specifies the label to which control is transferred.

Because labels must be declared in the innermost block in which they occur as statement labels, a GO TO statement cannot lead from outside a block to a point inside that block. Each block must be entered at the BEGIN so that the declarations associated with that block are invoked. For more information on labels, refer to “LABEL Declaration” in Section 3, “Declarations.”

Bad GO TO

A bad GO TO occurs when a GO TO statement in an inner block transfers control to a label that is global to that block. A necessary side effect of a bad GO TO is that the block in which it occurs is exited abruptly and local variables are deallocated immediately.

A bad GO TO requires cutting back the lexical (lex) level to a more global block. To perform a bad GO TO, the operating system is invoked to cut back the stack and discard any locally declared items that occupy memory space outside of the stack, sometimes referred to as nonstack items, such as files, arrays, and interrupts.

Examples of GO TO Statements

In the following example, control is transferred to the statement with the label LABEL1.

```
GO TO LABEL1
```

In the following example, control is transferred to the statement with the label LABEL2.

```
GO LABEL2
```

In the following example, control is transferred to the statement with the label designated by the subscripted switch label identifier SELECTIT[INDX].

```
GO TO SELECTIT[INDX]
```

In the following example, if K is equal to 1, control is transferred to the statement with the label designated by the subscripted switch label identifier SELECT[2]. Otherwise control is transferred to the statement with the label START.

```
GO TO IF K=1 THEN SELECT[2] ELSE START
```

I/O Statement

An I/O statement causes information to be exchanged between a program and a peripheral device, and allows the programmer to perform certain control functions.

<I/O statement>

<accept statement>
<close statement>
<display statement>
<lock file statement>
<open statement>
<read statement>
<rewind statement>
<seek statement>
<space statement>
<write statement>

ALGOL I/O is handled by a part of the operating system called the I/O subsystem. For more information on the I/O subsystem, refer to the *I/O Subsystem Programming Guide*.

The ACCEPT statement and DISPLAY statement are unique in that the file to or from which data is transferred need not be specified. For more information, refer to "ACCEPT Statement" and "DISPLAY Statement" earlier in this section.

The remaining I/O statements reference a file that must be declared by the program. For more information, refer to "FILE Declaration" in Section 3, "Declarations."

Two distinct methods of I/O are available. The first and typical method is referred to as normal I/O; the second method is called direct I/O. The major differences between normal I/O and direct I/O have to do with buffering, the overlap of program execution, and the overlap of I/O operations. Their effect on a particular I/O statement is presented in the description of the statement.

Normal I/O

Normal I/O is indicated when direct files and direct arrays are not used. Normal I/O includes many automatic facilities provided by the operating system, such as the following:

- Buffering: the automatic overlap of program processing and I/O traffic to and from the peripheral units
- Blocking: more than one logical record per physical block
- Translation as needed between the character set of the unit and that required by the program

The amount of buffering between the I/O statements and program execution depends on the number of buffers allocated for the file. Refer to "FILE Declaration" in Section 3, "Declarations," for information on how to specify the number of buffers.

I/O Statement

In normal I/O, a READ statement causes the automatic testing of the availability of the needed record. The program is suspended in the READ statement until the record is actually available for use.

In normal I/O, a WRITE statement transfers the specified data to a buffer; the program is immediately released to begin execution of the next statement. If all the buffers are full when the WRITE statement is executed, the program is suspended until a buffer is available.

Direct I/O

Direct I/O is indicated when direct files and direct arrays are used.

Direct I/O allows more direct control of the actual I/O operations. In certain situations, avoiding suspension of the program is desirable. In other situations, nonstandard I/O operations and masking of certain types of error conditions which could arise are desirable.

When direct I/O is used, the program is responsible for the buffering, blocking, and translation.

The syntax for a direct read or direct write operation employs the *<arithmetic expression>*, *<array row>* form of *<format and list part>*. An event designator is the only allowable form of action labels or finished event for direct I/O. The value of the arithmetic expression has the following meaning:

Field	Contains
[16:17]	Number of words to be transferred
[19:3]	Number of trailing characters to be transferred

The array row is called the I/O area of the user. A direct array identifier must be used for the *<array name>* part in the array row construct. Thus, the following statement could be used to perform a direct read of 10 words from file FID into direct array A using the event EVT as the finished event:

```
READ(FID, 10, A[*]) [EVT]
```

The operating system establishes a relationship between the I/O area and the finished event, if one is specified. Before any subsequent use of the I/O area can be made in the program, either for calculations or for further I/O, the direct I/O operation must be finished. The finished event can be inspected by one of the following methods:

- By using the HAPPENED function
- By obtaining the value of the STATE file attribute using the WAIT statement as a Boolean function and specifying a direct array row as a parameter
- By using the WAIT statement on the event to deactivate the process until the event is caused

Once the operation has been completed, the happened state of the event should be set to FALSE (not happened) before reusing it. Refer to "WAIT Statement" later in this section for more information.

The finished event can be associated with a direct array row that is declared in a different block. For example, a formal event can be associated with a local array. Such an association can cause compile-time or run-time up-level event errors if the block containing the finished event can be exited before the block that contains the direct array is exited.

In direct I/O, the I/O operations analogous to the SPACE and REWIND statements are performed as if they were read or write operations, except that the IOCW direct array attribute is specifically assigned the proper hardware instructions for the operation.

When performing direct I/O with the SPACE operation, the device's spacing limitation overrides any user-specified spacing. In the case of a line printer, this limitation is two.

IF Statement

The IF statement causes a statement to be executed or not executed based on the value of a Boolean expression.

<if statement>

—<if clause>—<statement> — ELSE —<statement> —

<if clause>

— IF —<Boolean expression>— THEN —

Forms of the IF Statement

Assume the IF statement you are using is of the following form:

```
IF BEXP THEN S1
```

If the value of the Boolean expression BEXP is TRUE, the statement S1 is executed. If BEXP is FALSE, then S1 is not executed. In either case, execution continues with the statement following the IF statement.

Assume the IF statement you are using is of the following form:

```
IF BEXP THEN S1 ELSE S2
```

If the value of BEXP is TRUE, the statement S1 is executed and the statement S2 is ignored. If the value of BEXP is FALSE, then the statement S2 is executed, and S1 is ignored. In either case, execution continues with the statement following the IF statement.

Note that both block statements and compound statements can be substituted for <statement> in the IF statement.

IF statements can be nested; that is, the statements following the reserved words THEN or ELSE (or both) can also be IF statements.

When IF statements are nested, the correct correspondence between the reserved words THEN and ELSE must be maintained. The compiler matches the innermost THEN to the first ELSE that follows it and that yields a syntactically correct IF statement. Consider the following IF statement:

```
IF BEXP1 THEN IF BEXP2 THEN S2 ELSE S1
```

The ELSE is paired with the innermost THEN, which is the THEN following BEXP2, as illustrated in the following:

```
IF BEXP1 THEN
  IF BEXP2 THEN
    S2
  ELSE
    S1
```

If the program pairs the ELSE with the THEN following BEXP1, the inner IF statement must be made a compound statement by using BEGIN and END as follows:

```
IF BEXP1 THEN
  BEGIN
    IF BEXP2 THEN
      S2
    END
  ELSE
    S1
```

A GO TO statement can lead to a labeled statement within an IF statement. The subsequent action is equivalent to the action that would result if the IF statement were entered at the beginning and evaluation of the Boolean expression caused execution of the labeled statement.

Examples of IF Statements

In the following example, if ALLDONE is TRUE, control is transferred to the statement with the label AWAY. If ALLDONE is FALSE, the statement following the IF statement is executed.

```
IF ALLDONE THEN
  GO AWAY
```

In the following example, if the value of X is greater than the value of LIMIT, procedure ERROR is called. If the value of X is less than or equal to the value of LIMIT, the value of X is incremented by 1. In either case, execution continues with the statement following the IF statement.

```
IF X > LIMIT THEN
  ERROR
ELSE
  X := * + 1
```

INTERRUPT Statement

Interrupts provide a way to interrupt a process when a specific event occurs. Interrupt statements allow interrupts to be attached to and detached from events, and allow interrupts to be enabled and disabled.

<interrupt statement>



The ATTACH statement is used to associate an interrupt with an event.

The DETACH statement is used to sever the association between an interrupt and the event to which it is attached.

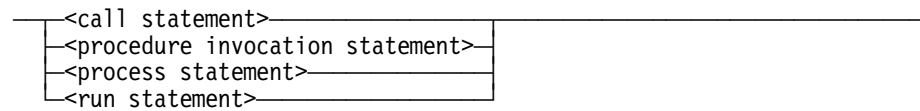
The ENABLE statement and DISABLE statement are used to explicitly enable and disable, respectively, an interrupt.

For more information on interrupts, refer to “INTERRUPT Declaration” in Section 3, “Declarations.”

INVOCATION Statement

An INVOCATION statement causes a previously declared procedure to be executed as a subroutine, an asynchronous process, a coroutine, or an independent program.

<invocation statement>



The CALL statement invokes a procedure to execute as a coroutine. The PROCEDURE INVOCATION statement invokes a procedure to execute as a subroutine. The PROCESS statement invokes a procedure to run as an asynchronous process. The RUN statement invokes a procedure to run as an independent program.

With the exception of the PROCEDURE INVOCATION statement, a separate stack is initiated and the specified procedure cannot be a typed procedure.

With the exception of the RUN statement, parameters can be call-by-name or call-by-value. All parameters passed in the RUN statement must be call-by-value.

LIBERATE Statement

The LIBERATE statement activates all tasks waiting on the specified event. It changes the happened state of the event to TRUE (happened) unless a waiting task uses the WAITANDRESET statement.

<liberate statement>

— LIBERATE — (—<event designator>—) —————|

Execution of Implicit CAUSE Statement

The LIBERATE statement causes the execution of an implicit CAUSE statement for the specified event. This implicit CAUSE statement results in a change to the happened state of the event, if no waiting task has used the WAITANDRESET statement. For more information, refer to “CAUSE Statement” and “WAITANDRESET Statement” in this section. The available state of the event is set to TRUE (available).

Although all waiting tasks are activated, they are linked into the ready queue in priority order. At that point, all tasks that were waiting to procure the event are in the ready queue in priority order. For more information about procuring events, refer to “PROCURE Statement” later in this section.

Examples of LIBERATE Statements

The following example causes the event ANEVENT and sets its available state to TRUE (available).

```
LIBERATE(ANEVENT)
```

The following example causes the event designated by EVENTARRAY[INDEX] and sets its available state to TRUE (available).

```
LIBERATE(EVENTARRAY[INDEX])
```

LOCK File Statement

The LOCK file statement causes the specified file to be closed.

<lock file statement>

```
— LOCK — ( —<file designator> [ , —<lock option> ] ) —————
```

<lock option>

```
[ CRUNCH  
* ] —————
```

The LOCK file statement cannot be used as a function.

Lock Options

If the specified file is a tape file, it is rewound and the tape unit is made inaccessible to the system until the operator readies it again. If the file is a disk file, it is retained as a permanent file on disk. The file buffer areas are returned to the system.

A LOCK file statement with a lock option performs the same action as a CLOSE statement that specifies CRUNCH. Whether CRUNCH or an asterisk (*) appears as the lock option, the action of the LOCK file statement is the same. The file must be a disk file. The unused portion of the last row of disk space, beyond the end-of-file indicator, is returned to the system. The disk file can no longer be expanded without being copied into a new file; however, data can be written to existing records.

Examples of LOCK File Statements

In the following example, if FILEA is a disk file it is retained as a permanent file.

```
LOCK(FILEA)
```

In the following example, the unused portion of the last row of disk file FYLE is returned to the system.

```
LOCK(FYLE,CRUNCH)
```

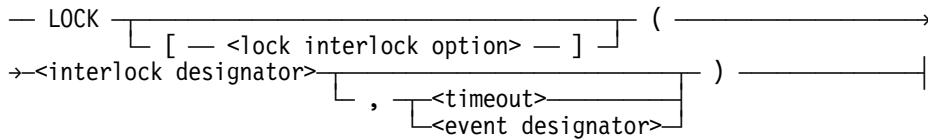
In the following example, the unused portion of the last row of disk file FYLE is returned to the system.

```
LOCK(FYLE,*)
```

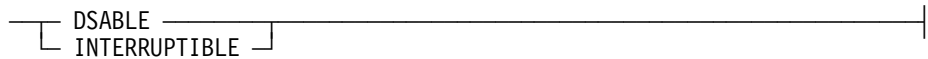
LOCK Interlock Statement

The LOCK interlock statement suspends the program until the interlock is acquired.

<lock interlock statement>



<lock interlock option>



<timeout>



The LOCK interlock statement attempts to acquire the interlock. The LOCK interlock can be used as a function. If LOCK interlock is used as a statement, the process is discontinued when the result is a value other than 1 (successfully acquired).

If an event designator is present and the interlock cannot be acquired immediately, the caller waits until either the interlock can be acquired or the event state is HAPPENED. The event cannot have an interrupt attached or be inscribed for use with a POBOX. If the event is used in either of these two ways, a result of 10 is returned.

When an isolated procedure is waiting to acquire an interlock while being run by another process, the DSABLE option specifies that the wait can be terminated if the calling process is discontinued. If the wait was terminated before the interlock could be acquired, the function value returned is zero.

If the interlock cannot be acquired immediately and the INTERRUPTIBLE option is used, the caller waits until either the interlock can be acquired, the timeout or event happens, or a signal is sent to the program. If the process is awakened by receipt of a signal, the function value is zero (0).

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

Value	Meaning
0	A signal was received before the interlock could be acquired.
1	The interlock was successfully acquired.
2	The timeout elapsed or the event happened (event state HAPPENED), before the interlock could be acquired.
10	The event used has a conflicting usage, such as an attached INTERRUPT.

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, a result of 2 is returned. If the caller that timed out is the only contender, the interlock becomes LOCKED_UNCONTENDED.
- If the interlock is LOCKED_CONTENTENDED, its state does not change, and the caller is added to the contender list.

Timeout Option

The timeout option, 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 specified, a timeout of -1 is assumed. If the timeout is 0 (zero), the lock succeeds only if the interlock is FREE.

Examples of LOCK Interlock Statements

The following examples show valid uses of the LOCK interlock statement:

```
LOCK (MYLOCK);  
  
LOCK (OURLOCKS [2]);  
  
I := LOCK (CONN_LIB[7].MYLOCK);  
  
I := LOCK (MYLOCK, 17);  
  
I := LOCK (MYLOCKS [31]);  
  
I := LOCK (MYLOCK, FINISHEVENT);
```

MERGE Statement

The MERGE statement causes data in the specified files to be combined and returned.

<merge statement>

— MERGE — (—<output option>— , —<compare procedure>— , —————→
→<record length>— , —<merging option list>—) —————|

<merging option list>

—<merging option>—, —<merging option>— |

<merging option>

—<input option>— |

Merge Options

The compare procedure determines the manner in which the data is combined. The output option specifies how the data is to be returned from the merge operation.

The merging option list must contain between two and eight input options, inclusive, which must be files or Boolean procedures.

Example of a MERGE Statement

The following example merges records from files IN1 and IN2 according to a scheme given in compare procedure COMP. The merged result is written to file LINEOUT. The records of IN1 and IN2 have a maximum record size of 14.

```
MERGE(LINEOUT,COMP,14,IN1,IN2)
```

MESSAGESEARCHER Statement

The MESSAGESEARCHER statement returns a completed output message based on the information passed to it.

<messagesearcher statement>

```

— MESSAGESEARCHER — ( —<output message array identifier>— [ —————→
→└──────────────────────────────────┘<arithmetic expression>— ] — , —→
→└──────────────────────────────────┘<result pointer>— , —<result length>————→
→└──────────────────────────────────┘ ) —————→
└──────────────────────────────────┘
└──────────────────────────────────┘ , —<parameter element>└──────────────────┘

```

<language specification>

```

└──────────────────────────────────┘
└──────────────────────────────────┘<string expression>
└──────────────────────────────────┘<pointer expression> FOR —<arithmetic expression>└──────────┘

```

<result pointer>

```

—<pointer expression>—————└──────────────────┘

```

<result length>

```

—<variable>—————└──────────────────┘

```

<parameter element>

```

└──────────────────────────────────┘
└──────────────────────────────────┘<string expression>
└──────────────────────────────────┘<pointer expression> FOR —<arithmetic expression>└──────────┘

```

The output message array identifier indicates the output message array from which the output message is to be obtained. For more information on output message arrays, see "OUTPUTMESSAGE ARRAY Declaration" in Section 3, "Declarations."

The language specification indicates the preferred language for the requested output message. The language specification must not have a trailing dot.

The arithmetic expression within the square brackets ([]) indicates the output message number of the message that is to be completed. The arithmetic expression cannot be a double-precision value.

The result pointer is a call-by-value EBCDIC pointer that points to where the completed output message is to be stored. An EBCDIC null character (48"00") is placed after the last character of the message. The null character is not included in the returned message length.

The result length is an integer or real variable that is assigned the length of the returned output message, not counting the null character that is appended at the end.

Finding a Requested Message

Each parameter element contains the actual value of a parameter that was specified in the declaration of the requested output message. The first parameter element refers to parameter <1>, the second to parameter <2>, and so on.

The following method is used to find the requested message so that it can be completed.

First, an initial language in which to search for the message must be selected. If a language specification is given as a parameter to the MESSAGESEARCHER statement, that language is selected; otherwise, the language in the language specification of the task requesting the message is used. If the task does not have a language specification, the system default language is used.

If the requested message cannot be found in the initial language and the initial language is not the system default language, the message is searched for in the system default language. If the message still cannot be found, then the message is searched for in the languages that exist in the specified output message array, beginning with the first language, the second language, and so on. If none of the languages in the output message array contains the message, an error message that specifies the message number is produced in place of the message.

MESSAGESEARCHER Statement as an Arithmetic Function

The MESSAGESEARCHER statement can be used as an arithmetic function that returns an integer result indicating whether or not the message was successfully found and formatted. The possible values for this result are as follows:

Value	Meaning
1	The message is not in the requested language; it is in MYSELF.LANGUAGE or the system default language.
0	The message was found and formatted as requested.
-1	Too few parameters were specified.
-2	No matching <output message case part> was found.
-3	The message is in the first available language.
-4	The array row referenced by the result pointer is too small.
-5	The message was not found.
-6	The version of the output message array is incompatible with the version of the operating system.
-7	The output message array is in error.
-8	A fault occurred while obtaining the output message.
-9	The length passed with a parameter is too long.
-11	The result pointer has a type that is not valid.

Example of a MESSAGESEARCHER Statement

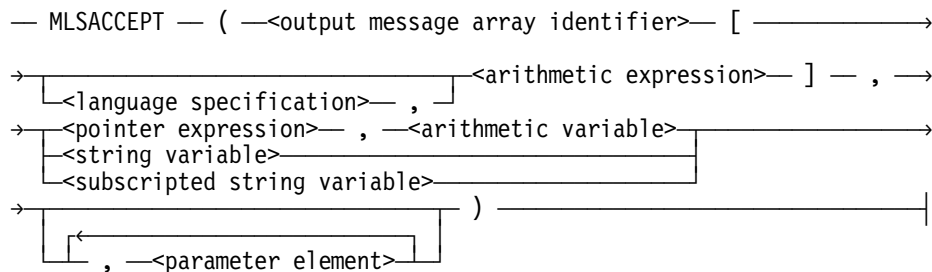
In the following example, the value of RTN, the returned integer, notes whether or not the message was successfully found and formatted. ERRORS is the output message array identifier. The language specification is ENGLISH. POSINTX is the arithmetic expression. MSG is the result pointer and MSG_LEN is the result length.

```
RTN:= MESSAGESEARCHER (ERRORS ["ENGLISH", POSINTX], MSG, MSG_LEN);
```

MLSACCEPT Statement

The MLSACCEPT statement either displays or prints a message and causes the program to wait for input, or it returns a Boolean value indicating whether or not a message is waiting for the program.

<MLSaccept statement>



MLSACCEPT Used for Data Input

When the MLSACCEPT statement is used for data input, it displays a message and causes the program to wait for input. No MLS translation is performed on the input text.

The input text can be entered at an Operator Display Terminal (ODT) or at a user terminal. If the input text is entered from a user terminal, the user must use the mix number of the task and be logged on to the usercode that originated the job.

The input text is placed in the specified pointer expression, string variable, or subscripted string variable. It is placed left-justified with leading blanks discarded. No translation is performed on the input text. The program continues execution with the statement following the MLSACCEPT statement.

MLSACCEPT Used as a Boolean Function

The MLSACCEPT statement can be used as a Boolean function to determine whether or not a message was entered before the MLSACCEPT statement was executed. If a message was entered, the result returned by the MLSACCEPT statement is TRUE and the message is placed in the pointer expression, string variable, or subscripted string variable. If no message was entered, the result is FALSE. In either case, the program does not wait for input.

Additional MLSACCEPT Options

The language specification defines the language to be used for messages that are displayed or printed.

The arithmetic expression indicates the output message number of the message to be displayed or printed. It cannot be a double-precision value.

The arithmetic variable is an integer or real variable that is assigned the length of the returned response.

Each parameter element contains the actual value of a parameter that was specified in the declaration of the requested output message. The first parameter element refers to parameter 1, the second to parameter 2, and so on.

No more than 430 characters can be displayed or printed. A maximum of 960 characters can be accepted as input. The input response can be entered before the MLSACCEPT statement is executed.

For additional information on output messages, refer to "MESSAGESEARCHER Statement" earlier in this section.

Example of an MLSACCEPT Statement

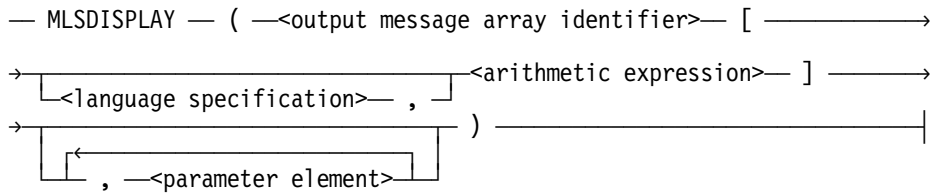
In the following example, the message number POSINDX from output message array ERRORS is displayed in the ENGLISH language. The program waits to accept an input which, when received, is placed in INPT for length specified by INPT_LEN.

```
MLSACCEPT (ERRORS ["ENGLISH", POSINDX], INPT, INPT_LEN);
```

MLSDISPLAY Statement

The MLSDISPLAY statement displays a message on the Operator Display Terminal (ODT) and prints the message in the job summary listing.

<MLsdisplay statement>



MLSDISPLAY Options

The MLSDISPLAY statement displays the specified message on the ODT and prints the message between the beginning-of-task (BOT) and end-of-task (EOT) messages on the job summary listing. If the program that invokes the statement was started with either the CANDE *RUN* or *START* command, and if the MESSAGES option is currently set to TRUE for the session, the text is also displayed on the user's terminal.

The output message array identifier indicates the output message array that contains the message to be displayed.

The language specification defines the language to be used for messages displayed on the user's terminal.

The arithmetic expression indicates the message number of the output message to be displayed. It cannot be a double-precision value.

Each parameter element contains the actual value of a parameter that was specified in the declaration of the requested output message. The first parameter element refers to parameter 1, the second to parameter 2, and so on.

No more than 430 characters can be displayed.

The MLSDISPLAY statement returns a Boolean result. If an error occurred, a result of TRUE is returned.

For additional information on output messages, refer to "MESSAGESEARCHER Statement" earlier in this section.

MLSTRANSLATE Statement

The MLSTRANSLATE statement returns an integer value indicating whether or not the message was successfully found in both the input and output languages. The translated message is returned to the caller using a pointer parameter.

<mlstranstate statement>

```

— MLSTRANSLATE — ( —<output message array identifier>— [ —————→
→ ┌──────────────────────────┐ <output language>— ] — , —————→
  └──┬──<input language>— , ─┘
→ <input message>— , —<result message number>— , —<result message>→
→ ) —————→

```

<input language>

<output language>

```

—<language specification>————→

```

<input message>

```

—┌<string expression>————→
  └──┬──<pointer expression>— FOR —<arithmetic expression>┘

```

<result message number>

```

—<arithmetic variable>————→

```

<result message>

```

—┌<result pointer>— , —<result length>
  └──┬──<string primary>————→

```

<result pointer>

```

—<pointer expression>————→

```

<result length>

```

—<arithmetic variable>————→

```

MLSTRANSLATE Options

A message that is used with the MLSTRANSLATE statement must not contain parameters.

The output message array identifier indicates the output message array from which the translated message is to be obtained.

The input language indicates the language from which the message is being translated. If no input language is specified, the language of the task is used as the input language. If the input message does not exist in the task language, the system language is used as the input language. If the input message is still not found, an error is returned. The input language must not have a trailing period.

The output language indicates the language into which the message is to be translated. The output language must not have a trailing period.

The input message contains the text to be translated. For the translation to be successful, the input message must be the same as the message contained in the output message array, including blanks. However, the translation is not case-sensitive.

The result message number is an integer or real variable that is assigned the message array index corresponding to the message looked up. This value can be used to acquire text in the language understood by the program through a MESSAGESEARCHER call.

The result message can be either a pointer/length pair of variables or a string variable, providing flexibility for the destination of the translated message.

The result pointer is a call-by-value EBCDIC pointer that points to where the translated message is to be stored. An EBCDIC null character (48"00") is placed after the last character of the message. The null character is not included in the returned message length.

The result length is an integer or real variable that is assigned the length of the returned output message, not counting the null character that is placed at the end.

The MLSTRANSLATE statement can be used only with output message arrays that have been declared with a CCSVERSION, a DECIMALPOINTIS, or a THOUSANDSEPARATORIS output message parameter.

MLSTRANSULATE as an Arithmetic Function

The MLSTRANSULATE statement can be used as an arithmetic function that returns an integer result indicating whether or not the message was successfully found in both the input and output languages. The possible values for this result are as follows:

Value	Meaning
1	The message is not in the requested language. It is in MYSELF.LANGUAGE or the system default language.
0	The message was found and returned as requested.
-1	The message was not found in the <input language>.
-2	The message was not found in the <output language>.
-3	The requested <output language> does not exist for this output message array.
-4	The array row referenced by the <result pointer> is too small.
-5	The message was not found in any language tried.
-6	The version of the output message array is incompatible with the version of the operating system.
-7	The output message array is corrupted: cannot obtain output message number <num>.
-8	A fault occurred while obtaining the output message.
-9	The length passed with a parameter is too long.
-11	The <result pointer> has a type that is not valid for the MLSTRANSULATE statement.
-12	The output message array created by this program cannot be used for message translation.
-13	The ccsversion of the input or output language messages is invalid.
-14	The CENTRALSUPPORT library is not available.

Example of an MLSTRANSLATE Statement

The following example illustrates the use of the MLSTRANSLATE statement:

```
BEGIN
  DEFINE NO_V = 0#,
         YES_V = 1#;
  OUTPUTMESSAGE ARRAY INPUT_MESSAGES
    (ENGLISH SYSTEMDEFAULT
     (NO_V = "NO",
      YES_V = "YES"),
     FRENCH
     (NO_V = "NON",
      YES_V = "OUI"));

  INTEGER I, MSG_NUMBER;
  STRING OUTP, INP;

  INP := "YES";
  I := MLSTRANSLATE (INPUT_MESSAGES ["ENGLISH", "FRENCH"],
                    INP, MSG_NUMBER, OUTP);

  IF I GEQ 0 THEN
    BEGIN
      DISPLAY (OUTP);
      CASE MSG_NUMBER OF
        BEGIN
          NO_V: MLSDISPLAY (INPUT_MESSAGES ["FRENCH", NO_V]);
          YES_V: MLSDISPLAY (INPUT_MESSAGES ["FRENCH", YES_V]);
        ELSE;;
      END;
    END
  ELSE
    DISPLAY ("ERROR:" CAT STRING8(I,5));
END.
```


MULTIPLE ATTRIBUTE ASSIGNMENT Statement

The multiple attribute assignment statement is used to assign values at run time to one or more attributes of a specified file.

<multiple attribute assignment statement>

—<file identifier>— (—<attribute specifications>—) —————|

Assignment of Values

If the name of a Boolean file attribute in the attribute specifications is not followed by an equal sign (=) and a value, it is assigned a value of TRUE; that is, the following attribute specifications have the same effect as each other:

DEPENDENTSPECS, KIND = DISK

DEPENDENTSPECS = TRUE, KIND = DISK

An assignment specified in a MULTIPLE ATTRIBUTE ASSIGNMENT statement occurs at run time and overrides any assignment made to the attribute in a FILE declaration or through file equation.

One intrinsic call is generated to assign all attributes, except when a pointer-valued file attribute name is assigned a pointer expression. In this case, the compiler generates a separate intrinsic call for the pointer-valued attribute assignment.

Valid mnemonics for file attributes are also type 3 reserved words in ALGOL. When it is not certain whether an identifier is a variable or a mnemonic, the compiler always assumes that the identifier is a variable or other local identifier if it is enclosed in parentheses. If the identifier is not enclosed in parentheses, the compiler always assumes it is a mnemonic. This rule can be used to resolve ambiguities when new attributes with names that conflict with local variables are added to the system.

Examples of MULTIPLE ATTRIBUTE ASSIGNMENT Statements

In the following example, at run time the BUFFERS attribute of file AFILE is assigned the value 3, the INTMODE attribute is assigned EBCDIC, and the KIND attribute is set to DISK.

AFILE(BUFFERS = 3, INTMODE = EBCDIC, KIND = DISK)

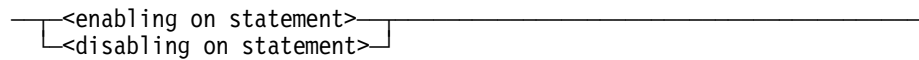
In the following example, at run time the TITLE attribute of file LINE is assigned the value pointed to by pointer P, and the INTNAME attribute is assigned the value pointed to by pointer Q.

LINE(TITLE = P, INTNAME = Q)

ON Statement

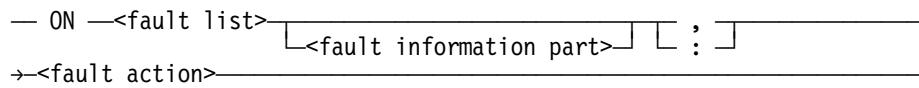
The ON statement is used to enable or disable an interrupt for one or more fault conditions.

<on statement>



Enabling ON Statements

<enabling on statement>



The two forms of enabling ON statements are the implicit call and the implicit branch. The implicit call form causes the fault termination of the block in which it appears. The implicit branch form permits the block in which it appears to continue to run.

Once an interrupt is enabled, it remains enabled until one of the following conditions occurs:

- The procedure or block that contains the ON statement is exited.
- The interrupt is explicitly disabled.
- A new interrupt is enabled for the same fault condition.

Whenever the block that contains an ON statement is exited, the interrupt status (enabled or disabled) for that fault condition reverts to the status it had just before the block was entered.

No call on the block exit intrinsic is required to deactivate the armed faults for a block.

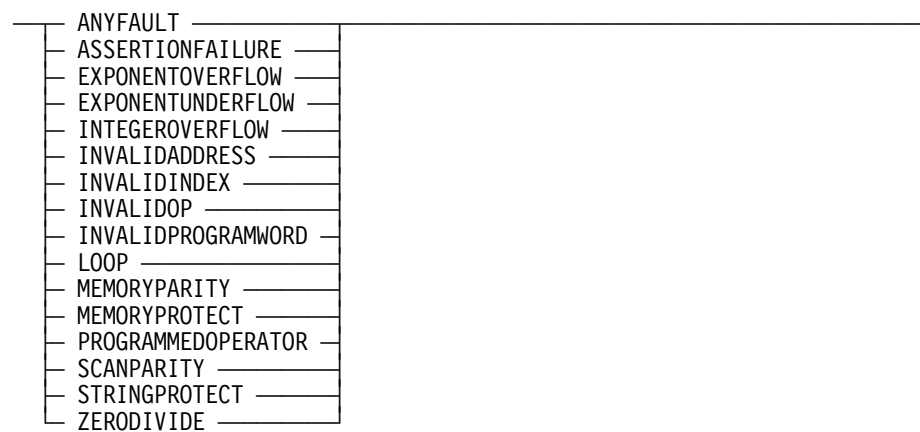
Each execution of an ON statement adds one stack cell to the block in which it is used.

Fault List

<fault list>



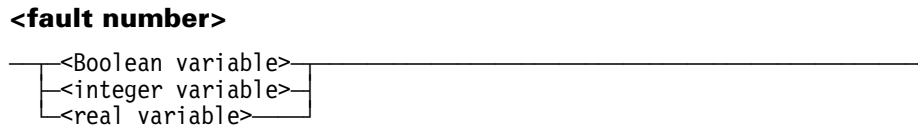
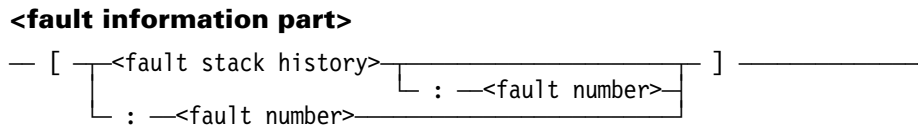
<fault name>



The fault list allows several fault interrupts to be enabled (armed) or disabled (disarmed) at the same time. When it is used to enable several interrupts, they all use the same fault action if any of the faults occur. The occurrence of any one of the faults in the fault list is sufficient to cause transfer of control to the fault action. The fault name ANYFAULT is used to enable or disable all faults.

Refer to the *Task Management Programming Guide* for more information about determining the internal cause of a fault.

Fault Information Part



The fault information part provides access to the stack history at the time of the occurrence of the fault and to the number corresponding to the fault kind. The variable that contains the fault number is assigned one of the following values when the corresponding fault occurs:

Value	Fault
1	DIVIDEBYZEROV
2	EXPOVERFLOWV
3	EXPUNDERFLOWV
4	INVALIDINDEXV
5	INTEGEROVERFLOWV
6	INACTIVEQV
7	MEMORYPROTECTV
8	INVALIDOPV
9	LOOPV
10	MEMORYPARITYV
11	SCANPARITYV
12	INVALIDADDRESSV
13	STACKOVERFLOWV
14	STRINGPROTECTV
16	FALSEASSERTV
17	SEQUENCEERRORV
18	INVALIDPCWV
19	STACKUNDERFLOWV
21	LIBLINKERRORV
22	INVALIDINTV
23	MEMFAIL1V
26	MEMORYFAIL2V

Value	Fault
30	PROCINTERNALV
35	PROCDIEDV
40	DISKPARITYV
41	EMODEVIOLATIONV
42	NOACTIVELINKV
43	PROCLINKPARITYV
45	BOTTOMOFSTACKV
46	RUNLIGHTOUTV
47	STACKSTRUCTUREV
48	BADMSCWV

Fault Stack History

<fault stack history>



If the fault stack history option is used, a string of EBCDIC characters representing the stack history is stored into the array row or the array specified by the pointer expression. The stack history information is always stored as EBCDIC characters regardless of the character type of the array row or pointer expression.

The format of the stack history is either of the following:

SSS:AAAA:Y,;SSS:AAAA:Y,;...;SSS:AAAA:Y.

SSS:AAAA:Y;(DDDDDDDD),;...;SSS:AAAA:Y;(DDDDDDDD).

In these formats, the following applies:

EBCDIC Characters	Meaning
SSS	A code segment number
AAAA	A code word address
Y	A code syllable number
1	A blank space
DDDDDDDD	A sequence number (present only if the compiler control option LINEINFO was TRUE during program compilation). The element DDDDDDDD can have the form DDDDDDDD/DDDDDDDD/... when INLINE procedures are referenced. The rightmost line number indicates the most recently invoked procedure reference.

ON Statement

One of these entries is generated for each activation record in the stack when the fault is encountered. Each entry is followed by a comma (,), and the last complete entry is terminated by a period (.). If the user-specified array is sufficiently long, the entire stack history is stored. If it is not long enough, then only a portion of the stack history is stored, with the last complete entry in the array terminated by a period. The code segment number field, SSS, is expanded to four characters, SSSS, for segment numbers greater than 4095; that is, for segment numbers whose hexadecimal representation requires four characters.

The array row or pointer expression that makes up the fault stack history and the variable that makes up the fault number are evaluated once when the ON statement is executed, and not at the time the fault occurs. Thus, in the following ON statement, array row A[I,*] is determined by the value of I at the execution of the ON statement and not when a ZERODIVIDE fault actually occurs. This determination is also true for the variables B[J] and J.

```
ON ZERODIVIDE[A[I,*]:B[J]]: GO TO ERROR_HANDLING
```

Fault Action

<fault action>

—<statement>—————|

The form of the ON statement that includes a comma, instead of a colon (:), before the fault action is the implicit call form. With this form of ON statement, when a specified fault occurs, the program calls the fault action statement as a procedure. If the fault action statement does a bad GO TO, the fault condition is discarded and the program continues running.

When a bad GO TO branches to a label outside the block in which the fault occurred, the block is terminated. When it branches out of the fault action statement into the block in which the fault occurred, the block continues to run.

If the fault action statement exits without doing a bad GO TO, the fault condition for which the fault action statement was called still exists. If an ON statement is enabled for that condition in a more global block, then control is passed to that ON statement; otherwise, the program is discontinued as a result of that fault.

A GO TO statement cannot be executed from outside the fault action statement to a label inside the fault action statement. Undefined results occur when a GO TO statement specifies a label passed as a parameter (a formal label).

The form of the ON statement that includes a colon, instead of a comma, before the fault action is the implicit branch form of the ON statement. With this form of ON statement, the program branches to the statement given as the fault action when a specified fault occurs. The fault condition is discarded, as though it had never happened, and the program continues execution at the first statement of the fault action. When there is no branch out of the fault action statement, the program flow continues with the next statement following it. When the ON statement is in the block in which the fault occurred, it permits that block to continue to run.

Disabling ON Statement

<disabling on statement>

```
— ON —<fault list>—————|
```

The disabling ON statement disables or disarms the interrupts corresponding to the fault names in the fault list. This has the same effect as if none of those interrupts had been enabled in the block in which it appears. It has no effect, however, on any interrupts that were enabled by ON statements in more global blocks, once this block is exited.

Note: *Excessive arming and disarming of faults within a single activation of the block can cause the stack limit of the program to be exceeded and the program to be terminated.*

Examples of ON Statements

In the following example, if either a divide-by-zero fault or an invalid index fault occurs at run time, the fault condition is discarded and control transfers to the compound statement in this ON statement. The stack history information is written to the array row FAULTARRAY, and the fault number of the fault that occurred is stored in FAULTNO.

```
ON ZERODIVIDE OR INVALIDINDEX [FAULTARRAY:FAULTNO]:
  BEGIN
  REPLACE FAULTARRAY[8] BY FAULTNO FOR * DIGITS;
  WRITE(LINE, 22, FAULTARRAY);
  REPLACE FAULTARRAY BY " " FOR 22 WORDS;
  CASE FAULTNO OF
    BEGIN
      1: DIVISOR := 1;
      4: INDEX := 100;
    END;
  GO BACK;
  END
```

In the following example, if either of the specified faults occurs at run time, the fault condition is discarded and control is transferred to the assignment statement in the ON statement. After execution of the assignment statement, execution continues with the statement following the ON statement.

```
ON MEMORYPROTECT OR LOOP: Q:= 2
```

The following example disables the interrupt associated with the exponent under flow fault.

```
ON EXPONENTUNDERFLOW % Disabling ON Statement
```

ON Statement

In the following example, if any fault occurs, the statement HANDLEFAULTS(Z) is called as a procedure. The stack history information is written to the location indicated by the pointer expression POINTR + 2, and the fault number of the fault that occurred is stored in Z.

```
ON ANYFAULT [POINTR + 2:Z],HANDLEFAULTS(Z)
```


OPEN Statement

The OPEN statement causes the referenced file or subfile to be opened.

<open statement>

```
— OPEN — ( —<open file part> —<open options> ) —————
```

<open file part>

```
—<file designator> —————
  |
  | [ — SUBFILE —<subfile index>— ] —
  |
  |<task designator>— . —<file-valued task attribute name>—
```

The OPEN statement can be used as an arithmetic function. For information on the values returned, see the *File Attributes Programming Reference Manual*.

The subfile index specifies the subfile to be opened. For more information on the subfile index, see "CLOSE Statement" earlier in this section. For more information on the file designator, see "SWITCH FILE Declaration" in Section 3, "Declarations." For more information on the task designator, see "TASK and TASK ARRAY Declarations" in Section 3, "Declarations."

OPEN Options

<open options>

```
—————
  |
  | , —<open control option>— | , —<associateddata option>—
  |
  | , —<connecttimelimit option>—
  |
  |>
```

<open control option>

```
—————
  | ATEND —————
  | AVAILABLE ———
  | AVAILATEND ———
  | CONDITIONAL ———
  | DONTWAIT ———
  | MUSTBENEW ———
  | OFFER —————
  | WAIT —————
```

Open Control Options

If no open control option is specified, the WAIT option is assumed. The WAIT option sets the current position to the beginning of the file. Use the ATEND option to set the current position to the end of the file.

Use the AVAILABLE option to prevent the OPEN operation from being suspended if the file cannot be opened. The AVAILABLE option sets the current position to the beginning of the file. Use the AVAILATEND option to set the current position to the end of the file.

The DONTWAIT and OFFER options are meaningful only for port files. Refer to the following examples for information about when to use these options.

The CONDITIONAL and MUSTBENEW options are meaningful only for disk files. When creating a new file, use either of these options to ensure that an existing file is not replaced. Use the MUSTBENEW option to cause the OPEN operation to fail if an existing file would be replaced. Use the CONDITIONAL option to cause the existing file to be opened instead of replaced.

ASSOCIATEDDATA Option

This option is only meaningful for port files and is used to send associated data with an OPEN request.

For more information on the ASSOCIATEDDATA option, see "CLOSE Statement" earlier in this section.

CONNECTTIMELIMIT Option

This option is only meaningful for port files and is used to specify the maximum amount of time (in minutes) that the system allows for a successful OPEN operation on a subfile.

For more information on the CONNECTTIMELIMIT option, see "AWAITOPEN Statement" earlier in this section.

Examples of OPEN Statements

The following example opens file FILEID. Execution of the program is suspended until FILEID is open.

```
OPEN(FILEID)
```

The following example opens subfile I of port file FILEID and offers it for matching. Control is returned to the program when it is determined whether the host can be reached.

```
OPEN(FILEID[SUBFILE I],OFFER)
```

The following example opens subfile I of port file FILEID and a dialog request is sent. The program is suspended until dialog establishment is complete.

```
OPEN(FILEID[SUBFILE I],WAIT)
```

The following example opens subfile I of port file FILEID. The AVAILABLE control option is a nonpreferred way of verifying OPEN WAIT with AVAILABLEONLY = TRUE. When AVAILABLEONLY = TRUE, the OPEN attempt fails if the correspondent endpoint cannot be reached immediately. If dialog can currently be established, the subfile is opened, the result returned by the OPEN statement is 1, and PROCESSOPEN is called; otherwise, an error result is returned and PROCESSOPEN is not called.

```
IF OPEN(FILEID[SUBFILE I],AVAILABLE) = 1 THEN PROCESSOPEN
```

The following example opens subfile 1 of port file FILEID. The program is suspended until a dialog is established or until T minutes have elapsed.

```
OPEN (FILEID [SUBFILE 1], WAIT, CONNECTTIMELIMIT = T)
```

The following example opens subfile 1 of port file FILEID. When the dialog request is sent, the information "MYDATA" is sent to the correspondent process as associated data.

```
OPEN (FILEID [SUBFILE 1], ASSOCIATEDDATA = "MYDATA")
```

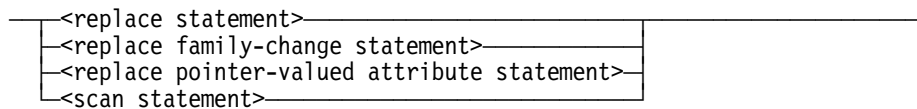
The following example opens subfile I of port file FILEID. Control returns to the program as soon as the open process has begun, because the DONTWAIT option is included. When a matching subfile is found, 14 characters of information are taken, beginning at the location pointed to by PTR; the 14 characters are sent to the correspondent process as associated data.

```
OPEN (FILEID [SUBFILE I], DONTWAIT, ASSOCIATEDDATALENGTH = 14,  
      ASSOCIATEDDATA = PTR)
```

POINTER Statement

Pointer statements are used to examine, transfer, and edit character data stored in arrays.

<pointer statement>



POINTER Statement Options

The REPLACE statement can be used to move character data into an array row. Within a single REPLACE statement, the character data to be moved can be taken from several sources. Each of these sources can be one of several different types. A source can be another array row, a string literal, the value of an arithmetic expression, the value of a string expression, or the value of a pointer-valued attribute. Furthermore, as the character data is moved from a source to the destination, the characters can be translated or edited. Also, an arithmetic expression source can be treated as a binary value and converted into the equivalent decimal number expressed as a string of numeric characters.

The REPLACE FAMILY-CHANGE statement is the language construct provided to add datacomm stations to or remove datacomm stations from a family of stations.

The REPLACE POINTER-VALUED ATTRIBUTE statement is the language construct provided to assign character data to pointer-valued file and task attributes.

The SCAN statement can be used to examine character data located in an array row.

POINTER statements process character data from left to right.

Temporary Storage

Many of the operations performed by POINTER statements require the use of temporary storage for intermediate results. In describing the actions of a POINTER statement, a discussion of how this temporary storage is initialized, changed, and disposed of is necessary. These discussions use the following names for these temporary storage locations:

- Stack-source-pointer
- Stack-destination-pointer
- Stack-auxiliary-pointer
- Stack-integer-counter
- Stack-test-character
- Stack-source-operand

The prefix, *stack*, denotes that none of these parameters correspond to any program variables. They exist only until execution of the **POINTER** statement is completed.

The *stack-source-pointer*, the *stack-destination-pointer*, and the *stack-auxiliary-pointer* have the same internal structure as a pointer variable that can be declared in a program. These temporary storage locations are initialized either from pointer expressions in the pointer statement or from previous corresponding temporary storage locations.

Stack-Source-Pointer

The initial value of the *stack-source-pointer* points to the first source character to be used by the associated operation. As the execution of the instruction progresses, the *stack-source-pointer* is modified to point to each successive source character. When the operation is complete, the *stack-source-pointer* points to the first unprocessed character in the source data (the process is determined by the particular form of the **POINTER** statement). This final value can be stored into a pointer variable, or it can be discarded.

Stack-Destination-Pointer

The initial value of the *stack-destination-pointer* points to the first destination character position to be used by the associated operation. As the execution of the operation progresses, the *stack-destination-pointer* is modified to point to each successive destination character position. When the operation is complete, the *stack-destination-pointer* points to the first unfilled character position in the destination. If more than one source is to be processed, the *stack-destination-pointer* value corresponding to the completed processing of one element in the source list is used as the initial value for the subsequent source. If no more sources are to be processed, this final value can be stored into a pointer variable, or it can be discarded.

Stack-Auxiliary-Pointer

The initial value of the *stack-auxiliary-pointer* points to the first entry in a table of data to be used by the operation in its execution. This table can be a translate table if the operation to be performed is extracting characters from the source data, translating the characters to different characters (possibly containing a different number of bits per character), and storing the translated characters in the destination. This table can be a truth set describing a particular set of characters if the operation to be performed requires a membership test. Finally, this table can be a picture: a table that contains instructions of a special type describing how the source data is to be edited before being stored in the destination.

Stack-Integer-Counter

The stack-integer-counter, when required by a POINTER statement, is initialized by an arithmetic expression supplied in the POINTER statement. The value of this arithmetic expression is integerized before it is used. The stack-integer-counter has different meanings depending on the type of POINTER statement involved. In some cases, the number of characters in a source string to be processed is dictated solely by this parameter. The number of numeric characters to be placed in the destination while converting the value of an arithmetic expression to character form is also dictated by the stack-integer-counter.

In some forms of the POINTER statement, two controlling factors exist that dictate how many characters are to be processed from a source string. One factor depends on the source data and is called a condition. The other factor is a maximum count contained in the stack-integer-counter and is provided by an arithmetic expression in the POINTER statement. For example, with such a POINTER statement, the following instructions could be written: *Translate characters from the source string to the destination until either 14 characters have been transferred or a period is encountered in the source string, whichever comes first.* The final value of the stack-integer-counter is available for storage, or it is discarded.

Stack-Test-Character

The stack-test-character is initialized by an arithmetic expression usually, but not necessarily, of the form of a single-character string, such as B. Although the stack-test-character parameter is one entire word of memory that contains the single-precision value of the arithmetic expression, only the rightmost character position of the word is used. When a condition employing a relational operator is used in a POINTER statement, the stack-test-character must contain the character against which the individual characters in the source string are to be compared.

Stack-Source-Operand

The stack-source-operand is used when the source data is given by the value of an arithmetic expression rather than a value located in an array row into which the stack-source-pointer points. The stack-source-operand is initialized by the arithmetic expression.

PROCEDURE INVOCATION Statement

A PROCEDURE INVOCATION statement causes a previously declared procedure to be executed as a subroutine.

<procedure invocation statement>

—<procedure identifier> —
<actual parameter part>
 ()

<actual parameter part>

— (—
<actual parameter> ,
<parameter delimiter>
) —

When a procedure is invoked, program control is transferred from the point of the PROCEDURE INVOCATION statement to the referenced procedure. When the procedure is completed, program control is transferred back to the statement following the PROCEDURE INVOCATION statement, unless a bad GO TO is executed in the referenced procedure. Bad GO TO statements are described in "GO TO Statement" earlier in this section.

A typed procedure returns a value. However, when a typed procedure is used in a PROCEDURE INVOCATION statement, this value is discarded.

Calling Procedures with Parameters

<actual parameter>

<expression>
<array designator>
<string array designator>
<connection block reference variable>
<connection library instance designator>
<direct file identifier>
<direct switch file identifier>
<event designator>
<event array designator>
<file designator>
<switch file identifier>
<format designator>
<switch format identifier>
<interlock designator>
<interlock array designator>
<label identifier>
<switch label identifier>
<list designator>
<switch list identifier>
<null value>
<picture identifier>
<procedure identifier>
<procedure reference array designator>
<procedure reference array element>
<procedure reference identifier>
<structure block array designator>
<structure block array element>
<structure block reference variable>
<structure block variable>
<task designator>
<task array designator>
<this intrinsic>

The actual parameter part of a PROCEDURE INVOCATION statement must have the same number of entries as the formal parameter list in the declaration of the procedure. Correspondence between the actual parameters and formal parameters is obtained by matching the parameters that occur in the same relative position in the two lists. Corresponding formal and actual parameters must be of compatible types. Parameters can be call-by-name or call-by-value.

For more information on procedures and formal parameters, refer to "PROCEDURE Declaration" in Section 3, "Declarations."

If a formal parameter is a call-by-name INTEGER or REAL simple variable, then the actual parameter can be either an INTEGER or a REAL expression; no type conversion is performed. If a formal parameter is a call-by-value INTEGER, REAL, or DOUBLE simple variable, then the actual parameter can be either an INTEGER, a REAL, or a DOUBLE expression, and automatic type conversion is performed on the actual parameter at the time the procedure is invoked.

If the formal parameter of a nonformal procedure is a simple variable of type COMPLEX, then the corresponding actual parameter can be of type INTEGER, REAL, DOUBLE, or COMPLEX. However, if the COMPLEX formal parameter is call-by-name and the corresponding actual parameter is not of type COMPLEX, an assignment to that formal parameter within the procedure body causes the program to be discontinued with a fault.

The types of actual and formal parameters must match exactly for all cases not mentioned above. For more information, see “Type Coercion of One-Word and Two-Word Operands” in Appendix C, “Data Representation.”

Imported events and event arrays cannot be used as actual parameters in a PROCEDURE INVOCATION statement.

Examples of PROCEDURE INVOCATION Statements

The following example invokes the procedure SIMPL, which has no parameters.

```
SIMPL
```

The following example invokes the procedure HEAVY and passes it four parameters: X, Y, the array row A[*], and the expression SQRT(BINGO+BASE).

```
HEAVY(X,Y,A[*],SQRT(BINGO+BASE))
```

PROCEDURE REFERENCE Statement

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

<procedure reference statement>

—<procedure reference variable> ————
 └─<actual parameter part>┘

<procedure reference variable>

—└─<procedure reference identifier> ————
 └─<procedure reference array element>┘

Using Procedure References

If the procedure reference variable is invoked while it is uninitialized or has had NULL assigned to it, the program is terminated with the message INVALID STACK ARGUMENT.

When a typed procedure reference variable is used in a PROCEDURE REFERENCE statement, the value returned by the procedure reference 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 identifier or procedure reference array. 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 variable in a PROCEDURE REFERENCE statement is equivalent to invoking the procedure directly in a PROCEDURE INVOCATION statement. For more information, see "PROCEDURE INVOCATION Statement" earlier in this section.

Imported events and event arrays cannot be used as actual parameters in a PROCEDURE REFERENCE statement.

Example of a PROCEDURE REFERENCE Statement

The following example assigns a reference to procedure SWAPPER in the first element of the procedure reference array PROCARRAY and to the procedure reference PROCREF. SWAPPER is then invoked through PROCARRAY and PROCREF.

```
BEGIN
*
*
*
REAL
  SORT1,
  SORT2,
  SORT3;

PROCEDURE REFERENCE ARRAY PROCARRAY[0:9] (A,B);
                                REAL A,B;
                                NULL;

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

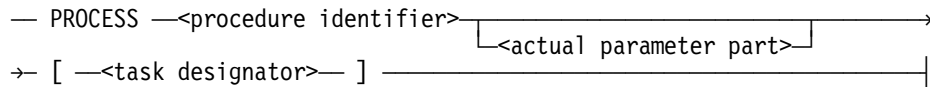
PROCEDURE SWAPPER(X,Y);
  REAL X,Y;
  BEGIN
  X :=: Y;
  END;

PROCARRAY[0] := SWAPPER;
PROCREF := SWAPPER;
READ (MYFILE, *, SORT1, SORT2);
IF SORT2 > SORT1 THEN
  PROCARRAY[0] (SORT1, SORT2);
READ (MYFILE, *, SORT3);
IF SORT3 > SORT2 THEN
  PROCREF(SORT2, SORT3);
END.
```

PROCESS Statement

The PROCESS statement initiates a procedure as an asynchronous process.

<process statement>



Initiation of an Asynchronous Process

Initiation of an asynchronous process consists of setting up a separate stack for the process, passing any parameters (call-by-name or call-by-value), and beginning the execution of the procedure. The initiating program continues execution, and both the initiating program and the initiated procedure run in parallel.

If the specified procedure is a typed procedure, the return value is discarded.

If the procedure identifier is a system supplied process, such as an intrinsic, the library GENERALSUPPORT must be declared using a library entry point specification. The procedure identifier must be declared in the program or the syntax error *PROCEDURE MUST BE USER DECLARED* results.

The actual parameter part must agree in number and type with the formal parameter part in the declaration of the procedure; otherwise, a run-time error occurs.

The task designator associates a task with the process at initiation; the values of the task attributes of that task, such as COREESTIMATE, STACKSIZE, and DECLARED PRIORITY, can be used to control execution of the process. For information about assigning values to task attributes, refer to "Task Assignment," <arithmetic task attribute> under "Arithmetic Assignment," and <Boolean task attribute> under "Boolean Assignment" earlier in this section. Many task attributes can be interrogated while the process is running.

Critical Block

An asynchronous process depends on its initiator for global variables and call-by-name actual parameters. Thus, for each process, a critical block is present in the initiator that cannot be exited until the process is terminated. The critical block is the block of highest lexical level that contains one or more of the following items:

- The declaration of the procedure itself
- The declarations of the actual parameters passed to the call-by-name formal parameters
- The declaration of the task designator
- Any compiler-generated code for evaluating arithmetic expressions passed to call-by-name parameters

The critical block can be the block that contains the PROCESS statement, the outer block of the program, or a block in between. An attempt by the initiator to exit the critical block before the process is terminated causes the initiator and all tasks it has initiated through CALL or PROCESS statements to be terminated.

A process is terminated by exiting its own outermost block or by execution in the initiator of the following statement where the task designator specifies the task associated with the process to be terminated:

```
<task designator>.STATUS := VALUE(TERMINATED)
```

Note: A processed procedure must not declare an OWN array or reference another procedure that declares an OWN array. An attempt to do so results in a run-time error. A string expression cannot be passed as an actual parameter to a call-by-name parameter of a procedure in a PROCESS statement.

When an item of a structure or connection block is used in the PROCESS or CALL statement, either as the procedure being invoked, an actual parameter passed to a call-by-name formal parameter, or the task designator, the calculation for the block that contains the highest lexical level includes the lexical level of the block that declared the structure block variable, structure block array, structure or connection block reference, or connection library used to qualify the structure or connection block item.

Examples of PROCESS Statements

In the following example, the procedure AGENT, which has no parameters, is invoked as an asynchronous process. The task TSK is associated with the process.

```
PROCESS AGENT [TSK]
```

In the following example, the procedure ACHILD is invoked as an asynchronous process and passed the three parameters OUTARRAY, YOUREVENT[INDX], and COUNT. The task designated by TSKARAY[INDX] is associated with the process.

```
PROCESS ACHILD(OUTARRAY, YOUREVENT[INDX], COUNT) [TSKARAY[INDX]]
```

PROCURE Statement

The PROCURE statement tests the available state of an event.

<procure statement>

— PROCURE — (—<event designator>—) _____|

Testing the Available State

If the available state of the event is FALSE (not available), the program is suspended and put in the procure list until some other task executes the LIBERATE statement for that event. If the available state of the event is TRUE (available), the available state is set to FALSE (not available), and the program continues execution with the statement following the PROCURE statement.

Sharing Resources Among Programs

The PROCURE statement provides a means for different programs to share resources. For example, a convention could be established that a certain shared resource that is available for use by more than one program is not to be used by a program unless that program has procured the event that is used as the interlock. When the program has completed its use of the resource, it should execute a LIBERATE statement on the event.

Examples of PROCURE Statements

In the following example, if the available state of EVNT is TRUE (available), EVNT is procured by setting its available state to FALSE (not available). Otherwise, the program is suspended until EVNT is made available.

```
PROCURE(EVNT)
```

In the following example, if the available state of the event designated by EVNTARAY[INDX] is TRUE (available), then that event is procured by setting its available state to FALSE (not available). Otherwise, the program is suspended until the event designated by EVNTARAY[INDX] is made available.

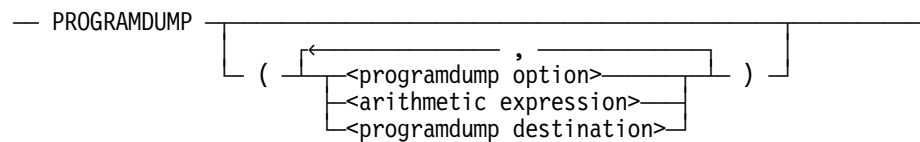
```
PROCURE(EVNTARAY[INDX])
```

PROGRAMDUMP Statement

The PROGRAMDUMP statement can be used to generate a program dump. After the dump is taken, the program continues and executes the next statement.

A program dump is an expanded listing of the internal stack as it existed when the dump was requested. Several options are available to specify which items of the stack are to be included in the dump.

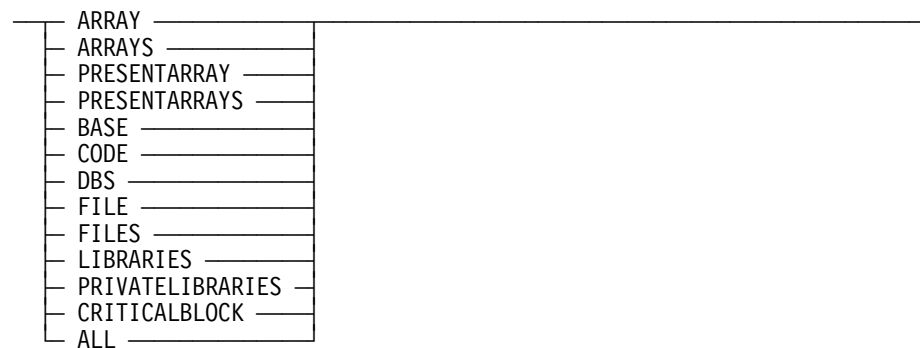
<programdump statement>



The information produced by the PROGRAMDUMP statement is written to the file specified by the TASKFILE task attribute of the program, unless the TODISK destination option is specified. See the discussion of the TODISK destination option later in this section.

PROGRAMDUMP Options

<programdump option>



PROGRAMDUMP Statement

The information included in the dump depends on the options specified. If no program dump options are specified, the stack is dumped according to the specifications in the task attribute OPTION of the program. The following table describes the results of specifying each program dump option:

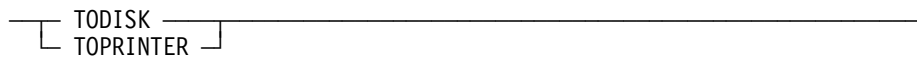
Option	Result
ARRAY or ARRAYS	Causes the contents of all arrays declared in the program to be dumped.
PRESENTARRAY or PRESENTARRAYS	Causes only the arrays in the present state (at the time the program dump is taken) to be dumped. If the ARRAY or ARRAYS program dump option is set along with the PRESENTARRAY or PRESENTARRAYS program dump option, all arrays will be dumped.
BASE	Causes the base of the stack to be dumped. The operating system uses a portion of each stack to contain various words needed to control, identify, and log the program. If the TODISK option is also specified, the base of the stack and the program information block (PIB) are always dumped for any stack dumped.
CODE	Causes segment dictionary information to be included in the dump. The actual code is dumped only for segments that have been referenced by the program when the program dump occurs. Value arrays in the segment dictionary are dumped when both the CODE option and either the ARRAY or ARRAYS option are specified.
DBS	Causes the output of database stacks to be dumped.
FILE or FILES	Causes information about each file declared in the program to be dumped. For each file, each word of the file information block (FIB) is separately named and, in some cases, analyzed.
LIBRARIES	Causes the stacks of all libraries that are being used by the program to be dumped.
PRIVATELIBRARIES	Causes the stacks of all private libraries that are being used by the program to be dumped.
CRITICALBLOCK	Causes the stack that contains the critical block visible to the calling stack to be dumped. Dumping starts at the offset specified by the critical block reference in the Process Information Block of the calling stack.
ALL	Equivalent to specifying all the other options. The ALL option has no effect on the program dump destination. If the TODISK or TOPRINTER destination options are needed, they must be explicitly mentioned.

If the arithmetic expression option is used, the value in the arithmetic expression corresponds to the bit values in the OPTION word. The value of the expression is interpreted as follows:

Value	Meaning
[7:1] = 1	The base of the user stack is dumped.
[8:1] = 1	Array contents are dumped.
[9:1] = 1	The segment dictionary is dumped.
[10:1] = 1	Files are dumped.
[11:1] = 1	Present array contents are dumped.
[15:1] = 1	Database stacks are dumped.
[19:1] = 1	Stacks for libraries that the program is linked to are dumped.
[20:1] = 1	Stacks for private libraries are dumped.
[23:1] = 1	Destination of the dump is the disk.
[24:1] = 1	Destination of the dump is the printer.
[25:1] = 1	The stack containing the critical block is dumped.

Programdump Destination Options

<programdump destination>



When the TODISK option is specified and a program dump is taken, a disk file is created in a format acceptable to DUMPANALYZER. The listing normally produced by PROGRAMDUMP is suppressed.

The program dump file name is as follows. The portion of the task name included in the program dump file name is limited to eight nodes. The date format is YYMMDD, and the time format is HHMMSS.

PDUMP/<task name>/<date>/<time>/<mix number>

In a program dump to disk, the base of the stack and the PIB are always dumped. All other program dump option settings work the way they work in a dump to the printer.

If the TODISK option and the TOPRINTER option are set, a program dump to the printer is taken after the disk file program dump has been produced. The two dumps might not be identical because the dump to disk has some side effects that might change the contents of memory.

The current destination option can be overridden by either destination option.

If neither destination option is specified, the destination is determined by the PDTODISK system options.

Relation to OPTION Task Attribute

Options specified in the PROGRAMDUMP statement apply only to the program dump taken at that time and temporarily override the values specified in the OPTION word of the program. The bits of the OPTION word are set with the OPTION task attribute for the program. Refer to the *Task Attributes Programming Reference Manual* for information about the OPTION task attribute.

A program dump taken with the PROGRAMDUMP statement has an advantage over a program dump taken with the OPTION task attribute. A program dump can be taken with the OPTION task attribute only upon a fault or discontinue condition. A program dump taken with the PROGRAMDUMP statement can be taken at any time, and the program can continue after the dump is taken. For example, the PROGRAMDUMP statement might be useful as part of an ON statement, within an INTERRUPT statement, or within a piece of newly developed code.

The PROGRAMDUMP statement displays identifier name and compiler class information along with the stack variables when binding information (bindinfo) is present in the code file. ALGOL generates binding information by default unless the program is compiled with the compiler control option NOBINDINFO set to TRUE.

Retrieval of Binding Information

When the BEGINSEGMENT and ENDSEGMENT compiler control options are used, a situation can occur where the binding information cannot be retrieved. This situation is related to two factors:

- Two or more variables have the same address. Normally there is no conflict when this happens because the addresses relate to different code segments.
- Procedures that are encountered between a BEGINSEGMENT and ENDSEGMENT option are placed in the same code segment.

When these two conditions occur, and multiple variables have the same address within the same code segment, the compiler cannot retrieve the binding information for those variables.

Diagnostic and debugging information also can be written to the TASKFILE so that the program dump and the information can be coordinated.

Examples of PROGRAMDUMP Statements

The following example analyzes and prints the program stack according to the value of the OPTION task attribute of the program.

```
PROGRAMDUMP
```

The following example analyzes and prints the basic information plus the contents of all arrays.

```
PROGRAMDUMP (ARRAYS)
```

The following example analyzes and prints the contents of arrays, value arrays, the base of the stack, the segment dictionary, referenced code segments, and files.

```
PROGRAMDUMP (ARRAYS, BASE, CODE, FILE)
```

The following example analyzes the maximum amount of information about the program stack. The program dump is written to the printer.

```
PROGRAMDUMP (ALL, TOPRINTER)
```

The following example analyzes and prints the program stack according to the value of DUMPPARAM.

```
PROGRAMDUMP (DUMPPARAM)
```

The following example is equivalent to the statement PROGRAMDUMP(FILE). This statement analyzes and prints the contents of files of the program.

```
PROGRAMDUMP (0 & 1 [10:1])
```

PROGRAMDUMP Statement

The following example analyzes the basic information plus the contents of all arrays and database stacks. The information is written to a disk file.

```
PROGRAMDUMP (ARRAYS,DBS,TODISK)
```

The following example analyzes the information specified in the OPTION word of the program because no program dump option was specified in the statement. The dump is written first to the disk file and then to the printer file.

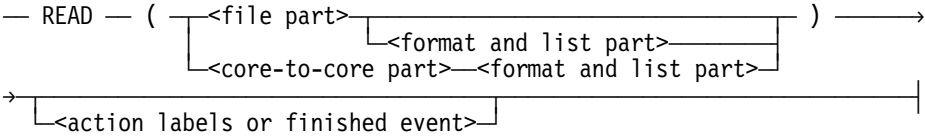
```
PROGRAMDUMP (TODISK,TOPRINTER)
```

READ Statement

The READ statement allows data to be read from files and assigned to program variables.

Note: The syntax of the READ statement and the syntax of the WRITE statement are nearly identical. Differences in the semantics are discussed following the syntax for each statement.

<read statement>



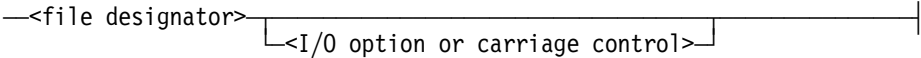
The action of the READ statement depends on the form of the <file part> element or <core-to-core part> element and on the form of the <format and list part> element. If the file was declared as a direct file, the <record number or carriage control> part interpretation is governed in ways specific to the device type; this is less general than for nondirect files.

The file part or the core-to-core part specifies the location of the data to be read.

The READ statement can be used as a Boolean function. When the read operation fails, the value TRUE is returned. When the read operation succeeds, the value FALSE is returned. The READ statement returns a value identical to that returned by the file attribute STATE. For more information, refer to the discussion of the STATE attribute in the *File Attributes Programming Reference Manual*.

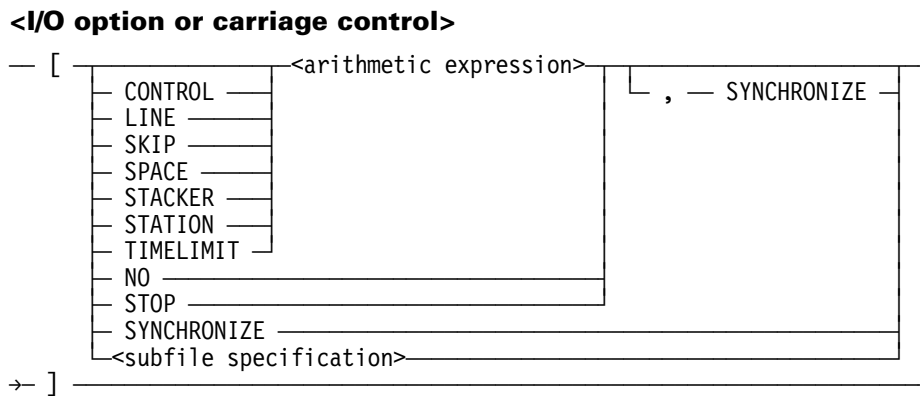
File Part

<file part>



The file designator specifies the file to be read. For more information on the file designator, refer to "SWITCH FILE Declaration" in Section 3, "Declarations."

I/O Option or Carriage Control



If the I/O option or carriage control element is not specified, the record currently addressed by the record pointer is read, and the record pointer is adjusted to point to the next record in the file.

If the I/O option or carriage control element is invalid for the physical file associated with the file designator, it is in general ignored. If the file was declared as a direct file, the I/O option or carriage control is overridden by the limitations of the particular I/O device type.

If the I/O option or carriage control element is an arithmetic expression, its value indicates the zero-relative record number of the record in the file that is to be read. The record pointer is adjusted to point to the specified record before the read is performed, and the record pointer is adjusted after the read operation to point to the next record.

The *[CONTROL <arithmetic expression>]* construct is meaningful only for KIND=LBP direct files. The construct *[CONTROL 1]* is a READ modifier used to perform a Read Information operation.

If the I/O option or carriage control element is NO, then the record pointer is not adjusted following the read operation. That is, the record can be read again. This I/O option or carriage control element has no effect if the KIND attribute of the file being read is equal to REMOTE.

If the I/O option or carriage control element is of the form *[SPACE <arithmetic expression>]*, then the number of records specified by the value of the arithmetic expression are skipped. Spacing is forward if the arithmetic expression has a positive value and backward if the arithmetic expression has a negative value.

The *[TIMELIMIT <arithmetic expression>]* construct, which is meaningful only for remote files, assigns the value of the arithmetic expression to the TIMELIMIT attribute of the file. Refer to the *File Attributes Programming Reference Manual* for information on the TIMELIMIT attribute. The value of this attribute applies to all subsequent READ and WRITE statements on that file. If the value of the TIMELIMIT attribute is greater than zero and if no input is received within that number of seconds (the value can be fractional), then a time-out error is reported.

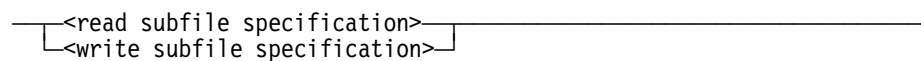
The *[STATION <arithmetic expression>]* construct is meaningful only for remote files. The value of the arithmetic expression is assigned to the LASTSUBFILE attribute of the file. Refer to the *File Attributes Programming Reference Manual* for information on the LASTSUBFILE attribute.

The *[SYNCHRONIZE]* construct is meaningful for the WRITE statement only.

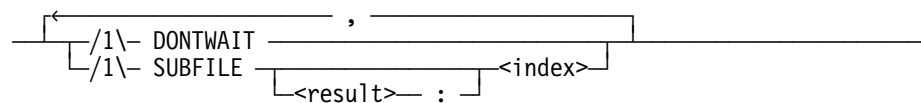
Imported events and event arrays cannot be used as <event designator> s in a direct I/O READ statement.

Subfile Specification

<subfile specification>



<read subfile specification>



<result>



If the file to be read is a port file (a file for which the KIND attribute is equal to PORT), an array row read containing a subfile specification must be used. For more information, refer to “Array Row Read” later in this section.

The subfile specification is meaningful only for port files. It is used to specify the subfile to be used for the read operation and the type of read operation to be performed.

If the subfile index is used, the value of the subfile index is assigned to the LASTSUBFILE attribute of the file. It specifies the subfile to be used for the read operation. If the subfile index is zero, a nonselective read is performed. If the subfile index is nonzero, then a read from the specified subfile is performed. The result variable, if any, is assigned the resultant value of the LASTSUBFILE attribute. For more information on the LASTSUBFILE attribute, refer to the *File Attributes Programming Reference Manual*.

If DONTWAIT is specified in a READ statement, and if no input is available, no data is returned and the program is not suspended.

Core-to-Core Part

<core-to-core part>

—<core-to-core file part> [<core-to-core blocking part>]

<core-to-core file part>

[<array row> | <pointer expression> | <subscripted variable>]

If the core-to-core part is specified in the READ statement, then a core-to-core read is performed. A core-to-core read operation reads from a location in memory, not from a physical device; therefore, it is much faster than a physical read. Editing is performed exactly as it is performed when reading from a physical device.

If the core-to-core file part is a hexadecimal, or EBCDIC array row or pointer, then the default record size (the number of characters considered to be in the record) depends on the character size of the array row or pointer and is determined by the actual length of the designated string.

The maximum size of the core-to-core file part for hexadecimal arrays is 65,535 words. Core-to-core I/O on hexadecimal arrays longer than 65,535 words is permitted only if the core-to-core file part is indexed far enough into the array such that the length between that point and the end of the array does not exceed 65,535 words. If an attempt is made to use an array or array segment more than 65,535 words long, a run-time error occurs.

For single-precision and double-precision array rows or subscripted variables, the default record size is computed by multiplying the length of the array row (or remaining length of the array row when a subscripted variable is used) by the number of characters per word. The characters per word is 6 for single-precision and 12 for double-precision.

Core-to-Core Blocking Part

<core-to-core blocking part>

— (—<core-to-core record size> [, —<core-to-core blocking>])

<core-to-core record size>

—<arithmetic expression>

<core-to-core blocking>

—<arithmetic expression>

To specify a record size smaller than the default size, a value can be provided for core-to-core record size. This value is in terms of characters. By supplying a value for core-to-core blocking, the file can be blocked into more records than the default number, which is one.

With formatted I/O, if the format requires more records than indicated by the core-to-core blocking value, a run-time error is given. Also, the format can require more characters than the core-to-core file part contains; this situation also results in a run-time error. In such cases, the number of characters indicated in the core-to-core blocking part (this number is computed by multiplying the core-to-core record size by the core-to-core blocking) can appear to be large enough to satisfy the format, but the core-to-core blocking part can indicate more characters than the core-to-core file part actually contains. The core-to-core file part, the core-to-core blocking part, and the format must be compatible or run-time errors occur.

For example, the following statements result in errors:

```

BEGIN
  ARRAY A[0:9];
  REAL B,C;
  READ (A(80),<T50,A6,I10>,B,C);      % Example 1
  WRITE(A(15,3),<X5,I15>,1,2,3);      % Example 2
  WRITE(A(20,2),<X5,I15>,1,2,3);      % Example 3
  B := " ITEM";
  WRITE(A(15,4),<" .",X2,A6,I2,X4>,B,1,B,2,B,3,B,4);  % Example 4
END.

```

The statement labeled "Example 1" in the preceding program results in a run-time error (format error 217), because the format requires 65 characters, but the file part (array A) contains only 60 characters.

The statement labeled "Example 2" results in a run-time error (format error 117), because the format requires 20-character records, but 15-character records were specified in the blocking part.

The statement labeled "Example 3" results in a run-time error (format error 120), because the three list elements require three repetitions of the format. Thus, three records are required, but only two records were specified in the blocking part.

The statement labeled "Example 4" fills array A with the following EBCDIC data ("I" denotes the end of the data):

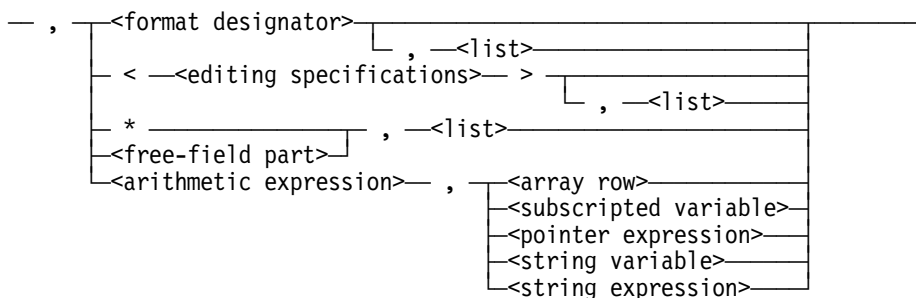
```

.  ITEM 1  .  ITEM 2  .  ITEM 3  .  ITEM 4  |

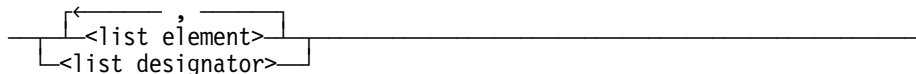
```

Format and List Part

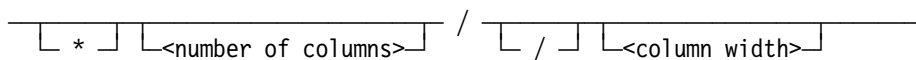
<format and list part>



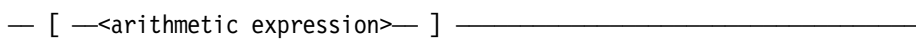
<list>



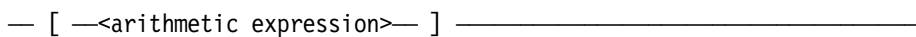
<free-field part>



<number of columns>



<column width>



The format and list part element indicates the interpretation of the data in the file and the variables to which the data is assigned.

If the format and list part element does not appear, the input record is skipped.

Formatted Read

A READ statement that contains a format designer, editing specifications, or a free-field part is called a formatted read.

A format designer without a list indicates that the referenced format contains a string literal into which corresponding characters of the input data are to be placed. The string literal in the FORMAT declaration is replaced by the string literal in the input data.

A format designer with a list indicates that the input data is to be edited according to the specifications of the format and assigned to the variables of the list.

Editing specifications can appear in place of a format designer and have the same effect as if they had been declared in a FORMAT declaration and had been referenced through a format designer. For more information, refer to "FORMAT Declaration" in Section 3, "Declarations."

On any formatted I/O statement (excluding core-to-core I/O), the number of characters allowed in the record is determined solely by the value of the file attribute MAXRECSIZE of the file. If the format requires more characters than are contained in the record, a format error occurs at run time.

The free-field part is discussed under "Data Format for Free-field Input" later in this section.

Binary Read

A READ statement of the following form is called a binary read:

```
READ(<file part>,*,<list>)
```

An asterisk (*) followed by a list specifies that the input data is to be processed as full words and assigned to the elements of the list without being edited. The number of words read is determined by the number of elements in the list or the maximum record size, whichever is smaller.

When data is read into character arrays, only full words are read. If there is a partial word left at the end of the data, it is ignored. For example, if A is an EBCDIC array and FILEID contains the string *12345678*, the following statement reads only the characters *123456*:

```
READ(FILEID,*,A)
```

When a string is read into a string variable using a binary READ statement, the first word read from the record is assumed to specify the length of the string. This word is evaluated, and the resulting value is the number of characters read beginning with the next word of the record. The binary WRITE statement automatically writes a word of length information before the text of each string variable; therefore, the following WRITE statement can later be read by the following READ statement:

```
WRITE(F,*,STR,STRARRAY[5],STR || "ABC")
```

```
READ(F,*,STR1,STR2,STRARRAY[0])
```

For more information, see "Binary Write" under "WRITE Statement" later in this section.

The results are undefined for binary READ statements that attempt to read data not containing length information into string variables.

Array Row Read

A READ statement of any of the following forms is called an array row read:

```
READ(<file part>,<arithmetic expression>,<array row>)
READ(<file part>,<arithmetic expression>,<subscripted variable>)
READ(<file part>,<arithmetic expression>,<pointer expression>)
READ(<file part>,<arithmetic expression>,<string variable>)
```

The first three forms of the array row read specify that input data is to be read without editing and assigned left-justified to the array specified by the array row, subscripted variable, or pointer expression. The arithmetic expression specifies the number of words or the number of characters, depending on the value of the FRAMESIZE attribute for the file, to be read. Refer to the *File Attributes Programming Reference Manual* for information on the FRAMESIZE attribute. The number of words or characters actually read equals whichever of the following values is smallest:

- The MAXRECSIZE attribute of the file being read
- The length of the array row (or portion of the array to the right of where the pointer expression points or to the right of the element specified by the subscripted variable)
- The absolute value of the arithmetic expression

A READ statement of the following form specifies that input data is to be read without editing and assigned to the string variable:

```
READ (<file part>,<arithmetic expression>,<string variable>)
```

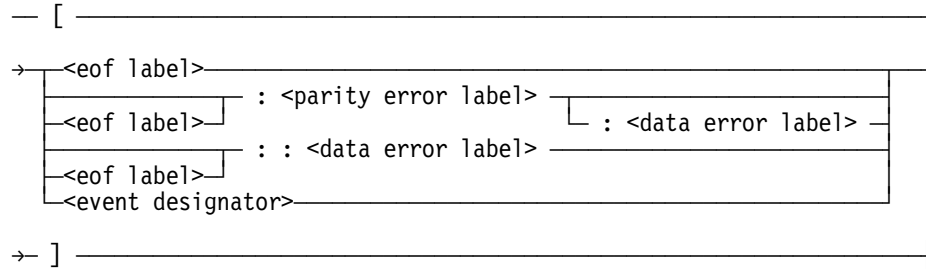
The number of characters read is the smaller of the value of the MAXRECSIZE attribute of the file being read or of the absolute value of the arithmetic expression. The value of the arithmetic expression always specifies the number of characters (not words) to be read.

The following is an example of an array row read:

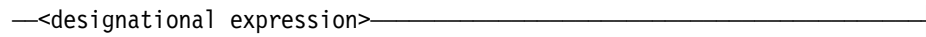
```
BEGIN
  FILE IN(TITLE="TEST.", UNITS=CHARACTERS, MAXRECSIZE=20);
  STRING S1,S2;
  READ(IN,15,S1); % READS 15 CHARACTERS INTO S1
  READ(IN,50,S2); % READS 20 CHARACTERS INTO S2
END.
```

Action Labels or Finished Event

<action labels or finished event>



<eof label>



<parity error label>



<data error label>



The action labels or finished event element provides a means of transferring control from a READ statement, WRITE statement, or SPACE statement when exception conditions occur. A branch to the eof label takes place when an end-of-file condition occurs. A branch to the parity error label takes place if an irrecoverable parity error is encountered. A branch to the data error label takes place if a conflict exists between the format and the data. If the appropriate label is not provided when an exception condition occurs, the program is terminated.

The [<event designator>] syntax can be used only for direct I/O. The event is caused when the I/O operation is finished. For more information, refer to "Direct I/O" under "I/O Statement" earlier in this section.

Exception conditions occurring during a READ statement can also be handled without the use of the action labels or finished event syntax. The READ statement can be used as a Boolean function, and the value returned can be tested to determine if any exception conditions exist. For more information, refer to the discussion of the STATE attribute in the *File Attributes Programming Reference Manual*. When exception conditions are handled in this manner, the action labels or finished event syntax cannot be used. The user assumes all responsibility for handling exception conditions. Core-to-core I/O statements of the following forms cannot be used with the action labels or finished event syntax and cannot be used as Boolean functions.

```
READ(<array row>,<arithmetic expression>,<array row>)
WRITE(<array row>,<arithmetic expression>,<array row>)
```

Attempting to do either results in a syntax error.

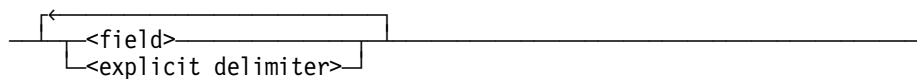
Data Format for Free-Field Input

The use of a free-field part element in a READ statement allows input to be performed with editing but without using editing specifications. The appropriate format is selected automatically.

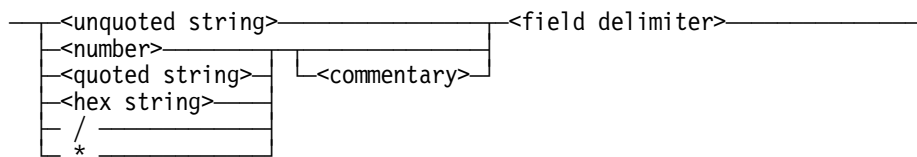
On input, only the simplest forms of the free-field part, a single slash (/) or double slash (/ /), can be used. These formats allow input from records in the form of free-field data records. A single slash indicates that data items are delimited by a comma; a double slash indicates that data items are delimited by one or more blanks.

Free-Field Data Format

The format of a free-field input data record is as follows:



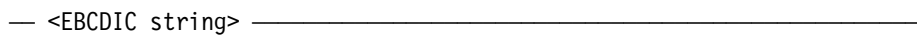
<field>



<unquoted string>

Any string not containing an <explicit delimiter>.

<quoted string>



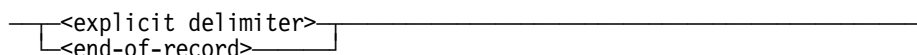
<hex string>



<commentary>

Any string not containing an <explicit delimiter>.

<field delimiter>



<explicit delimiter>

Comma (,) for the single-slash form or one or more blanks for the double-slash form. An empty record is not considered an explicit delimiter.

<end-of-record>

The end of the input record.

Each record of free-field input data must be in the form described earlier.

Empty records are ignored. The commentary option is ignored.

Each field except the slash is associated with the list element to which it corresponds by position.

Fields

The single-slash format interprets a field that contains only a comma or a comma preceded by blanks as a null field. Such a field is skipped along with its associated list element, which is left unaltered.

The different types of fields are described in the following paragraphs.

Unquoted String

If an unquoted string is read into a list element of type string or pointer, all characters preceding the explicit delimiter (including quotation marks if present) are transferred to the list element. The end-of-record is not recognized as a delimiter.

If an unquoted string is read into a list element of type string, characters are read until an explicit delimiter is detected or until the maximum string length ($2^{*}5 - 2$) is reached.

If an unquoted string is read into a list element of type pointer, characters are read until an explicit delimiter is detected or until the end of the array is reached.

If an unquoted string is read into a list element of type Boolean, the value TRUE is assigned to the list element if the first character of the string is T. If the first character is not the letter T, the value FALSE is assigned to the list element. The unquoted string is read until a field delimiter is detected.

If an unquoted string is read into a list element of any type other than string, pointer, or Boolean, it is treated as commentary.

Number

A number that is represented as an integer is treated as type INTEGER unless it is larger than the largest allowable integer, in which case it is treated as type REAL. Numbers that contain a decimal fraction are treated as type REAL. However, when the list element is double-precision, results are treated as type DOUBLE. When the field delimiter is a comma, blanks within numbers are ignored.

Complex values are divided into real and imaginary values. When a complex variable or complex subscripted variable appears in the list of a free-field READ statement, two fields are necessary to complete the read operation. The value in the first field is assigned to the real part, and the value in the second field is assigned to the imaginary part.

READ Statement

Quoted String

A quoted string of any length can be read into single-precision or double-precision list elements. Each single-precision EBCDIC list element receives six characters (12 characters for double-precision list elements), until either the list or the string is exhausted. If the number of characters in the string is not a multiple of six (for EBCDIC) then the last list element receives the remaining characters of the string. The string characters are stored, right-justified, in the list elements.

Hex String

A hexadecimal string can be read into a single-precision or double-precision list element. If fewer than 12 hexadecimal digits are read into a single-precision variable (or fewer than 24 hexadecimal digits into a double-precision variable), the string is stored right-justified in the variable. If a minus sign precedes the string (for example, `-4"A`), bit 46 of the resulting value is complemented.

Slash (/)

The slash field causes the remainder of the current buffer to be ignored. The buffer following the slash is considered the beginning of a new field. The slash is a field by itself and must not be placed within another field or between a field and its explicit delimiter.

Asterisk (*)

The asterisk field terminates the READ statement. The program continues with the statement following the READ statement. The list element corresponding to the asterisk remains unchanged, as do any subsequent elements in the list.

Examples of Fields

1,
2.5, / anything to the right of a slash is ignored
2.48 @ -20, / blanks are ignored if using single-slash editing
3 4 / two data elements if the delimiter is a blank
3,4, / two data elements if the delimiter is a comma
"THIS IS A QUOTED STRING"
THIS IS AN UNQUOTED STRING AND THE DELIMITER IS A COMMA, 123
THIS-IS-AN-UNQUOTED-STRING-AND-THE-DELIMITER-IS-A-BLANK 456
2.5 ANY COMMENT OR NOTE NOT CONTAINING A COMMA,
4"AB" / A HEX STRING
-4"40000000000A" / BIT 46 IS COMPLEMENTED, THE RESULT = +10
,,, / null fields; the three corresponding list elements are
/ skipped with no alteration to their contents.
4, ,5 / null field is ignored
* THIS DATA RECORD TERMINATES THE READ STATEMENT

Examples of READ Statements

```
READ(FILEID)
READ(FILEID,FMT)
READ(FILEID,FMT,LISTID)
READ(FILEID,*,LISTID)
READ(SPOFILE,FMT,A,B,C)
READ(SPOFILE,/,SIZE,LENGTH,MASS)
READ(FILEID,FMT,7,2,A,B,C,ARRAY[A],B+C,F)
READ(FILEID,/,J,FOR I:= 0 STEP 1 UNTIL J DO ARRAY[I])
READ(FILEID,*,A,B,C,FOR A:= B*A STEP C UNTIL J DO ARY[I])
READ(SWFILEID[IF X > N THEN X+N ELSE 0],25,ARRAY[2,*])
READ(FILEID,/,SWLISTID[I])
READ(FILEID,FMT,SWLISTID[I])
READ(SPOFILE,SWFMT[16],A,B,C)
READ(FILEID,50,STR)
READ(FILEID,/,L,M,N,ARRAY[2]) [EOFL]
READ(FILEID[3][NO]) [:PARL]
READ(SWFILEID[14][NO],FMT,A+EXP(B),ARRAY[I,J,*]) [:PARSWL[M]]
READ(FILEID[NO],SWFMT[6+J],LISTID) [EOFSWL[Q*3]::DATAERRORL]
READ(SWFILEID[A+B],*,SWLISTID[2+H/K]) [EOFL:PARL]
READ(FILEID[NO]) [EOFSWL[I]:PARSWL[J]]
READ(FYLE) [EOFL:PARL:DATAERRL]
READ(DIRFYLE) [EVNT]
READ(DIRFYLE,30,DIRARRAY) [EVNT]
```

REMOVEFILE Statement

The REMOVEFILE statement removes files without opening them.

<removefile statement>

```
— REMOVEFILE — ( —<directory element>— ) —————|
```

Directory Element

The syntax and semantics of the directory element appear under “CHANGEFILE Statement” earlier in this section.

If the directory element is a directory name, all files in that directory are removed. If the directory element is both a file name and a directory name, that file and all files in the directory are removed.

A directory element of the form *<file name>/=* removes only files in that directory. It does not remove a file named *<file name>*.

If a pointer expression is used as a directory element, it must point to an array that contains the name of the file or directory to be removed.

REMOVEFILE Statement as a Boolean Function

The REMOVEFILE statement can be used as a Boolean function, in which case it returns a value of TRUE if an error occurs. The value in field [39:20] of the result defines the failure as follows:

Value	Meaning
10	File name or directory name is in error.
30	Files have not been removed.

Family Substitution

Family substitution is used if the task has an active family specification and the family name involved in the REMOVEFILE statement is the target family name that the FAMILY specification substitutes.

If a family substitution specification is in effect, the REMOVEFILE statement affects only the substitute family, not the alternate family.

REMOVEFILE Statement

Example of a REMOVEFILE Statement

The following statement removes the file MYTEST and, if the remove is successful, assigns FALSE to the variable BOOL.

```
BOOL := REMOVEFILE("MYTEST ON PACKFOUR.")
```

REPLACE Statement

The REPLACE statement causes character data from one or more sources to be stored in a designated portion of an array row.

<replace statement>

— REPLACE —<destination>— BY —<source part list>—————|

<destination>

—————|<pointer expression>—————|
 └<update pointer>┘

<update pointer>

—<pointer variable>— : —————|

The REPLACE statement stores character data from one or more data sources into a designated portion of an array row. The array row and the starting character position within the array row are both determined by the pointer expression part of the destination syntax. The value of this pointer expression initializes the stack-destination-pointer. As each character is moved into the destination array row, the stack-destination-pointer is correspondingly incremented one character position. When the last character has been stored in the destination array row, the corresponding final value of the stack-destination-pointer is stored in the pointer variable of the update pointer, if specified; otherwise, it is discarded. For more information on temporary storage locations, see "POINTER statement" earlier in this section.

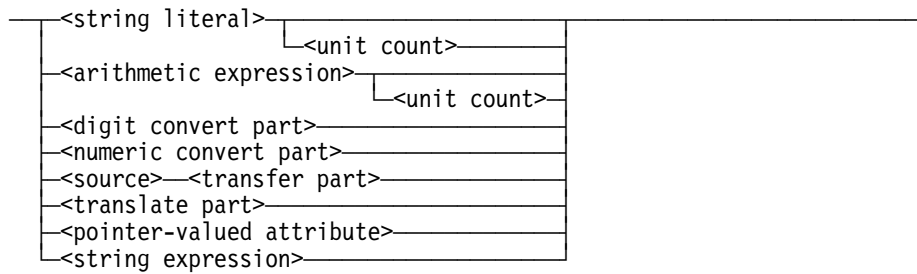
If either the source or destination of the REPLACE statement is an 8-bit character-based pointer, and the array pointed to is larger than 65535 words, an attempt to index into and beyond word 65536 produces an undefined result or the program fails. Refer to "POINTER Function" in Section 5, "Expressions and Functions", for the description of an 8-bit (EBCDIC) character-based pointer.

Source Part List

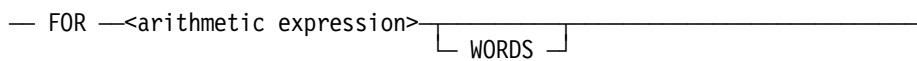
<source part list>



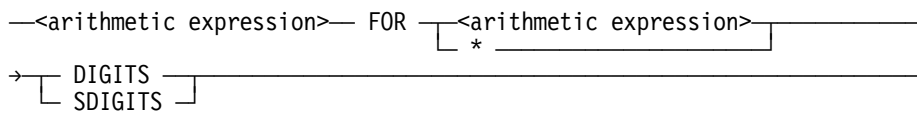
<source part>



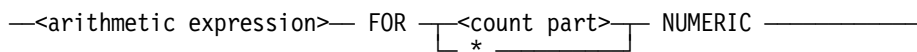
<unit count>



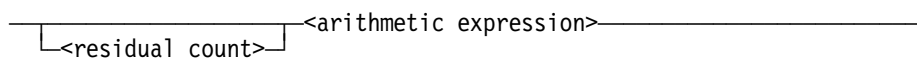
<digit convert part>



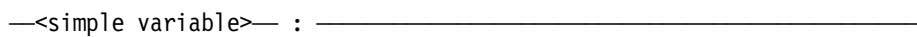
<numeric convert part>



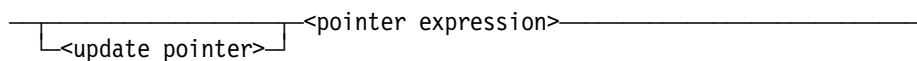
<count part>



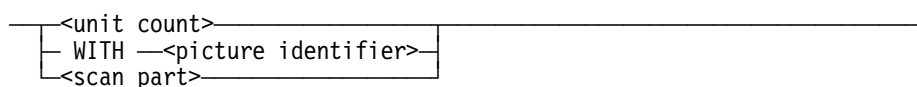
<residual count>



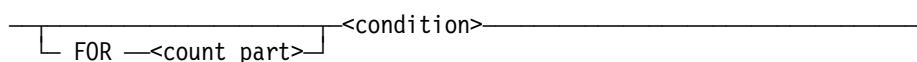
<source>



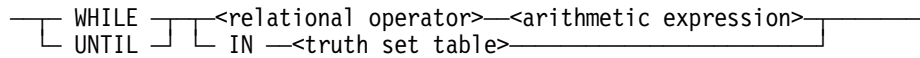
<transfer part>



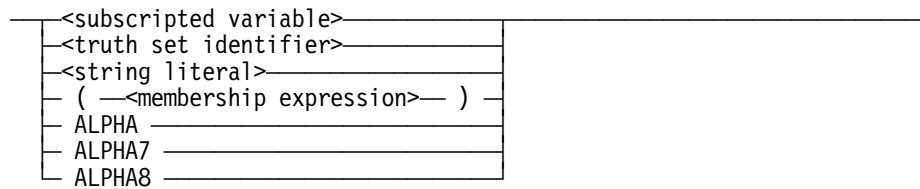
<scan part>



<condition>



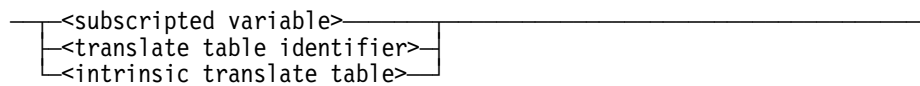
<truth set table>



<translate part>



<translate table>



<intrinsic translate table>



The source part list consists of one or more source parts. Each source part specifies source data and the processing to be performed on the data. All the data specified by a single source part is processed by a single method, but the various source parts of the source part list can specify a variety of processing methods.

With certain forms of the source part, provisions are made to store the final value of the stack-source-pointer. With several source parts in a single REPLACE statement, several final values for the stack-source-pointer arise. Corresponding to these final values are values of the stack-destination-pointer. These latter values are not accessible to the programmer but serve as the initial values of the stack-destination-pointer for the processing of the next source part.

REPLACE Statement

The syntactic construct <source> is the same construct encountered in the SCAN statement. The source construct contains a pointer expression that initializes the stack-source-pointer to a particular character position in an array row. The character size associated with this pointer expression must be the same as that associated with the pointer expression that initialized the stack-destination-pointer. If the update pointer option for the source is present, the pointer variable specified by the update pointer is assigned the final value of the stack-source-pointer for this source part.

The stack-source-pointer and the stack-destination-pointer can both reference the same array during a REPLACE statement. However, if the stack-source-pointer references a character position between the initial position of the stack-destination-pointer and its current position, the result is undefined. For example, the following REPLACE statement produces an undefined result:

```
REPLACE POINTER(A)+6 BY POINTER(A) FOR 12
```

On the other hand, the following REPLACE statement produces a well-defined result:

```
REPLACE POINTER(A) BY POINTER(A) FOR 12
```

Source Part Combinations

The formal syntax of the source part can be reduced to the following combinations:

```
<string literal>
<string literal> FOR <arithmetic expression>
                    FOR <arithmetic expression> WORDS

<arithmetic expression>
<arithmetic expression> FOR <arithmetic expression>
                        FOR <arithmetic expression> WORDS
                        FOR <arithmetic expression> DIGITS
                        FOR * DIGITS
                        FOR <arithmetic expression> SDIGITS
                        FOR * SDIGITS
                        FOR <count part> NUMERIC
                        FOR * NUMERIC

<source> FOR <arithmetic expression>
          FOR <arithmetic expression> WORDS
          FOR <arithmetic expression> WITH <translate table>

<source> WITH <picture identifier>

<source> WHILE <relational operator> <arithmetic expression>
          UNTIL <relational operator> <arithmetic expression>
          WHILE IN <truth set table>
          UNTIL IN <truth set table>
```



```

<source> FOR <count part> WHILE <relational operator>
                                     <arithmetic expression>
<source> FOR <count part> UNTIL <relational operator>
                                     <arithmetic expression>

<source> FOR <count part> WHILE IN <truth set table>
      FOR <count part> UNTIL IN <truth set table>

<pointer-valued attribute>

<string expression>

```

Each of these combinations is discussed in turn in the sections that follow. In all examples, P and Q are 8-bit pointers and the default character type is EBCDIC.

String Literal Source Parts

A string literal of 96 bits or less is a short string literal. A short string literal is evaluated at compilation time and stored, left-justified, in a one-word or two-word operand. Character size information is discarded.

A string literal of more than 96 bits is a long string literal. A long string literal is evaluated at compilation time and stored in a portion of an array called a pool array. The character size and address of the string literal are stored in a pointer called a pool array pointer.

The compiler calculates the number of characters in a string literal in terms of the largest character size specified by the string literal. The following are calculations for various string literals:

String Literal	Number of Characters
4"C1"	2
8"AB"	2
48"01"	1
4"01""A"	2 (if the default character type is EBCDIC)

<string literal>

If the source part is a short string literal, it is processed as follows:

1. At compilation time, the number of characters in the string is calculated.
2. At run time, the string literal is stored, left-justified with zero fill, in a one-word or two-word stack-source-operand.
3. The stack-integer-counter is assigned the value for the string length calculated at compilation time (see step 1).
4. Characters are copied from the stack-source-operand to the destination specified by the stack-destination-pointer. The stack-integer-counter specifies the number of characters copied, and the stack-destination-pointer specifies the character size. If the destination is specified by a noncharacter array row or array element, the character size is eight bits.

If the source part is a long string literal, it is processed as follows:

1. At compilation time, the number of characters in the string is calculated.
2. At run time, the stack-source-pointer is assigned the value of the pool array pointer to the long string literal, which includes the character size and address.
3. The character sizes of the stack-source-pointer and the stack-destination-pointer are compared. If they are not equal, the program is discontinued with a fault.
4. The stack-integer-counter is assigned the value for the string length calculated at compilation time (see step 1).
5. The number of characters specified by the stack-integer-counter are copied from the pool array to the destination specified by the stack-destination-pointer.

Examples of a <string literal>

In the following example, the three EBCDIC characters ABC are copied to the destination pointed to by P.

```
REPLACE P BY "ABC"
```

In the following example, the 20-character EBCDIC string is copied to the destination pointed to by P. At the end of the statement, P is left pointing to the first character position after the last character copied.

```
REPLACE P:P BY "A MUCH LONGER STRING"
```

In the following example, because the source, a string literal, is only four 4-bit characters long, it is evaluated as a single word; the character size is not retained. Because the destination is an 8-bit pointer, and four characters are to be replaced, four 8-bit characters are copied to the destination. At the end of the statement, the destination contains 4"12340000", which are the leftmost 32 bits of the stack-source-operand 4"123400000000".

```
REPLACE P BY 4"1234"
```

<string literal> FOR <arithmetic expression>

If the source part is a short string literal, it is processed as follows:

1. At compilation time, the string literal is stored in a one- or two-word operand. If the string literal is less than or equal to 48 bits long, it is stored, left-justified, and repeated for fill in a one-word operand. If the string literal is more than 48 bits long, it is stored, left-justified with zero fill, in a two-word operand.
2. At run time, this operand is assigned to the stack-source-operand.
3. If the arithmetic expression yields a positive value, this value is rounded to an integer, if necessary, and assigned to the stack-integer-counter; otherwise, zero is assigned to the stack-integer-counter.
4. The number of characters specified by the stack-integer-counter are copied to the destination specified by the stack-destination-pointer. If the stack-source-operand contains fewer than the specified number of characters, it is reused as many times as necessary. The character size is specified by the stack-destination-pointer. If the destination is specified by a noncharacter array row or array element, the character size is eight bits.

In the following examples, the first column shows a source part, and the second column shows the resulting string. A question mark (?) represents a null character.

Source Part	Result
"A" FOR 20	AAAAAAAAAAAAAAAAAAAAA
"AB" FOR 20	ABABABABABABABABAB
"ABC" FOR 20	ABCABCABCABCABCABC
"ABCD" FOR 20	ABCDABABCDABABCDAB
"ABCDE" FOR 20	ABCDEFABCDEFABCDEF
"ABCDEFGH" FOR 20	ABCDEFGH????ABCDEFGH

REPLACE Statement

If the source part is a long string literal, it is processed as follows:

1. The stack-source-pointer is assigned the value of the pool array pointer to the long string literal, which includes the character size and address.
2. The character sizes of the stack-source-pointer and the stack-destination-pointer are compared. If they are not equal, the program is discontinued with a fault.
3. If the arithmetic expression yields a positive value, this value is rounded to an integer, if necessary, and assigned to the stack-integer-counter; otherwise, zero is assigned to the stack-integer-counter.
4. The number of characters specified by the stack-integer-counter are copied to the destination specified by the stack-destination-pointer.

The *<string literal> FOR <arithmetic expression>* syntax is undefined for a long string literal if the integerized value of the arithmetic expression is greater than the length of the string literal in characters. For example, the result of the following statement is undefined:

```
REPLACE POINTER(A) BY "ABCDEFGHIJKLMNO" FOR 30
```

<string literal> FOR <arithmetic expression> WORDS

If the source part is a short string literal, it is processed as follows:

1. At compilation time, the string literal is stored in a one- or two-word operand. If the string literal is less than or equal to 48 bits long, is stored, left-justified and repeated for fill, in a one-word operand. If the string literal is more than 48 bits long, it is stored, left-justified with zero fill, in a two-word operand.
2. At run time, this operand is assigned to the stack-source-operand.
3. If the arithmetic expression yields a positive value, this value is rounded to an integer, if necessary, and assigned to the stack-integer-counter; otherwise, zero is assigned to the stack-integer-counter.
4. The stack-destination-pointer is moved forward, if necessary, to the nearest word boundary.
5. The number of words specified by the stack-integer-counter are copied from the stack-source-operand to the destination specified by the stack-destination-pointer. If the stack-source-operand contains fewer than the specified number of words, it is reused as often as necessary.

In the following examples, the first column shows a source part, and the second column shows the resulting string. A question mark (?) represents a null character.

Source Part	Result
"ABCD" FOR 2 WORDS	ABCDABABCDAB
"ABCDEFGH" FOR 2 WORDS	ABCDEFGH????
"ABCDEFGH" FOR 3 WORDS	ABCDEFGH????ABCDEF

If the source part is a long string literal, it is processed as follows:

1. The stack-source-pointer is assigned the value of the pool array pointer to the long string literal, which includes the character size and address.
2. The character sizes of the stack-source-pointer and the stack-destination-pointer are compared. If they are not equal, the program is discontinued with a fault.
3. If the arithmetic expression yields zero or a positive value, this value is rounded to an integer, if necessary, and assigned to the stack-integer-counter; otherwise, zero is assigned to the stack-integer-counter.
4. The stack-destination-pointer is moved forward, if necessary, to the nearest word boundary.
5. The number of words specified by the stack-integer-counter are copied from the stack-source-operand to the destination indicated by the stack-destination-pointer.

The *<string literal> FOR <arithmetic expression> WORDS* syntax is undefined for a long string literal if the integerized value of the arithmetic expression is greater than the length of the string literal in 48-bit words. For example, the result of the following statement is undefined:

```
REPLACE POINTER(A) BY "ABCDEFGHJKLMNOP" FOR 6 WORDS
```

Arithmetic Expression Source Parts

When a string literal is to be interpreted as an arithmetic expression, it must be enclosed in parentheses. Without the parentheses, the compiler interprets it as a string literal and generates code or issues syntax errors accordingly. For example the following is an invalid statement and results in a syntax error:

```
REPLACE POINTER(A) BY "A" FOR 3 DIGITS
```

On the other hand, the following statement is valid:

```
REPLACE POINTER(A) BY ("A") FOR 3 DIGITS
```

<arithmetic expression>

A source part of this form is processed as follows:

1. The arithmetic expression is evaluated and assigned to a one-word stack-source-operand.
2. The stack-source-operand is copied once to the destination specified by the stack-destination-pointer.

The character size of the stack-destination-pointer is not used to determine the default value to be assigned to the stack-integer-counter. Since the <unit count> is not specified, the stack-integer-counter is assigned the value 6.

Examples of Arithmetic Expressions

In the following examples, the first column shows a REPLACE statement, and the second column shows, in hexadecimal format, the resulting string.

Statement	Result
REPLACE P BY 7.5	267800000000
REPLACE P BY 3	000000000003
REPLACE P BY 1.68@@@2	248540000000
REPLACE P BY ("A")	0000000000C1
REPLACE POINTER(W,4) BY "ABCDEF"	C1C2C3xxxxxx
REPLACE POINTER(W) BY "ABCDEF"	C1C2C3C4C5C6

An x in the result indicates that the hexadecimal character at that position was unchanged by the given statement.

<arithmetic expression> FOR <arithmetic expression>

A source part of this form is processed as follows:

1. The first arithmetic expression is evaluated and assigned to a one-word stack-source-operand. Unpredictable results can occur if the absolute value of the stack-source-operand exceeds the maximum possible single-precision integer value. An INTEGEROVERFLOW execution-time fault occurs if the absolute value of the stack-source-operand exceeds the maximum possible double-precision integer value. Refer to "Compiler Number Conversion" in Section 2, "Language Components," for information on the maximum single and double-precision integer values.
2. If evaluation of the second arithmetic expression yields a positive value, this value is rounded to an integer, if necessary, and assigned to the stack-integer-counter; otherwise, zero is assigned to the stack-integer-counter.
3. The number of characters specified by the stack-integer-counter are copied from the stack-source-operand to the destination specified by the stack-destination-pointer. If the stack-source-operand contains fewer than the specified number of characters, it is reused as often as necessary. The character size is specified by the stack-destination-pointer. If the destination is specified by a noncharacter array row or array element, the character size is eight bits.

Examples of the <arithmetic expression> FOR <arithmetic expression> Form

The following example copies the character 48"00" to P. The stack-source-operand is 4"000000000003", and the leftmost character of this operand is copied to P.

```
REPLACE P BY 3 FOR 1
```

The following example copies the character 48"03" to P.

```
REPLACE P BY (3).[7:48] FOR 1
```

The following example copies the EBCDIC character A to P.

```
REPLACE P BY ("A").[7:48] FOR 1
```

<arithmetic expression> FOR <arithmetic expression> WORDS

A source part of this form is processed as follows:

1. If the evaluation of the first arithmetic expression yields a double-precision value, this double-precision value is assigned to a two-word stack-source-operand. Otherwise, the value of the first arithmetic expression is assigned to a one-word stack-source-operand.
2. If evaluation of the second arithmetic expression yields a positive value, this value is rounded to an integer, if necessary, and assigned to the stack-integer-counter; otherwise, zero is assigned to the stack-integer-counter.
3. The stack-destination-pointer is moved forward, if necessary, to the nearest word boundary.
4. The number of words specified by the stack-integer-counter is copied from the stack-source-operand to the destination specified by the stack-destination-pointer. If the stack-source-operand is double-precision, it is copied to the destination one word at a time (first word first, second word second) with TAGs. If the stack-integer-counter specifies more than one word (when the stack-source-operand is single-precision) or more than two words (when the stack-source-operand is double-precision), then the stack-source-operand is reused until the number of words specified by the stack-integer-counter has been copied.

Example of the <arithmetic expression> FOR <arithmetic expression> WORDS Form

The following example copies a single-precision zero into every element of array A.

```
REPLACE POINTER(A) BY 0 FOR SIZE(A) WORDS
```


<arithmetic expression> FOR <arithmetic expression> DIGITS

A source part of this form is processed as follows:

1. The absolute value of the first arithmetic expression is rounded to an integer value, if necessary, and assigned to the stack-source-operand.

Unpredictable results might occur if the absolute value of the stack-source-operand exceeds the maximum possible single-precision integer value.

An INTEGER OVERFLOW execution-time fault occurs if the absolute value of the stack-source-operand exceeds the maximum possible double-precision integer value.

For information about the maximum single- and double-precision integer values, refer to "Compiler Number Conversion," in Section 2.

2. If evaluation of the second arithmetic expression yields a positive value, this value is rounded to an integer, if necessary, and assigned to the stack-integer-counter; otherwise, zero is assigned to the stack-integer-counter.
3. A string of 12 hexadecimal characters that represents the absolute decimal value of the stack-source-operand is generated. If the value of stack-source-operand can be expressed in fewer than 12 digits, the string is filled on the left with zeros.
4. The N rightmost hexadecimal characters, where N is the number specified by the stack-integer-counter, are copied from this hexadecimal string to the destination. If the character size of the stack-destination-pointer is four bits, the characters are copied without change; if it is six or eight bits, the appropriate zone field is supplied.

If the value of the stack-integer-counter is greater than 12, the program is discontinued with a fault.

Examples of the <arithmetic expression> FOR <arithmetic expression> DIGITS Form

In the following examples, the first column shows the source part, the second column shows the resulting string when the destination is an 8-bit pointer, and the third column shows the resulting string when the destination is a 4-bit pointer.

Source Part	8-Bit Destination	4-Bit Destination
1234 FOR 6 DIGITS	8"001234"	4"001234"
7.5 FOR 3 DIGITS	8"008"	4"008"
-10 FOR 3 DIGITS	8"010"	4"010"
1234 FOR 3 DIGITS	8"234"	4"234"

<arithmetic expression> FOR * DIGITS

This source part functions similarly to a source part of the following form except that the stack-integer-counter is assigned a value equal to the minimum number of characters required to express accurately the value of the stack-source-operand.

```
<arithmetic expression> FOR <arithmetic expression> DIGITS
```

Examples of the <arithmetic expression> FOR * DIGITS Form

In the following examples, the first column shows the source part, the second column shows the resulting string when the destination is an 8-bit pointer, and the third column shows the resulting string when the destination is a 4-bit pointer.

Source Part	8-Bit Destination	4-Bit Destination
1234 FOR * DIGITS	8"1234"	4"1234"
7.5 FOR * DIGITS	8"8"	4"8"
-10 FOR * DIGITS	8"10"	4"10"

<arithmetic expression> FOR <arithmetic expression> SDIGITS

This source part functions similarly to a source part of the following form except that the sign of the first arithmetic expression is also recorded.

```
<arithmetic expression> FOR <arithmetic expression> DIGITS
```

If the \$TARGET option or the COMPILETARGET ODT option requests object code generation for predecessors of the enterprise server, explicitly or by omission, and the first <arithmetic expression> evaluates to (-0), the generated object code can produce a negative sign and a value of zero (0) in the destination. This does not occur when object code generation tailored for any enterprise server is requested. For further information on the object code generation, see the TARGET option in Section 6, "Compiling Programs".

If the character size of the stack-destination-pointer is four bits, then a 4"D" (1"1101") character, indicating a negative value, or a 4"C" (1"1100") character, indicating a positive value, is copied before the first digit. If the character size is eight bits, the zone field of the rightmost digit is changed to 1"1101" for negative values or 1"1100" for positive values.

When the character size of the stack-destination-pointer is four bits, the 4"C" or 4"D" character, indicating the sign of the value, is not counted as a digit.

For example, the statement REPLACE POINTER(A,4) BY -123 FOR 3 SDIGITS yields D123. Four, not three, characters are copied to the destination.

Strings produced by this form of source part can later be converted to an integer value with the correct sign using the INTEGER function. For example, the statement in the above example could be followed by the following statement, after which integer I would contain the value -123:

```
I := INTEGER(POINTER(A,4),3)
```

Examples of the <arithmetic expression> FOR <arithmetic expression> SDIGITS

In the following examples, the first column shows the source part, the second column shows the resulting string when the destination is an 8-bit pointer, and the third column shows the resulting string when the destination is a 4-bit pointer.

Source Part	8-Bit Destination	4-Bit Destination
1234 FOR 6 SDIGITS	4"F0F0F1F2F3C4"	4"C001234"
-1234 FOR 6 SDIGITS	4"F0F0F1F2F3D4"	4"D001234"

<arithmetic expression> FOR * SDIGITS

This source part functions similarly to a source part of the following form, except that the stack-integer-counter is assigned a value equal to the minimum number of characters required to express accurately the value of the stack-source-operand:

```
<arithmetic expression> FOR <arithmetic expression> SDIGITS
```

Examples of the <arithmetic expression> FOR * SDIGITS Form

In the following examples, the first column shows the source part, the second column shows the resulting string when the destination is an 8-bit pointer, and the third column shows the resulting string when the destination is a 4-bit pointer.

Source Part	8-Bit Destination	4-Bit Destination
1234 FOR * SDIGITS	4"F1F2F3C4"	4"C1234"
-1234 FOR * SDIGITS	4"F1F2F3D4"	4"D1234"

<arithmetic expression> FOR <count part> NUMERIC

A source part of this form is processed as follows:

1. If the arithmetic expression in the count part yields a positive value, this value is rounded to an integer, if necessary, and assigned to the stack-integer-counter; otherwise, zero is assigned to the stack-integer-counter.
2. The first arithmetic expression is evaluated, and an internal procedure is called. This procedure generates an EBCDIC character string representing the decimal value of the arithmetic expression as precisely and concisely as possible given the field width specified by the stack-integer-counter.
3. If the character size of the stack-destination-pointer is eight bits, the string is copied to the destination without translation. If the character size is four bits, the program is discontinued with a fault.

If a residual count does not appear in the count part, the string is copied to the destination, right-justified with blank fill, in a field with a width equal to the value of the stack-integer-counter. If a residual count does appear in the count part, the string is copied to the destination, left-justified, and the simple variable is assigned the difference between the initial value of the stack-integer-counter and the number of characters copied.

The form of the decimal representation is determined by the operand type (single or double-precision), whether or not the operand value is an integer, the magnitude of the operand, the number of significant digits in its decimal representation, and the field width. The basic rule is that the number is represented as compactly as possible using integer, simple decimal, or exponential notation, as appropriate.

For example, the following source parts generate the decimal representations shown:

Source Part	Decimal Representation
12345678 FOR 8 NUMERIC	12345678
12345678 FOR 6 NUMERIC	1.23+7
123/100 FOR N:6 NUMERIC	1.23 (N := 2)

<arithmetic expression> FOR * NUMERIC

This source part functions similarly to a source part of the following form except that no maximum field width is specified:

```
<arithmetic expression> FOR <count part> NUMERIC
```

Thus, the internal procedure that generates the string is allowed to use as many as 36 characters to represent the decimal value of the arithmetic expression.

For example, the following source parts generate the decimal representations shown:

Source Part	Decimal Representation
123 FOR * NUMERIC	123
1/3 FOR * NUMERIC	0.333333333333333333333333333333

Pointer Expression (<source>) Source Parts**<source> FOR <arithmetic expression>**

A source part of this form is processed as follows:

1. The pointer expression in the source is evaluated and assigned to the stack-source-pointer.
2. If the arithmetic expression yields a positive value, this value is rounded to an integer, if necessary, and assigned to the stack-integer-counter; otherwise, zero is assigned to the stack-integer-counter.
3. The character sizes of the stack-source-pointer and the stack-destination-pointer are compared. If they are not equal, the program is discontinued with a fault. If both the source and the destination are specified by noncharacter array rows or array elements, the character size of both the stack-source-pointer and the stack-destination-pointer is eight bits.
4. The number of characters specified by the stack-integer-counter are copied from the location specified by the stack-source-pointer to the destination specified by the stack-destination-pointer.

Example of a <source> FOR <arithmetic expression> Form

In the following example, the 20 EBCDIC characters pointed to by Q are copied to the location pointed to by P.

```
REPLACE P BY Q FOR 20
```

<source> FOR <arithmetic expression> WORDS

A source part of this form is processed as follows:

1. The pointer expression in the source is evaluated and assigned to the stack-source-pointer.
2. If the arithmetic expression yields a positive value, this value is rounded to an integer, if necessary, and assigned to the stack-integer-counter; otherwise, zero is assigned to the stack-integer-counter.
3. The stack-source-pointer and the stack-destination-pointer are moved forward, if necessary, to the nearest word boundary.
4. The number of 48-bit words specified by the stack-integer-counter are copied from the location specified by the stack-source-pointer to the destination specified by the stack-destination-pointer.

The character sizes of the source and destination pointer expressions are irrelevant.

Example of <source> FOR <arithmetic expression> WORDS

Both P and Q are advanced to the nearest word boundary, if necessary, and 20 words are copied from the location pointed to by Q to the location pointed to by P.

```
REPLACE P BY Q FOR 20 WORDS
```

<source> FOR <arithmetic expression> WITH <translate table>

This construct retrieves characters from a source location, translates each character (through the use of the specified translate table) into a possibly different character with a possibly different character size, and stores each resulting character in the location indicated by the stack-destination-pointer.

The value of the pointer expression in the source points to the first character to be translated. The stack-source-pointer is initialized to this value. The stack-destination-pointer and the stack-source-pointer need not have the same character size. Instead, the stack-source-pointer must have a character size equal to that of the characters being translated, and the stack-destination-pointer must have a character size equal to that of the resulting translated characters; otherwise a Memory Protect fault or undefined results occur.

The value of the arithmetic expression indicates the number of characters to be translated and written to the destination. This value is integerized, if necessary, and assigned to the stack-integer-counter. The stack-auxiliary-pointer is initialized to point to the first character of the first word of the translate table, and its character size is absent. Normally, when a pointer is used and its character size is absent, a default value of six or eight is used, depending on the default character type. However, the character size of the pointer used to initialize the stack-auxiliary-pointer is irrelevant. The translate table is not examined sequentially (one character at a time); instead, the data in the table is accessed by special indexing techniques implemented in the hardware, as described in the following text.

<intrinsic translate table>

If the translate table is of this form, the stack-auxiliary-pointer is initialized to point to the appropriate intrinsic translate table. The function of each translate table can be deduced from its name. For example, the HEXTOEBCDIC table is used to translate characters from hexadecimal to EBCDIC.

<translate table identifier>

If the translate table is of this form, a translate table must have been declared in a TRANSLATETABLE declaration. For a detailed discussion regarding the construction of a translate table, refer to "TRANSLATETABLE Declaration" in Section 3, "Declarations."

<subscripted variable>

If the translate table is of this form, the programmer is responsible for creating a properly structured translate table that does not cross array row or page boundary and begins with the word in the array row indicated by the subscripted variable. From the subscripted variable to the end of the array row or page, there must be enough words for the entire translate table: 4 or 64 words, depending on whether the character type is 4-bit, 7- or 8-bit, respectively. If there are not enough words for the translate table, an invalid index error can result at run time. For more information on translate table indexing, refer to "Translate Table Indexing" under "TRANSLATETABLE Declaration" in Section 3, "Declarations."

Examples of the <source> FOR <arithmetic expression> WITH <translate table> Form

```
REPLACE POINTER(B,4) BY POINTER(A,8) FOR 20 WITH EBCDICTOHEX
```

```
A = 8"0123456789ABCDEFGHIJ"
B = 4"0123456789ABCDEFFFFF"
```

```
REPLACE POINTER(B,7) BY POINTER(A,8) FOR 14 WITH EBCDICTOASCII
```

```
A = 4"F0F1F2F3F4F5F6F7F8F9C1C2C3C4"
B = 4"3031323334353637383941424344"
```

```
REPLACE POINTER(B,8) BY POINTER(A,4) FOR 12 WITH HEXTOEBCDIC
```

```
A = 8"012345" = 4"F0F1F2F3F4F5"
B = 8"F0F1F2F3F4F5"
```

<source> WITH <picture identifier>

The character data specified by the source (which must be a pointer) is processed under control of the picture specified by the picture identifier. The source and destination pointers must be 4-bit, 8-bit, or word-oriented. If the source is a word-oriented pointer, it is changed to a 4-bit pointer if the destination is a 4-bit pointer; otherwise, it is changed to an 8-bit pointer. If the destination is a word-oriented pointer, it is changed to a 4-bit pointer if the source is a 4-bit pointer; otherwise, it is changed to an 8-bit pointer. If neither the source nor the destination pointer is a word-oriented pointer, the source and destination pointers must either both be 4-bit pointers or both be 8-bit pointers. Details regarding the formation and action of pictures are described under "PICTURE Declaration" in Section 3, "Declarations."

When using a <picture identifier> that provides for character insertion, note the possibility of receiving an undefined result. This is described earlier in the explanation of the REPLACE statement, where both the source and destination are the same.

Source Parts with Boolean Conditions

The next eight forms of the source part copy characters from the source to the destination until a source character fails or passes the specified test. The number of characters copied can also be limited by an optional count part. For more information on the use of these Boolean conditions, refer to "SCAN Statement" later in this section.

In the source parts containing a condition of either of the following forms the source characters are tested against bits [7:8], [5:6], or [3:4] of the arithmetic expression, depending on the character size of the source:

```
WHILE <relational operator> <arithmetic expression>  
UNTIL <relational operator> <arithmetic expression>
```

In all cases, the stack-source-pointer is left pointing to the character that failed or passed the test.

The count part consists of an arithmetic expression and, optionally, a residual count. The value of the arithmetic expression specifies the maximum number of characters to be copied. The residual count, when it appears, is a simple variable in which is stored the difference between the value of the arithmetic expression and the number of source-part characters copied.

<source> WHILE <relational operator> <arithmetic expression>

The stack-source-pointer is initialized to the source pointer. Characters are then copied from the source to the destination as long as source characters pass the test.

A paged (segmented) array error fault could occur at run time if all of the following conditions occur:

- The stack-destination-pointer references the first character beyond the end of the destination array.
- The stack-source-pointer references the first character to fail the test.
- The stack-integer-counter is nonzero.

Example of a <source> WHILE <relational operator> <arithmetic expression> Form

```
REPLACE P BY Q WHILE NEQ " "

      Q = "LONG STRING"
      P = "LONG"
```

<source> UNTIL <relational operator> <arithmetic expression>

The stack-source-pointer is initialized to the source pointer. Characters are then copied from the source to the destination until a source character passes the test.

A paged (segmented) array error fault could occur at run time if all of the following conditions occur:

- The stack-destination-pointer references the first character beyond the end of the destination array.
- The stack-source-pointer references the first character to pass the test.
- The stack-integer-counter is nonzero.

Example of a <source> UNTIL <relational operator> <arithmetic expression> Form

```
REPLACE P BY Q UNTIL = "."

      Q = "FILE/TITLE ON PACK.XXX"
      P = "FILE/TITLE ON PACK"
```

<source> WHILE IN <truth set table>

The stack-source-pointer is initialized to the source pointer. Characters are then copied from the source to the destination as long as the source characters are members of the truth set. For further information on truth sets, see "TRUTHSET Declaration" in Section 3, "Declarations."

Example of a <source> WHILE IN <truth set table> Form

```
REPLACE P BY Q WHILE IN ALPHA8
```

```
Q = "ABCD1234.56"
```

```
P = "ABCD1234"
```

<source> UNTIL IN <truth set table>

The stack-source-pointer is initialized to the source pointer. Characters are then copied from the source to the destination until a source character is encountered that is a member of the truth set. For further information on truth sets, see "TRUTHSET Declaration" in Section 3, "Declarations."

Example of a <source> UNTIL IN <truth set table> Form

```
REPLACE P BY Q UNTIL IN ALPHA8
```

```
Q = ", *,$1234"
```

```
P = ", *,$"
```

<source> FOR <count part> WHILE <relational operator> <arithmetic expression>

The stack-source-pointer is initialized to the source pointer. The stack-integer-counter is initialized to the value of the arithmetic expression in the count part. Characters are then copied from the source to the destination and the stack-integer-counter is decremented for each character copied as long as the stack-integer-counter is not zero and the source characters pass the test.

A paged (segmented) array error fault could occur at run time if all of the following conditions occur:

- The stack-destination-pointer references the first character beyond the end of the destination array.
- The stack-source-pointer references the first character to fail the test.
- The stack-integer-counter is nonzero.

Example of a <source> FOR <count part> WHILE <relational operator><arithmetic expression> Form

```
REPLACE P BY Q FOR N:11 WHILE NEQ " "
```

```
Q = "LONG STRING"
P = "LONG"          (and N = 7)
```

<source> FOR <count part> UNTIL <relational operator> <arithmetic expression>

The stack-source-pointer is initialized to the source pointer. The stack-integer-counter is initialized to the value of the arithmetic expression in the count part. Characters are then copied from the source to the destination and the stack-integer-counter is decremented for each character copied until either the stack-integer-counter is zero or a source character passes the test.

A paged (segmented) array error fault could occur at run time if all of the following conditions occur:

- The stack-destination-pointer references the first character beyond the end of the destination array.
- The stack-source-pointer references the first character to pass the test.
- The stack-integer-counter is nonzero.

Example of a <source> FOR <count part> UNTIL <relational operator> <arithmetic expression> Form

```
REPLACE P BY Q FOR N:22 UNTIL = "."
```

```
Q = "FILE/TITLE ON PACK.XXX"
P = "FILE/TITLE ON PACK"    (and N = 4)
```

<source> FOR <count part> WHILE IN <truth set table>

The stack-source-pointer is initialized to the source pointer. The stack-integer-counter is initialized to the value of the arithmetic expression in the count part. Characters are then copied from the source to the destination and the stack-integer-counter is decremented for each character copied as long as the stack-integer-counter is not zero and the source characters are members of the truth set. For further information on truth sets, see "TRUTHSET Declaration" in Section 3, "Declarations."

Example of a <source> FOR <count part> WHILE IN <truth set table> Form

```
REPLACE P BY Q FOR N:11 WHILE IN ALPHA8
```

```
Q = "ABCD1234.56"
P = "ABCD1234"      (and N = 3)
```

<source> FOR <count part> UNTIL IN <truth set table>

The stack-source-pointer is initialized to the source pointer. The stack-integer-counter is initialized to the value of the arithmetic expression in the count part. Characters are then copied from the source to the destination and the stack-integer-counter is decremented for each character copied until either the stack-integer-counter is zero or a source character is a member of the truth set. For further information on truth sets, see "TRUTHSET Declaration" in Section 3, "Declarations."

Example of a <source> FOR <count part> UNTIL IN <truth set table> Form

```
REPLACE P BY Q FOR N:10 UNTIL IN ALPHA8
```

```
Q = ", *,$1234"  
P = ", *,$"      (and N = 4)
```

Other Source Parts

<pointer-valued attribute>

The string of characters that forms the value of the pointer-valued attribute is copied to the location indicated by the stack-destination-pointer. The string of characters is formatted in the destination array row in a form suitable to serve in a replace pointer-valued attribute statement that assigns a value to the same attribute. The character string ends with an EBCDIC period (8".).

For example, if P is a pointer identifier and F1 and F2 are file identifiers, then the following sequence of statements is valid:

```
REPLACE P BY F1.TITLE;  
REPLACE F2.TITLE BY P;
```

All pointer-valued attributes have a character size of eight bits. At run time, if the destination pointer does not also have a character size of eight bits, the program is discontinued with a fault.

If a pointer-valued attribute appears as the source part in a REPLACE statement, a call is made on an operating system procedure to perform this part of the REPLACE statement.

<string expression>

When a string expression appears as the source part in a REPLACE statement, it is evaluated and stored in a pool array. The stack-source-pointer is initialized to point to the first character of the string in the pool array. The entire string is copied to the destination.

Example of a String Expression

The following example copies the EBCDIC string ABCDEFG to the location pointed to by P.

```
STR:= "ABCDEFGH";
REPLACE P BY STR;
```

Examples of REPLACE Statements

```
REPLACE PTR BY "A"
REPLACE PTR:PTR BY "*" FOR 75
REPLACE PTR BY ITEM
REPLACE PTR BY (4"03").[7:48] FOR 1
REPLACE PTR BY " " FOR N WORDS
REPLACE PTR:PTR BY PST FOR 18
REPLACE PTR BY PST:PST FOR NUM WORDS
REPLACE PTR BY PINFO WITH PIC
REPLACE PTR:PTR BY PST WHILE NEQ " "
REPLACE PTR BY PST WHILE IN ALPHA
REPLACE P BY X FOR * DIGITS
REPLACE P BY X FOR 50 NUMERIC
REPLACE P BY X FOR * NUMERIC
REPLACE PTR BY PST WHILE IN MYTRUTHTABLE
REPLACE PTR BY PST UNTIL = ","
REPLACE PTR:PTR BY PST:PST UNTIL IN ALPHA8
REPLACE PTR BY PST FOR THELENGTH WHILE > "0"
REPLACE PTR BY PST FOR LEFT:25 WHILE IN ACCEPTABLE
```

REPLACE Statement

```
REPLACE PTR BY PST FOR 120 UNTIL NEQ " "  
REPLACE PTR BY PST FOR M:N UNTIL IN ALPHA  
REPLACE PTR:PTR BY SUMTOTAL FOR 6 DIGITS, "."  
REPLACE PTR BY FYLE.TITLE  
REPLACE PTR BY PST:PST FOR L WITH XLATTABLE  
REPLACE PTR BY STR  
REPLACE PTR BY SARRAY[4,J]  
REPLACE P BY S1 || S2  
REPLACE P BY PTR:PTR FOR 10  
REPLACE P BY TAKE(S,2) || SA[4]  
REPLACE P BY HEAD(S,ALPHA)
```

REPLACE FAMILY-CHANGE Statement

The REPLACE FAMILY-CHANGE statement adds stations to, or deletes stations from, the family of an open, remote file.

<replace family-change statement>

```
— REPLACE —<family designator>— BY —<up or down>—<simple source>—|
```

<family designator>

```
—<file designator>— . — FAMILY —————|
```

<up or down>

```
— * [ + ]—————|
   [ - ]
```

<simple source>

```
—<string literal>—————|
  |<pointer expression>—|
```

The file designator specifies the file whose FAMILY file attribute is to be changed. For more information on the file designator, see “SWITCH FILE Declaration” in Section 3, “Declarations.” If a station is to be added to the family, <up or down> is *+. If a station is to be deleted from the family, <up or down> is *- . The simple source specifies the title of the station involved. Because the simple source is a value for a pointer-valued attribute, its value must end with a period (.). For more specific information, refer to “REPLACE POINTER-VALUED ATTRIBUTE Statement” later in this section.

Specification of Valid Stations

If the simple source does not reference a valid station title, as specified for the current network in the Network Definition Language II (NDLII) description, then the REPLACE FAMILY-CHANGE statement has the following effects:

- <file designator>.FAMILY is unchanged.
- <file designator>.ATTERR is given the value TRUE, and <file designator>.ATTYPE is set to the appropriate value.
- An appropriate error message is displayed on the Operator Display Terminal (ODT).
- The program continues.

If <up or down> is *-and the simple source specifies a valid station as defined by the current NDLII description, but the specified station is not currently a member of the family, then the REPLACE FAMILY-CHANGE statement makes no change to the specified family. No error condition is indicated (such a situation is not considered to be an error) and control passes to the next statement of the program.

REPLACE FAMILY-CHANGE Statement

If, after execution of a REPLACE FAMILY-CHANGE statement, the remote file is closed with release and later reopened, the family reverts to its NDLL-specified value. However, if the remote file is closed with retention and later reopened, the family retains its changed value.

When the REPLACE FAMILY-CHANGE statement is executed, a call is made on an operating system procedure to perform the desired function.

Examples of REPLACE FAMILY-CHANGE Statements

The following example adds the station with the title "ACCT7" to the family of the remote file NETWORK.

```
REPLACE NETWORK.FAMILY BY *+ "ACCT7."
```

The following example deletes the station with the title given by the pointer STATIONNAMEPTR from the family of the remote file DATACollectors.

```
REPLACE DATACollectors.FAMILY BY *- STATIONNAMEPTR
```


REPLACE POINTER-VALUED ATTRIBUTE Statement

The REPLACE POINTER-VALUED ATTRIBUTE statement changes the value of a pointer-valued attribute.

<replace pointer-valued attribute statement>

```
— REPLACE —<pointer-valued attribute>— BY —————>
→ <simple source>—————|
  | <pointer-valued attribute> |
```

<pointer-valued attribute>

```
— <pointer-valued file attribute>—————|
  | <pointer-valued task attribute> |
  | <string-valued library attribute> |
```

<pointer-valued file attribute>

```
—<file designator>—————|
  | <attribute parameter specification> | . —————>
→ <pointer-valued file attribute name>—————|
→ | ( —<arithmetic variable>— ) |
```

The attribute error number returned from the operating system can be captured in the <arithmetic variable>.

<pointer-valued task attribute>

```
—<task designator>— . —<pointer-valued task attribute name>—————|
```

<pointer-valued task attribute name>

Task attributes of type string can be used according to the constraints listed in the *Task Attributes Programming Reference Manual*.

<string-valued library attribute>

```
—<library attribute designator>— . —————>
→ <string-valued library attribute name>—————|
```

When the REPLACE POINTER-VALUED ATTRIBUTE statement is executed, a call is made on an operating system procedure to perform the desired function.

Specification of the Simple Source

The simple source specifies the string of characters that is to become the new value of the pointer-valued attribute. This string of characters must end in a terminator that is a null character (48"00") for the PATHNAME file attribute and the CURRENTDIRECTORY task attribute, and a period (.) for other attributes.

If the simple source is a string literal, the last character of the string literal must be the terminator character. The effective part of the string literal is terminated by the first terminator in the string. A maximum string length is associated with each pointer-valued attribute. If the effective part of the string literal has a string length that is greater than the maximum length allowed for the pointer-valued attribute, then the new value of the pointer-valued attribute is the value of the string literal truncated on the right to the required length.

If the simple source is a pointer expression, the pointer expression must point to the string of characters that is to become the new value of the pointer-valued attribute. Starting with the first character pointed to by the pointer expression, characters are copied as the new value of the pointer-valued attribute until the terminator is encountered, the maximum number of characters for the attribute are copied, or the end of the array row is encountered. The last case results in a run-time error.

If a pointer-valued task attribute is used as the destination and the source is a pointer-valued attribute, the source attribute and the destination attribute must be the same attribute. Some pointer-valued task attributes used as a destination require a simple source. These attributes receive a run-time INVALID PARAMETER error if the source is a pointer-valued task attribute. If a pointer-valued file attribute is used as the destination, the source must be a simple source. If a string-valued library attribute is used as the destination, the source can be either a simple source or another string-valued library attribute.

Examples of REPLACE POINTER-VALUED ATTRIBUTE Statements

In the following example, the TITLE attribute of file FYLE is assigned "MASTER/PAYROLL."

```
REPLACE FYLE.TITLE BY "MASTER/PAYROLL."
```

In the following example, the NAME attribute of the task TSK is assigned "SECOND/STACK."

```
REPLACE TSK.NAME BY "SECOND/STACK."
```

In the following example, the NAME attribute of task T is assigned the value of the NAME attribute of task TS.

```
REPLACE T.NAME BY TS.NAME
```

In the following example, the NAME attribute of the task TSK is assigned the value of the NAME attribute of the task T.PARTNER.

```
REPLACE TSK.NAME BY T.PARTNER.NAME
```

In the following example, the INTNAME attribute of library L is assigned "INTLIB."

```
REPLACE L.INTNAME BY "INTLIB."
```

In the following example, the TITLE attribute of library LIB_A is assigned the value of the TITLE attribute library LIB_B.

```
REPLACE LIB_A.TITLE BY LIB_B.TITLE
```

RESET Statement

The RESET statement sets the happened state of the designated event to FALSE (not happened).

<reset statement>

— RESET — (—<event designator>—) —————|

The RESET statement does not change the status of any tasks waiting on the event.

WAIT and WAITANDRESET Statements

If a RESET statement is used after a WAIT statement to restore the happened state of an event to FALSE (not happened), a period of time exists during which another task could cause the event. For this reason, a WAITANDRESET statement might be more useful than a WAIT statement followed by a RESET statement.

Examples of RESET Statements

The following example sets the happened state of the event EVNT to FALSE (not happened).

```
RESET (EVNT)
```

The following example sets the happened state of the event designated by EVNTARRAY[INDX] to FALSE (not happened).

```
RESET (EVNTARRAY [INDX])
```

RESIZE Statement

The RESIZE statement modifies the size of the designated array row, subarray, or array.

Note: The RESIZE statement cannot be used for task arrays.

<resize statement>

— RESIZE — ($\left[\begin{array}{l} \text{<array row resize parameters>} \\ \text{<special array resize parameters>} \end{array} \right]$) —————|

The RESIZE statement changes the upper bounds of the appropriate dimensions of an array. The resize parameters designate the array row or rows to be changed and the new sizes of those rows.

Note that if the RESIZE statement is used on other than the highest-order dimension of an array, the array can contain subarrays of different sizes.

When the initial size of an array is to be chosen dynamically by a program, the most efficient technique is to declare the array with a variable upper bound, the bound being a global variable or a parameter computed before the procedure or block is entered.

There are two forms of the RESIZE statement: the <array row resize parameters> form and the <special array resize parameters> form.

Array Row Resize Parameters

<array row resize parameters>

— $\left[\begin{array}{l} \text{<array row>} \\ \text{<procedure reference array row>} \end{array} \right]$, —<new size>—————→
 → $\left[\begin{array}{l} \text{,} \\ \text{RETAIN} \\ \text{DISCARD} \\ \text{PAGED} \end{array} \right]$ —————|

<new size>

—<arithmetic expression>—————|

In this form of RESIZE statement, the first parameter is an array row, a one-dimensional array whose elements are of some array class: BOOLEAN, COMPLEX, DOUBLE, INTEGER, REAL, or a character type.

RESIZE Statement

The RESIZE statement causes the size of the designated row to be modified as specified by the new size. The resize options have the following effects:

Option	Effect
DISCARD	The current contents of the array row are discarded; the new contents of the array row are undefined.
RETAIN	As much of the current information in the array row is retained as fits in the new size. If the new size is smaller than the old, data in the lost elements is discarded. If the new size is larger, the data in the new elements is undefined.
PAGED	The resized array is to be a paged (segmented) array. The new paged array is considered to be touched (referenced) after the resize is complete. PAGED also implies RETAIN. PAGED is not valid for a procedure reference array row because a procedure reference array cannot be segmented.

If no third parameter appears, DISCARD is assumed.

RETAIN is typically used for an array being employed as a stack. When the array is not large enough to accept a push of the next entry, the array can be enlarged without losing the data already present. If no data has been assigned to the array, or if the old data is no longer relevant, DISCARD is more efficient for the resize of an array row.

It is possible to resize a referenced paged array. However, such arrays can only be resized downward on certain systems. An array is referenced, or touched, if a statement referring to the array has been executed in the block. An array row is paged if its declared length exceeds the array segmentation start size, unless it is declared LONG or DIRECT. The array segmentation start size is typically 1024 words or more. The start size can be displayed or set with the system command SEGARRAYSTART. Note that if an array is initially declared shorter than or equal to the array segmentation start size, then it is unpagged, and resizing it larger without using the PAGED option does not cause it to become paged.

An array that is initially declared to be shorter than or equal to the array segmentation start size and is, therefore, an unpagged array can be resized to become a paged array by using the PAGED option. The new paged array is considered to be touched after the resize is complete.

The PAGED option is useful in cases where the desired size of a paged array is not known at the time the array is declared. The PAGED option offers an alternative to declaring an array larger than the array segmentation start size and avoiding references to the array until the desired size is known. The PAGED option achieves the same results and makes errors less likely. The only restriction on the use of the PAGED option is that on certain systems the new size of the resized array must be larger than the current array size.

If the new array row size is less than the old, any pointer variable that now points beyond the end of the array row is set to the uninitialized state.

The value of the new size is integerized with rounding, if necessary, to specify a new size, S_n , which is interpreted as a number of elements for the resized array row. If the

array row is an original array, then its size is changed to Sn. If the array row is a referred array and the original array has a different element size, the original array is resized to have just enough elements to hold Sn elements of the referred array row.

When an original array is resized, any referred arrays with element widths different from those of the original array are assigned the size they would have had if the original array had been declared at its new size and the referred array had been created from the original by array equivalence or array reference assignment.

When a resize of a referred array causes a resize of an original array, the size calculations are performed with the element widths Wn for the resized referred array and Wo for the original array. The new size of the original array, So, is the following:

$$So := (Sn * Wn + Wo - 1) \text{ DIV } Wo$$

So * Wo can exceed Sn * Wn. The new size, Sr, of any referred array with element width Wr that is based on the resized original array is the following:

$$Sr := (So * Wo) \text{ DIV } Wr$$

Wr is equal to Wn for the explicitly resized referred array, and the net calculation is the following, which can exceed Sn.

$$Sr := ((Sn * Wn + Wo - 1) \text{ DIV } Wo) * Wo \text{ DIV } Wn$$

The explicitly resized array row can be slightly larger than requested, if the original array has a wider element width. For example, if arrays RA and EA are declared as follows then EA contains 36 elements:

```
REAL ARRAY RA[0:5];
EBCDIC ARRAY EA[0] = RA;
```

If the statement RESIZE(EA,50,RETAIN) is executed, the original array, RA, is resized to a new size of 9 words, calculated from the following:

$$So = (50 * 8 + 48 - 1) \text{ DIV } 48 = 9$$

The actual new size of the referred array, EA, is then 54, calculated from the following:

$$Sr = (9 * 48) \text{ DIV } 8 = 54$$

Because the second calculation truncates, Sr * Wr can be less than So * Wo, just as with array row equivalence or array reference assignment.

For example, consider the following statement, where H is a hexadecimal array row:

```
RESIZE(H,29,DISCARD)
```

The following table shows the size assigned to referred arrays for several combinations of referred and original classes. The diagonal of the table shows the size assigned to each original.

RESIZE Statement

Original	Referred Arrays			
	Hexadecimal	EBCDIC	Real	Double
Hex	29	14	2	1
EBCDIC	30	15	2	1
Real	36	18	3	1
Double	48	24	4	2

Special Array Resize Parameters

<special array resize parameters>

```

—<multidimensional array designator>— , —<new size>— , —————>
|
| —<event array designator>—
| —<string array designator>—
| —<structure block array row>—
→ RETAIN —————>

```

In this form of RESIZE statement, the first parameter is an array whose elements do not have an array class. Events and strings are special classes of objects. A multidimensional array can be considered an array of arrays.

The RESIZE statement sets the size of the parameter array to the new size, unless the new size is less than the existing size, in which case the RESIZE statement is ignored and the warning message *ATTEMPTED DOWNWARD RESIZE IGNORED* is generated.

If the first parameter includes a subarray selector, the dimension corresponding to the first asterisk (*) is changed; otherwise, the first dimension of the designated array is changed. Whenever a higher-order dimension of an array is enlarged, new subarrays are created with the same dimensions as in the original ARRAY declaration. Any existing subarrays are unaffected by the resize operation. For example, given the declaration `DOUBLE ARRAY A[1:2,0:5,-4:4]` the statement `RESIZE(A[1,*,*],8,RETAIN)` increases the size of `A[1,*,*]` from 6 to 8 (new bound pair = 0:7). The statement causes array rows `A[1,6,*]` and `A[1,7,*]` to be established as one-dimensional double arrays of size 9, even if all existing rows of A had already been resized to some other size.

Multidimensional Array Designator

A multidimensional array designator is an array designator or a procedure reference array designator with dimensionality greater than one; that is, a multidimensional array name or procedure reference array identifier, optionally suffixed by a subarray selector with at least two asterisks (*).

If the array to be resized is specified by a multidimensional array designator, then the new subarrays have the same type as the original array; their contents are undefined.

Event Array Designator

If the array to be resized is specified by an event array designator, then enlarging the low-order dimension creates new events with the happened state equal to FALSE (not happened) and the available state equal to TRUE (available); existing elements are unaffected.

String Array Designator

If the array to be resized is specified by a string array designator, then enlarging the low-order dimension creates new empty strings; existing elements are unaffected.

Interlock Array Designator

If the array to be resized is specified by an interlock array designator, then enlarging the low-order dimension creates new interlocks with a state of FREE; existing elements are unaffected.

Run-Time Error Messages

When an illegal resize is attempted on an array, one of the following messages is displayed at run time:

BAD RESIZE/DEALLOCATE – NOT AN ARRAY

An attempt was made to resize something that is not an array, such as an uninitialized array reference.

BAD RESIZE/DEALLOCATE – ARRAY NOT MULTI-DIMENSIONAL

An attempt was made to resize a one-dimensional array when a multidimensioned array was expected.

BAD RESIZE/DEALLOCATE – RESIZING IMMOVABLE INSTACK ARRAY

An attempt was made to resize an immovable instack array.

BAD RESIZE/DEALLOCATE – RESIZING READONLY ARRAY

An attempt was made to resize a value array.

ILLEGAL DOWNWARD RESIZE ON A PAGED ARRAY

An attempt was made to resize a paged array to a length smaller than its current length. A segmented array that has been referenced can be resized downward only on certain systems.

RESIZE ABORTED – INSUFFICIENT MEMORY

The job performing the resize has been terminated because there is insufficient memory to complete the resize.

BAD RESIZE/DEALLOCATE – RESIZE PAGED TO INVALID LENGTH

An attempt was made to resize a long array and make it a paged array where the new size is not greater than the current size.

Examples of RESIZE Statements

In the following example, the size of one-dimensional array A is changed to NEWSZ, and the previous contents of A are discarded.

```
RESIZE(A,NEWSZ,DISCARD)
```

In the following example, the size of one-dimensional array ARAY is changed to NEWSZ, and ARAY is changed to a paged array. The contents of ARAY are retained.

```
RESIZE(ARAY,NEWSZ,PAGED)
```

In the following example, the size of one-dimensional array INPUTDATA is changed to equal the value of the MAXRECSIZE attribute of file F. The previous contents of INPUTDATA are discarded.

```
RESIZE(INPUTDATA,F.MAXRECSIZE,DISCARD)
```

In the following example, the size of the specified row of array A is changed to 5, and the previous contents of that row are discarded. The other rows of A are not affected.

```
RESIZE(A[2,*],5,DISCARD)
```

In the following example, the size of one-dimensional array A is increased by 100 elements, and the previous contents of A are retained.

```
RESIZE(A,SIZE(A)+100,RETAIN)
```

In the following example, the size of the one-dimensional event array EVENTARRAY is changed to 20, and the previous contents of the array are retained. Note that RETAIN must be specified for event and interlock arrays.

```
RESIZE(EVENTARRAY,20,RETAIN)
```

In the following example, the size of the second dimension of array STUFF is changed to M. New array rows are created for the new size of the second dimension. The previous contents of the array are retained.

```
RESIZE(STUFF[I,*],M,RETAIN)
```

In the following example, the size of the specified row of array STUFF is changed to N, and the previous contents of that row are retained.

```
RESIZE(STUFF[I,J,*],N,RETAIN) % RESIZE an array row
```

RESPOND Statement

The RESPOND statement is used to enable a program to issue a positive or negative response to an offer for subfile dialog establishment or a request for orderly termination. Networks that support this function can be found in the *I/O Subsystem Programming Guide*. The program is notified of requests for dialog establishment or termination through the CHANGEEVENT and FILESTATE attributes.

<respond statement>

— RESPOND — (—<respond file part>— , —<respond options>—) ———|

<respond file part>

—<file designator>—|
| [SUBFILE —<subfile index>—] |

<respond options>

—<respondtype option>—|
| , —<associateddata option>—|

<respondtype option>

— RESPNDTYPE — = | ACCEPTOPEN |
| REJECTOPEN |
| ACCEPTCLOSE |

The RESPOND statement can be used only when the KIND of the file designator is PORT and only when the SERVICE file attribute is set to a network type that supports this feature.

RESPOND Statement Options

The subfile index, if present, specifies the subfile to which the RESPOND statement applies. If 0 (zero) is specified, the RESPOND is invoked on all subfiles in a FILESTATE that are awaiting a response.

The RESPNDTYPE option indicates the type of the response. The type of the response must be consistent with the FILESTATE. The ACCEPTOPEN and ACCEPTCLOSE options indicate that a positive response is to be generated and issued for an open indication or close indication received from the correspondent application. The REJECTOPEN option indicate that a negative response is to be generated and issued for an open indication.

The ASSOCIATEDDATA option can be used to send associated data to the correspondent endpoint with the response. If a string expression is used, the length of the expression is calculated automatically and used for the ASSOCIATEDDATALength value. Otherwise, the ASSOCIATEDDATALength option indicates how many characters are to be sent as the ASSOCIATEDDATA value.

If the ASSOCIATEDDATA value is of type HEX, the ASSOCIATEDDATALENGTH option indicates the number of HEX characters, otherwise the number of EBCDIC characters. If the ASSOCIATEDDATALENGTH is not a single-precision integer it is integerized.

The RESPOND statement can be used as an arithmetic function. It returns the same values as the file attribute AVAILABLE. For a description of these values, see the *File Attributes Programming Reference Manual*. If the result of this statement is not interrogated by the program, the program terminates if the respond action fails.

Examples of RESPOND Statements

The following program responds affirmatively to a request to close from the correspondent endpoint for the subfile of port file FILEID.

```
RESPOND (FILEID, RESPONDTYPE = ACCEPTCLOSE)
```

The following program responds to an offer for subfile I on port file FILEID by accepting the offer.

```
RESPOND (FILEID [SUBFILE I], RESPONDTYPE = ACCEPTOPEN)
```

The following program responds to an offer for subfile 1 on port file FILEID by rejecting the offer. The associated data that is stored in the string STR is sent with the response.

```
RESPOND (FILEID [SUBFILE 1], RESPONDTYPE = REJECTOPEN,  
        ASSOCIATEDDATA =STR)
```

The following program responds to an offer for subfile I on port file FILEID by accepting the offer. Twelve characters of associated data are taken from the array RA, beginning at index 0, and are sent with the response.

```
RESPOND (FILEID [SUBFILE I], RESPONDTYPE = ACCEPTOPEN,  
        ASSOCIATEDDATALENGTH = 12, ASSOCIATEDDATA = RA [0])
```

REWIND Statement

The REWIND statement causes the designated file to be closed and the file buffer areas to be returned to the system.

<rewind statement>

— REWIND — (—<file designator>—) _____|

Effects on Designated Files

If the file is a magnetic tape file, it is rewound. For disk files, the record pointer is set to the first record of the file. For more information on the file designator option, see “SWITCH FILE Declaration” in Section 3, “Declarations.”

Line printer units are released from program control. When the REWIND statement is used for a magnetic tape file that is positioned past the first reel of a multireel file, the second and subsequent reels are released from program control. Other kinds of units remain under program control.

For random access files, if the file is to be reused immediately, the statement *SEEK(<file designator>[0])* positions the file at its first record while avoiding the overhead of closing the file and then reopening it. For more information, refer to “SEEK Statement” later in this section.

Example of a REWIND Statement

In the following example, if FILEA is a disk file, the file is closed and the record pointer is set to the first record of the file. If FILEA is a magnetic tape file, the file is closed and the tape is rewound.

```
REWIND(FILEA)
```

RUN Statement

The RUN statement initiates a procedure as an independent program.

<run statement>

```

— RUN —<procedure identifier>—————→
      |—————<actual parameter part>—————|
→ [ —<task designator>— ] —————|
  
```

Initiating Procedures

Initiation of a procedure as an independent program consists of setting up a separate stack, passing any parameters (call-by-value only), and beginning the execution of the initiated procedure. For more information on the procedure identifier, see “PROCEDURE Declaration” in Section 3, “Declarations.”

The initiating program continues execution, and both the initiated procedure and the initiating program run in parallel. The initiated procedure must be compiled separately and declared EXTERNAL in the initiating program.

A procedure initiated by a RUN statement, as opposed to a PROCESS statement, is independent of the initiating program. No critical block exists for the initiated procedure, and the initiating program can finish processing while the external procedure continues running.

The contents of the designated task are copied by the operating system so that the initiated procedure has its own task variable. Before initiation, the values of the task attributes of the task, such as COREESTIMATE, STACKSIZE, and DECLARED PRIORITY, can be used to control the execution of the procedure. For information about assigning values to task attributes, refer to <arithmetic task attribute> under “Arithmetic Assignment,” <Boolean task attribute> under “Boolean Assignment,” and “Task Assignment” earlier in this section. For information about the task designator, see “TASK and TASK ARRAY Declarations” in Section 3, “Declarations.”

Because array and file parameters cannot be call-by-value, procedures with array or file parameters cannot be invoked with a RUN statement. Also, a procedure that has a pointer or string as a parameter, whether or not it is specified as call-by-value, cannot be invoked with a RUN statement.

If the specified procedure is a typed procedure, the return value is discarded.

If the procedure identifier is a system supplied process, such as an intrinsic, the library GENERALSUPPORT must be declared using a library entry point specification. The procedure identifier must be declared in the program or the syntax error PROCEDURE MUST BE USER DECLARED results.

Examples of a RUN Statement

The following example invokes procedure SIMPL, which has no parameters, as an independent program. The task TSK is copied by the operating system for SIMPL to use as its task variable.

```
RUN SIMPL [TSK]
```

The following example invokes procedure DOOER as an independent program, passing the four parameters X, Y, Z, and the string literal "ABCD". Though the value "ABCD" appears as a string literal, it is passed to a call-by-value REAL parameter. The task designated by TSKARRAY[INDEX] is used by DOOER as its task variable.

```
RUN DOOER(X,Y,Z,"ABCD") [TSKARRAY[INDEX]]
```


SCAN Statement

The SCAN statement examines a contiguous portion of character data in an array row, one character at a time, in a left-to-right direction.

<scan statement>

— SCAN —<source>—<scan part>—————|

For more information on <source> and <scan part>, see “REPLACE Statement” earlier in this section.

The source is always a pointer expression, and at the completion of the SCAN statement the final value of the stack-source-pointer can be stored in a pointer variable.

The scan part is basically a testing operation that determines when the SCAN statement is to stop. The scan part can specify that scanning is to stop after a given number of source characters, or when a source character fails or passes a specified test.

The count part is used in a scan part when a limited number of source characters are to be scanned. A residual count can be used, in which case the value of the remaining count is stored in the specified simple arithmetic variable at the completion of the SCAN statement.

The relational operator in the condition option specifies the comparison to be made between the arithmetic expression and the source characters. The arithmetic expression can be of any valid form, but most often takes the form of a one-character string literal.

Before the scan operation begins, the arithmetic expression in the condition option is evaluated and the value of bits [7:8] or [3:4] (depending on the character size of the source pointer) of the arithmetic expression is assigned to the stack-source-operand.

Scan Part Combinations

The formal syntax of the <scan part> can be reduced to the following combinations:

```
WHILE <relational operator> <arithmetic expression>  
UNTIL <relational operator> <arithmetic expression>
```

```
WHILE IN <truth set table>  
UNTIL IN <truth set table>
```

```
FOR <count part> WHILE <relational operator> <arithmetic expression>  
FOR <count part> UNTIL <relational operator> <arithmetic expression>
```

```
FOR <count part> WHILE IN <truth set table>  
FOR <count part> UNTIL IN <truth set table>
```

Each of these combinations is discussed in the following separate sections. Because all combinations of the SCAN statement begin with <source>, each description of a combination begins with the assumption that the stack-source-pointer has been initialized to the source pointer.

The scan parts that contain a count part examine, or scan, source characters until either the number of characters specified by the arithmetic expression in the count part have been examined or a source character fails or passes the test specified by the condition syntax. The scan parts that do not contain a count part examine source characters until either a source character fails or passes the test specified by the condition syntax or the end of the array is reached. If the end of the array is reached, the program is discontinued with a paged (segmented) array error.

Scan Parts without Count Parts

WHILE <relational operator> <arithmetic expression>

Characters are scanned as long as they pass the test. For example, the following statement scans the characters pointed to by P as long as a period (.) is not encountered:

```
SCAN P WHILE NEQ "."
```

UNTIL <relational operator> <arithmetic expression>

Characters are scanned until a source character passes the test. For example, the following statement scans the characters pointed to by P until a blank character is encountered:

```
SCAN P:P UNTIL = " "
```

P is updated to point to the blank character that passed the test.

WHILE IN <truth set table>

Characters are scanned as long as they are members of the truth set. For example, the following statement scans the characters pointed to by P as long as they are members of the truth set ALPHA8:

```
SCAN P:P WHILE IN ALPHA8
```

P is updated to point to the first character that is not a member of ALPHA8.

UNTIL IN <truth set table>

Characters are scanned until a source character is found that is a member of the truth set. For example, the following statement scans the characters pointed to by P until a member of the truth set ALPHA8 is encountered:

```
SCAN P:P UNTIL IN ALPHA8
```

P is updated to point to the first character that is a member of ALPHA8.

Scan Parts with Count Parts**FOR <count part> WHILE <relational operator> <arithmetic expression>**

The stack-integer-counter is initialized to the value of the arithmetic expression in the count part. Characters are then scanned and the stack-integer-counter is decremented for each character as long as the stack-integer-counter is not zero and a source character passes the test.

For example, the following statement scans the first 20 characters pointed to by P as long as a period (.) is not encountered:

```
SCAN P FOR N:20 WHILE NEQ "."
```

Because N reflects how many of the 20 characters have yet to be scanned, it can be used to determine whether a period was encountered and, if so, where the period is.

FOR <count part> UNTIL <relational operator> <arithmetic expression>

The stack-integer-counter is initialized to the value of the arithmetic expression in the count part. Characters are then scanned and the stack-integer-counter is decremented for each character until either the stack-integer-counter is zero or a source character passes the test. For example, the following statement scans the first N characters pointed to by P until the first nonblank character is encountered:

```
SCAN P:P FOR N:N UNTIL NEQ " "
```

If, when the statement is invoked, the value of N is the number of characters between P and the end of the array row, then because both P and N are updated in this statement, at the completion of the statement, N gives the number of characters between the updated P and the end of the array row.

FOR <count part> WHILE IN <truth set table>

The stack-integer-counter is initialized to the value of the arithmetic expression in the count part. Characters are then scanned and the stack-integer-counter is decremented for each character as long as the stack-integer-counter is not zero and source characters are members of the truth set. For further information on truth sets, see "TRUTHSET Declaration" in Section 3, "Declarations."

For example, the following statement scans the first 20 characters pointed to by P as long as they are members of the truth set ALPHA8:

```
SCAN P:P FOR N:20 WHILE IN ALPHA8
```

P is updated to point to the first character that is not a member of ALPHA8, or, if all of the 20 characters scanned are members of ALPHA8, to the character that is 20 characters beyond the initial position of P. N is assigned the number of characters yet to be scanned.

FOR <count part> UNTIL IN <truth set table>

The stack-integer-counter is initialized to the value of the arithmetic expression in the count part. Characters are then scanned and the stack-integer-counter is decremented for each character until either the stack-integer-counter is zero or a source character is a member of the truth set. For further information on truth sets, see "TRUTHSET Declaration" in Section 3, "Declarations."

For example, the following statement scans the first 20 characters pointed to by P until a member of the truth set ALPHA8 is encountered:

```
SCAN P:P FOR 20 UNTIL IN ALPHA8
```

P is updated to point to the first character that is a member of ALPHA8, or, if none of the 20 characters scanned are members of ALPHA8, to the character that is 20 characters beyond the initial position of P.

Examples of SCAN Statements

SCAN PTR WHILE = " "

SCAN PTR UNTIL NEQ 48"00"

SCAN PTR:PTR WHILE IN ALPHA

SCAN PTR UNTIL IN ALPHA8

SCAN PTR:PTR WHILE IN ACCEPTABLE[0]

SCAN PTR FOR 50 WHILE > "Z"

SCAN PTR:PTR FOR X:80 UNTIL = "."

SCAN PTR FOR RMNDR:960 WHILE NEQ 48"1D"

SCAN PTR:PTR FOR ZED:ZED WHILE IN ALPHA8

SCAN PTR FOR 80 UNTIL IN GOODSTUFF[5]

SEEK Statement

The SEEK statement positions the record pointer for the designated file at the specified record. This record is read or written by the next serial I/O operation.

<seek statement>

— SEEK — (—<file designator>— [—<record number>—] —) ———|

<record number>

—<arithmetic expression>—————|

A serial I/O operation is a READ statement or WRITE statement that does not include a record number in the record number or carriage control part. The SEEK statement does not affect any nonserial I/O statements. The value of the record pointer is not saved when the file is closed.

SEEK Statement as a Boolean Function

The SEEK statement can be used as a Boolean function. When the statement fails, the value TRUE is returned. When the statement is successful, the value FALSE is returned. Specifically, the SEEK statement returns a value identical to that returned by the file attribute STATE. For more information, refer to the discussion of the STATE attribute in the *File Attributes Programming Reference Manual*.

The file designator must not reference a direct file or a direct switch file.

When the record number is less than one, the record pointer points at the first record.

Example of a SEEK Statement

The following example positions the record pointer of file FILEA to record number $X + 2 * Y$.

```
SEEK(FILEA[X+2*Y])
```

SET Statement

The SET statement sets the happened state of the designated event to TRUE (happened).

<set statement>

— SET — (—<event designator>—) _____|

SET Statement Options

For more information on the event designator, see “EVENT and EVENT ARRAY Declarations” in Section 3, “Declarations.” The SET statement does not activate any tasks waiting on the event.

To set the happened state of an event to TRUE (happened) and activate the tasks waiting on the event, use the CAUSE statement. For more information, see “CAUSE Statement” earlier in this section.

Examples of SET Statements

The following example sets the happened state of EVNT to TRUE (happened).

```
SET(EVNT)
```

The following example sets the happened state of the event designated by EVNTARRAY[INDX] to TRUE (happened).

```
SET(EVNTARRAY[INDX])
```

SETABSTRACTVALUE Statement

The SETABSTRACTVALUE statement causes the file attribute to be set to the abstract value.

<setabstractvalue statement>

```
— SETABSTRACTVALUE — ( —<file designator>—————>
→
→ [ ( —<attribute parameter list>— ) ] , —————>
→ <arithmetic-valued file attribute name> , —————>
→ [ <Boolean-valued file attribute name>
  <pointer-valued file attribute name> ]
→ <abstract value mnemonic>— ) —————>
```

<attribute parameter list>

```
— [ /2\—<arithmetic expression>— ] —————>
```

ALGOL supports all file attributes and file attribute values described in the *File Attributes Programming Reference Manual*.

Setting the File Attribute

The SETABSTRACTVALUE statement can be used to set the file attribute to one of the abstract value mnemonics allowed for that file attribute.

Examples of SETABSTRACTVALUE Statements

The following example sets the arithmetic-valued file attribute AREAS to its default value:

```
SETABSTRACTVALUE (F, AREAS, DEFAULT)
```

The following example sets the arithmetic-valued file attribute AREAS to its maximum value:

```
SETABSTRACTVALUE (F, AREAS, MAXIMUM)
```

The following example sets the arithmetic-valued file attribute AREAS to its minimum value:

```
SETABSTRACTVALUE (F, AREAS, MINIMUM)
```

The following example sets the mnemonic-valued file attribute INTMODE to its default value:

```
SETABSTRACTVALUE (F, INTMODE, DEFAULT)
```


SORT Statement

The SORT statement invokes the sort intrinsic, which provides a means for designated data to be sorted and placed in a file or returned to a procedure.

<sort statement>

```
— SORT — ( —<output option>— , —<input option>— , —————→
→<number of tapes>— , —<compare procedure>— , —<record length>————→
→┌—————┐ ) ┌—————┐┌—————┐
   <size specifications>   <restart specifications>
```

The data to be sorted is indicated by the input option. The output option indicates where the sorted data is to be placed. The order in which the data is sorted is determined by the compare procedure.

Output Option

<output option>

```
—┌<file designator>┐—————┐
  └<output procedure>┘
```

<output procedure>

```
—<procedure identifier>—————┐
```

If a file designator is specified as the output option, the sort intrinsic writes the sorted output to this file. When sorting is completed, the sort intrinsic closes the file. If the file is a disk file for which the file attribute SAVEFACTOR has a nonzero value, it is closed and locked. The output file must not be open when it is passed to the sort intrinsic by the program.

If an output procedure is specified as the output option, the sort intrinsic calls the output procedure once for each sorted record and once to allow end-of-output action. This procedure must be untyped, must not be declared EXTERNAL, and must have two parameters. The first parameter must be a call-by-value Boolean variable, and the second parameter must be a one-dimensional array with a lower bound of zero. The Boolean parameter is FALSE as long as the second parameter contains a sorted record. When all records are returned, the first parameter is TRUE and the second parameter must not be accessed.

SORT Statement

The following is an example of an output procedure:

```
PROCEDURE OUTPROC(B,A);
VALUE B;
BOOLEAN B;
ARRAY A[0];
BEGIN
  IF B THEN
    CLOSE(FILEID,PURGE)
  ELSE
    WRITE(FILEID,RECSIZE,A[*]);
END OUTPROC;
```

Input Option

<input option>

—<file designator>—
└─<input procedure>┘

<input procedure>

—<procedure identifier>—

If a file designator is used as the input option, the file supplies input records to the sort intrinsic. This file is closed after the last record is read. Disk files are closed with regular close action, and nondisk files are closed with release action. The input file must not be open when it is passed to the sort intrinsic by the program. The input file cannot be a file that is declared to be DIRECT.

If an input procedure is used as the input option, the procedure is called to furnish input records to the sort intrinsic. The input procedure must be a Boolean procedure, must not be declared EXTERNAL, and must have a one-dimensional array with a lower bound of zero as its only parameter. This procedure, on each call, either inserts the next record to be sorted into its array parameter or returns the value TRUE, which indicates the end of the input data.

When TRUE is returned by the input procedure, the sort intrinsic does not use the contents of the array parameter and does not call the input procedure again.

The following is an example of an input procedure that can be used when sorting N elements of array Q:

```

BOOLEAN PROCEDURE INPROC(A);
ARRAY A[0];
  BEGIN
    N:= *-1;
    IF N GEQ 0 THEN
      A[0]:= Q[N]
    ELSE
      INPROC:= TRUE;
    END INPROC;
  
```

Note: *The sort intrinsic maintains a logical record structure in memory. Be careful not to exceed the length of the record when manipulating records; data corruption might occur.*

Number of Tapes

<number of tapes>

—<arithmetic expression>—————|

The value of <number of tapes> specifies the number of tape files that can be used, if necessary, in the sorting process. If the value of the arithmetic expression is zero, no tapes are used. If the value of the arithmetic expression is between 1 and 3, inclusive, three tapes are used. If the value of the arithmetic expression is between 3 and 8, the specified number of tapes are used. If the value of the arithmetic expression is 8 or more, a maximum of eight tapes are used.

Compare Procedure

<compare procedure>

—<procedure identifier>—————|

The compare procedure is called by the sort intrinsic to apply the appropriate sort criteria to a pair of input records. The procedure must be a Boolean procedure, must not be declared EXTERNAL, and must have exactly two parameters. Each of the parameters must be a one-dimensional array with a lower bound of zero. Every time two input records are to be compared, the sort intrinsic calls the compare procedure and passes the two records to the compare procedure through the array parameters. If the compare procedure returns TRUE, the record passed to the first array precedes, in the sorted output, the record passed to the second array. If the compare procedure returns FALSE, the record passed to the second array precedes the record passed to the first array.

SORT Statement

The following is an example of a compare procedure that can be used to sort arithmetic data in ascending sequence:

```
BOOLEAN PROCEDURE CMP(A,B);
ARRAY A,B[0];
BEGIN
  CMP:= A[0] < B[0];
END CMP;
```

For alphanumeric comparisons, the following compare procedure can be used to sort data in ascending sequence:

```
BOOLEAN PROCEDURE CMP(A,B);
ARRAY A,B[0];
BEGIN
  CMP:= POINTER(A) LSS POINTER(B) FOR 6;
END CMP;
```

The CMP procedures above return TRUE if the value in A[0] compares as less than the value in B[0] and return FALSE if the value in A[0] compares as greater than or equal to the value in B[0]. Therefore, if A[0] is less than B[0], the content of array A is passed to the output file or procedure before the content of array B, and if A[0] is greater than or equal to B[0], the content of array B is passed to the output file or procedure before the content of array A. If either of these compare procedures is used, word zero of the input records is considered to be the key on which sorting is done.

For the actual comparison, a string relation can be used to compare a string from each record (according to the EBCDIC collating sequence), or an arithmetic relation can be used to compare an arithmetic value from each record. The comparison can be done on one or more fields, called keys, from each record or on the entire record. The manner in which the comparison is done is specified entirely by the programmer.

Record Length

<record length>

—<arithmetic expression>—————|

The record length specifies the length, in words or characters (depending on whether the array parameters of the compare procedure are word or character arrays, respectively) of the largest item that is to be sorted. If the value of the arithmetic expression is not a positive integer, the largest integer that is not greater than the absolute value of the expression is used; for example, a record length of 12 is used if the expression has a value of -12.995. If the value of the arithmetic expression is zero, the program terminates.

Size Specifications

<size specifications>

```
— , —<memory size> —————|
      | , —<disk size> —————|
      | —<pack size> —————|
```

<memory size>

```
—<arithmetic expression>—————|
```

<disk size>

```
—<arithmetic expression>—————|
```

<pack size>

```
— PACK —————|
      | —<arithmetic expression> —————|
```

The size specifications allow the programmer to specify the maximum amount of main memory and disk storage to be used by the sort intrinsic.

The memory size specifies the maximum amount, in words, of main memory that is to be used. If the memory size is unspecified, a value of 12,000 is assumed.

The disk size specifies the maximum amount, in words, of disk storage that can be used. If the disk size is unspecified, a value of 600,000 is assumed.

If the pack size is specified, temporary files created by the sort intrinsic have PACK, instead of DISK, as the value of their FAMILYNAME attribute. For an explanation of the FAMILYNAME attribute, refer to the *File Attributes Programming Reference Manual*. If the arithmetic expression option does not appear in the pack size element, a value of 600,000 words is assumed.

Restart Specifications

<restart specifications>

— [— RESTART — = —<arithmetic expression>—] _____|

The restart specifications allow the sort intrinsic to resume processing at the most recent checkpoint after discontinuation of a program. The program must provide logic to restore and maintain variables, arrays, files, pointers, and so forth, which are defined for, and by, the program. In other words, the program must provide the means to restore everything that is necessary for the program to continue from the point of interruption. The restart capability is implemented only for disk sorts.

The sort intrinsic inspects the least significant (rightmost) five bits of the value of the arithmetic expression in the restart specifications to determine the course of action it is to take. To control the sort, these bits can be set by the program. The meanings of these bits are explained in the following table.

Bit	Value	Description
0	1	The program is restarting a previous sort. The sort intrinsic tries to open its two disk files and obtain restart information. If it is successful in obtaining this information, the sort intrinsic tries to continue from the most recent restart point.
0	0	The sort is starting from the beginning. If the sort is restartable, and previous sort files with identical titles exist, they are removed and replaced by new sort files.
1	1	The program is requesting a restartable sort. The sort intrinsic saves its two internal files and can be restarted on program request. If bit 2 is 1, bit 1 is set to 1 by default.
1	0	A normal sort is requested, and no sort files are saved (unless bit 2 is 1, which sets bit 1 to 1 by default).
2	1	The program is requesting a restartable sort and desires extensive error recovery from I/O errors. If bit 2 is 1, the sort intrinsic attempts to backtrack and remerge strings, as necessary, when I/O errors occur during the accessing of either of the two sort files. To use this option, the program must provide at least three times as much disk space as required to contain the input data. If less disk space is provided, the sort intrinsic emits an error message, changes to restartable-only mode, and continues the sort without further use of backtracking capability.
2	0	Recovery from internal errors is not requested.
3	–	Bit 3 has meaning only if a restartable sort is requested. The use of this option controls the sort during the stringing phase as the user input is being read by the sort intrinsic. Use of this bit determines how the sort restarts (when a restart is requested) only if the restart occurs while the sort is in the stringing phase.

Bit	Value	Description
3	1	The program requires that the sort restart at the beginning of the user input. This restart is the equivalent of starting an entirely new sort. In case the restarted sort passes from the stringing phase into the merge phase, it continues from the merge phase. This bit can be set to 1 during a restart, even if it is not 1 initially. Once set to 1, it cannot be set to 0 by subsequent restarts.
3	0	The program requires the ability to restart at the last restart point that occurred during the stringing phase. If the sort is still in the stringing phase, it skips over the records already processed and continues from the last restart point. If the sort is in the merge phase, it continues from the last merge phase restart point. If bit 3 is 0, the sort is normally less efficient because more strings are created during the stringing phase.
4	–	This bit is reserved for expansion and is not currently used by the sort intrinsic.

Arrays in Sort Procedures

The array parameters used by the input procedure, output procedure, and compare procedure must be similarly specified. For example, if one procedure declares its array parameter as an EBCDIC array, then all must declare their array parameters as EBCDIC arrays.

When character arrays are used in the procedures passed to the sort intrinsic, the record length parameter is interpreted as a length in characters.

For more detailed information about the sort intrinsic, refer to "SORT" in the *System Software Utilities Operations Reference Manual*.

SORT Mode

The combination of the disk size and number of tapes determines the sort mode as follows:

Number of Tapes	Disk Size	Sort Mode
NEQ 0	0	Tape Only
NEQ 0	NEQ 0	Integrated-Tape-Disk (ITD)
0	NEQ 0	Disk Only
0	0	Core Only

Examples of SORT Statements

The following example sorts the records of file FILEIN according to compare procedure COMPARE and writes the sorted data to file FILEOUT. Three tapes are used in the sort and the record length is 10.

```
SORT(FILEOUT,FILEIN,3,COMPARE,10)
```

The following example sorts the records provided by procedure INPROC according to compare procedure COMPARER, and writes sorted data out according to procedure OUTPROC. The number of tapes is given by NUMOFTAPES, and the record size is given by DSKSZ. A restart specification is given by PARAM.

```
SORT(OUTPROC,INPROC,NUMOFTAPES,COMPARER,DSKSZ) [RESTART = PARAM]
```


SPACE Statement

The SPACE statement is used to bypass records in a file without reading those records.

<space statement>

```
— SPACE — ( —<file designator>— , —<arithmetic expression>— ) —>
→ ┌──────────────────────────────────────────────────────────────────────────────────┐
  │ <action labels or finished event> ───────────────────────────────────────────┘
```

The value of the arithmetic expression determines the number of records to be spaced and the direction of the spacing. If the value of the arithmetic expression is positive, the records are spaced in a forward direction; if it is negative, the records are spaced in the reverse direction.

SPACE Statement as a Boolean Function

The SPACE statement can be used as a Boolean function. When the statement fails, the value TRUE is returned. When the statement is successful, the value FALSE is returned. The SPACE statement returns a value identical to that returned by the file attribute STATE. For more information, refer to the discussion of the STATE attribute in the *File Attributes Programming Reference Manual*.

The file designator must not reference a direct file or a direct switch file. For more information on the file designator see "SWITCH FILE Declaration" in Section 3, "Declarations."

Examples of SPACE Statements

The following example spaces file FYLE forward 50 records.

```
SPACE(FYLE,50)
```

The following example spaces file FILEID a number of records and a direction given by the value of N. If an end-of-file condition occurs, the program continues execution with the statement associated with the label LEOF.

```
SPACE(FILEID,N) [LEOF]
```

The following example spaces file FILEID backward 3 records. A value is assigned to B indicating the success or failure of the spacing. If an end-of-file condition occurs, the program continues execution with the statement associated with the label LEOF.

```
B := SPACE(FILEID,-3) [LEOF]
```

SWAP Statement

The SWAP statement assigns the value of the variable on the right side of the swap operator (:=) to the variable on the left side of the swap operator, and assigns the value of the variable on the left side of the swap operator to the variable on the right side of the swap operator. The lexical levels of the references must match, along with the type of the procedure reference variables and the parameters of the procedure reference variables.

<swap statement>

<integer variable>	:=	<integer variable>
<real variable>	:=	<real variable>
<double variable>	:=	<double variable>
<Boolean variable>	:=	<Boolean variable>
<complex variable>	:=	<complex variable>
<array reference variable>	:=	<array reference variable>
<pointer variable>	:=	<pointer variable>
<procedure reference variable>	:=	<procedure reference variable>

<integer variable>

A variable of type INTEGER.

<real variable>

A variable of type REAL.

<double variable>

A variable of type DOUBLE.

Variable Type Matching

The declared types of the variables on either side of the swap operator (:=) must be the same. Partial word swaps are not permitted.

Descriptions of the processes of an assignment are found under "ASSIGNMENT Statement" earlier in this section.

Example of a SWAP Statement

This example program uses the SWAP statement to sort a real array.

```

BEGIN
  FILE REM(KIND=DISK,TITLE="SORT/OUT.",PROTECTION=SAVE);
  BOOLEAN SWAP_DONE;
  INTEGER I,J;
  DEFINE LASTONE = 5;;
  INTEGER ARRAY ARY[0:LASTONE];

  PROCEDURE SORTER;
  BEGIN

    BOOLEAN PROCEDURE NEED_TO_SWAP(A,B);
    VALUE A,B;
    INTEGER A,B;
    BEGIN
      IF (A < B) THEN
        NEED_TO_SWAP:= TRUE
      ELSE
        NEED_TO_SWAP:= FALSE;
      END NEED_TO_SWAP;

    SWAP_DONE:= TRUE;
    FOR I:= 0 STEP 1 WHILE (I < LASTONE AND SWAP_DONE) DO
      BEGIN
        SWAP_DONE:= FALSE;
        FOR J:= I+1 STEP 1 UNTIL LASTONE DO
          IF (NEED_TO_SWAP(ARY[I],ARY[J])) THEN
            BEGIN
              SWAP_DONE:= TRUE;
              ARY[I]:= ARY[J];
              END;
          END FORLOOP;
        END SORTER;

    ARY[0]:= "  SAM";
    ARY[1]:= "  AL";
    ARY[2]:= "  HAL";
    ARY[3]:= "  BOB";
    ARY[4]:= "  TOM";
    ARY[5]:= "  SID";
    SORTER;
    FOR I:= 0 STEP 1 UNTIL LASTONE DO
      WRITE(REM,<A6>,ARY[I]);
    END.

```

THRU Statement

The THRU statement executes a statement a specified number of times.

<thru statement>

— THRU —<arithmetic expression>— DO —<statement>—————|

Value of the Arithmetic Expression

The absolute value of the arithmetic expression is evaluated and rounded to an integer, if necessary. This value determines the number of times the statement following DO is executed. The upper limit of this value is $2^{**}39 - 1$. Figure 4-6 illustrates the THRU loop.

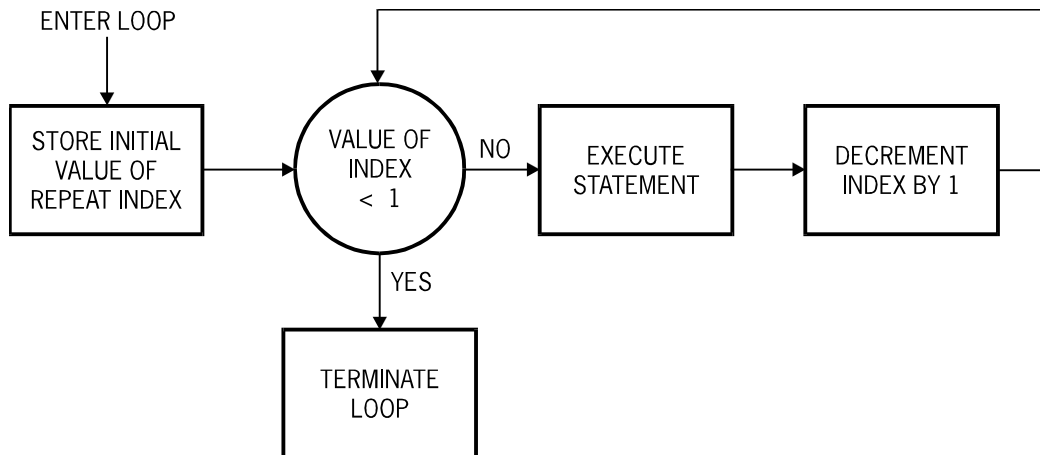


Figure 4-6. THRU Loop

Examples of THRU Statements

In the following example, the statement LOADCHAR is executed 255 times.

```
THRU 255 DO
LOADCHAR
```

In the following example, the REAL function is evaluated, the value is assigned to MAXI, and the statement SKIP1 is executed MAXI times.

```
THRU MAXI := REAL(PTR,3) DO
SKIP1
```

TRY Statement

The TRY statement provides a general error/fault protection mechanism that allows a program to maintain normal flow of control.

<try statement>

```
— TRY —<try normal form> —
      |
      |<try limited form>|
```

<try normal form>

```
—<statement> —<statement>
      |
      |ELSE —<statement>|
```

<try limited form>

```
— [ —<error procedure>— ] —<procedure invocation statement> —
      |
      |<procedure reference statement>|
```

<error procedure>

A procedure identifier that is not imported or declared external or formal.

Explanation

TRY Normal Form

The normal form of the TRY statement can be used to protect any statement, including a compound statement or a procedure invocation statement. This form of the TRY statement enables multiple sets of error recovery code to be specified. If the statement following the TRY verb fails to finish normally, the statement following the first ELSE clause is executed. For an explanation of the phrase “finish normally,” see “Considerations for Use” later in this section.

If the statement following the first ELSE clause then fails to finish normally, the statement following the second ELSE clause is executed, if one exists. This pattern is repeated through all remaining ELSE clauses until the last one. No protection is provided if the statement following the last ELSE clause fails to finish normally. ELSE clauses are not executed if any of the preceding statements are executed successfully without fault. In the normal form of the TRY statement, another normal-form TRY statement cannot be used as the <statement> portion unless it is part of a block statement or compound statement.

TRY Limited Form

The limited form of the TRY statement can be used only to protect a procedure invocation statement. This form of TRY has a more limited syntax and provides more limited protection. Unlike the normal form, the protection afforded by the limited form does not begin until the procedure begins execution. For example, if an exception occurs during parameter building, or the procedure being invoked is a null procedure reference, no protection is available. However, if an attempt is made to invoke an imported procedure that is not available, the limited form of TRY provides protection if it is executed on machines capable of running with TARGET = LEVEL4 (or higher).

TRY Statement

If a procedure fails to finish normally, the error procedure is automatically invoked. No protection is provided if the error procedure fails to finish normally.

The error procedure provided must be a procedure declared with no parameters. The error procedure cannot be a procedure imported from a library or declared EXTERNAL, and cannot be a procedure reference variable.

If a typed procedure is being invoked for the procedure call, a typed procedure must be specified for the error procedure. If an untyped procedure is being invoked for the procedure call, an untyped procedure must be specified for the error procedure.

Considerations for Use

The phrase “finish normally,” in the previous explanations, means completion of the statement or procedure without run-time errors, faults, or conditions that could cause a process to be terminated. These conditions include, but are not limited to, faults such as divide by zero, invalid index, stack overflow, task attribute errors, security violations, untrapped I/O errors, and termination causes such as programmatic termination. These conditions include only those exception conditions that would result in program termination.

Some termination causes ignore all TRY occurrences that are invoked after a specified point in the execution history of a process. These causes include stack overflow and library delink errors. For example, if a process has invoked an imported library procedure that has not yet exited when a library delink error occurs, any TRY verbs that occurred after the procedure invocation are ignored. If a stack overflow occurs, TRY occurrences are ignored unless they are sufficiently far down the stack that the stack can be cut back to the TRY statement and still be at a safe size.

If TRY is used in the same process as ON statements, whichever one is closest to the exception point and capable of handling the condition is used.

If TRY is used in a block that declares an EXCEPTION procedure, the EXCEPTION procedure is invoked only if the TRY does not handle the condition.

A program dump may be generated if the OPTION task attribute has the FAULT option selected, even if the fault is handled by the TRY statement. Messages associated with the fault are suppressed if the fault could have been handled by the ON statement.

UNLOCK Statement

The UNLOCK statement relinquishes an acquired interlock.

<unlock statement>

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

The UNLOCK statement is used to relinquish an interlock that was previously acquired by a LOCK interlock statement. UNLOCK can be used as a function. If UNLOCK is used as a statement, the process is discontinued when the result is a value other than 1 (successfully unlocked).

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

Value	Meaning
1	The interlock was successfully unlocked.
6	The interlock has a state of FREE and therefore cannot be locked.

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_CONTENTENDED, a result of 1 is returned, and the interlock is given to the first contended process in the contender list. If only one contender is in the list, the state is changed to LOCKED_UNCONTENDED; otherwise it is not changed.

Examples of UNLOCK Statements

The following examples show valid uses of the UNLOCK statement:

```
I := UNLOCK (CONN_LIB[7].MYLOCK);
```

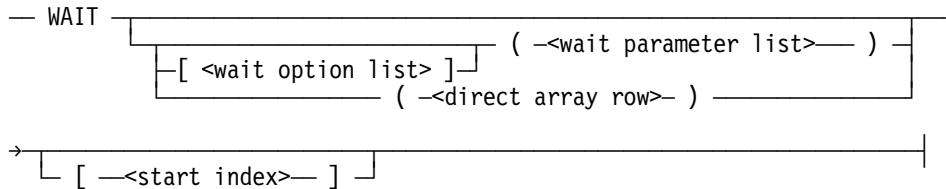
```
I := UNLOCK (MYLOCK);
```

```
UNLOCK (YOURLOCKS [2]);
```

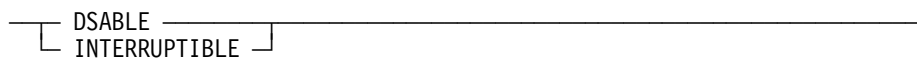
WAIT Statement

The WAIT statement suspends a program until a designated condition occurs. The program can be suspended until a given length of time elapses, an event is caused, a previously initiated direct I/O statement is finished, a software interrupt occurs, or a signal occurs.

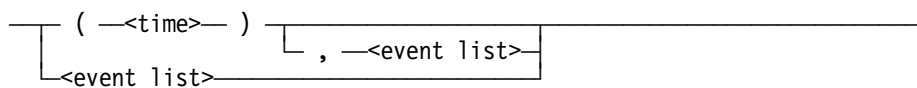
<wait statement>



<wait option list>



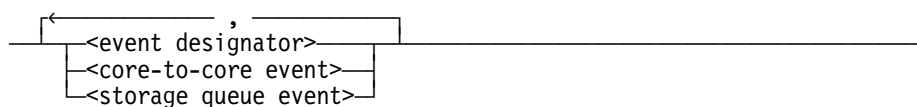
<wait parameter list>



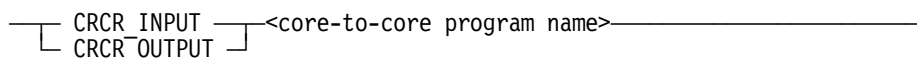
<time>



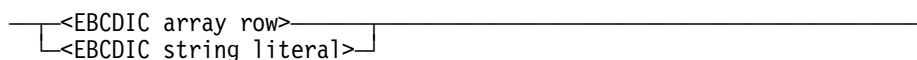
<event list>



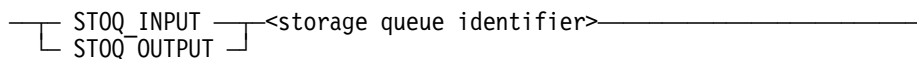
<core-to-core event>



<core-to-core program name>



<storage queue event>



<storage queue identifier>



<start index>

—<arithmetic expression>—————|

If the WAIT statement consists solely of WAIT, an operating system procedure is called that suspends the program until an attached and enabled interrupt is invoked as a result of the associated event being caused. For more information, refer to “INTERRUPT Declaration” in Section 3, “Declarations.” This form of the WAIT statement cannot be used as a function.

Wait Option List

The DSABLE option specifies that a wait in an isolated procedure can be terminated when the isolated procedure is being run by another process and that process was discontinued. If this occurs, the value returned by a WAIT function is zero.

The INTERRUPTIBLE option specifies that a wait can be terminated when the program receives a signal. If this occurs, the value returned by a WAIT function is zero.

WAIT Parameter List

When only one event appears in a statement of the form *WAIT(<event list>)*, the event is examined to determine whether its happened state is TRUE (happened) or FALSE (not happened). If the happened state of the event is TRUE, the program continues executing with the next statement. If the happened state of the event is FALSE, the program is suspended until the event is caused.

Generally, an IMPORTED EVENT or IMPORTED EVENT ARRAY element cannot be used as an event designator in an event list. However, an IMPORTED EVENT that is declared in a CONNECTION BLOCK can be used as an event designator if the compiler control option WAITIMPORT is set.

When a statement of the form *WAIT((<time>))* is executed, execution of the program is suspended for <time> seconds. Refer to “WHEN Statement” later in this section for a discussion of <time>.

When the statement includes an event list in the wait parameter list, the program is suspended until any one event in the event list is caused or until <time> seconds, if specified, have elapsed.

The *WAIT(<wait parameter list>)* form can be used as an arithmetic function that returns an integer value, starting at 1, that represents the position in the wait parameter list of the item that caused the program to be activated. For example, in the following statement the value of T is 1 if elapsed time caused the program to be activated:

```
T := WAIT((.001),E1,E2)
```

WAIT Statement

In the following statement the value of T is 2 if a cause operation on event E2 activated the program:

```
T := WAIT (E1,E2,E3)
```

This mechanism guarantees that one and only one parameter activates the program.

In the following statement, the value of T is 0 (zero) if a signal was received by the process before any of the events were caused:

```
T := WAIT [INTERRUPTIBLE] (E1,E2,E3)
```

When STOQ_INPUT is specified the program is suspended until a STOQUE entry is available. When STOQ_OUTPUT is specified the program is suspended until space is available for storage of data. The storage queue identifier specifies the appropriate STOQUE parameter block.

When CRCR_INPUT is specified the program is suspended until the sending program is ready to send the data. When CRCR_OUTPUT is specified the program is suspended until the receiving program is ready to receive the data. The core-to-core program name specifies the name of the program subject to the receive and send by way of core-to-core communication. It is a file title ended by a period or the null character.

The WAIT statement with a wait parameter list and the WAITANDRESET statement are identical, except for the state to which the caused event is set during the cause process. If a program is waiting on an event because of the WAIT statement, then the happened state of the event is set to TRUE (happened). If a program is waiting on an event because of a WAITANDRESET statement, then the happened state of the event is set to FALSE (not happened).

Start Index

The *WAIT(<wait parameter list> [<start index>]* form is used to specify a round-robin or other sequence of testing. The arithmetic expression <start index> represents the position of the item in the <wait parameter list> that should be tested first. The index of the first item is one, the second is two, and so on. If the value of the <start index> is less than one or greater than the last item in the list, it is set to one. Note that if a value for (<time>) is specified in the wait parameter list, it is always tested first.

Direct Array Row

The form *WAIT(<direct array row>)* is one of the ways in which a program can determine if a previously initiated direct I/O statement has finished. This form can be used as a Boolean function. When the I/O statement fails, the value TRUE is returned. When the statement is successful, the value FALSE is returned. Specifically, this form of the WAIT statement returns a value similar to that returned by the file attribute IORESULT, with certain exceptions. Refer to the discussion of the IORESULT attribute in the *File Attributes Programming Reference Manual*.

Examples of WAIT Statements

In the following example, if the happened state of event EVNT is TRUE (happened), the program continues with the next statement. Otherwise, the program is suspended until EVNT is caused, and the happened state of EVNT is set to TRUE (happened).

```
WAIT(EVNT)
```

In the following example, if the happened state of event EVNT1, EVNT2, or EVNT3 is TRUE (happened), the program continues with the next statement. Otherwise, the program is suspended until one of the events EVNT1, EVNT2, or EVNT3 is caused, and the happened state of that event is set to TRUE (happened).

```
WAIT(EVNT1, EVNT2, EVNT3)
```

The following example is identical to the previous one except that the first event tested is determined by the value of START_EVENT.

```
WAIT(EVNT1, EVNT2, EVNT3) [START_EVENT]
```

In the following example, if the happened state of WAKEUP or GOAWAY is TRUE (happened), the program continues with the next statement. Otherwise, the program is suspended until NAPTIME seconds have elapsed or until event WAKEUP or GOAWAY is caused. If WAKEUP or GOAWAY is caused, its happened state is set to TRUE (happened). The value stored in X is 1, 2, or 3, indicating which of the three items reactivated the program.

```
X := WAIT((NAPTIME), WAKEUP, GOAWAY)
```

In the following example, if the happened state of WAKEUP or GOAWAY is TRUE (happened), the program continues with the next statement. Otherwise, the program is suspended until NAPTIME seconds have elapsed or until event WAKEUP or GOAWAY is caused or the process receives a signal. If a signal is received before the WAKEUP or GOAWAY is caused (happened) and before NAPTIME seconds elapses, the value stored in X is 0 (zero), indicating that the wait was interrupted by a signal.

```
X := WAIT [INTERRUPTIBLE] ((NAPTIME), WAKEUP, GOAWAY)
```

In the following example, if DIRINPUT is a direct array row, the program is suspended until the direct I/O operation associated with DIRINPUT is completed. If the I/O operation fails, the value TRUE is assigned to RSLT. If the operation is successful, the value FALSE is assigned to RSLT.

```
RSLT := WAIT(DIRINPUT)
```

In the following example, the program is suspended until an attached and enabled interrupt is invoked as a result of the associated event being caused.

```
WAIT
```

WAIT Statement

In the following example, if a STOQUE entry is available in the storage queue identified by STOQ1 or if OBJECT/TEST is ready to send data, the program continues with the next statement. Otherwise, the program is suspended until either of these conditions is true. The value stored in WHICH is 1 or 2, indicating which of the two items reactivated the program.

```
WHICH := WAIT (STOQ_INPUT STOQ1, CRCR_INPUT "OBJECT/TEST.")
```

WAITANDRESET Statement

The WAITANDRESET statement suspends a program until a designated condition occurs.

<waitandreset statement>

```

— WAITANDRESET —————→
    | [ <wait option list> ] |
→ ( —<wait parameter list>— ) ————— |
    | [ —<start index>— ] |
  
```

The WAITANDRESET statement and the WAIT statement with a wait parameter list are identical, except for the state to which the caused event is set during the cause process. If a program is waiting on an event because of a WAITANDRESET statement, then the happened state of the event is set to FALSE (not happened). If a program is waiting on an event because of the WAIT statement, then the happened state of the event is set to TRUE (happened).

Generally, an IMPORTED EVENT or IMPORTED EVENT ARRAY element cannot be used as an event designator in an event list. However, an IMPORTED EVENT that is declared in a CONNECTION BLOCK can be used as an event designator if the compiler control option WAITIMPORT is set.

WAITANDRESET Statement as an Arithmetic Function

The WAITANDRESET statement can be used as an arithmetic function that returns an integer value, starting at 1, that represents the position in the wait parameter list of the item that caused the program to be activated. For example, in the following statement the value of T is 1 if elapsed time caused the program to be activated:

```
T := WAITANDRESET(.001),E1,E2)
```

In the following statement the value of T is 2 if a cause operation on event E2 activated the program:

```
T := WAITANDRESET(E1,E2,E3)
```

This mechanism guarantees that one and only one parameter activates the program.

WAITANDRESET Statement

In the following statement, the value of T is 0 (zero) if a signal was received by the process before any of the events were caused:

```
T := WAITANDRESET [INTERRUPTIBLE] (E1,E2,E3)
```

The *WAITANDRESET* (<wait parameter list>) [<start index>] form is used to specify a round-robin or other sequence of testing. The arithmetic expression <start index> represents the position of the item in the <wait parameter list> that should be tested first. The index of the first item is one, the second is two, and so on. If the value of the <start index> is less than one or greater than the last item in the list, it is set to one. Note that if a (<time>) is specified in the wait parameter list, it is always tested first.

Note that the <direct array row> syntax is not allowed as a parameter to the *WAITANDRESET* statement.

Examples of WAITANDRESET Statements

In the following example, if the happened state of event EVNT is TRUE (happened), the program continues with the next statement. Otherwise, the program is suspended until EVNT is caused, and the happened state of EVNT is set to FALSE (not happened).

```
WAITANDRESET (EVNT)
```

In the following example, if the happened state of event EVNT1, EVNT2, or the event designated by EVNTARRAY[INDX] is TRUE (happened), the program continues with the next statement. Otherwise, the program is suspended until one of the three events is caused, and the happened state of that event is set to FALSE (not happened).

```
WAITANDRESET (EVNT1, EVNT2, EVNTARRAY [INDX])
```

The following example is identical to the previous one except that the first event tested is determined by the value of START_EVENT.

```
WAIT (EVNT1, EVNT2, EVNTARRAY [INDX]) [START_EVENT]
```

In the following example, if the happened state of event FINI or GOAWAY is TRUE (happened), the program continues with the next statement. Otherwise, the program is suspended until .5 second has elapsed or until event FINI or GOAWAY is caused. If FINI or GOAWAY is caused, its happened state is set to FALSE (not happened).

```
WAITANDRESET ((.5), FINI, GOAWAY)
```

In the following example, if the happened state of event WAKEUP or LOOKAROUND is TRUE (happened), the program continues with the next statement. Otherwise, the program is suspended until SLEEPMAX seconds have elapsed or until event WAKEUP or LOOKAROUND is caused. The value stored in REASON is 1, 2, or 3, indicating which of the three items reactivated the program. If WAKEUP or LOOKAROUND is caused, its happened state is set to FALSE (not happened).

```
REASON := WAITANDRESET ((SLEEPMAX), WAKEUP, LOOKAROUND)
```

In the following example, if the happened state of WAKEUP or GOAWAY is TRUE (happened), the program continues with the next statement. Otherwise, the program is suspended until NAPTIME seconds have elapsed or until either event WAKEUP or GOAWAY is caused, or the process receives a signal. If a signal is received before the WAKEUP or GOAWAY event is caused (happened), or before NAPTIME expires, the value stored in X is 0 (zero), indicating that the wait was interrupted by a signal.

```
X := WAITANDRESET [INTERRUPTIBLE] ((NAPTIME),WAKEUP,GOAWAY)
```

WHEN Statement

The WHEN statement suspends processing of a program for the specified number of seconds.

<when statement>

— WHEN — (—<time>—) —————|

Characteristics of the Time Option

Program processing is suspended for <time> seconds. The value of <time> need not be an integer. If <time> is a double-precision value, it is rounded to single-precision. If <time> is less than approximately 0.0000023, the program resumes execution immediately. If <time> is larger than this value, then the number of seconds that the program is suspended is the smaller of <time> and 164,925 seconds (approximately 45.8 hours).

Depending on the amount of multiprocessing being performed and the priorities of other programs in execution, the actual time that a program is suspended can vary widely with respect to <time> but is at least <time> seconds.

Examples of WHEN Statements

In the following example, the program is suspended for 10 seconds.

```
WHEN(10)
```

In the following example the program is suspended for $2 * Y + Z$ seconds.

```
WHEN(2*Y+Z)
```


WHILE Statement

The WHILE statement executes a statement as long as a specified condition is met.

<while statement>

— WHILE —<Boolean expression>— DO —<statement>—————|

Execution of the WHILE Statement

The iterative WHILE statement is executed as follows: the Boolean expression is evaluated and, if the result is TRUE, the statement following DO is executed. This sequence of events continues until the value of the Boolean expression is FALSE or until the statement following DO transfers control outside the WHILE statement. Figure 4-7 illustrates the WHILE-DO loop.

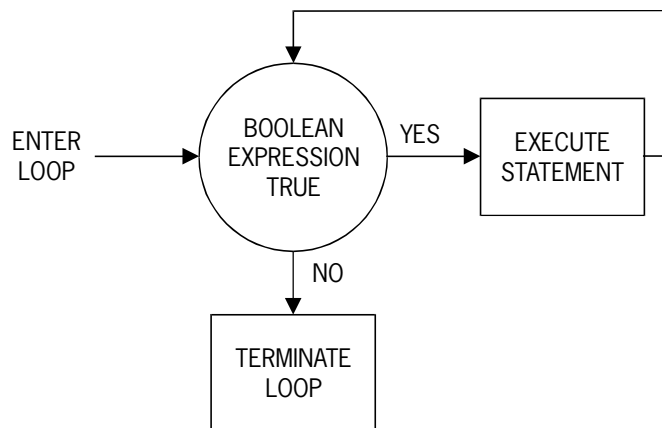


Figure 4-7. WHILE-DO Loop

Examples of WHILE Statements

In the following example, as long as INDX is less than or equal to MAXVAL, the value of X is incremented by the value A[INDX].

```
WHILE INDX LEQ MAXVAL DO  
  X := *+A[INDX];
```

In the following example, as long as J is less than LIMIT, the compound statement is executed. Contiguous elements of array SU are assigned the values of the elements of array SVALUES with the same indexes.

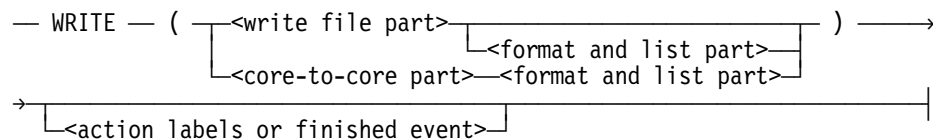
```
WHILE J LSS LIMIT DO  
  BEGIN  
    SU[J] := SVALUES[J];  
    J := *+1;  
  END;
```

WRITE Statement

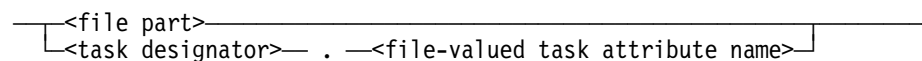
The WRITE statement causes data to be transferred from various program variables to a file.

Note: The syntax of the WRITE statement and the syntax of the READ statement are nearly identical. Differences in the syntax are discussed separately under each statement. See the "READ Statement" earlier in this section for a more detailed breakdown of those syntactic elements of the WRITE statement that are not discussed here.

<write statement>



<write file part>



The action of the WRITE statement depends on the form of the <write file part> element or <core-to-core part> element and on the form of the <format and list part> element.

The write file part or core-to-core part indicates where the data is to be written.

The WRITE statement can be used as a Boolean function. When the write operation fails, the value TRUE is returned. When the write operation succeeds, the value FALSE is returned. Specifically, the WRITE statement returns a value identical to that returned by the file attribute STATE. For more information, refer to the discussion of the STATE attribute in the *File Attributes Programming Reference Manual*.

For BNA Host Services, error results for WRITE statements are reported one WRITE statement after the WRITE statement that reuses the buffer that originally had the error. That is, the error is reported one buffer later than normal. Normally, error results are reported exactly at the WRITE statement that reuses the buffer having the error.

WRITE statements that do not contain format designators or editing specifications provide a faster output operation than those that specify that data is to be edited.

I/O Option or Carriage Control

If the <I/O option or carriage control> element is *[LINE <arithmetic expression>]* and the file is a printer file, then the printer spaces forward to the specified line before printing. The PAGESIZE file attribute of the file must be nonzero. Because the default action for ALGOL is to print before carriage action, a subsequent WRITE statement can overprint the line.

The *[CONTROL <arithmetic expression>]* construct is meaningful only for KIND=LBP direct files. The construct *[CONTROL 2]* is a WRITE modifier used to perform a Flush LBPI Buffers operation. Higher CONTROL values are WRITE modifiers used to perform Mode Change operations that set the type of data to be returned by the next Read Information operation, as follows:

Value	Meaning
3	Fault Character information
4	Count of output pages to a stacker
5	LBP Sense Bytes information

The *[SKIP <arithmetic expression>]* construct causes the printer to skip to the channel indicated by the value of the arithmetic expression after printing the current record. The LINENUM file attribute of the file is reinitialized to 1 when [SKIP 1] is used, but the PAGE file attribute is not incremented.

The *[SPACE <arithmetic expression>]* construct causes the printer to space the number of lines specified by the arithmetic expression after printing the current record, or before printing the current record if the WRITEAFTER compiler control option is set. For nonbinary writes, the default of single-spacing is equivalent to [SPACE 1]. Overprinting can be achieved with [SPACE 0]. For binary writes, a carriage control is done before the SPACE carriage control option is evaluated. The default of single-spacing is equivalent to [SPACE 0], and overprinting is not possible. On other types of devices, this construct causes the number of records specified by the value of the arithmetic expression to be spaced.

For disk files the *[SPACE <arithmetic expression>]* construct works in the following manner. A WRITE statement that is not a binary write spaces before writing. The disk file might be extended as a result. A binary write spaces after writing unless the WRITEAFTER compiler control option is used to indicate that spacing should take place before writing. The file cannot be extended. The effect of this construct is not necessarily the same as a WRITE statement following or preceding a SPACE operation.

If the specified file is a remote file, the *[STOP]* construct causes the normal line feed and carriage return action to be omitted.

The *[STACKER <arithmetic expression>]* value is ignored by the operating system at run time.

The *[SYNCHRONIZE]* construct enforces synchronization between the logical and physical file. Synchronization means that output must be written to physical file before the program initiating the output can resume execution. The SYNCHRONIZE file attribute designates the synchronization for all output records, and the *[SYNCHRONIZE]* construct overrides the specified synchronization for the record being written. The *[SYNCHRONIZE]* construct is available for use by disk files with FILEORGANIZATION=NOTRESTRICTED or for tape files, and is not available for use with port files.

The *[TIMELIMIT <arithmetic expression>]* construct is meaningful only for remote files. The write operation is terminated with a TIMELIMIT error if the buffer is not available within the number of seconds specified by the value of the arithmetic expression.

The *[STATION <arithmetic expression>]* construct is meaningful only for remote files. It assigns the value of the arithmetic expression to the LASTSUBFILE file attribute of the file.

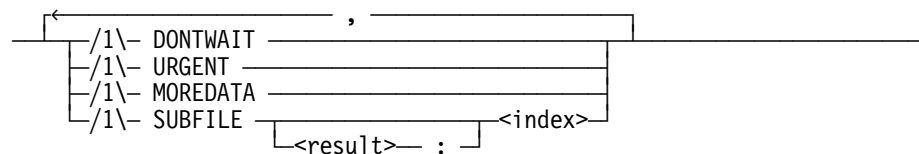
Imported events and event arrays cannot be used as <event designator> s in a direct I/O WRITE statement.

For more information on the FILEORGANIZATION, LASTSUBFILE, LINENUM, PAGESIZE, or SYNCHRONIZE attributes, refer to the *File Attributes Programming Reference Manual*.

On output to the printer, the *[NO]* construct causes the line feed after a write to be suppressed.

Write Subfile Specification

<write subfile specification>



<file-valued task attribute name>



If the file to be written is a port file (a file for which the KIND attribute is equal to PORT), any form of write, other than a binary write, can be used. Refer to "Binary Write" later in this section.

The subfile specification is meaningful only for port files. It is used to specify the subfile to be used for the write operation and the type of write operation to be performed.

WRITE Statement

If a subfile index is used, the value of the subfile index is assigned to the LASTSUBFILE attribute of the file. It specifies the subfile to be used for the write operation. For a WRITE statement, if the subfile index is zero, a broadcast write is performed. If the subfile index is nonzero, then a write to the specified subfile is performed. The result variable, if specified, is assigned the resultant value of the LASTSUBFILE attribute.

For more information on the LASTSUBFILE file attribute, refer to the *File Attributes Programming Reference Manual*.

If DONTWAIT is specified and no buffer is available, the program is not suspended.

The URGENT clause is meaningful only when the Transmission Control Protocol/Internet Protocol (TCP/IP) is being used. This clause sets the urgent indication associated with the data. For more information on TCP/IP, refer to the *Distributed Systems Service (DSS) Operations Guide*.

If the URGENT clause is used for port subfiles that have the SERVICE attribute set to BNANATIVESERVICE, then the BADWRITEOPTION (43) write error is returned in field [26:10] of the write result and the STATE attribute. The error bit [0:1] is set also in the write result and the STATE attribute.

The MOREDATA specification is valid only for port files. The MOREDATA specification is used to indicate that only part of a message is being written by the WRITE statement. There is no restriction on the size of the partial message that can be passed in a buffer except that the maximum data length cannot be greater than 64K characters.

<core-to-core part>
<core-to-core file part>
<core-to-core blocking part>

Refer to "READ Statement" earlier in this section for a discussion of these constructs.

Format and List Part

For the syntax of <format and list part>, see "READ Statement" earlier in this section. The format and list part element indicates which variables contain the data and how the data is to be interpreted.

If the format and list part element is omitted in a WRITE statement, a logically empty record is written. The actual output is device-dependent. Printers interpret this as a blank record; disks and tapes interpret this as a record with undefined contents.

Formatted Write

A WRITE statement that contains a format designator, editing specifications, or a free-field part is called a formatted write.

A format designator without a list indicates that the referenced format contains one or more string literals that constitute the entire output of the WRITE statement.

A format designator with a list indicates that the variables in the list are to be written in the format described by the referenced format.

Editing specifications can appear in place of a format designator and have the same effect as if they had been declared in a FORMAT declaration and had been referenced through a format designator. For more information, refer to "FORMAT Declaration" in Section 3, "Declarations."

Binary Write

A WRITE statement of the following form is called a binary write:

```
WRITE(<write file part>,*,<list>)
```

A binary WRITE statement can be used to write variables to a file in an internal form that can later be read with a binary READ statement.

An asterisk (*) followed by a list specifies that the elements in the list are to be processed as full words and are to be written without being edited. The number of words written is determined by the number of elements in the list or the maximum record size, whichever is smaller. When unblocked records are used, the block size is the maximum record size.

In a binary write, when the record number or carriage control element is LINE, SKIP, STATION, TIMELIMIT, or STOP, it is ignored and treated as a serial binary write. If the carriage control element is SPACE, the number of lines specified plus one is spaced after writing the current record. The extra space occurs because in a binary write normal carriage control is performed first and then any extra spacing is done.

When writing a character array, only full words are written. If there is a partial word left at the end of the array, it is ignored. For example, if A is an EBCDIC array that contains the characters 12345678, the following statement writes only the characters 123456:

```
WRITE(FILEID,*,A)
```

When a string variable occurs in the list of a binary WRITE statement, a word containing the string length is written to the file before the contents of the string are written. This feature allows the program to write string information that can later be read through a binary READ statement. For more information, see "Binary Read" under "READ Statement" earlier in this section.

Array Row Write

A WRITE statement of any of the following forms is called an array row write:

```
WRITE(<write file part>,<arithmetic expression>,<array row>)
WRITE(<write file part>,<arithmetic expression>,<subscripted variable>)
WRITE(<write file part>,<arithmetic expression>,<pointer expression>)
WRITE(<write file part>,<arithmetic expression>,<string variable>)
```

The first three forms of the array row write specify that the elements of the designated array row, subscripted variable, or item referenced by the pointer expression are to be processed as full words and are to be written without being edited. The number of words written is determined by the smallest of the following:

- The number of elements in the array row, subscripted variable, or item referenced by the pointer expression
- The maximum record length
- The greater of the absolute value of the arithmetic expression or one (1)

If the FILETYPE attribute of the file has a value of 6, then the maximum record length is ignored and records span block boundaries. When unblocked records are used, the block size is the maximum record size. If the UNITS attribute equals CHARACTERS and the INTMODE attribute does not equal SINGLE, then all counts represent characters, not words.

A WRITE statement of the following form specifies that the characters in the string variable are to be written without being edited:

```
WRITE(<write file part>,<arithmetic expression>,<string variable>)
```

The number of characters written is determined by the maximum record size, the absolute value of the arithmetic expression, or the length of the string, whichever is smallest. If the UNITS attribute equals CHARACTERS and the INTMODE attribute does not equal SINGLE, then all counts represent characters, not words.

A WRITE statement of the following form is not an array row write:

```
WRITE (<write file part>,<arithmetic expression>,<string literal>)
```

This kind of statement can have unexpected results because the number of words or characters written is determined by the arithmetic expression and the UNITS attribute of the file part. If the arithmetic expression in the units of the file is greater than the length of the string literal, unexpected output can occur. Use a formatted write of the following form to write a string literal:

```
WRITE(<file part>,<" <string literal> ">)
```


Free-Field Part

The free-field part allows output to be performed with editing but without using editing specifications. The appropriate format is selected automatically, but variations of the free-field part give the programmer some control over the form of the output.

On output, each value is edited into an appropriate format. An edited item is never split across a record boundary. If the record is too short to hold the representation of the item, a string of pound signs (#) is written in place of the item.

When a complex expression appears in the list of a free-field WRITE statement, two values are written. The first value corresponds to the real part, and the second value corresponds to the imaginary part.

Data items are normally separated by a comma and a space (,). If the free-field part contains two slashes, data items are separated by two spaces.

If the optional asterisk (*) is used, the name of the data item and an equal sign (=) are written to the left of the value of the data item. If the data item is not a variable, then the expression is written as the name of the data item.

If the free-field part includes the <number of columns> and <column width> elements, each list element is written in a separate column. This process is controlled by two column factors: the number of columns per record (r) and the width of each column (w), where w is measured in characters. Both r and w are integerized, if necessary.

If r is 0 (zero), the number of columns per record is determined from the value of w and the record length. If w is 0, the width of each column is determined from the value of r and the record length. If both r and w are 0, the output has no column structure. If r and w are such that r columns of w characters cannot fit on one record, adjustments are made to both r and w . The width of a column does not include the two-character delimiter; therefore, $r*(w+2)$ must be less than or equal to the length of the record.

Example of a Free-Field Part

```
BEGIN
  FILE DCOM(KIND=REMOTE,MAXRECSIZE=12,MYUSE=I0);
  INTEGER I;
  REAL R;
  DOUBLE D;
  STRING S;
  I := 25;
  R := 1002459;
  D := 25@@5;
  S := "string";
  WRITE(DCOM,*/, I, R, D, S, I+R, R-D, S||" ABCDE", 7.2);
END.
```

WRITE Statement

The following output results when this program is executed:

```
I=25, R=1002459.0, D=2.5D+6, S=string, I+R=1002484.0, R-D=-1497541.0,  
S||" ABCDE"=string ABCDE, <CNST>=7.2,
```

Additional information about I/O operations can be found under "I/O Statement" and "READ Statement" earlier in this section.

Action Labels or Finished Event

This construct provides a means of transferring program control from a READ statement, WRITE statement, or SPACE statement when exception conditions occur (for normal I/O) or when the I/O is complete (for direct I/O). Exception conditions can also be handled by using the WRITE statement as a Boolean function. For more information, refer to "READ Statement" earlier in this section.

Examples of WRITE Statements

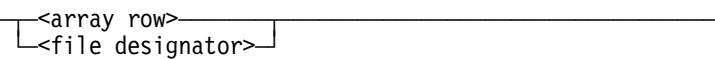
```
WRITE(FILEID)  
  
WRITE(SPOFILE,FMT,LISTID)  
  
WRITE(FILEID[NO],FMT)  
  
WRITE(SPOFILE,10,ARRY[3,*])  
  
WRITE(SWFILEID[0],X+Y-Z,ARRY[X,I,*])  
  
WRITE(SPOFILE,/,LISTID)  
  
WRITE(FILEID,FMT,LISTID)  
  
WRITE(SWFILEID[3][PAGE])  
  
WRITE(FILEID,/,A,B,C)  
  
WRITE(FILEID,SWFMT[A*I])  
  
WRITE(FILEID,*,LISTID)  
  
WRITE(FILEID[5+I],/,SWLISTID[4])  
  
WRITE(FILEID,/,LISTID)  
  
WRITE(FILEID,*,A,B,C)  
  
WRITE(FILEID,FMT,A,B,C,D+SIN(X)) [:PARL]  
  
WRITE(FILEID,FMT,LISTID) [:PARSWL[M]]
```

```
WRITE(SWFILEID[1],SWFMT[2],SWLISTID[3]) [:PARSWL[4]]  
WRITE(DIRFYLE,30,DIRARAY) [EVNT]  
WRITE(MYSELF.TASKFILE,<"ABOVE DUMP BEFORE TRANSACTION">)  
WRITE(OUT,10,S1 || S2)  
WRITE(OUT,<2A10>,TAKE(S1,2),S1 || "ABC")  
WRITE(OUT[5,SYNCHRONIZE],30,ARRAYNAME)
```

ZIP Statement

The ZIP statement causes the Work Flow Language (WFL) compiler to begin compiling the designated source code.

<zip statement>

— ZIP — WITH — 

The ZIP statement passes to the WFL compiler the source code in the array row or in the file referenced by the file designator. The source code in the array row or file must be valid WFL source input; otherwise, it is not executed. WFL syntax requirements are described in the *Work Flow Language (WFL) Programming Reference Manual*.

The EXCEPTIONEVENT is caused for a user program when the program executes a ZIP statement.

ZIP WITH <array row>

The array row can be an EBCDIC array row or a row of a word array. If the array row is a word array, the character type of the contents of the array row is the default character type. For more information, refer to “Default Character Type” in Appendix C, “Data Representation.” The array row is processed as one record, but it can include more than 72 characters. A semicolon (;) is used to separate statements within the array row. Only one question mark character can appear in the array row.

WFL examines the contents of the array row for correct syntax, and if errors occur it reports this fact to the Operator Display Terminal (ODT). If no errors are detected, the compiled job is run. In either case, program control passes to the next statement in the program.

Note: For a ZIP WITH <array row> involving Job Transfer (“AT <host name> BEGIN JOB...”), the maximum amount of text allowed is 65,500 bytes.

ZIP WITH <file designator>

When this form of the ZIP statement is executed, the file referenced by the file designator is passed to the WFL compiler. The file is compiled in the same manner as any other WFL source file. If the source compiles without syntax errors, it is executed, and control passes to the statement following the ZIP statement. If syntax errors occur when the source is compiled, the WFL job is not executed, and control passes to the statement following the ZIP statement.

On execution of a ZIP statement, control of the file referenced by the file designator is passed to the operating system.

It is recommended that the file used in the ZIP statement be declared with the file attributes (BLOCKSIZE=420, MAXRECSIZE=15) or (DEPENDENTSPECS = TRUE).

Examples of ZIP Statements

In the following example, the WFL source input in array ARAY is compiled and executed.

```
ZIP WITH ARAY
```

In the following example, the WFL source input in file FYLE is compiled and executed.

```
ZIP WITH FYLE
```

ZIP Statement

Section 5

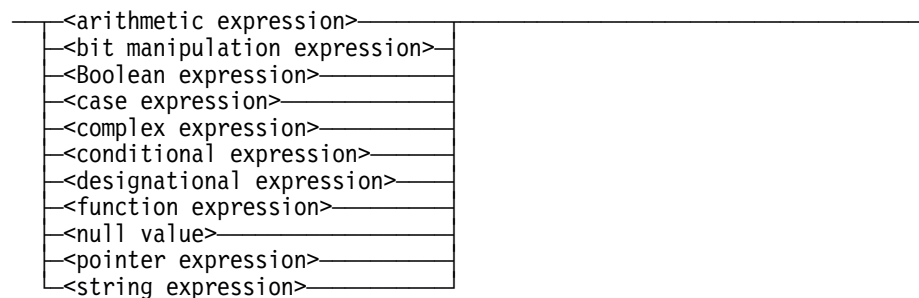
Expressions and Functions

This section contains two main parts: one for expressions and one for intrinsic functions. In each part, the expressions or functions are presented in alphabetical order.

Expressions

An expression describes how a value can be obtained by applying specified operations to designated operands or primaries.

<expression>



The evaluation of each expression returns a different value, as follows:

Expression	Value
Arithmetic	Numerical
Boolean	Boolean (TRUE or FALSE)
Complex	Complex: a real numerical part and an imaginary numerical part
Designational	Label
Pointer	A value that can be used to reference a character position in an array row
String	A value that is an EBCDIC, ASCII, or hexadecimal string

Expressions and Functions

A bit manipulation expression operates on bits within words and makes it possible to build the contents of a word from groups of bits contained in other words.

Case expressions and conditional expressions allow one of several alternative expressions to be chosen for evaluation based on a selection value.

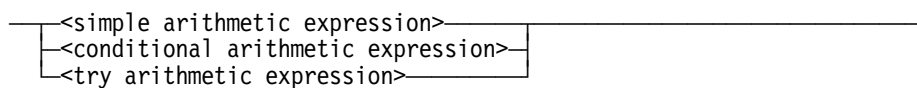
A function expression is a call on a typed procedure. The procedure can be declared in the program or it can be an intrinsic which is a typed procedure that is a predefined part of the ALGOL language. Intrinsic functions exist that are predefined arithmetic expressions, predefined Boolean expressions, predefined complex expressions, predefined pointer expressions, and predefined string expressions. For example, *SQRT*(*<arithmetic expression>*) is a function expression that returns the square root of the value of *<arithmetic expression>* construct. Because the *SQRT* returns a numeric value, it is an arithmetic function, a predefined arithmetic expression.

Note: Expressions that are very large or deeply nested can cause the compiler to receive a stack overflow fault. The fault can be avoided by breaking very large or deeply nested expressions into several separate expressions or by increasing the maximum stack size by using the task attribute *STACKLIMIT*.

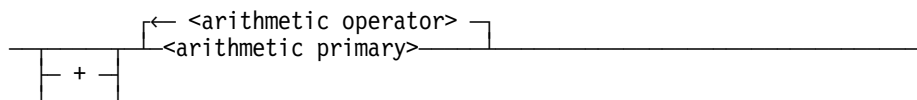
Arithmetic Expression

Arithmetic expressions perform specified operations on designated arithmetic primaries to return numerical values.

<arithmetic expression>

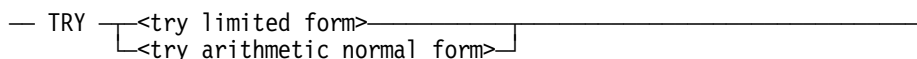


<simple arithmetic expression>

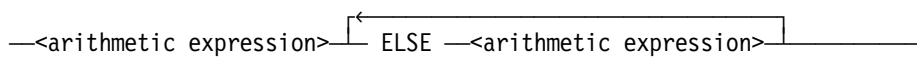


The evaluation of a conditional arithmetic expression is described in "Conditional Expression" later in this section.

<try arithmetic expression>



<try arithmetic normal form>



An explanation of <try arithmetic expression> is given in "TRY Expression" later in this section.

Precision of Arithmetic Expressions

The value of an arithmetic expression can be expressed in single or double-precision, depending on the precision of its constituents or, in the case of MUX, on the operator involved. The value of an arithmetic expression is double-precision if any variable, function, or number of which it is composed is of type DOUBLE, or if two primaries are combined by the double-precision operator MUX. The MUX operator allows a double-precision result to be obtained from the multiplication of two single-precision arithmetic primaries.

The value of a case expression is double-precision if any expression in its arithmetic expression list is of type DOUBLE. Likewise, the value of a conditional arithmetic expression is double-precision if either arithmetic expression is double-precision. In either case, single-precision arithmetic expressions are converted to double-precision, when necessary.

Arithmetic Operators

<arithmetic operator>

+	_____
-	_____
*	_____
TIMES	_____
MUX	_____
/	_____
DIV	_____
MOD	_____
**	_____

The operators +, -, *, and / have the conventional mathematical meaning of addition, subtraction, multiplication, and division, respectively. No two operators can be adjacent, and implied multiplication is not allowed.

The TIMES operator also denotes multiplication.

The DIV operator denotes integer division. It has the following mathematical meaning:

$$Y \text{ DIV } Z = \text{SIGN}(Y/Z) * \text{ENTIER}(\text{ABS}(Y/Z))$$

The MOD operator denotes remainder division. If Z is greater than or equal to 1, MOD has the following meaning:

$$Y \text{ MOD } Z = Y - (Z * (Y \text{ DIV } Z))$$

If Z is less than 1, the MOD operator produces undefined results.

The MUX operator multiplies either single-precision or double-precision arithmetic primaries, and yields a double-precision result.

The ** operator denotes exponentiation. The semantics of the exponentiation operator depend on the types and values of the primaries involved. Figure 5–1 explains the various meanings of $Y^{**}Z$.

	Integer			Real		
	Z > 0	Z = 0	Z < 0	Z > 0	Z = 0	Z < 0
Y > 0	Note 1	1	Note 2	Note 3	1	Note 3
Y < 0	Note 1	1	Note 2	Note 4	1	Note 4
Y = 0	0	Note 4	Note 4	0	Note 4	Note 4

Legend

Note 1: $Y^{**}Z = Y * Y * Y \dots * Y$ (Z times)

Note 2: $Y^{**}Z = \text{Reciprocal of } Y * Y * Y \dots * Y$ (ABS(Z) times)

Note 3: $Y^{**}Z = \text{EXP}(Z * \text{LN}(Y))$

Note 4: Value of the expression is undefined.

Figure 5–1. Exponentiation: Meaning of $Y^{}Z$**

Precedence of Arithmetic Operators

The sequence in which the operations of an arithmetic expression are performed is determined by the precedence of the operators involved. The order of precedence is as follows:

1. ** (highest precedence)
2. *, /, MOD, DIV, MUX, TIMES
3. +, –

Operators with the same precedence are applied in their order of appearance in an expression, from left to right.

The precedence of the assignment operator (:=) is as follows:

1. An expression to the right of an assignment operator is evaluated before the assignment.
2. The assignment is done before the evaluation of an expression involving the variable that is the target of the assignment.

Parentheses can be used in normal mathematical fashion to override the defined order of precedence. An expression in parentheses is evaluated by itself, and the resulting value is subsequently combined with the other elements of the expression. In the following expression, for example, the addition is performed before the division because of the parentheses:

$$(X+1)/Y$$

In the following expression, 1 is first divided by Y and then the result is added to X:

$$X + 1/Y$$

For information on the order of evaluation of subscripts, see "Arithmetic Update Assignment" in Section 4, "Statements."

Figure 5-2 illustrates how mathematical notation can be translated to an ALGOL arithmetic expression.

Mathematical Expression	Equivalent ALGOL Expression
$A \times B$	$A * B$
$A + \frac{B}{2}$	$A + B / 2$
$\frac{X + 1}{Y}$	$(X + 1) / Y$
$\frac{D + E^2}{2A}$	$(D + E**2) / (2 * A)$
$4(X + Y)^3$	$4 * (X + Y) ** 3$
$\frac{M - N}{(M + N) \times 10^6 (P + 5@-6)}$	$(M - N) / (M + N) ** (P + 5@-6)$

Figure 5-2. Mathematical Notation

Types of Resulting Values

The type of the value resulting from an arithmetic operation depends on the arithmetic operator and the types of the primaries being combined, except when the resulting value is undefined. Figure 5–3 describes the types of quantities that result from various combinations of arithmetic primaries.

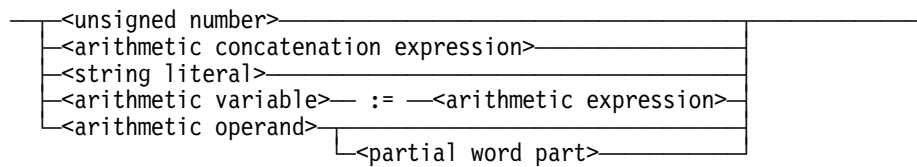
Operand on Left	Operand on Left	+ - *	/	DIV	MOD	**	MUX
INTEGER	INTEGER	Note 3	REAL	INTEGER	INTEGER	Note 1	DOUBLE
INTEGER	REAL	REAL	REAL	INTEGER	REAL	Note 2	DOUBLE
INTEGER	DOUBLE	DOUBLE	DOUBLE	DOUBLE	DOUBLE	Note 2	DOUBLE
REAL	INTEGER	REAL	REAL	INTEGER	REAL	Note 2	DOUBLE
REAL	REAL	REAL	REAL	INTEGER	REAL	Note 2	DOUBLE
REAL	DOUBLE	DOUBLE	DOUBLE	DOUBLE	DOUBLE	DOUBLE	DOUBLE
DOUBLE	(any)	DOUBLE	DOUBLE	DOUBLE	DOUBLE	DOUBLE	DOUBLE
Note 1: If the operand on the right is negative or the absolute value of the result is greater than or equal to 2**39, REAL; otherwise, INTEGER Note 2: If the operand on the right is zero, INTEGER; otherwise, REAL Note 3: If the absolute value of the result is less than 2**39, INTEGER; otherwise, REAL							

Figure 5–3. Types of Values Resulting from Arithmetic Operations

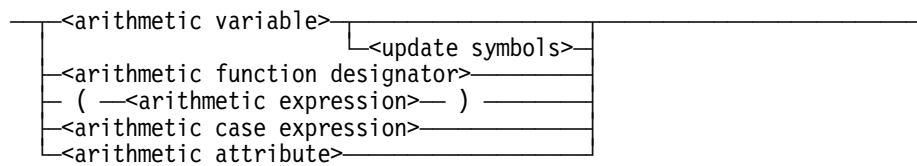
The type of an arithmetic case expression or a conditional arithmetic expression is DOUBLE if any of its constituent expressions are of type DOUBLE. For more information, refer to “Precision of Arithmetic Expressions” earlier in this section. If the conditional arithmetic expression or arithmetic case expression contains only expressions of type INTEGER and REAL, its type is REAL. A conditional arithmetic expression or arithmetic case expression is of type INTEGER only if all its constituent expressions are of type INTEGER.

Arithmetic Primaries

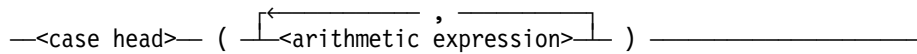
<arithmetic primary>



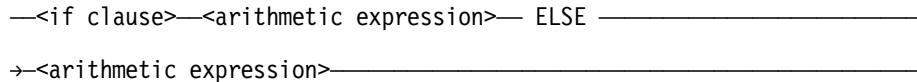
<arithmetic operand>



<arithmetic case expression>



<conditional arithmetic expression>



<constant arithmetic expression>

An arithmetic expression that can be fully evaluated at compilation time.

The items on which arithmetic operators act are called arithmetic primaries.

A variable or function designator used as an arithmetic primary in an arithmetic expression must be of an arithmetic type: INTEGER, REAL, or DOUBLE.

An attribute used as an arithmetic primary must have a type that is INTEGER, REAL, or DOUBLE. For information on file attribute and direct array attribute types, see the *File Attributes Programming Reference Manual*. The arithmetic-valued task attributes are described in "Arithmetic Assignment" in Section 4, "Statements."

The length of a string literal used as an arithmetic primary must not exceed 48 bits. A string literal used as an arithmetic primary is interpreted as either type REAL or type INTEGER, depending on its value.

The arithmetic concatenation expression is described in "Concatenation Expression" later in this section.

The partial word part is described in "Partial Word Expression" later in this section.

The evaluation of an arithmetic case expression is described in “Case Expression” later in this section.

Examples of Arithmetic Primaries

Valid

5.678
X:= * + 3
(14 + 3.142)
MABEL
R & 3 [1:2]
Y.[30:4]
"ABCD"
SQRT(X)
CASE I OF (5,15,17)
FYLE.MAXRECSIZE

Invalid

X:= *:= Y
+ DC8
B -A
-(A + B)
TRUE

Bit Manipulation Expression

Bit manipulation expressions provide a means of isolating a field of one or more bits from a word, and these expressions allow words to be constructed from fields of one or more bits from other words.

<bit manipulation expression>

—<concatenation expression>—
└─<partial word expression>┘

Concatenation Expression

The concatenation expression forms a primary from selected parts of two or more primaries.

<concatenation expression>

—<arithmetic concatenation expression>—
└─<Boolean concatenation expression>┘

<arithmetic concatenation expression>

—<arithmetic primary>— & —<arithmetic expression>—<concatenation>—

<Boolean concatenation expression>

—<Boolean primary>— & —<Boolean expression>—<concatenation>—

A concatenation expression is formed by taking a specified part of the bit pattern of an expression value and copying it into the specified portion of a primary. The rest of the destination primary is not changed by this operation.

Note that only arithmetic expressions can be concatenated with arithmetic primaries, and only Boolean expressions can be concatenated with Boolean primaries.

Because the concatenation expression is a primary and the syntax for a concatenation expression is of the form *<primary> & <expression> <concatenation>*, concatenation expressions of the following form are allowed:

```

<primary> & <expression> <concatenation>
      & <expression> <concatenation>
      .
      .
      .
      & <expression> <concatenation>
    
```

If, as in the preceding example more than one concatenation term is used in a concatenation expression, then these terms are evaluated from left to right.

Concatenation

<concatenation>

— [—<left bit to>— : —————> <number of bits>—>
 └─<left bit from>— : ┘
 →] —————|

<left bit to>

—<arithmetic expression>—————|

<left bit from>

—<arithmetic expression>—————|

<number of bits>

—<arithmetic expression>—————|

The concatenation construct describes the location in the expression of the field to be copied and the location in the destination primary where the field is to be copied.

The <left bit to> element defines the leftmost bit location of the field in the destination word. The <left bit from> element defines the leftmost bit location of the field in the source word. The <number of bits> element specifies the length of the field to be copied from the source to the destination.

If the [*<left bit to>:<left bit from>:<number of bits>*] form is used, the field of bits to be copied starts at <left bit from> in the source word and is <number of bits> long.

If the [*left bit to*:*number of bits*] form is used, then the field of bits to be copied from the source word starts at bit number (*number of bits*–1) and extends through bit 0. That is, the source field is assumed to be the low-order *number of bits* bits in the source word.

The values of the *left bit to* and *left bit from* elements must lie within the range 0 through 47, where bit 0 is the rightmost, or least significant, bit in the word.

The value of the number of bits must lie within the range 1 through 48. If the value of the number of bits exceeds the number of bits to the right of the starting bit in either the source or destination words, these fields wrap around and are continued at bit 47, the leftmost bit, of the same word.

If, through the use of variables, the ranges for the *left bit to*, *left bit from*, or *number of bits* elements are exceeded, then the program is discontinued with a fault. If 0 is used as the number of bits, no bits are copied.

Because a concatenation expression is a primary, when it appears as an operand in a larger expression, the concatenation expression is evaluated before any other operation is executed. The following is an example of a concatenation expression evaluation:

Expression	Evaluated as
$2^{**}4 \& 1 [0:0:1]$	$2^{**}(4 \& 1 [0:0:1]) = 2^{**}5 = 32$

The expression is not evaluated as the following:

$$(2^{**}4) \& 1 [0:0:1] = 16 \& 1 [0:0:1] = 17$$

Examples of Concatenation Expressions

Assume that the real variables X, Y, and Z have the following values:

```
X = 32767 = 4"000000007FFF"
Y = 1024 = 4"00000000400"
Z = 1 = 4"000000000001"
```

Given these values, the following are examples of arithmetic concatenation expressions and their values:

Expression	Value
X & Y [47:11:4]	4"400000007FFF"
X & Y [47:12]	4"400000007FFF"
Y & X [39:20]	4"0007FFF00400"
Y & Z [46:1]	4"400000000400"
0 & Y [11:11]	4"000000000800"
0 & X [23:48]	4"007FFF000000"
X & Y [39:12] & Z [47:1]	4"804000007FFF"
Y & X [19:15:8]	4"00000007F400"

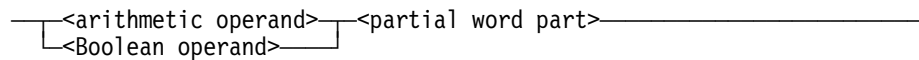
Assume that the elements of the real array INFO contain information about the data in a file. Each element of INFO contains a record number in the field [19:20], and the length of the data in that record is in field [39:20]. That is, each element of INFO stores the location and length of a record in the data file. Let N be a variable that contains a record number, and let L be a variable that contains the length of that record. The following declarations and assignment statement can be used to store a value into an element of INFO.

```
DEFINE
  REC_NUMF = [19:20]#,
  LENGTHF = [39:20]#;
  .
  .
  .
  INFO[I] := 0 & N REC_NUMF
            & L LENGTHF;
```

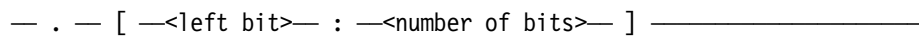
Partial Word Expression

A partial word expression isolates the value of a field of one or more bits of a specified word.

<partial word expression>



<partial word part>



<left bit>



The <partial word part> construct describes the location in the operand of the field to be isolated. The isolated field is copied to the low order (rightmost) field of a word of all zeros.

The <left bit> element defines the leftmost bit location of the field in the source word. The value of the left bit must lie within the range 0 through 47, where bit 0 is the rightmost, or least significant, bit in the word.

The <number of bits> element specifies the length of the field. The value of the number of bits must lie within the range 1 through 48. If the value of the number of bits exceeds the number of bits to the right of the left bit in the source word, the field wraps around and is continued at bit 47, the leftmost bit, of the same word.

If, through the use of variables, these ranges are exceeded, the program is discontinued with a fault. If 0 is used for the number of bits, no bits are copied.

Examples of Partial Word Expressions

Assume that real variables X, Y, and Z have the following values:

```
X = 32767 = 4"000000007FFF"  
Y = 1024 = 4"000000000400"  
Z = 2 = 4"000000000002"
```

Given these values, the following are examples of arithmetic partial word expressions and their values:

Expression	Value
X.[5:6]	4"00000000003F"
Y.[11:4]	4"000000000004"
Z.[19:48]	4"000020000000"
X.[23:24]	4"000000007FFF"
X.[23:20]	4"0000000007FF"

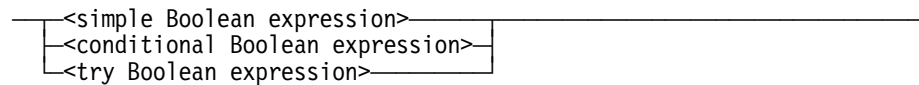
Using the INFO array example from "Concatenation Expression" earlier in this section, the following assignments could be used to extract information from INFO.

```
N := INFO[I].REC_NUMF;
L := INFO[I].LENGTHF;
```

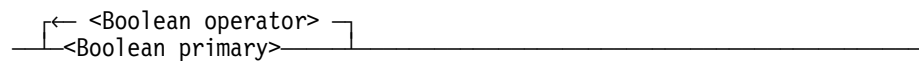
Boolean Expression

Boolean expressions are expressions that return logical values by applying specified operations to designated Boolean primaries.

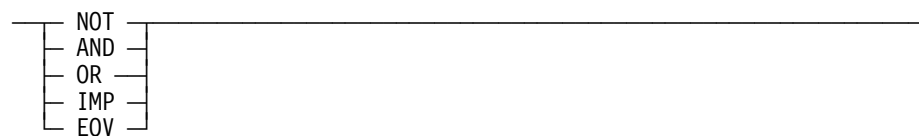
<Boolean expression>



<simple Boolean expression>

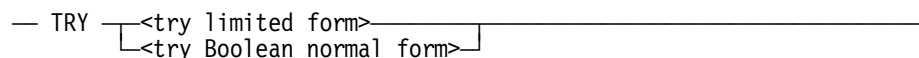


<Boolean operator>

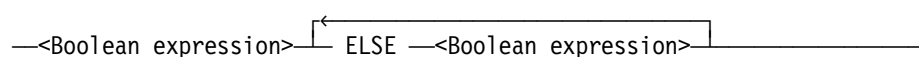


The evaluation of a conditional Boolean expression is described in "Conditional Expression" later in this section.

<try Boolean expression>



<try Boolean normal form>



An explanation of <try Boolean expression> is given in “TRY Expression” later in this section.

Operators in Boolean Expressions

The following table lists the operators that can be used in Boolean expressions, along with their meanings. When two operators are listed on the same line, they are equivalent to each other.

Operator		Meaning
NOT	^	Logical NOT
AND		Logical AND
OR	!	Logical inclusive OR
IMP		Logical implication
EQV		Logical equivalence
IS		Identical to
ISNT		Not identical to
EQL	=	Equal to
NEQ	^=	Not equal to
GTR	>	Greater than
GEQ	>=	Greater than or equal to
LSS	<	Less than
LEQ	<=	Less than or equal to

Logical Operators

The values returned by the logical operators are defined in Figure 5–4.

Operand A	Operand B	NOT A	A AND B	A OR B	A IMP B	A EQV B
TRUE	TRUE	FALSE	TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE
FALSE	TRUE	TRUE	FALSE	TRUE	TRUE	FALSE
FALSE	FALSE	TRUE	FALSE	FALSE	TRUE	TRUE

Figure 5–4. Results of Logical Operators

The Boolean operations defined in this table are performed on all 48 bits of the Boolean primaries on a bit-by-bit basis. For example, the constant TRUE (4"000000000001") does not have the same bit pattern as the Boolean expression NOT FALSE, because NOT complements all 48 bits of the constant FALSE (4"000000000000"), generating 4"FFFFFFFF". Even so, the constant TRUE and the Boolean expression NOT FALSE have the same Boolean value of TRUE because the Boolean value of a Boolean primary is based upon the value of the low-order bit (bit 0) of the Boolean primary (0 is FALSE; 1 is TRUE).

Note: Exception: When NOT operates on an arithmetic relation, the low-order bit (bit zero) is complemented; however, the other 47 bits are not necessarily complemented. For example, if $X = Y$ evaluates to TRUE, then $NOT(X = Y)$ evaluates to FALSE, not necessarily to NOT TRUE as would be expected.

IS and ISNT Operators

The IS relational operator performs a bit-for-bit comparison on its two operands. The operator returns the value TRUE if the corresponding bits of each operand are the same. The ISNT operator is the negation of the IS operator.

If the IS operator is used to compare a double-precision arithmetic quantity to a single-precision (integer or real) arithmetic quantity, the result is always FALSE.

The IS operator differs from the EQL or = operator, which does an arithmetic comparison of its operands. Two operands can have the same arithmetic value with different bit patterns. Thus, the following pairs yield the value TRUE when compared with the EQL or = operator, but yield the value FALSE when compared with the IS operator:

- +0 and –0
- A normalized number and the same number not in normalized format
- A number with an exponent of +0 and the same number with an exponent of –0
- A number with bit 47 = 0 and the same number with bit 47 = 1

Relational Operators

The action of the relational operators GTR, >, GEQ, >=, LSS, <, LEQ, <=, EQL, =, NEQ, and ^= depends on the kinds of items being compared. For more information, refer to "Arithmetic Relation," "Complex Relation," "String Relation," "Pointer Relation," and "String Expression Relation" later in this section.

Precedence in Boolean Expressions

The components of Boolean expressions are evaluated in the following order:

1. All arithmetic, complex, pointer, and string expressions are evaluated.
2. All relations, table memberships, and assignments are evaluated.
3. Logical operators are then applied.

The order of precedence of the logical operators is as follows:

1. NOT (highest precedence)
2. AND
3. OR
4. IMP
5. EQV

Operators with the same precedence are applied in their order of appearance in an expression, from left to right.

The precedence of the assignment operator (:=) is as follows:

1. A primary to the right of an assignment operator is evaluated before the assignment.
2. The assignment is done before the evaluation of an expression involving the variable that is the target of the assignment.

Parentheses can be used in normal mathematical fashion to override the defined order of precedence. An expression in parentheses is evaluated by itself, and the resulting value is subsequently combined with the other elements of the expression. In the following expression, for example, the OR is performed before the AND operator because of the parentheses:

X AND (Y OR Z)

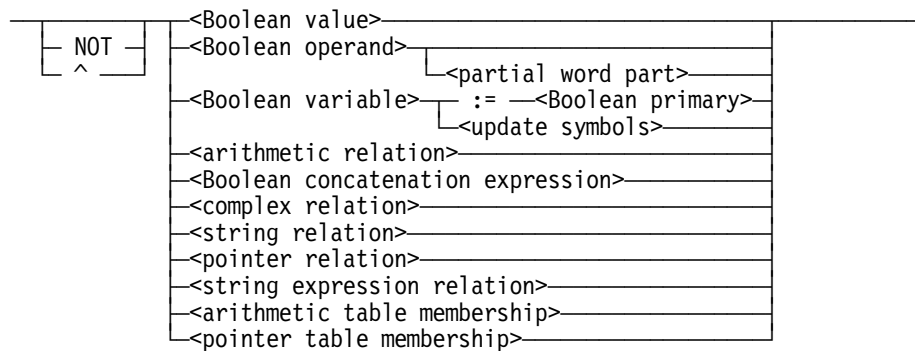
In the following expression the AND is performed before the OR:

X AND Y OR Z

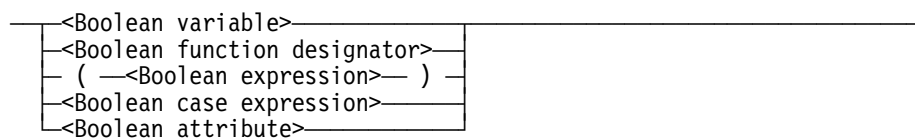
For information on the order of evaluation of subscripts, see "Arithmetic Update Assignment" in Section 4, "Statements."

Boolean Primaries

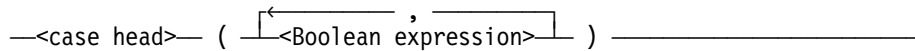
<Boolean primary>



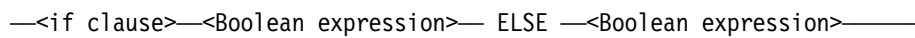
<Boolean operand>



<Boolean case expression>



<conditional Boolean expression>



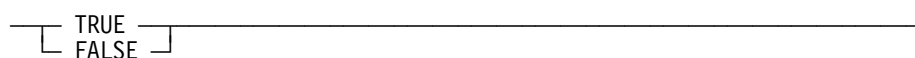
The Boolean concatenation expression is described in "Concatenation Expression" earlier in this section.

The partial word part is described in "Partial Word Expression" earlier in this section.

The evaluation of a Boolean case expression is described in "Case Expression" later in this section.

Boolean Value

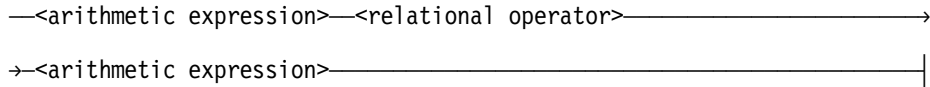
<Boolean value>



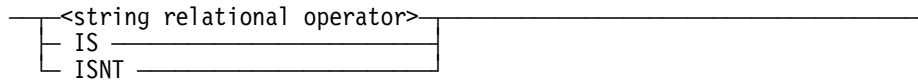
The Boolean value TRUE is represented internally as a 48-bit word containing 4"0000000000001", and the Boolean value FALSE is represented internally as a 48-bit word containing 4"0000000000000".

Arithmetic Relation

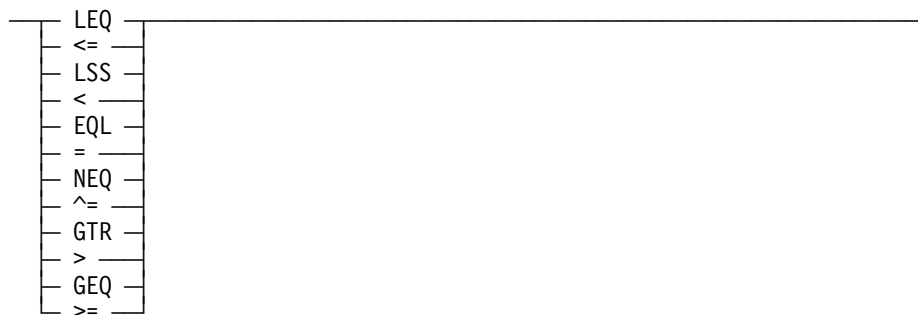
<arithmetic relation>



<relational operator>



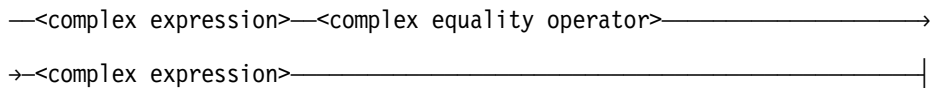
<string relational operator>



An arithmetic relation performs an arithmetic comparison of the values of two arithmetic expressions. The value of the relation is either TRUE or FALSE.

Complex Relation

<complex relation>



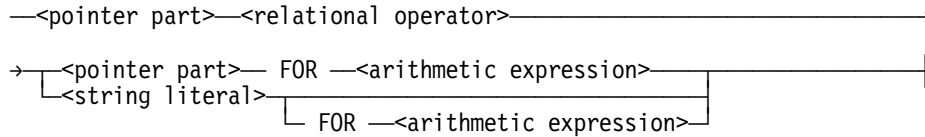
<complex equality operator>



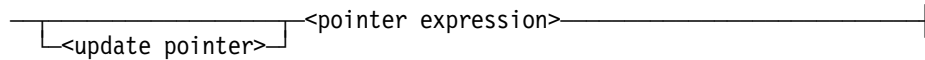
A complex relation performs a comparison between the values of two complex expressions. Because one of the forms of a complex expression is an arithmetic expression, the complex relation can also be used to compare a complex value and an arithmetic value. In a complex relation, the only allowed relational operators are `=`, `EQL`, `^=`, and `NEQ`.

String Relation

<string relation>



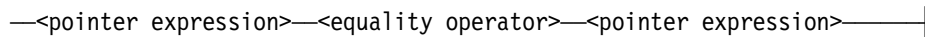
<pointer part>



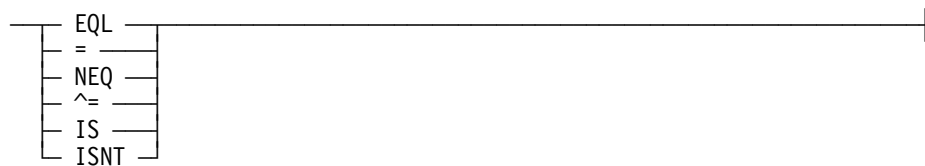
The string relation syntax causes a comparison to be performed between two character strings referenced by two pointer expressions, or between the character string referenced by a pointer expression and a string literal. The character strings are compared according to the EBCDIC collating sequence. The arithmetic expression specifies the number of characters to be compared (the repeat count). If a string literal follows the relational operator and a repeat count has been specified, then the string literal is concatenated with itself, if necessary, to form a 48-bit literal. The comparison is repeated until the repeat count is exhausted. If no repeat count is specified, the string characters are compared once. If a repeat count is specified, and its value is zero, the relation unconditionally produces a TRUE value.

Pointer Relation

<pointer relation>



<equality operator>



A pointer relation determines whether two pointer expressions refer to the same character position in the same array row. If the character sizes of the two pointer expressions are unequal, the comparison always yields the value FALSE.

A pointer relation should not be confused with a string relation. A string relation compares the character strings referenced by pointer expressions. A pointer relation compares the pointer expressions themselves. For example, assume that pointers P1 and P2 are initialized as follows:

```
POINTER P1,P2;  
REAL ARRAY A,B[0:1];  
P1 := POINTER(A,8);  
P2 := POINTER(B,8);  
REPLACE P1 BY "A";  
REPLACE P2 BY "A";
```

Given these initializations, the following string relation has a value of TRUE:

```
P1 EQL P2 FOR 1
```

The following pointer relation has a value of FALSE because P1 refers to array A and P2 refers to array B:

```
P1 EQL P2
```

String Expression Relation

<string expression relation>

```
—<string expression>—<string relational operator>—————→  
→<string expression>—————|
```

The string expression relation compares two string expressions according to the collating sequence of the character type of the string expressions. Only string expressions of the same character type can be compared.

Two strings are equal only if the lengths of the two strings are equal and if every character in one string is equal to the corresponding character in the other string. Two null strings are equal.

One string is strictly greater than a second string only if at least one of the following conditions is true:

- The leftmost character in the first string that is not equal to the corresponding character in the second string compares as greater than the corresponding character in the second string.
- The length of the first string is greater than the length of the second string, and the two strings compare as equal for the length of the second string.

Whenever two string literals can be compared as two arithmetic primaries, they are so compared. For example, the following Boolean expression yields a value of FALSE, because the two quoted strings are treated as two arithmetic primaries:

```
"B" > "AA"
```

However, if the two string literals "B" and "AA" are assigned to two string variables T and S, respectively, then the following Boolean expression yields a value of TRUE, because the first character of T is greater than the first character of S:

T > S

For example, if a string S1 is assigned the value "AAB12+", then all of the following comparisons are TRUE:

```
S1 EQL TAKE(S1,6)
S1 NEQ HEAD(S1,ALPHA)
S1 GTR HEAD(S1,ALPHA)
S1 LSS DROP(S1,3)
S1 LSS DROP(S1,1)
```

Table Membership

<arithmetic table membership>

—<arithmetic expression>— IN —<truth set table>—————|

<pointer table membership>

—<pointer expression>— IN —<truth set table>—————>

→ [FOR —<arithmetic expression>—]—————|

The table membership constructs allow testing to determine whether a character is a member of a truth set. The character can be either a character in a string literal or a character in an array row referenced by a pointer expression. In a pointer table membership primary, the *FOR* <arithmetic expression> part applies the membership test to the first arithmetic expression characters to which the pointer expression points. If the pointer expression does not have the same character size as the truth set table, unpredictable results can occur.

The subscripted variable form of the truth set table construct allows several truth set tables to be contained in one array row. The value of the subscript indicates the beginning of the desired table within the array row. For a description of truth sets, refer to "TRUTHSET Declaration" in Section 3, "Declarations."

Examples of Boolean Expressions

Valid

TRUE
BOOL
B.[11:1]
B := TRUE
B1 := * AND B2
CSIN(C1) = CSIN(C2)
X > Y
NOT B
R = 3
PTR = "ABCD"
P1 < P2 FOR 20
S1 !! S2 = S3
HAPPENED(E)
CASE I OF (B1, TRUE, X = Y)
NOT FYLE.OPEN
P1 = P2
BOOL & TRUE [19:1]
X IN ALPHA8
P1 IN ALPHA8 FOR 6

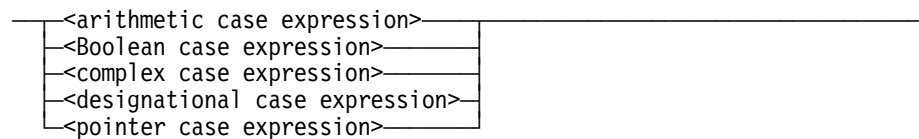
Invalid

SQRT(X)
5.67
-X
R + TRUE
CSIN(C1) > CSIN(C2)
P1 > P2
S1 IS "ABCD"

Case Expression

Case expressions provide a means of selecting one expression from a list of expressions for evaluation.

<case expression>



In a case expression, the list of expressions must be composed of expressions of the same kind: for example, all arithmetic expressions or all Boolean expressions. The expression to be evaluated is selected as follows:

1. The arithmetic expression in the case head is evaluated and rounded by an integer value, if necessary.
2. This value is used as an index into the expression list. The component expressions of the expression list are numbered sequentially from 0 through N-1, where N is the number of expressions in the list.
3. The expression selected by the index is evaluated, and its value is the value of the case expression.

If the value of the index lies outside the range 0 through N-1, the program is discontinued with a fault.

The type of an arithmetic case expression is DOUBLE if any of its constituent expressions is of type DOUBLE; in this case, any constituent expression that is not of type DOUBLE is extended to double-precision. The type of an arithmetic case expression is INTEGER if and only if all of its constituent expressions are of type INTEGER. Otherwise, an arithmetic case expression is of type REAL.

Examples of Case Expressions

```
CASE N OF (2, 20, 100, 37)
```

```
CASE X.[27:2] OF (TRUE, FALSE, TRUE, TRUE)
```

```
CASE I OF (C1, C2, COMPLEX(X,Y))
```

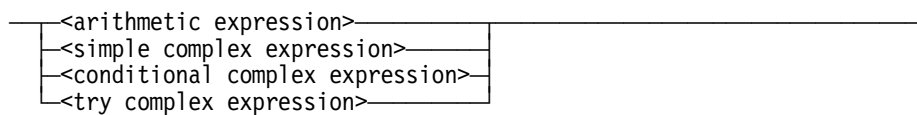
```
CASE TSTS[INDEX] OF (LBL1, LBL2, AGAIN, NEXT, MORE)
```

```
CASE CHAR.SZF OF (PTR, PTS, POINTER(A), PTEMP, POLO)
```

Complex Expression

Complex expressions are expressions that return complex values (arithmetic values that consist of a real part and an imaginary part) by applying specified operations to designated complex primaries.

<complex expression>



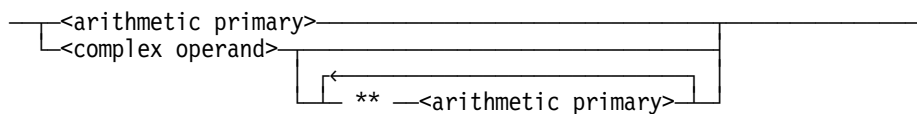
<simple complex expression>



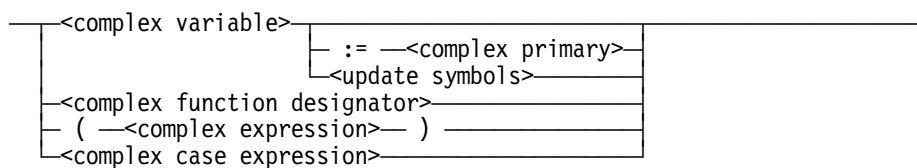
<complex operator>



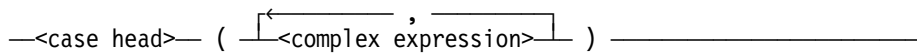
<complex primary>



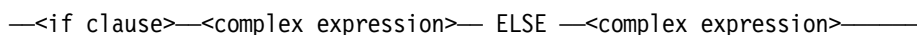
<complex operand>



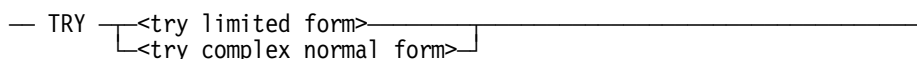
<complex case expression>



<conditional complex expression>



<try complex expression>



<try complex normal form>

—<complex expression> [ELSE —<complex expression>]

An explanation of <try complex expression> is given in “TRY Expression” later in this section.

The imaginary part of a complex value can be equal to zero. Therefore, an arithmetic expression is, whenever necessary, considered to be the real part of a complex expression with a zero imaginary part. No automatic type conversion from complex to arithmetic exists.

The sequence in which the operations of a complex expression are performed is determined by the precedence of the operators involved. The order of precedence is as follows:

1. *, / (highest precedence)
2. +, -

Operators with the same precedence are applied in their order of appearance in an expression, from left to right.

The precedence of the assignment operator (:=) is as follows:

1. A primary to the right of an assignment operator is evaluated before the assignment.
2. The assignment is done before the evaluation of an expression involving the variable that is the target of the assignment.

For information on the order of evaluation of subscripts, see “Arithmetic Update Assignment” in Section 4, “Statements.”

Parentheses can be used in normal mathematical fashion to override the defined order of precedence. An expression in parentheses is evaluated by itself, and the resulting value is subsequently combined with the other elements of the expression. In the following expression, for example, the addition is performed before the division because of the parentheses:

$$(C2 + C1)/C2$$

In the following expression, C1 is first divided by C2 and then the result is added to C2.

$$C2 + C1/C2$$

The evaluation of a conditional complex expression is described in “Conditional Expression” later in this section.

The evaluation of a complex case expression is described in “Case Expression” earlier in this section.

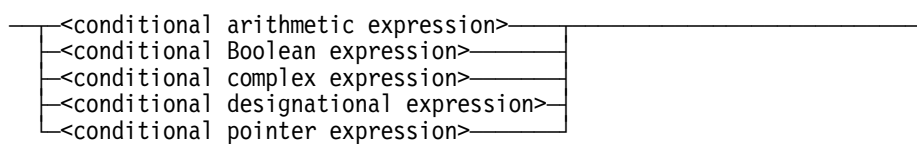
Examples of Complex Expression

```
C1
C1+3
C1 ** X
C1 := COMPLEX(X,Y)
C1 := * - C2
CABS(C1)
(COMPLEX(R,S) * CCON)
CASE I OF (C1, C2, CLN(C2))
IF BOOL THEN C1 ELSE CONJUGATE(C1)
```

Conditional Expression

Conditional expressions are expressions that return one of two possible values, depending upon a specified condition.

<conditional expression>



Conditional expressions are of the following form:

```
IF <Boolean expression> THEN <expression> ELSE <expression>
```

Either the first or the second expression is selected for evaluation, depending on the value of the Boolean expression. The two alternative expressions must be of the same kind: for example, two arithmetic expressions or two Boolean expressions.

The expression to be evaluated is selected as follows:

1. The Boolean expression following IF is evaluated.
2. If the resulting value is TRUE, the expression following THEN is evaluated, and the expression following ELSE is ignored.
3. If the resulting value is FALSE, the expression following THEN is ignored, and the expression following ELSE is evaluated.

If either of the two expressions is itself a conditional expression, the process is repeated until an unconditional expression is selected for evaluation.

The type of a conditional arithmetic expression is DOUBLE if either of its constituent expressions is of type DOUBLE; in this case, a constituent expression that is not of type DOUBLE is extended to double-precision. The type of a conditional arithmetic expression is INTEGER if and only if both of its constituent expressions are of type INTEGER. Otherwise, a conditional arithmetic expression is of type REAL.

Examples of Conditional Expressions

```
IF BOOL THEN 47 ELSE 95

IF A = B THEN BOOL ELSE FALSE

IF NOT BOOL THEN C1 ELSE C2

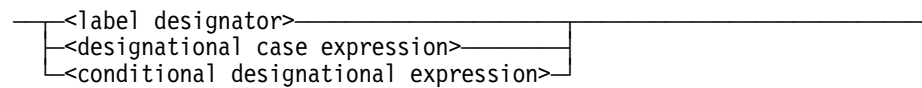
IF ALLDONE THEN EOJLBL ELSE NEXTLBL

IF CHAR.SZF = 8 THEN PTRINEBCDIC ELSE PTRINHEX
```

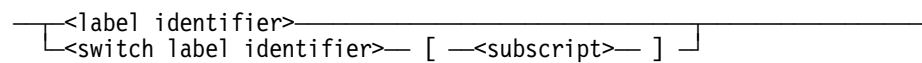
Designational Expression

Designational expressions are expressions that return a value that is a label.

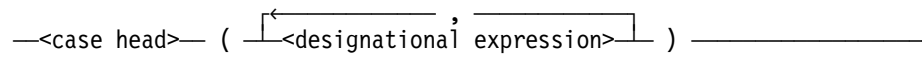
<designational expression>



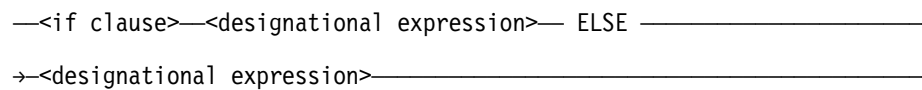
<label designator>



<designational case expression>



<conditional designational expression>



If a designational expression is a label identifier, then the value of the expression is that label.

Expressions and Functions

If a designational expression is a subscripted switch label identifier, then the numerical value of the subscript designates one of the elements in the switch label list. The value of the subscript is rounded, if necessary, to an integer. This value is used as an index into the switch label list. The entries of the list are numbered sequentially from 1 through N, where N is the number of entries in the list. The entry corresponding to the value of the subscript is selected. If the value of the subscript is outside the range of the switch label list, program control continues to the next statement without any error indication. For more information about the switch label list, refer to "SWITCH LABEL Declaration" in Section 3, "Declarations."

The evaluation of a designational case expression is described in "Case Expression" earlier in this section.

The evaluation of a conditional designational expression is described in "Conditional Expression" earlier in this section.

Examples of Designational Expression

```
ENDLABEL
```

```
CHOOSELABEL[I+2]
```

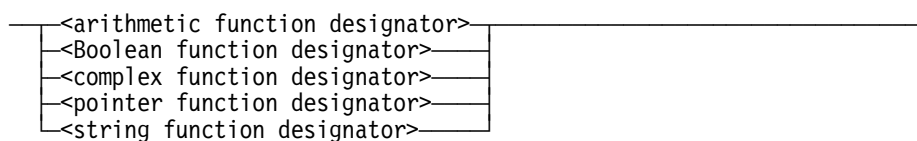
```
CASE X OF (GOTDATA, GOTERR, GOTREAL, GOTCHANGE, ESCAPE)
```

```
IF K = 1 THEN SELECT[2] ELSE START
```

Function Expression

A function expression is an expression that returns a single value that is the result of invoking a procedure. The procedure can be declared in the program, or it can be an intrinsic procedure.

<function expression>



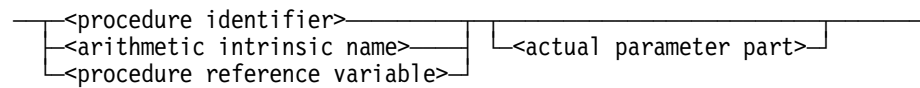
The two kinds of functions are predefined, or intrinsic, functions which are part of the ALGOL language, and programmer-defined functions, which are typed procedures that are declared in the program.

The intrinsic functions are described later in this section in "Intrinsic Functions."

Arithmetic Function Designator

An arithmetic function designator specifies a function that returns an arithmetic value: a value of type INTEGER, REAL, or DOUBLE.

<arithmetic function designator>



<arithmetic intrinsic name>

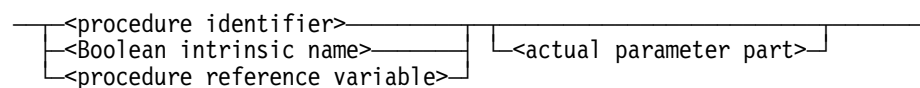
Any of the names listed under "Arithmetic Intrinsic Names" later in this section.

The procedure specified by the procedure identifier or the procedure reference variable must be of type INTEGER, REAL, or DOUBLE.

Boolean Function Designator

A Boolean function designator specifies a function that returns a Boolean value (a value of TRUE or FALSE).

<Boolean function designator>



<Boolean intrinsic name>

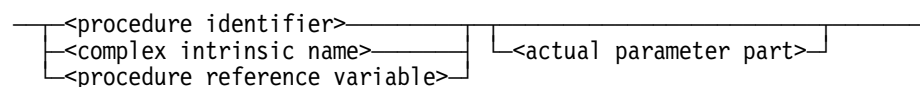
Any of the names listed under "Boolean Intrinsic Names" later in this section.

The procedure specified by the procedure identifier or the procedure reference variable must be of type BOOLEAN.

Complex Function Designator

A complex function designator specifies a function that returns a complex value: a value with a real part and an imaginary part.

<complex function designator>



The procedure specified by the procedure identifier or the procedure reference variable must be of type COMPLEX.

Pointer Function Designator

A pointer function designator specifies a function that returns a pointer value: a value that can be used to refer to a character position in an array row.

<pointer function designator>

—<pointer intrinsic name>—<actual parameter part>—————|

Unlike the other function designators, the syntax for pointer function designator does not allow a procedure identifier as part of the syntax. There is no PROCEDURE declaration that allows declaring a procedure of type POINTER.

String Function Designator

A string function designator specifies a function that returns a string value: a hexadecimal string, an ASCII string, or an EBCDIC string.

<string function designator>

—<string procedure identifier>—|
|<string intrinsic name>—|<actual parameter part>—|
|<procedure reference variable>—|

<string intrinsic name>

Any of the names listed under “String Intrinsic Names” later in this section.

The procedure reference variable must designate a procedure declared with a type of HEX STRING, ASCII STRING, or EBCDIC STRING.

NULL Value

<null value>

— NULL —————|

A null value can be used to uninitialized procedure references, pointers, array references, structure block references, or connection block references to null through the use of <pointer assignment>, <array reference assignment>, <procedure reference assignment>, <structure block reference assignment>, or <connection block reference assignment>. NULL can also be used as an actual parameter for a formal procedure, array, call-by-value pointer parameter, structure block variable parameter, or connection block parameter.

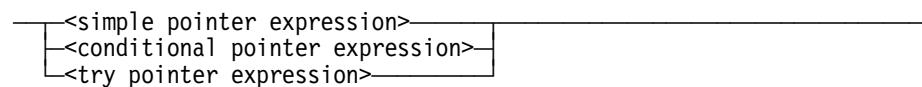
When used in a procedure reference assignment, if NULL is specified and there is an environment called NULL, then a reference to the procedure called NULL is assigned. If NULL is specified and there is no environment called NULL, then a NULL value is assigned to the procedure reference variable.

If a pointer or array reference variable contains NULL and an attempt to use that variable is made, a program interrupt occurs. If the procedure reference variable is invoked while it is NULL, a program interrupt occurs.

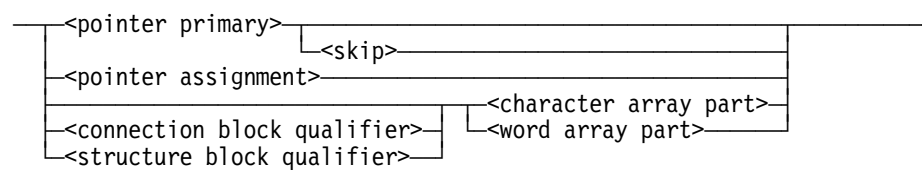
Pointer Expression

A pointer expression is an expression that returns a value that is a pointer, which can be used to reference a character position in an array row.

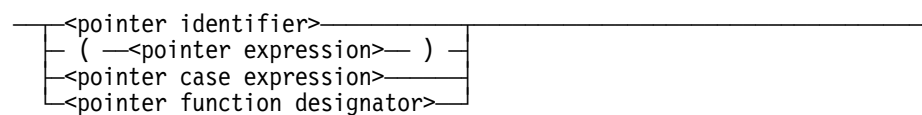
<pointer expression>



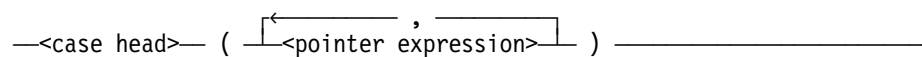
<simple pointer expression>



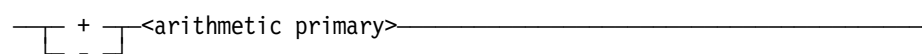
<pointer primary>



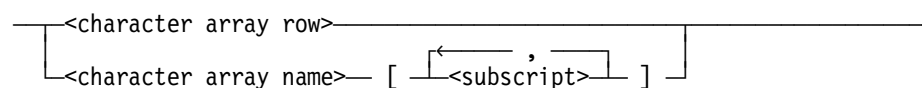
<pointer case expression>



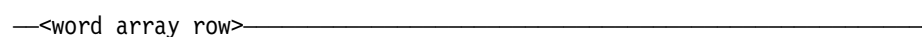
<skip>



<character array part>



<word array part>



<character array row>

An array row whose identifier is declared with a character type.

<word array row>

An array row whose identifier is declared with a noncharacter type.

<character array name>

An array name whose identifier is declared with a character type.

<conditional pointer expression>

—<if clause>—<pointer expression>— ELSE —<pointer expression>—|

<try pointer expression>

— TRY —<try pointer normal form>—|

<try pointer normal form>

—<pointer expression>—| ELSE —<pointer expression>—|

An explanation of <try pointer expression> is given in “TRY Expression” later in this section.

A pointer must be initialized before it can be used; otherwise, a run-time error occurs. A pointer can be initialized in the following ways:

- By the use of a pointer assignment
- By the appearance of an update pointer in any of the following:
 - A REPLACE statement
 - A SCAN statement
 - A string relation in a Boolean expression
 - The DINTEGER function
 - The DOUBLE function
 - The INTEGER function

When a one-dimensional array designator is used as a pointer expression, it references the beginning of the array, no matter what its lower bounds are.

The use of a <word array part> results in a word pointer that is not acceptable when a character pointer is required.

The evaluation of a conditional pointer expression is described in “Conditional Expression” earlier in this section.

A one-dimensional array designator or a fully subscripted variable can be interpreted as a pointer primary whenever context determines that no conflict exists with other valid constructs (for example, when a pointer expression is required). This syntax can be used for such constructs as the following, where A and B are one-dimensional arrays:

```
REPLACE A BY B FOR 10 WORDS
```

The evaluation of a pointer case expression is described in "Case Expression" earlier in this section.

If the skip construct is used, the value of the arithmetic primary determines the adjustment to the value of the pointer primary. If N is the value of the arithmetic primary, the pointer is adjusted as follows:

- If N is less than or equal to 0, the pointer is not adjusted.
- If N is greater than 0, then the pointer is adjusted N characters to the right when the skip construct specifies "+", or N characters to the left if it specifies "-". Skipping to the right is defined as incrementing the value of the character index. Skipping to the left is defined as decrementing this value.

If the adjustment to the value of the pointer primary is given by the value of an arithmetic expression, note that the arithmetic expression must be enclosed in parentheses. For example, the following expression is invalid and generates a compile-time error:

```
PTR + X*Y
```

The expression is written correctly as follows:

```
PTR + (X*Y)
```

The use of a pointer expression to skip up and down an array for more than a few words is expensive. Each word of the array is accessed in order to ensure that no memory-protected words are encountered. For pointer moves of more than a few words, it is faster to reindex the array and use the POINTER function for word arrays, or to reindex the array for character arrays.

Examples of Pointer Expressions

```
PTR
```

```
PTS+15
```

```
PTR := POINTER(A)
```

```
(PTEMP + (X*Y))
```

```
HEXARRAY
```

```
HEXARRAY[N]
```

```
CASE VAL OF (PTR,PTS,PTEMP,PSORCE)
```

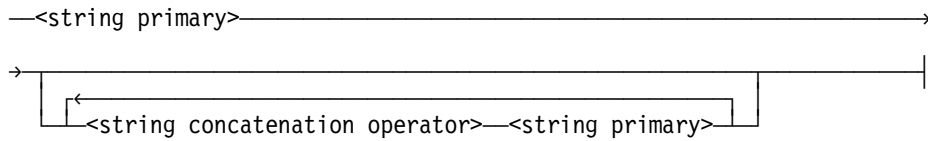
```
POINTER(INFO,8)
```

```
IF BOOL THEN P1 ELSE POINTER(A)
```

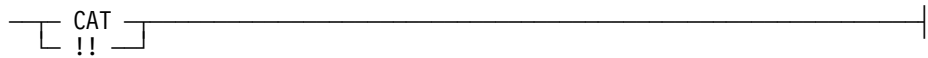
String Expression

A string expression is an expression that returns a value that is a hexadecimal string, EBCDIC string, or ASCII string.

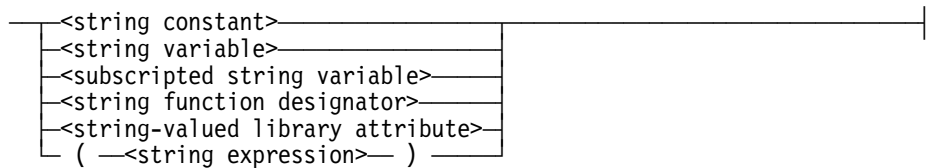
<string expression>



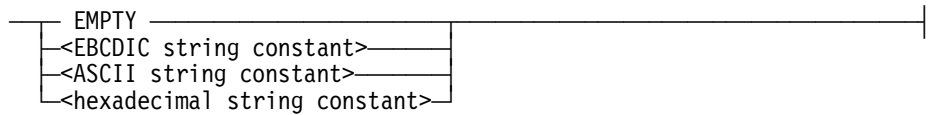
<string concatenation operator>



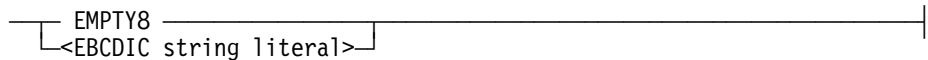
<string primary>



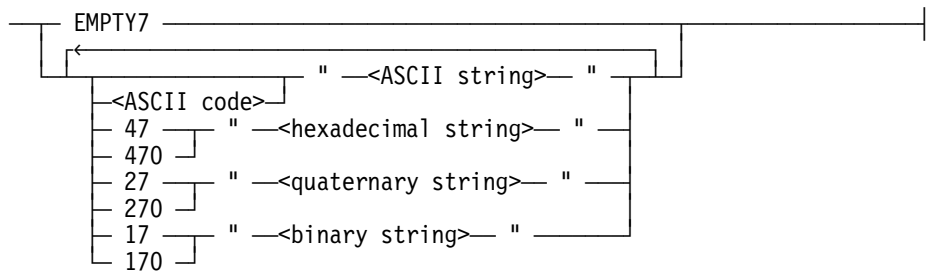
<string constant>



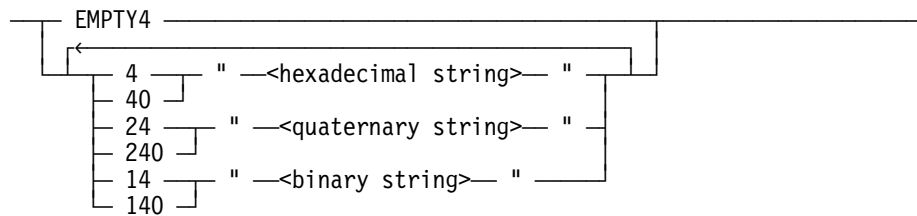
<EBCDIC string constant>



<ASCII string constant>



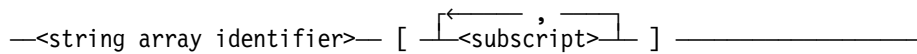
<hexadecimal string constant>



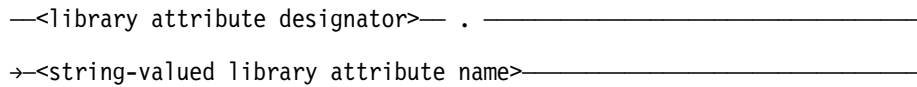
<string variable>



<subscripted string variable>



<string-valued library attribute>



<constant string expression>

A string expression that can be fully evaluated at compilation time.

In the syntax for EBCDIC string constant, ASCII string constant, and hexadecimal string constant, the string codes determine the interpretation of the characters between the quotation marks (") and have no effect on the justification of a string. A string is always left-justified; therefore, any 0 (zero) in a string code is ignored.

The <EBCDIC code> construct is optional in an EBCDIC string constant containing an EBCDIC string only if the default character type is EBCDIC. The <ASCII code> construct is optional in an ASCII string constant containing an ASCII string only if the default character type is ASCII. For more information, refer to "String Code" in Section 2, "Language Components," and "Default Character Type" in Appendix C, "Data Representation."

The reserved words EMPTY8, EMPTY7, and EMPTY4 represent null strings of the character types EBCDIC, ASCII, and hexadecimal, respectively. The reserved word EMPTY represents a null string of the default character type.

For more information about string-valued library attributes, refer to "Library Attributes" in Section 8, "Library Facility."

String Concatenation

The operators CAT and !! are used to concatenate two strings. The concatenation of two strings yields a new string whose length is the sum of the lengths of the two original strings, and whose value is formed by joining a copy of the second string immediately onto the end of a copy of the first string.

Only strings of the same character type can be concatenated.

If more than one string primary is used in a concatenation operation, the string primaries are evaluated from left to right.

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.

Examples of String Expressions

8"ABCD123" % result = ABCD123

"WHY"48"6F" "" % result = "WHY?"

EMPTY8 % result =

S2 !! S3

"AC" !! S2 !! "123"

S1 !! TRANSLATE(S2,HEXTOEBCDIC)

HEAD(S,ALPHA)

TAIL(S,NOT "-")

REPEAT("ABC",3)

STRING(256,*)

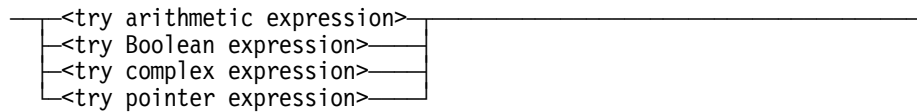
TAKE(S,2)

DROP(TAKE(S,4),2)

TRANSLATE(S,HEXTOEBCDIC)

TRY Expression

<try expression>



Explanation

The TRY expression is similar to the TRY statement, except that the expression is used when a returned value is needed. The value returned by the TRY expression is the value returned by the expression or procedure call that finished normally. When the <try limited form> is used in the <try arithmetic expression>, the procedure identifier specified for the error procedure must be of type INTEGER, REAL, or DOUBLE. Also, the <procedure invocation statement> or <procedure reference statement> must invoke a procedure of type INTEGER, REAL, or DOUBLE.

When the <try limited form> is used in the <try Boolean expression>, the procedure identifier specified for the error procedure must be of type BOOLEAN. Also, the <procedure invocation statement> or <procedure reference statement> must invoke a procedure of type BOOLEAN. In all cases of <try limited form> used in the TRY expression, the types of the error procedure and the invoked procedure must match exactly.

When the <try *kind* normal form> is used, the expressions must be of the same kind—for example, all arithmetic expressions or all Boolean expressions. For <try arithmetic normal form>, the type of a <try arithmetic expression> is DOUBLE if any of its constituent expressions are of type DOUBLE; in this case, a constituent expression that is not of type DOUBLE is extended to double-precision. The type of a <try arithmetic expression> is INTEGER only if all its constituent expressions are of type INTEGER. Otherwise, a <try arithmetic expression> is of type REAL.

Intrinsic Functions

Intrinsic functions are typed procedures that are predefined in the ALGOL language; that is, intrinsic functions can be used without being declared.

Intrinsic Names by Type Returned

The intrinsic functions of ALGOL return values of type INTEGER, REAL, DOUBLE, BOOLEAN, COMPLEX, POINTER, and STRING.

Arithmetic Intrinsic Names

The following intrinsic functions return arithmetic values of type DOUBLE, INTEGER, or REAL.

Table 5-1. Arithmetic Intrinsic Functions

Function	DOUBLE	INTEGER	REAL
ABS			X
ARCCOS			X
ARCSIN			X
ARCTAN			X
ARCTAN2			X
ARRAYSEARCH		X	
ATANH			X
CABS			X
CHECKSUM			X
CLOSE		X	
COMPILETIME			X
COS			X
COSH			X
COTAN			X
DABS	X		
DALPHA	X		
DAND	X		
DARCCOS	X		
DARCSIN	X		

Table 5-1. Arithmetic Intrinsic Functions

Function	DOUBLE	INTEGER	REAL
DARCTAN	X		
DARCTAN2	X		
DCOS	X		
DCOSH	X		
DECIMAL	X		
DELINKLIBRARY		X	
DELTA		X	
DEQV	X		
DERF	X		
DERFC	X		
DEXP	X		
DGAMMA	X		
DIMP	X		
DINTEGER	X		
DINTEGERT	X		
DLGAMMA	X		
DLN	X		
DLOG	X		
DMAX	X		
DMIN	X		
DNABS	X		
DNORMALIZE	X		
DNOT	X		
DOR	X		
DOUBLE	X		
DSCALELEFT	X		
DSCALERIGHT	X		
DSCALERIGHTT	X		
DSIN	X		
DSINH	X		
DSQRT	X		
DTAN	X		

Table 5-1. Arithmetic Intrinsic Functions

Function	DOUBLE	INTEGER	REAL
DTANH	X		
ENTIER		X	
ERF			X
ERFC			X
EXP			X
FIRST			X
FIRSTONE		X	
FIRSTWORD			X
GAMMA			X
IMAG			X
INTEGER		X	
INTEGRT		X	
LENGTH		X	
LINENUMBER		X	
LINKLIBRARY		X	
LISTLOOKUP		X	
LN			X
LNGAMMA			X
LOCK		X	
LOCKSTATUS			X
LOG			X
MASKSEARCH		X	
MAX			X
MESSAGESEARCHER		X	
MIN			X
MLSTRANSLATE		X	
NABS			X
NORMALIZE			X
OFFSET		X	
ONES		X	
OPEN		X	
POTC	X		

Table 5-1. Arithmetic Intrinsic Functions

Function	DOUBLE	INTEGER	REAL
POTH	X		
POTL	X		
PROCESSID		X	
RANDOM			X
READLOCK			X
READYCL		X	
REAL			X
REMAININGCHARS		X	
SCALELEFT		X	
SCALERIGHT		X	
SCALERIGHTF			X
SCALERIGHTT		X	
SECONDWORD			X
SETACTUALNAME		X	
SIGN		X	
SIN			X
SINGLE			X
SINH			X
SIZE		X	
SQRT			X
TAN			X
TANH			X
TIME			X
UNLOCK		X	
UNREADYCL		X	
VALUE		X	
WAIT		X	
WAITANDRESET		X	

Boolean Intrinsic Names

The following intrinsic functions return values of type BOOLEAN:

ACCEPT	FREE	REMOVEFILE
AVAILABLE	HAPPENED	SEEK
BOOLEAN	ISVALID	SPACE
CHANGEFILE	MLSACCEPT	WAIT
CHECKPOINT	READ	WRITE
FIX	READLOCK	

Complex Intrinsic Names

The following intrinsic functions return values of type COMPLEX:

CCOS	CONJUGATE
CEXP	CSIN
CLN	CSQRT
COMPLEX	

Pointer Intrinsic Names

The following intrinsic functions return values of type POINTER:

POINTER	READLOCK
---------	----------

String Intrinsic Names

The following intrinsic functions return values of type STRING:

DROP	STRING7
HEAD	STRING8
REPEAT	TAIL
STRING	TAKE
STRING4	TRANSLATE

Intrinsic Function Descriptions

For arithmetic intrinsic functions, all arithmetic parameters are assumed to be called-by-value. For a further description of some of the arithmetic intrinsic functions, refer to the *System Software Utilities Operations Reference Manual*.

ABS Function

— ABS — (—<arithmetic expression>—) _____|

The ABS function returns, as a real value, the absolute value of the specified arithmetic expression.

ACCEPT Statement

The ACCEPT statement returns a Boolean value. For more information, refer to “ACCEPT Statement” in Section 4, “Statements.”

ARCCOS Function

— ARCCOS — (—<arithmetic expression>—) _____|

The ARCCOS function returns, as a real value, the principal value of the arccosine (in radians) of the specified arithmetic expression. The arithmetic expression is greater than -1 and less than 1 . If the value of the arithmetic expression is not in this range, a run-time error occurs.

ARCSIN Function

— ARCSIN — (—<arithmetic expression>—) _____|

The ARCSIN function returns, as a real value, the principal value of the arcsine (in radians) of the specified arithmetic expression. The arithmetic expression is greater than -1 and less than 1 . If the value of the arithmetic expression is not in this range, a run-time error occurs.

ARCTAN Function

— ARCTAN — (—<arithmetic expression>—) _____|

The ARCTAN function returns, as a real value, the principal value of the arctangent (in radians) of the specified arithmetic expression.

ARCTAN2 Function

— ARCTAN2 — (—<arithmetic expression>— , —————→
→<arithmetic expression>—) —————|

The ARCTAN2 function returns, as a real value, the arctangent (in radians) of the following:

$$\text{first } \langle \text{arithmetic expression} \rangle / \text{second } \langle \text{arithmetic expression} \rangle$$

The returned value is adjusted according to the following formulas so that the value falls in the range $(-\pi$ to $+\pi)$.

In the following equations, let X be the value of the first arithmetic expression, and let Y be the value of the second arithmetic expression. If Y is greater than 0, then

$$\text{ARCTAN2}(X,Y) = \text{ARCTAN}(X/Y)$$

If Y equals 0, then

$$\text{ARCTAN2}(X,Y) = \text{SIGN}(X) * \pi / 2$$

If Y is less than 0, then

$$\text{ARCTAN2}(X,Y) = \text{ARCTAN}(X/Y) + \text{SIGN}(X) * \pi$$

ARRAYSEARCH Function

— ARRAYSEARCH — (—<arithmetic expression>— , —————→
→<arithmetic expression>— , —————|
|<array row>—————|
|<subscripted variable>|) —————|

The ARRAYSEARCH function searches for a specific value within an array. The first arithmetic expression is the value being searched for (the target value). The second arithmetic expression is the mask to be used in the search. The third parameter is a row or subscripted variable of an array of type INTEGER, REAL, BOOLEAN, or COMPLEX. For information on the subscripted variable, see "Arithmetic Assignment" in Section 4, "Statements."

If the third parameter is an array row, the search begins with the last element of the specified array row; otherwise, the search begins with the element specified by the subscripted variable. Each element, in turn, is retrieved and the Boolean operation AND is performed using the value of the mask and the element as the operators. The Boolean operation AND is also performed on the target value. The results of these two operations are then compared with each other, using the IS operator. For more information on the IS operator, see "IS and ISNT Operators" earlier in this section.

If the comparison yields the value TRUE, the function returns an integer value equal to the difference between the subscript of the element where the value is found and the subscript of the first element in the array. If the comparison yields the value FALSE, the subscript of the element to be retrieved is decremented by 1, and the search continues until either a match is found or the first element of the array has been examined. If no match is found, -1 is returned. For information on <array row>, see "ARRAY Declaration" in Section 3, "Declarations."

The ARRAYSEARCH function can be used on paged (segmented) arrays and unpaged (unsegmented) arrays.

ATANH Function

— ATANH — (—<arithmetic expression>—) _____|

The ATANH function returns, as a real value, the hyperbolic arctangent of the specified arithmetic expression.

AVAILABLE Function

— AVAILABLE — (—<event designator>—) _____|

The AVAILABLE function is a Boolean function that returns the value TRUE if the available state of the specified event is TRUE (available) and returns the value FALSE if the available state is FALSE (not available).

BOOLEAN Function

— BOOLEAN — (—<arithmetic expression>—) _____|

The BOOLEAN function returns the value of the arithmetic expression as a Boolean value. If the arithmetic expression is double-precision, its value is first truncated to single-precision.

CABS Function

— CABS — (—<complex expression>—) —————|

The CABS function returns, as a real value, the absolute value of the specified complex expression.

CCOS Function

— CCOS — (—<complex expression>—) —————|

The CCOS function returns, as a complex value, the complex cosine of the specified complex expression.

CEXP Function

— CEXP — (—<complex expression>—) —————|

The CEXP function returns the following as a complex value, where e is the base of the natural logarithms:

`e ** <complex expression>`

CHANGEFILE Statement

The CHANGEFILE statement returns a Boolean value. For more information, refer to “CHANGEFILE Statement” in Section 4, “Statements.”

CHECKPOINT Statement

The CHECKPOINT statement returns a Boolean value. For more information, refer to “CHECKPOINT Statement” in Section 4, “Statements.”

CHECKSUM Function

— CHECKSUM — (—<array row>— , —<starting index>— , —————→
→<ending index>—) —————|

<starting index>

—<arithmetic expression>—————|

<ending index>

—<arithmetic expression>—————|

The CHECKSUM function returns, as a real value, a hash function of all bits in the words of the array row beginning with the word indexed by the <starting index> and up to, but not including, the word indexed by the <ending index>. Both the starting index and ending index are rounded to an integer value before they are used to index the array row. The value of this function can be used to verify the integrity of data being transferred or stored. The array row must be single-precision and unpagged (unsegmented).

The CHECKSUM function can return the value of 0 or 1 as a valid result or as an error result. When either index is beyond the bounds of the array, the value returned is 1. If the span of the indexes is less than 2, then the result returned is 0.

CLN Function

— CLN — (—<complex expression>—) —————|

The CLN function returns, as a complex value, the natural logarithm of the specified complex expression.

CLOSE Statement

The CLOSE statement returns an integer value. For more information, refer to “CLOSE Statement” in Section 4, “Statements.”

COMPILETIME Function

— COMPILETIME — (—<constant arithmetic expression>—) —————|

The COMPILETIME function obtains various system time values at compilation time, for use by the object program. The form of the value returned by the COMPILETIME function is the same as that returned by the TIME function for the same argument. The compiler computes the returned value by using the TIME function at compilation time. For more information, refer to “TIME Function” later in this section.

COMPILETIME(20) returns, in integer form, the program version number as set by the most recent VERSION compiler control option.

COMPILETIME(21) returns, in integer form, the program cycle number as set by the most recent VERSION compiler control option.

COMPILETIME(22) returns, in integer form, the program patch number as set by the most recent VERSION compiler control option.

COMPLEX Function

— COMPLEX — (—<arithmetic expression>— , —————→
→<arithmetic expression>—) —————|

The COMPLEX function returns the following as a complex value, where i is the square root of -1 :

first <arithmetic expression> + i * second <arithmetic expression>

The arithmetic expressions are first rounded to single-precision, if necessary.

CONJUGATE Function

— CONJUGATE — (—<complex expression>—) —————|

The CONJUGATE function returns, as a complex value, the complex conjugate of the specified complex expression.

COS Function

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

The COS function returns, as a real value, the cosine of an angle of <arithmetic expression> radians.

COSH Function

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

The COSH function returns, as a real value, the hyperbolic cosine of the specified arithmetic expression.

COTAN Function

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

The COTAN function returns, as a real value, the cotangent of an angle of <arithmetic expression> radians.

CSIN Function

— CSIN — (—<complex expression>—) —————|

The CSIN function returns, as a complex value, the complex sine of the specified complex expression.

CSQRT Function

— CSQRT — (—<complex expression>—) —————|

The CSQRT function returns, as a complex value, the square root of the specified complex expression.

DABS Function

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

The DABS function returns, as a double-precision value, the absolute value of the specified arithmetic expression.

DALPHA Function

— DALPHA — (—<pointer expression>— , —<arithmetic expression>—
→) —————|

The DALPHA function returns, as a double-precision value, a bit image of the string of <arithmetic expression> characters starting with the character indicated by the specified pointer expression.

DAND Function

— DAND — (—<arithmetic expression>— , —<arithmetic expression>—
→) —————|

The DAND function returns the following as a double-precision value:

first <arithmetic expression> AND second <arithmetic expression>

The arithmetic expressions are first extended to double-precision, if necessary, and the AND operation is performed on all 96 bits.

DARCCOS Function

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

The DARCCOS function returns, as a double-precision value, the principal value of the arccosine (in radians) of the specified arithmetic expression, the arithmetic expression is greater than -1 and less than 1 . If the value of the arithmetic expression is not in this range, a run-time error occurs.

DARCSIN Function

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

The DARCSIN function returns, as a double-precision value, the principal value of the arcsine (in radians) of the specified arithmetic expression, the arithmetic expression is greater than -1 and less than 1 . If the value of the arithmetic expression is not in this range, a run-time error occurs.

DARCTAN Function

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

The DARCTAN function returns, as a double-precision value, the principal value of the arctangent (in radians) of the specified arithmetic expression.

DARCTAN2 Function

— DARCTAN2 — (—<arithmetic expression>— , —————→
 →<arithmetic expression>—) —————|

The DARCTAN2 function returns, as a double-precision value, the principal value of the arctangent (in radians) of the following:

first <arithmetic expression> / second <arithmetic expression>

DCOS Function

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

The DCOS function returns, as a double-precision value, the cosine of an angle of <arithmetic expression> radians.

DCOSH Function

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

The DCOSH function returns, as a double-precision value, the hyperbolic cosine of the specified arithmetic expression.

DECIMAL Function

— DECIMAL — (—<string expression>—) —————|

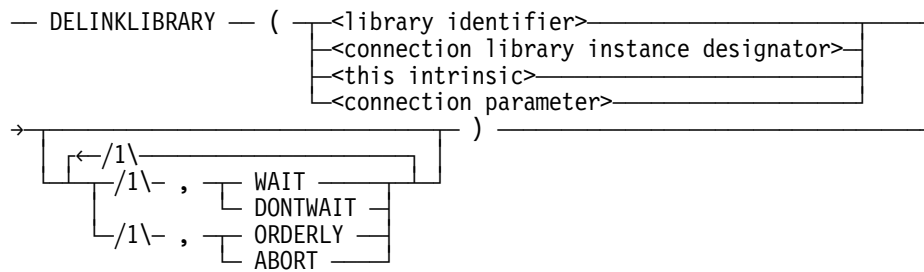
The DECIMAL function returns the double-precision value represented by the string expression. The string expression must yield a valid number on evaluation. For more information on the <number> construct, see Section 2, "Language Components." For example, the assignment *D := DECIMAL(STR)*; is valid for the following strings:

```
STR = "+5497823"
STR = "1.75@-46"
STR = "-4.31468"
STR = "@2"
STR = "+549"!!"7823"
```

However, the program receives a run-time error for the following strings:

```
STR = "50 00."
STR = "1,505,278,00"
STR = "1@2.5"
STR = ".573"!!"5.82"
```

DELINKLIBRARY Function



The DELINKLIBRARY function unlinks the program from the library program specified by the library identifier. The DELINKLIBRARY function affects only the linkage between the program and the indicated library; other programs using the library program are not affected.

The WAIT/DONTWAIT and the ORDERLY/ABORT options are used for connection libraries. The ORDERLY/ABORT option indicates what to do if there are still stacks calling through a connection library. The ORDERLY option indicates that the delinkage is delayed until the identified stacks no longer have a procedure activation from the connection library in their stack. No new calls are allowed through the library while it is delinking. The ABORT option indicates that the identified stacks should be discontinued (DSed).

The WAIT/DONTWAIT option indicates whether to wait for the delink action to finish if the first option is ORDERLY. If the value is WAIT, the connection library is completely delinked when the function returns. If the value is DONTWAIT, the delinkage might not be finished and the program has to check the STATE attribute to determine when it is delinked.

You can direct a program to wait for other stacks to stop using the connection library by having the program first call DELINKLIBRARY with the DONTWAIT and ORDERLY options. Then, after the time has passed, the program can check to see if the library is actually delinked. If the program is not delinked, then DELINKLIBRARY can be called again with the WAIT and ABORT options set.

When a connection library is block exited, DELINKLIBRARY is called with the options set to WAIT, ABORT.

This function returns an integer value that indicates success or failure and the reason for failure. The values returned by the function can be interpreted as follows:

Value	Meaning
2	The connection is already being delinked.
1	The library has been unlinked from the program.
0	The library was not linked to the program.
-1	The library structure could not be accessed; a system fault has occurred.
-2	The library designator is not a server library or a connection library.
-3	The DONTWAIT or ORDERLY option was specified for a Server Library.
-4	The delinkage was not completed because the process was terminated.
-5	DELINKLIBRARY was called from inside a CHANGE or APPROVAL procedure, or from a procedure called from a CHANGE or APPROVAL procedure.

DELTA Function

— DELTA — (—<pointer expression>— , —<pointer expression>—) —|

The DELTA function returns, as an integer value, the number of characters between the character position referenced by the first pointer expression and the character position referenced by the second pointer expression. The value is calculated as follows: the character position referenced by the first pointer expression is subtracted from the character position referenced by the second pointer expression.

For a function that returns the number of characters between the character position referenced by a pointer expression and the beginning of the array row, see "OFFSET Function" later in this section. For a function that returns the number of characters between the character position referenced by a pointer expression and the end of the array row, see "REMAININGCHARS Function" later in this section.

DEQV Function

— DEQV — (—<arithmetic expression>— , —<arithmetic expression>—
→) —————|

The DEQV function returns the following as a double-precision value:

first <arithmetic expression> EQV second <arithmetic expression>

Both arithmetic expressions are first extended to double-precision, if necessary, and the EQV operation is performed on all 96 bits.

DERF Function

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

The DERF function returns, as a double-precision value, the value of the standard error function at the specified arithmetic expression. For any valid N, DERF(-N) = -DERF(N).

DERFC Function

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

The DERFC function returns, as a double-precision value, the complement of the value of the standard error function at the specified arithmetic expression. For any valid N, DERFC(N) = 1 - DERF(N).

DEXP Function

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

The DEXP function returns the following as a double-precision value, where e is the base of the natural logarithms:

e ** <arithmetic expression>

DGAMMA Function

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

The DGAMMA function returns, as a double-precision value, the value of the gamma function at the specified arithmetic expression.

DIMP Function

— DIMP — (—<arithmetic expression>— , —<arithmetic expression>—
 →) —————|

The DIMP function returns the following as a double-precision value:

first <arithmetic expression> IMP second <arithmetic expression>

Both arithmetic expressions are first extended to double-precision, if necessary, and the IMP operation is performed on all 96 bits.

DINTEGER Function

The DINTEGER function has two forms, each of which returns a double-precision value.

The following form of DINTEGER function returns the value of the arithmetic expression as a double-precision value. The sign of the returned value is positive if the value of the arithmetic expression is greater than or equal to zero and is negative otherwise.

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

Specifically, the function returns the following for positive-valued arithmetic expressions:

DOUBLE (ENTIER (<arithmetic expression> + 0.5))

The function returns the following for negative-valued arithmetic expressions:

DNABS (ENTIER (ABS (<arithmetic expression>) + 0.5))

The following form of the DINTEGER function returns the decimal value represented by the string of characters indicated by the pointer expression, as a double-precision integer value. The absolute value of the integer value to be returned must be less than or equal to 3.0223145903657293676543@@23.

The arithmetic expression specifies the length of the string of characters and must have a value, when rounded to an integer, less than 24. If the rounded value of the arithmetic expression is 24 or greater, the program is terminated with a fault. If the length is 0 (zero), a result of 0 is returned.

A zone field configuration of 1"1101" in the least significant character position causes the result of the function to be negative. With 4-bit characters, a 1"1101" in the most significant character position results in a negative value.

When the pointer expression is a hexadecimal pointer or a hexadecimal array, if the character in the most significant character position is not in the range 0 through 9, the character is assumed to be a sign. Therefore, the number of characters processed is one greater than that specified by the arithmetic expression. This character affects the value of the update pointer.

If the string of characters is all 1"11111111" (48"FF") for 8-bit characters or 1"1111" (4"F") for 4-bit characters, the string is treated as though it were a sequence of nines. In the 4-bit character case, the first character is treated as a sign (positive).

Other than the above special cases, if the numeric field of each character in the string of characters contains anything other than a digit the results are undefined and can vary.

The state of the pointer expression when the count is exhausted can be preserved by using the update pointer construct.

— DINTEGER — (—<update pointer>—<pointer expression>— , —>—
→<arithmetic expression>—) —————|

DINTEGERT Function

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

The DINTEGERT function returns the value of the arithmetic expression integerized with truncation to a double-precision integer value.

DLGAMMA Function

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

The DLGAMMA function returns, as a double-precision value, the natural logarithm of the gamma function at the specified arithmetic expression.

DLN Function

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

The DLN function returns, as a double-precision value, the natural logarithm of the specified arithmetic expression.

DLOG Function

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

The DLOG function returns, as a double-precision value, the base-10 logarithm of the specified arithmetic expression.

DMAX Function

— DMAX — (— $\overbrace{\langle \text{arithmetic expression} \rangle}^{\text{ , } \overbrace{\langle \text{arithmetic expression} \rangle}^{\text{ , } \langle \text{arithmetic expression} \rangle}}$ —) —————|

The DMAX function returns, as a double-precision value, the maximum of the values of all the specified arithmetic expressions.

DMIN Function

— DMIN — (— $\overbrace{\langle \text{arithmetic expression} \rangle}^{\text{ , } \overbrace{\langle \text{arithmetic expression} \rangle}^{\text{ , } \langle \text{arithmetic expression} \rangle}}$ —) —————|

The DMIN function returns, as a double-precision value, the minimum of the values of all the specified arithmetic expressions.

DNABS Function

— DNABS — (— $\langle \text{arithmetic expression} \rangle$ —) —————|

The DNABS function returns, as a double-precision value, the negative of the absolute value of the specified arithmetic expression.

DNORMALIZE Function

— DNORMALIZE — (— $\langle \text{arithmetic expression} \rangle$ —) —————|

The DNORMALIZE function returns, as a double-precision value, the normalized form of the specified arithmetic expression. For more information on normalized format, refer to “Double-Precision Operand” in Appendix C, “Data Representation.”

DNOT Function

— DNOT — (— $\langle \text{arithmetic expression} \rangle$ —) —————|

The DNOT function returns, as a double-precision value, the logical complement of the value of the specified arithmetic expression. The arithmetic expression is first extended to double-precision, if necessary, and the NOT operation is performed on all 96 bits.

DOR Function

— DOR — (—<arithmetic expression>— , —<arithmetic expression>—
→) —————|

The DOR function returns the following as a double-precision value:

first <arithmetic expression> OR second <arithmetic expression>

Both arithmetic expressions are first extended to double-precision, if necessary, and the OR operation is performed on all 96 bits.

DOUBLE Function

The DOUBLE function has three forms, each of which returns a double-precision value.

The following form of the DOUBLE function returns the value of the arithmetic expression extended to a double-precision value.

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

The following form of the DOUBLE function returns a double-precision value in which the first word is equal to the value of the first arithmetic expression and the second word is equal to the value of the second arithmetic expression. The arithmetic expressions are first truncated to single-precision, if necessary.

— DOUBLE — (—<arithmetic expression>— , —<arithmetic expression>—
→) —————|

The following form of the DOUBLE function returns, as a double-precision value, the decimal value represented by the string of characters starting with the character pointed to by the pointer expression.

A zone field configuration of 1"1101" in the least significant character position causes the result of the function to be negative. With 4-bit characters, a 1"1101" in the most significant character position results in a negative value.

When the pointer expression is a hexadecimal pointer or a hexadecimal array, if the character in the most significant character position is not in the range 0 through 9, the character is assumed to be a sign. Therefore, the number of characters processed is one greater than that specified by the arithmetic expression. This character affects the value of the update pointer.

If the string of characters is all 1"11111111" (48"FF") for 8-bit characters or 1"1111" (4"F") for 4-bit characters, the string is treated as though it were a sequence of nines. In the 4-bit character case, the first character is treated as a sign (positive).

Other than the above special cases, if the numeric field of each character in the string of characters contains anything other than a digit the results are undefined and can vary.

The state of the pointer expression when the count is exhausted can be preserved by the use of the <update pointer> construct.

```

— DOUBLE — ( —————<pointer expression>— , —————>
               └──<update pointer>──┘
→<arithmetic expression>— ) —————|
    
```

DROP Function

```

— DROP — ( —<string expression>— , —<arithmetic expression>— ) —|
    
```

The DROP function returns a string with a value equal to the value of the string expression with the first arithmetic expression characters deleted. The value of the arithmetic expression is rounded to an integer, if necessary. An error occurs if the rounded value of the arithmetic expression is greater than the number of characters in the string expression or less than 0 (zero). If the rounded value of the arithmetic expression is 0, the result is the same as the value of the string expression. If the rounded value of the arithmetic expression is equal to the length of the string expression, the result is the null string.

The DROP function and the TAKE function are complementary functions. This means that for any string expression S and any arithmetic expression A in the range $0 \leq A \leq \text{LENGTH}(S)$, the following relation is always TRUE:

$$S = \text{TAKE}(S,A) \text{ CAT } \text{DROP}(S,A)$$

For more information, see "TAKE Function" later in this section.

Examples of the DROP Function

In the following examples, string S has a length of 6 and contains 8"ABCDEF".

$$\text{DROP}(S,2) = 8\text{"CDEF"}$$

$$\text{DROP}(\text{TAKE}(S,4),2) = 8\text{"CD"}$$

$$\text{DROP}(S,6) = \text{the null string}$$

DSCALELEFT Function

— DSCALELEFT — (—<arithmetic expression>— , —————→
→<arithmetic expression>—) —————|

The DSCALELEFT function returns the following as a double-precision value. The second arithmetic expression, rounded to an integer, has a value in the range 0 to 12.

first <arithmetic expression>*(10 ** second <arithmetic expression>)

The DSCALELEFT function is undefined when the integerized value of the second arithmetic expression is less than 0 or greater than 12.

DSCALERIGHT Function

— DSCALERIGHT — (—<arithmetic expression>— , —————→
→<arithmetic expression>—) —————|

The DSCALERIGHT function returns, as a double-precision value, the rounded result of the following, where the second arithmetic expression, rounded to an integer, has a value in the range 0 to 12:

first <arithmetic expression>/(10 ** second <arithmetic expression>)

A run-time error occurs if the integerized value of the second arithmetic expression is less than 0 or greater than 12.

DSCALERIGHTT Function

— DSCALERIGHTT — (—<arithmetic expression>— , —————→
→<arithmetic expression>—) —————|

The DSCALERIGHTT function returns, as a double-precision value, the truncated result of the following, where the second arithmetic expression, rounded to an integer, has a value in the range 0 to 12:

first <arithmetic expression>/(10 ** second <arithmetic expression>)

A run-time error occurs if the integerized value of the second arithmetic expression is less than 0 or greater than 12.

DSIN Function

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

The DSIN function returns, as a double-precision value, the sine of an angle of <arithmetic expression> radians.

DSINH Function

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

The DSINH function returns, as a double-precision value, the hyperbolic sine of the specified arithmetic expression.

DSQRT Function

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

The DSQRT function returns, as a double-precision value, the square root of the specified arithmetic expression. The value of the arithmetic expression must be greater than or equal to 0.

DTAN Function

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

The DTAN function returns, as a double-precision value, the tangent of an angle of <arithmetic expression> radians.

DTANH Function

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

The DTANH function returns, as a double-precision value, the hyperbolic tangent of the specified arithmetic expression.

ENTIER Function

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

The ENTIER function returns the largest integer that is not greater than the value of the arithmetic expression.

Because of the limitations of finite representation arithmetic, the ENTIER function erroneously returns 0, rather than -1, for negative numbers of small magnitude. For a single-precision expression, the threshold is $-0.5 * 8^{*-13}$ (approximately $-9.09E-13$). This number correctly yields -1, whereas the next smaller (in magnitude) single-precision number incorrectly yields 0. For a double-precision expression, the threshold can differ on various systems because of different algorithms for double-precision rounding, but it is in the vicinity of -8^{*-26} (approximately $-3.31E-24$).

The ENTIER function returns an incorrect result for single-precision, negative integers less than or equal to $-(8^{*32})$. The result of the function for an integer expression should be equal to the value of the expression. However, ENTIER (-68719476736) returns a value of -68719476737.

The ENTIER function is not a simple truncation function, as illustrated by the examples. For a simple truncation function, see "INTEGERT Function" later in this section.

Examples of the ENTIER Function

ENTIER(2.6) = 2

ENTIER(3.1) = 3

ENTIER(-0.01) = -1

ENTIER(-3.4) = -4

ENTIER(-1.8) = -2

ERF Function

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

The ERF function returns, as a real value, the value of the standard error function at the specified arithmetic expression. For any valid N, $ERF(-N) = -ERF(N)$.

ERFC Function

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

The ERFC function returns, as a real value, the complement of the value of the standard error function at the specified arithmetic expression. For any valid N, $ERFC(N) = 1 - ERF(N)$.

EXP Function

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

The EXP function returns the following as a real value, where e is the base of the natural logarithms:

$e^{**} \text{ <arithmetic expression>}$

FIRST Function

— FIRST — (—<string expression>—) —————|

The FIRST function returns, as a real value, the ordinal position in the EBCDIC, ASCII, or hexadecimal collating sequence of the first character in the string expression. This function returns an ordinal position in the EBCDIC collating sequence if the string expression is EBCDIC, returns an ordinal position in the ASCII collating sequence if the string expression is ASCII, and returns an ordinal position in the hexadecimal collating sequence if the string expression is hexadecimal. A run-time error occurs if the string expression is the null string.

Examples of the FIRST Function

$FIRST("ABC") = 193 (4"C1")$

$FIRST(7"NNXX") = 78 (4"4E")$

$FIRST(4"F1F2") = 15 (4"0F")$

FIRSTONE Function

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

The FIRSTONE function returns, as an integer value, the bit number plus 1 of the leftmost nonzero bit in the value of the arithmetic expression. The FIRSTONE function returns the number 0 if all the bits are 0. If the expression is double-precision, only the first word of its value is examined.

FIRSTWORD Function

— FIRSTWORD — (—<arithmetic expression> — , —<real variable>) —————|

The FIRSTWORD function returns, as a real value, the first word of the double-precision arithmetic expression. The arithmetic expression is first extended to double-precision, if necessary. If the real variable parameter is specified, the second word of the double-precision arithmetic expression is stored in the variable.

FIX Statement

The FIX statement returns a Boolean value. For more information, refer to “FIX Statement” in Section 4, “Statements.”

FREE Statement

The FREE statement returns a Boolean value. For more information, refer to “FREE Statement” in Section 4, “Statements.”

GAMMA Function

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

The GAMMA function returns, as a real value, the value of the gamma function at the specified arithmetic expression.

HAPPENED Function

— HAPPENED — (—<event designator>—) —————|

The HAPPENED function is a Boolean function that returns the value TRUE if the happened state of the specified event is TRUE (happened) and returns the value FALSE if the happened state is FALSE (not happened).

HEAD Function

— HEAD — (—<string expression>— , —<string character set>—) —|

<string character set>

—| —<string constant>—
 └ NOT ┘ └<truth set table>┘

The HEAD function returns a string whose value consists of the leftmost characters of the string expression up to, but not including, the first character that is not a member of the string character set. If the first character of the string expression is not a member of the string character set, the null string is returned. If all characters of the string expression are members of the string character set, the entire string expression is returned.

The string character set must be of the same character type as the string expression. If a truth set table is used, it must not be composed of characters of different character types. The option NOT indicates a string character set made up of all characters that are not specified in the string constant or truth set table, but that are of the same character type as the string expression.

The HEAD function and the TAIL function are complementary functions. This means that for any string expression S and any string character set C, the following relation is always TRUE:

$$S = \text{HEAD}(SC) \text{ CAT } \text{TAIL}(SC)$$

For more information, see "TAIL Function" later in this section.

Examples of the HEAD Function

In the following examples, S is a string of length 9 that contains 8"ABC/1-2+3".

HEAD(S,NOT "/") = 8"ABC"

HEAD(S,ALPHA) = 8"ABC"

HEAD(S,"123") = the null string

IMAG Function

— IMAG — (—<complex expression>—) —————|

The IMAG function returns, as a real value, the imaginary part of the specified complex expression.

For a function that returns the real part of a complex expression, see "REAL Function" later in this section.

INTEGER Function

The INTEGER function has two forms, each of which returns an integer value.

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

This form of the INTEGER function returns the value of the arithmetic expression integerized with rounding. Specifically, it returns the following:

ENTIER(<arithmetic expression> + 0.5)

— INTEGER — (—————|<pointer expression>— , —————|
|<update pointer>|
→<arithmetic expression>—) —————|

This form of the INTEGER function returns, as an integer value, the decimal value represented by the string of characters starting with the character pointed to by the pointer expression. The absolute value of the integer value to be returned must be less than or equal to 549755813887.

The arithmetic expression specifies the length of the string of characters and must, when rounded to an integer, have a value less than 24. If the rounded value of the arithmetic expression is 24 or greater, the program is terminated with a fault. If the length is 0 (zero), a result of 0 is returned.

A zone field configuration of 1"1101" in the least significant character position causes the result of the function to be negative. With 4-bit characters, a 1"1101" in the most significant character position results in a negative value.

When the pointer expression is a hexadecimal pointer, or a hexadecimal array, if the character in the most significant character position is not in the range 0 through 9, the character is assumed to be a sign. Accordingly, the number of characters processed is one greater than that specified by the arithmetic expression. Note that this extra character affects the value of the update pointer.

If the string of characters is 1"11111111" (48"FF") for 8-bit characters or 1"1111" (4"F") for 4-bit characters, the string is treated as though it were a sequence of nines. Note that in the 4-bit character case, the first character is treated as a sign (positive).

Other than the preceding special cases, if the numeric field of each character in the string of characters contains anything other than a digit, the results are undefined and can vary.

The state of the pointer expression when the count is exhausted can be preserved by the use of the <update pointer> construct.

INTEGERT Function

— INTEGERT — (—<arithmetic expression>—)

The INTEGERT function returns the value of the arithmetic expression integerized with truncation.

The INTEGERT function does not necessarily return the same value as the ENTIER function. For example, the function INTEGERT(-1.2) returns the value -1, but the function ENTIER(-1.2) returns the value -2.

ISVALID Function

The ISVALID intrinsic enables you to check whether the following items contain a valid reference:

- Procedure reference variable
- Externally declared procedure (bound-in procedure)
- Imported library procedure
- Imported data identifier
- Imported data array identifier
- Formal procedure parameter
- Structure block reference
- Connection block reference
- Array reference
- Pointer

— ISVALID — (—><procedure identifier>—)

→ <procedure reference array element>—)

<structure block reference identifier>—)

<array reference identifier>—)

<pointer identifier>—)

<data array identifier>—)

<non-array data identifier>—)

, READABLE

READWRITE

The ISVALID function returns a Boolean value indicating whether the procedure identifier or the procedure reference variable contains a proper link to a valid procedure, or whether the structure block reference identifier, connection block reference identifier, array reference identifier, or pointer identifier has been initialized.

The value FALSE is returned if the argument to ISVALID does not contain a proper link to a valid procedure or has not been initialized. The value TRUE is returned if the argument to ISVALID contains a proper link to a valid procedure or has been initialized. The procedure identifier can be a formal parameter in a procedure, a procedure imported from a library, or an externally declared procedure that is to be bound in. Externally declared procedures that are not bound in always return a FALSE result. This function does not check the validity of externally declared procedures that are to be linked with an external code file through a task identifier on the RUN, PROCESS, or CALL statement.

When an array, array reference, pointer, or non-array data identifier is used as the argument to ISVALID, the READABLE or READWRITE options can be specified to determine the access method through which these items are valid. If no option is specified for these types of data items, the default is READABLE.

When the READABLE option is specified, a TRUE result is returned by ISVALID if the item can be read. A FALSE result is returned if the item is uninitialized or is (or refers to) an IMPORTED array or IMPORTED non-array data item that has not been linked.

When the READWRITE option is specified, a TRUE result is returned by ISVALID if the item can be both read from and written to (that is, if the operand can be used on either side of an assignment operation). A FALSE result is returned if the item is uninitialized, is (or refers to) an IMPORTED array or IMPORTED non-array data item that has not been linked, or is (or refers to) an IMPORTED array or IMPORTED non-array data item that has been linked but has been IMPORTED or EXPORTED as READONLY.

LENGTH Function

— LENGTH — (—<string expression>—) _____|

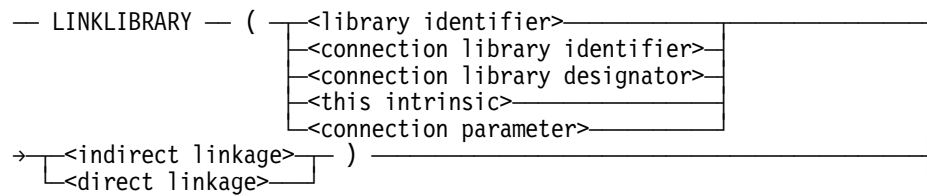
The LENGTH function returns, as an integer value, the number of characters in the string that results from evaluation of the string expression. The null string has a length of 0.

LINENUMBER Function

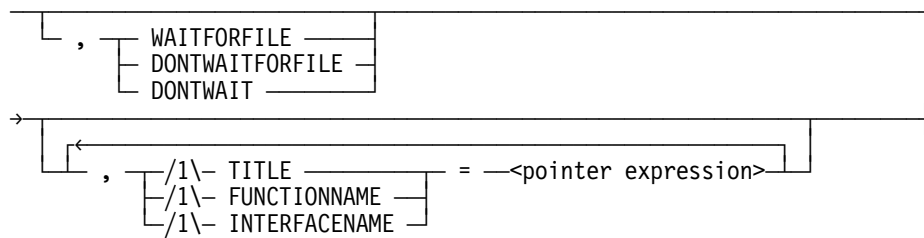
— LINENUMBER _____|

The LINENUMBER function returns, as an integer value, the sequence number of the source file record on which it appears.

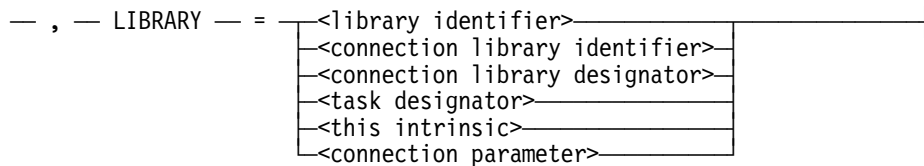
LINKLIBRARY Function



<indirect linkage>



<direct linkage>



The LINKLIBRARY function determines whether or not the program is currently linked to, or is capable of being linked to, the library program specified by the library designator. Results indicating a successful linkage are returned by the LINKLIBRARY function if the program is presently linked to the library or if the program is capable of being linked, in which case the LINKLIBRARY function performs the linkage. If the program cannot be linked to the library, the function returns a result indicating the reason for the failure. Whether or not linkage is achieved, the program continues to execute; that is, the use of the LINKLIBRARY function prevents the termination or suspension of a program upon an unsuccessful attempt to link to a library.

During the linkage process, an attempt is made to link to every entry point exported from the library whose name matches an entry point declared in the program. Only those names that match are checked for correct function type, number of parameters, and parameter types. Therefore, the LINKLIBRARY function does not check that every entry point exported from the library is declared in the program.

Direct Linkage

A direct linkage establishes linkage between two libraries that are specified in the LINKLIBRARY invocation. The linkage is the same as an <indirect linkage>, except that the TITLE, FUNCTIONNAME, INTERFACENAME, and LIBACCESS attribute values are irrelevant. If a task identifier is given in a <direct linkage>, the task must be for a frozen library. If the first parameter is a <library identifier>, then the second parameter must not be a <library identifier>.

The APPROVAL procedure associated with the library of the first parameter is invoked only if that parameter is a <connection library identifier> and only if it identifies a multiple-connection <connection library>. For more information on initiating connection library linkage and on the APPROVAL procedure, see the *Task Management Programming Guide*.

Indirect Linkage

If the WAITFORFILE option is set, it causes the stack to wait on an RSVP message. The WAITFORFILE option is also passed to the APPROVAL procedure of a connection library as the WAIT parameter in case the APPROVAL procedure is waiting before granting approval. If the DONTWAITFORFILE option is set, an error is returned if the library code file is not available. If the DONTWAIT option is specified, the caller is linked to the library only if the library is immediately available, that is, if it is already FROZEN or readied by the READYCL intrinsic. If the library is not immediately available, an error result is returned. Use of the DONTWAIT option never results in the initiation of the codefile containing target library.

The LINKLIBRARY function uses the TITLE, FUNCTIONNAME, or INTNAME value, depending on the LIBACCESS value, to find the process family to link to. If a TITLE is specified, the operating system overrides the LIBACCESS to BYTITLE for this request. Similarly, if a FUNCTIONNAME is specified, the operating system overrides the LIBACCESS to BYFUNCTION for this request. If both TITLE and FUNCTIONNAME are specified, the operating system uses the LIBACCESS attribute value assigned to the library. The LINKLIBRARY function then searches for a connection library with a matching INTERFACENAME that has been marked ready for linkage by the READYCL intrinsic. If a match is found, a linkage to that connection library is attempted. If a matching connection library is not found, and the parent stack of that process family is a server library provider, the LINKLIBRARY function attempts to link to that server library provider.

Value Returned

The values returned by the LINKLIBRARY function can be interpreted as follows:

Value	Meaning
2	Successful linkage was made to the library, but not all entry points were provided.
1	Successful linkage was made to the library, and all entry points were provided.
0	The program was already linked to the specified library at the time of the LINKLIBRARY call.
-1	The library code file is missing if LIBACCESS is BYTITLE, or the function name is not defined if LIBACCESS is BYFUNCTION.
-4	A LIBACCESS specification of BYINITIATOR was used, but the initiator program was not a library.
-5	The number or types of parameters provided by the library do not correspond to those declared in the program, or the declared entry points are not exported by the library.

Value	Meaning
-6	The library was terminated, canceled, or thawed before being frozen; therefore, the library was not successfully initiated.
-7	The linkage operation would have produced a two-way import/export relationship in conjunction with existing library and IPC relationships.
-8	A by-calling procedure never specified a library task.
-10	A library feature that was used is not implemented.
-11	The library template contained an invalid provision type.
-12	The library template level is obsolete. This program must be recompiled.
-13	The library directory level is incompatible with the library template level. The older program must be recompiled.
-15	The library was not initiated for one of the following reasons: <ul style="list-style-type: none">• A task array for a by-calling library was not declared in the library stack.• There was an attempt to link to a file that is not a code file.• The code file is not a library code file.• The code file is restricted.• The file structure is incompatible with the current release.• The call-by-function library was established with the MCPINIT SL attribute and the MCP has not initiated this library.
-18	A linkage was attempted from inside a CHANGE or APPROVAL procedure.
-19	An implicit linkage was attempted, and the attribute AUTOLINK is false for this library.
-20	An APPROVAL procedure approved the linkage but specified a connection index that is less than zero or greater than the CONNECTIONS attribute, or that refers to a connection that is not in the NOTLINKED state.
-21	An APPROVAL procedure returned an invalid control value.
-22	A fault occurred in an APPROVAL procedure.
-23	A fault occurred in a CHANGE procedure.
-24	The corresponding connection library was delinked before it became fully linked.
-25	The client terminated while linking.

Value	Meaning
-26	<p>Incorrect usage of an EXPORTING or IMPORTING connection library.</p> <p>Indicates one of the following circumstances:</p> <ul style="list-style-type: none">• An EXPORTING or IMPORTING connection library had no imports or exports.• An EXPORTING or IMPORTING connection library had both imports and exports. The compiler and the MCP check for this cause.• The importing side of the connection was a server library.
-27	<p>Attempt to connect two EXPORTING or two IMPORTING connection libraries.</p>
-500	<p>The APPROVAL procedure of the corresponding connection library did not approve the linkage, but returned an error that is outside the range of -501 to -1000.</p>
-501 to -1000	<p>The APPROVAL procedure of the corresponding connection library did not approve the linkage and returned the positive of this value as a user-defined error value.</p>

LISTLOOKUP Function

— LISTLOOKUP — (—<arithmetic expression>— , —<array row>— , —>
 →<arithmetic expression>—) _____|

The LISTLOOKUP function causes a linked list of words to be searched and returns, as an integer value, an index into the list as follows:

1. The array row is indexed by the value of the second arithmetic expression and the word is extracted. Each word contains a value (in field [47:28]) and a link (in field [19:20]) to the next word of the linked list.
2. If the value in the extracted word is greater than or equal to the value of the first arithmetic expression, the operation stops, and the index of the word whose link points to the extracted word is returned by the function.
3. If the value in the extracted word is less than the value of the first arithmetic expression, the link of the extracted word is used as an index into the array row, a new word is extracted, and the process is repeated.

A word with a link of 0 terminates the search. The value of a word is tested only if the link field is nonzero. If the linked list is exhausted, that is, (if a word with a link of 0 is encountered), a value of -1 is returned by the LISTLOOKUP function.

LN Function

— LN — (—<arithmetic expression>—) _____|

The LN function returns, as a real value, the natural logarithm of the specified arithmetic expression.

LNGAMMA Function

— LNGAMMA — (—<arithmetic expression>—) _____|

The LNGAMMA function returns, as a real value, the natural logarithm of the gamma function at the specified arithmetic expression.

LOCK Interlock Function

The LOCK Interlock statement returns an integer value. For more information, refer to "LOCK Interlock Statement" in Section 4.

LOCKSTATUS Function

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

The LOCKSTATUS function returns the status of the specified interlock. The LOCKSTATUS function is of the type REAL, and the result has the following subfields:

Subfield	Description
[47:24}	Owner stack number (24 least significant bits) 0 = no owner stack
[23:01]	Reserved
[22:15}	Owner stack number (15 most significant bits)
[07:06]	Reserved
[01:02}	Current lock state: 0 = free 1 = uncontended 2 = contended 3 = broken

The owner portion of the interlock status can be compared to the function PROCESSID.

Examples of the LOCKSTATUS Function

Examples of the LOCKSTATUS function are as follows:

```
R := LOCKSTATUS (MYLOCKS[3]);
```

```
OWNERID := (R := LOCKSTATUS (YOURLOCK)).[47:24] & R.[22:15] [38:15];
```

```
R := LOCKSTATUS (CONN_LIB[7].MYLOCK);
```


LOG Function

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

The LOG function returns, as a real value, the base-10 logarithm of the specified arithmetic expression.

MASKSEARCH Function

— MASKSEARCH — (—<arithmetic expression>— , —————→
→<arithmetic expression>— , —<array row>—————|
 |<subscripted variable>|) —————|

The MASKSEARCH function performs the same operations as the ARRAYSEARCH function.

MAX Function

— MAX — (—<arithmetic expression>— , —————|) —————|

The MAX function returns, as a real value, the maximum of the values of all the specified arithmetic expressions.

MESSAGESEARCHER Statement

The MESSAGESEARCHER statement returns an integer value. For more information, refer to “MESSAGESEARCHER Statement” in Section 4, “Statements.”

MIN Function

— MIN — (—<arithmetic expression>— , —————|) —————|

The MIN function returns, as a real value, the minimum of the values of all the specified arithmetic expressions.

MLSACCEPT Statement

The MLSACCEPT statement returns a Boolean value. For more information, refer to “MLSACCEPT Statement” in Section 4, “Statements.”

MLSDISPLAY Statement

The MLSDISPLAY statement returns a Boolean value. For more information, refer to “MLSDISPLAY Statement” in Section 4, “Statements.”

MLSTRANSULATE Statement

The MLSTRANSULATE statement returns an integer value. For more information, refer to “MLSTRANSULATE Statement” in Section 4, “Statements.”

NABS Function

— NABS — (—<arithmetic expression>—) _____|

The NABS function returns, as a real value, the negative of the absolute value of the specified arithmetic expression.

NORMALIZE Function

— NORMALIZE — (—<arithmetic expression>—) _____|

The NORMALIZE function returns, as a real value, the value of the arithmetic expression, normalized and rounded to a single-precision operand. For more information on the normalized format, refer to “Real Operand” in Appendix C, “Data Representation.”

OFFSET Function

— OFFSET — (—<pointer expression>—) _____|

The OFFSET function returns, as an integer value, the number of characters between the character position referenced by the pointer expression and the beginning of the array row, not including the character designated by the pointer expression. The function value is in terms of the character size of the pointer expression. If it is a word pointer, the value returned is given in terms of 8-bit characters. If the pointer expression points to a paged (segmented) array, the OFFSET function returns the total offset from the beginning of the first segment of the row.

If an uninitialized pointer is passed to the OFFSET function, the program is terminated with a fault. This fault can be trapped with an ON ANYFAULT statement or an ON INVALIDOP statement to prevent termination.

For a function that returns the number of characters between the character position referenced by a pointer expression and the end of the array row, see “REMAININGCHARS Function” later in this section.

ONES Function

— ONES — (—<arithmetic expression>—)

The ONES function returns, as an integer value, the number of nonzero bits in the value of the arithmetic expression. If the arithmetic expression is double-precision, all 96 bits of its value are examined.

OPEN Statement

The OPEN statement returns an integer value. For more information, refer to “OPEN Statement” in Section 4, “Statements.”

POINTER Function

— POINTER — (—<array row>—
 —<subscripted variable>—
 —<string literal>—)
 → , —<character size>—
 —<pointer primary>—)

<character size>

— 0 —
 — 4 —
 — 7 —
 — 8 —

The POINTER function generates a pointer to the specified location.

If the first parameter is an array row, the pointer references the first character of the first word of the specified array row. If the first parameter is a subscripted variable, the pointer references the first character of the array element specified by the subscripted variable. The <string literal> is stored in the constant pool. If the last character of the string literal is not already the null character (4"00"), a null character is appended to it implicitly.

If the second parameter is not given, the character size of the pointer is 8 bits, depending on the default character type. For more information on the default character type, refer to “Default Character Type” in Appendix C, “Data Representation.”

If the second parameter is a character size of 4 or 8, the character size of the pointer being generated is 4 bits or 8 bits, respectively. If the second parameter is a character size of 7, the character size of the pointer is 8 bits. If the second parameter is a character size of 0, the pointer is word-oriented, rather than character-oriented. In addition, the pointer is single-precision if the array it points to is single-precision (INTEGER, REAL, or BOOLEAN), and the pointer is double-precision if the array is double-precision (DOUBLE or COMPLEX).

If a pointer primary is given as the second parameter, the character size of that pointer primary is used for the character size of the pointer being generated.

Examples of the POINTER Function

```
BEGIN
  ARRAY A1,A2[0:9];
  POINTER P1,P2,P3,P4,P5;
  P1 := POINTER(A1);
  P2 := POINTER(A1[9]);
  P3 := POINTER(A1,4);
  P4 := POINTER(A2[6],P3);
END.
```

Execution of this program has the following results:

- P1 This pointer has a character size of 8 bits (the default character type is EBCDIC when P1 is assigned) and points to the first 8-bit character of the first element of array A1.
- P2 This pointer has a character size of 8 bits and points to the first 8-bit character of element 9 of array A1.
- P3 This pointer has a character size of 4 bits and points to the first 4-bit character of the first element of array A1.
- P4 This pointer has a character size of 4 bits and points to the first 4-bit character of element 6 of array A2.

POT Function

$\left. \begin{array}{l} \text{POTL} \\ \text{POTC} \\ \text{POTH} \end{array} \right\} [\text{---<arithmetic expression>---}]$

The POT (Power of Ten) functions-POTL (low), POTC (center), and POTH(high), together provide the following value from three tables that are double-precision, read-only arrays:

`10 ** <arithmetic expression>`

Each of the POT functions is defined only for integer values of the arithmetic expressions that fall in the range 0 through 29604. The complete value of `10 ** <arithmetic expression>` can be computed using the following arithmetic expression:

```
POTL[<arithmetic expression>.[5:6]]
* POTC[<arithmetic expression>.[11:6]]
* POTH[<arithmetic expression>.[14:3]]
```

Examples of the POT Function

```
RESULT := POTL[X]; % WHERE X < 64

DEFINE POT(T) = (POTL[T.[5:6]] * POTC[T.[11:6]] * POTH[T.[14:3]])#;
ONEDIVTENTOI := 1 / POT(I);
```

PROCESSID Function

— PROCESSID —————

The positive integer value uniquely identifies the process executing the function.

The PROCESSID function returns a value that remains unique to that process for the duration of its execution. However, the value is not guaranteed to be the same on every invocation of the PROCESSID function. For example, upon restarting from a checkpoint, the value of the PROCESSID function can have changed. Also, after a process terminates, its PROCESSID value can be assigned to a new process.

RANDOM Function

— RANDOM — (—<arithmetic variable>—) —————|

The RANDOM function returns, as a real value, a random number that is greater than or equal to 0 and less than 1. The arithmetic variable is a call-by-name parameter, and its value is changed each time the function is referenced. A compile-time or run-time error occurs if the parameter is not a single-precision arithmetic variable. An integer overflow error occurs if the integerized value of the arithmetic variable is not in the range $[(-2^{*}39) + 1]$ through $[(2^{*}39) - 1]$.

READ Statement

The READ statement returns a Boolean value. For more information, refer to “READ Statement” in Section 4, “Statements.”

READLOCK Function

Taking only one memory cycle, the READLOCK function stores the value of the specified expression in the designated variable and returns the previous contents of the variable.

The READLOCK function has three forms. One form returns a real value, one form returns a Boolean value, and one form returns a pointer. The following form of the READLOCK function stores the value of the arithmetic expression in the arithmetic variable and returns the previous contents of the variable as a real value. Both the variable and the expression must be single-precision.

— READLOCK — (—<arithmetic expression>— , —<arithmetic variable>→
→) —————|

The following form of the READLOCK function stores the value of the Boolean expression in the Boolean variable and returns the previous contents of the variable as a Boolean value.

— READLOCK — (—<Boolean expression>— , —<Boolean variable>—) —|

The following form of the READLOCK function stores the value of the pointer expression in the pointer variable and returns the previous contents of the variable as a pointer value.

— READLOCK — (—<pointer expression>— , —<pointer variable>—) —|

REAL Function

The REAL function has four forms, each of which returns a real value. The following form of the REAL function returns the value of the arithmetic expression rounded to a single-precision, real value.

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

The following form of the REAL function returns the value of the Boolean expression as a real value. All bits of the Boolean expression are used.

— REAL — (—<Boolean expression>—) —————|

The following form of the REAL function returns, as a real value, the real part of the specified complex expression.

— REAL — (—<complex expression>—) —————|

For a function that returns the imaginary part of a complex expression, see “IMAG Function” earlier in this section. The following form of the REAL function returns, as a real value, the string of <arithmetic expression> characters starting with the character indicated by the pointer expression.

— REAL — (—<pointer expression>— , —<arithmetic expression>—) —|

If the arithmetic expression indicates a string of characters that is less than or equal to 48 bits long, this string is right-justified in one word with leading zeros, if necessary, and this word is returned as the function result. If the arithmetic expression indicates a string of characters more than 48 bits long but less than or equal to 96 bits long, this string is right-justified with leading zeros, if necessary, in a two-word operand. Then only the first word is returned as the function result. If the arithmetic expression indicates a string of characters more than 96 bits long, a run-time error occurs.

REMAININGCHARS Function

— REMAININGCHARS — (—<pointer expression>—) —————|

The REMAININGCHARS function returns, as an integer value, the number of characters between the character position referenced by the pointer expression and the end of the array row, including the character designated by the pointer expression. The returned value is in terms of the character size of the pointer expression. If the pointer expression is a word pointer, the value returned is in terms of 8-bit characters. If the pointer expression points to a paged (segmented) array, the REMAININGCHARS function returns the number of characters left in the entire array row.

If an uninitialized pointer is passed to the REMAININGCHARS function, the program is terminated with a fault. This fault can be trapped with an ON ANYFAULT statement or an ON INVALIDOP statement to prevent termination.

For a function that returns the number of characters between the character position referenced by a pointer expression and the beginning of the array row, see "OFFSET Function" earlier in this section.

REMOVEFILE Statement

The REMOVEFILE statement returns a Boolean value. For more information, refer to "REMOVEFILEStatement" in Section 4, "Statements."

REPEAT Function

— REPEAT — (—<string expression>— , —<arithmetic expression>—)
→) —————|

The REPEAT function returns a string according to the following rules:

- If the arithmetic expression equals 0, the result of the REPEAT function is the null string.
- If the arithmetic expression is greater than 0, the result of the REPEAT function is the same as if <arithmetic expression> occurrences of the value of the string expression were all concatenated together.
- If the arithmetic expression is less than 0, the result of the REPEAT function is either a compile-time or a run-time error.

The value of the arithmetic expression is rounded to an integer, if necessary, before it is used as the repeat count.

Examples of the REPEAT Function

REPEAT("ABC",3) = "ABCABCABC"

REPEAT("WON'T WORK",0) = the null string

SCALELEFT Function

— SCALELEFT — (—<arithmetic expression>— , —————→
→<arithmetic expression>—) —————|

The SCALELEFT function returns the following as an integer value, where the second arithmetic expression, rounded to an integer, has a value in the range 0 to 12:

first <arithmetic expression>*(10 ** second <arithmetic expression>)

The SCALELEFT function is undefined when the integerized value of the second arithmetic expression is less than 0 or greater than 12.

SCALERIGHT Function

— SCALERIGHT — (—<arithmetic expression>— , —————→
→<arithmetic expression>—) —————|

The SCALERIGHT function returns, as an integer value, the rounded result of the following, where the second arithmetic expression, rounded to an integer, has a value in the range 0 to 12:

first <arithmetic expression>/(10 ** second <arithmetic expression>)

A run-time error occurs if the integerized value of the second arithmetic expression is less than 0 or greater than 12.

SCALERIGHTF Function

— SCALERIGHTF — (—<arithmetic expression>— , —————→
→<arithmetic expression>—) —————|

The SCALERIGHTF function returns, as a real value, a left-justified, packed decimal (4-bit decimal) number representing the following, where the second arithmetic expression, rounded to an integer, has a value in the range 0 to 12:

first <arithmetic expression> MOD (10 ** second <arithmetic expression>)

A run-time error occurs if the integerized value of the second arithmetic expression is less than 0 or greater than 12.

The number of significant digits returned by the function is equal to the integerized value of the second arithmetic expression. The external sign flip-flop (EXTF) is set to reflect the sign of the first arithmetic expression for use with the editing phrases specified in a PICTURE declaration.

Examples of the SCALERIGHTF Function

SCALERIGHTF(1234,4) = 4"123400000000"

SCALERIGHTF(12345678,12) = 4"000012345678"

SCALERIGHTT Function

— SCALERIGHTT — (—<arithmetic expression>— , —————→
 →<arithmetic expression>—) —————|

The SCALERIGHTT function returns, as an integer value, the truncated result of the following, where the second arithmetic expression, rounded to an integer, has a value in the range 0 to 12:

first <arithmetic expression>/ (10 ** second <arithmetic expression>)

A run-time error occurs if the integerized value of the second arithmetic expression is less than 0 or greater than 12.

SECONDWORD Function

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

The SECONDWORD function returns, as a real value, the second word of the double-precision arithmetic expression. The arithmetic expression is first extended to double-precision, if necessary.

For a function that returns the first word of a double-precision arithmetic expression, see "FIRSTWORD Function" earlier in this section.

SEEK Statement

The SEEK statement returns a Boolean value. For more information, refer to "SEEK Statement" in Section 4, "Statements."

SETACTUALNAME Function

— SETACTUALNAME — (—<library object identifier>— , —————→
→<pointer expression>—) —————|

<library object identifier>

—<library entry point identifier>—|
—<imported data identifier>—|

<library entry point identifier>

A procedure identifier declared with a library entry point specification.

<imported data identifier>

A <non-array data identifier> or <array identifier> declared within an <imported declaration>.

The SETACTUALNAME function determines whether or not the ACTUALNAME of the library object specified by the library object identifier can be changed to the name specified by the pointer expression. The function then makes the change, if it is possible. The results of a successful change are returned upon completing the ACTUALNAME change; otherwise, results indicating the reason for failure are returned.

The ACTUALNAME of an object of a linked library cannot be modified. Therefore, a linked library must be delinked before the SETACTUALNAME function can be called to change the ACTUALNAME of any of its objects. The function can be called to modify an object of a library that has not yet been linked.

The ACTUALNAME of an object in a connection library can be modified only if that object is the only object that is imported into the library. Setting the ACTUALNAME of an object in a connection library affects all connections. If a connection is linked when SETACTUALNAME is called, the connection is not affected until it is delinked. This functionality provides compatibility with server libraries. It is recommended that dynamic linkage redirections be performed by using the INTERFACENAME library attribute.

Starting with the first character pointed to by the pointer expression, characters are included as the new object name until a period is encountered, the maximum allowable number of characters is included, or the end of the array row is encountered. The last case results in an error condition.

The SETACTUALNAME function returns the following integer values:

Value	Meaning
1	A successful change was made to the ACTUALNAME of the object.
0	The new ACTUALNAME is the same as the current ACTUALNAME of the object.
-1	The library is linked. The library must be delinked before the SETACTUALNAME function is called.
-2	The library object identifier is not currently in an accessible library template.
-3	A parameter error occurred in the SETACTUALNAME function.
-4	The new ACTUALNAME is already in use by another object.
-5	The object is located in a connection library that imports multiple objects.

Example of the SETACTUALNAME Function

The following example changes the ACTUALNAME of the library entry point EP2 to "ENTRYPOINT2", if possible:

```
REPLACE NEWEPNAME BY "ENTRYPOINT2.";
SETACTUALNAME(EP2,NEWEPNAME);
```

SIGN Function

— SIGN — (—<arithmetic expression>—) _____|

The SIGN function returns the integer 1 if the value of the arithmetic expression is greater than 0, returns the integer 0 if the value of the arithmetic expression is equal to 0, and returns the integer -1 if the value of the arithmetic expression is less than 0.

SIN Function

— SIN — (—<arithmetic expression>—) _____|

The SIN function returns, as a real value, the sine of an angle of <arithmetic expression> radians.

SINGLE Function

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

The SINGLE function returns, as a real value, the value of the arithmetic expression normalized and truncated to single-precision.

SINH Function

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

The SINH function returns, as a real value, the hyperbolic sine of the specified arithmetic expression.

SIZE Function

— SIZE — (—<array designator>—
 |<event array identifier>—
 |<interlock array identifier>—
 |<pointer identifier>—
 |<procedure reference array designator>—
 |<structure block array designator>—) —————|

The *SIZE(<array designator>)*, the *SIZE(<event array identifier>)*, the *SIZE (<interlock array identifier>)*, the *SIZE(<procedure reference array designator>)*, and the *SIZE (<structure block array designator>)* forms of the SIZE function return, as an integer value, the size of one dimension of the specified array expressed in elements. The element size of a procedure reference array designator is always some number of words. If the array designator is an array name or the procedure reference array designator is a procedure reference array identifier, the SIZE function returns the size of the first dimension of the specified array. If the array designator or the procedure reference array designator contains a subarray selector, the SIZE function returns the size of the dimension that corresponds to the first asterisk (*) subscript.

The *SIZE(<pointer identifier>)* form of the SIZE function returns, as an integer value, the character size of the specified pointer. If the character size of the pointer is 4 or 8 bits, the value returned is 4 or 8, respectively. If the pointer is word-oriented, the value returned is 0 or 2, depending on whether the pointer is single-precision or double-precision, respectively. If the pointer is uninitialized, the SIZE function returns 0. See "POINTER Function" earlier in this section for a discussion of character size.

Note: *The SIZE function cannot be used for task arrays.*

SPACE Statement

The SPACE statement returns a Boolean value. For more information, refer to "SPACE Statement" in Section 4, "Statements."

SQRT Function

— SQRT — (—<arithmetic expression>—) _____|

The SQRT function returns, as a real value, the square root of the arithmetic expression. The value of the arithmetic expression must be greater than or equal to 0.

STRING Function

The STRING function returns a string value. The function STRING4 returns a hexadecimal string, the function STRING7 returns an ASCII string, and the function STRING8 returns an EBCDIC string. The function STRING returns a string of the default character type. For more information on the default character type, see "Default Character Type" in Appendix C, "Data Representation."

If the first parameter to the STRING function is a pointer expression, the STRING function converts the string of characters designated by the pointer expression into a string. The number of characters converted is indicated by the value of the arithmetic expression rounded to an integer, if necessary. If this rounded value is less than 0, a compile-time or run-time error occurs. If the rounded value is 0, the null string is returned.

When the first parameter is a pointer expression, the pointer expression and the character type of the STRING function differs from that of the pointer expression. The character referenced by the pointer expression must be the beginning of a valid character when the character type is changed to that of the STRING function. For example, if a hex pointer is being converted to an EBCDIC string, the 4-bit pointer must be referencing an even-numbered hex character within the word.

If the first parameter to the STRING function is an arithmetic expression, the STRING function returns a string consisting of a decimal representation of the value of that arithmetic expression. If the second parameter is also an arithmetic expression, then the value of this expression, rounded to an integer, specifies the length of the resulting string. If the value of the second arithmetic expression is less than 0, a compile-time or run-time error occurs. If this value is equal to 0, the null string is returned. If the second parameter is an asterisk (*), the resulting string is exactly long enough to represent the value of the arithmetic expression with no blanks. If the value of the first arithmetic expression is 0 and the second parameter is an asterisk, the resulting string is one character long.

When the STRING function is to return an ASCII or EBCDIC string and the first parameter is an arithmetic expression, its value is converted into the most efficient form, depending on the length specified by the second parameter. If both parameters are arithmetic expressions and the rounded value of the second parameter is greater than the minimum number of characters needed to represent the first parameter, leading blanks are inserted in the resulting string. If both parameters are arithmetic expressions and the rounded value of the second parameter is less than the number of characters needed to represent the first parameter, then a string of all asterisks is returned.

For the function STRING4, when the first parameter is an arithmetic expression, only the integer portion of the value of this expression is converted. If both parameters to STRING4 are arithmetic expressions and the rounded value N of the second parameter is less than the number of characters needed to represent the first parameter, then the rightmost N digits of the converted value of the first parameter are returned. If both parameters to STRING4 are arithmetic expressions and the rounded value of the second parameter is greater than the number of characters needed to represent the converted value of the first parameter, then leading zeros are inserted into the resulting string.

Examples of the STRING Function

STRING(P,20)

STRING(POINTER(A),N-3)

STRING(256,*) = 8"256"

STRING(-335.25,8) = 8" -335.25"

STRING(4.78@-2,5) = 8".0478"

STRING(555000,1) = 8"*"

STRING(456.789,7) = 8"456.789"

STRING4(123.456,8) = 4"00000123"

STRING4(12345678,4) = 4"5678"

TAIL Function

— TAIL — (—<string expression>— , —<string character set>—) —|

The TAIL function returns a string whose value consists of the rightmost characters of the string expression beginning with the first character that is not a member of the string character set. If all characters in the string expression are members of the string character set, the null string is returned. If the first character of the string expression is not a member of the string character set, the entire string expression is returned.

The string character set must be of the same character type as the string expression. For an explanation of the string character set, see "HEAD Function" earlier in this section.

The TAIL function and the HEAD function are complementary functions. This means that for any string expression S and any string character set C, the following relation is always TRUE:

$$S = \text{HEAD}(S,C) \text{ CAT } \text{TAIL}(S,C)$$

For additional information, see "HEAD Function" earlier in this section.

Examples of the TAIL Function

In the following examples, S is a string of length 9 that contains 8"ABC/1-2+3".

$$\text{TAIL}(S, \text{NOT } "-") = 8"-2+3"$$

$$\text{TAIL}(\text{DROP}(S,7), "+-") = 8"3"$$

TAKE Function

— TAKE — (—<string expression>— , —<arithmetic expression>—) —|

The TAKE function returns a string whose value is equal to the first <arithmetic expression> characters taken from the value of the string expression. The value of the arithmetic expression is rounded to an integer, if necessary. An error occurs if the rounded value of the arithmetic expression is greater than the number of characters in the string expression or less than 0(zero). If the rounded value of the arithmetic expression is 0, the result is the null string. If the rounded value of the arithmetic expression is equal to the length of the string expression, the result is the same as the value of the string expression.

The TAKE function and the DROP function are complementary functions. This means that for any string expression S and any arithmetic expression A in the range $0 \leq A \leq \text{LENGTH}(S)$, the following relation is always true:

$$S = \text{TAKE}(S,A) \text{ CAT } \text{DROP}(S,A)$$

For additional information, see "DROP Function" earlier in this section.

Examples of TAKE Function

In the following examples, string S has a length of 6 and contains 8"ABCDEF".

TAKE(S,2) = 8"AB"

TAKE(S,4) = 8"ABCD"

TAKE(DROP(S,2),2) = 8"CD"

TAN Function

— TAN — (—<arithmetic expression>—) _____|

The TAN function returns, as a real value, the tangent of an angle of <arithmetic expression> radians.

TANH Function

— TANH — (—<arithmetic expression>—) _____|

The TANH function returns, as a real value, the hyperbolic tangent of the specified arithmetic expression.

THIS Function

— THIS _____|
┌ (—<structure block type identifier>—) ─┐
└ (—<connection block type identifier>—) ─┘

The THIS intrinsic provides a self reference to the current structure or connection block when no type argument is provided, or to the structure or connection block that is of type <structure block type identifier> or <connection block type identifier>. The THIS function can only be invoked within a procedure that is embedded in the declaration of the specified structure or connection block type identifier.

While in an embedded structure or connection block, the THIS intrinsic can provide a reference to the outer structure block by using the outer structure or connection block type identifier as the argument to the THIS intrinsic.

The THIS intrinsic can be used in a subprogram only if \$ MAKEHOST is set in the host or the intrinsic is referenced in the host. Otherwise, a run-time fault occurs.

THISCL Function

— THISCL — (—<connection block type identifier>—) —

The THISCL function can be used to get a reference to the connection library in which the current procedure is being executed. It can be used in library attribute queries and in assignments.

If the THISCL function is used with a connection attribute, such as STATE, the current connection is assumed.

The THISCL function is allowed only inside a CONNECTION BLOCK TYPE declaration.

TIME Function

— TIME — (—<arithmetic expression>—) —

The TIME function makes various system time values available. The value of the arithmetic expression is rounded to an integer, if necessary, before being used. The results returned for different values of the integerized arithmetic expression are given in the following table. If the integerized value of the arithmetic expression is not one of the values listed in the table, the TIME function returns the value 0 (zero).

Parameter	Result Returned
1	The time of day, in 60ths of a second, as a rounded integer value.
2	The elapsed processor time of the program, in 60ths of a second, as a rounded integer value.
3	The elapsed I/O time of the program, in 60ths of a second, as a rounded integer value.
4	The value of a 6-bit clock that increments 60 times per second.

Parameter	Result Returned
6, 106	<p>The time and date (a timestamp) in the following form:</p> <p>0 & (JULIANDATE-70000) [47:16] & (TIME(11) DIV 16) [31:32]</p> <p>This Julian date is in YYYYDDD format where the value changes from 099365 to 100001 at midnight on December 31, 1999.</p> <p>The only difference between the parameters 6 and 106 is that 106 does not display warning 113 (year 2000 changes) when the system option SYSOPS DATEWARN has the value SET.</p> <p>Note: <i>The timestamp encoding is not a simple real numeric value; therefore, arithmetic relational operators cannot be used to compare timestamps accurately. For simple tests of equality or inequality, the IS or ISNT operators are appropriate. For testing relations, the timestamp values must be treated as unsigned 48-bit integers.</i></p> <p><i>The following Boolean expression correctly computes the truth of the proposition that T1 is less (earlier) than T2, assuming that T1 and T2 are real variables containing timestamp values:</i></p> <p style="text-align: center;">DOUBLE(1,T1) < DOUBLE(1,T2)</p>
7	<p>The current date and time in the following form:</p> <p>[47:12] Year (1900-2035)</p> <p>[35:06] Month (1-12)</p> <p>[29:06] Day (1-31)</p> <p>[23:06] Hour (0-23)</p> <p>[17:06] Minute (0-59)</p> <p>[11:06] Second (0-59)</p> <p>[05:06] Day of the week (0 = Sunday, 1 = Monday, and so on)</p>
8	<p>The elapsed Readyq time for the program, in increments of 1/60 seconds, as a rounded integer value.</p>
9	<p>The current time in EBCDIC characters in the format 8"HHMMSS," where HH is the hour, MM is the minute, and SS is the second.</p>
10	<p>The current date in EBCDIC characters in the Julian date format 8"YYDDD," where YY represents the year and DDD is the day of the year. Values of YY from 36 through 99, inclusive, represent years in the 20th century (1936 through 1999 inclusive). Values of YY less than 36 represent years in the 21st century (2000 through 2035, inclusive).</p>
11	<p>The same value as TIME(1), except that the time is expressed in multiples of 2.4 microseconds.</p>
12	<p>The same value as TIME(2), except that the time is expressed in multiples of 2.4 microseconds.</p>

Parameter	Result Returned
13	The same value as TIME(3), except that the time is expressed in multiples of 2.4 microseconds.
14	The time elapsed since the last halt/load expressed in multiples of 2.4 microseconds. Should be used as a type REAL expression.
15, 115	<p>The current date in EBCDIC characters in the format 8"MMDDYY," where YY represents the last two digits of the year. Values for YY in the range from 36 through 99, inclusive, represent years in the 20th century (1936 through 1999, inclusive). Values for YY less than 36 represent years in the 21st century (2000 through 2035, inclusive).</p> <p>The only difference between the parameters 15 and 115 is that 115 does not display warning 113 (year 2000 changes) when the system option SYSOPS DATEWARN has the value SET.</p>
16, 116	<p>The same value as TIME(6).</p> <p>The only difference between the parameters 16 and 116 is that 116 does not display warning 113 (year 2000 changes) when the system option SYSOPS DATEWARN has the value SET.</p>
18	The same value as TIME(8), except that the time is expressed in multiples of 2.4 microseconds.
19	<p>This parameter enables a program to check whether advanced features such as NX/Services are supported by the current hardware platform. The value returned does not determine whether the features are installed.</p> <ul style="list-style-type: none"> • Bit 1 = 0 The value in bit 0 is not meaningful on this MCP. • Bit 1 = 1 The data in bit 0 is valid. • Bit 0 = 1 This value is returned if the system supports HMP features such as NX/Services. • Bit 0 = 0 This value is returned if the system does not support HMP features such as NX/Services.
23	<p>The system identification information in the following format:</p> <pre style="margin-left: 40px;">[31:08] E-mode level 1 - Level Alpha 2 - Level Beta 3 - Level Gamma 4 - Level Delta</pre> <pre style="margin-left: 40px;">[23:16] System serial number</pre>
24	The system type as a real value containing six EBCDIC characters. The name is left-justified with blank fill.

Parameter	Result Returned
25	The value returned is the actual hardware TARGET level of the system. A value of 4 means TARGET=LEVEL 4.
26	The microcode version in the following format: [47:08] System dependent miscellaneous information [39:08] Microcode mark level [31:16] Microcode cycle number [15:16] Microcode creation date (Julian date-70000)
27	The number of initial presence bits taken on a stack.
28	The amount of time taken to process the initial presence bits.
29	The number of other presence bits taken on a stack.
30	The amount of time taken to process the other presence bits.
31	The SSR level of the MCP currently running on the system is returned in the following format: [47:16] The first 2 digits of the SSR number [31:16] A 3-digit cycle number whose high order digit is the last digit of the SSR number [15:16] A 4-digit patch number For example, SSR 48.1, TIME (31) returns a value of 44 in field [47:16] and a value of 2nn in field [31:16]. The element 2nn indicates the cycle displayed when the WM (What MCP) system command is used.

Parameter	Result Returned
32	<p>Offset from Universal Time (UT), in the form of direction, hours, minutes, and the one-character military-style mnemonic, if the offset matches one, encoded in partial word fields.</p> <p>The format is as follows:</p> <pre> TZ_VALIDF = [47:01] (1 if time zone valid) TZ_DIRECTIONF = [46:01] (which direction?) TZ_SUBV = 0 (UT offset = LT) TZ_ADDV = 1 (UT + offset = LT) (not used) = [45:02] (not used) TZ_CUSTOMF = [43:01] (1 if custom time zone) (not used) = [42:03] (not used) TZ_HOURSF = [39:08] (hours offset 0-24) TZ_MINUTESF = [31:08] (minutes offset 0-59) TZ_MNEMONICF = [23:08] (military mnemonic) (not used) = [15:04] (not used) TZ_NUMBERF = [11:12] (if predefined, number) </pre> <p>If a predefined time zone is configured, its number is also returned. This allows programs to use the exported procedure MCPMESSAGESEARCHER to obtain the time zone abbreviation or name. The message number of the who field (WHOF) is 80. The which field (WHICHF) is 1 for the time zone name, 2 for the time zone abbreviation. The message requires one parameter, the time zone number.</p> <p>The military mnemonic is a single letter that identifies 25 time zones with integral hours offset values from -12 to +12 hours, including zero hours. The letters used are A through M for -1 through -12. The letter J is not used. The letters N through Y are used for +1 through +12. The letter Z is used for zero</p>
33	<p>Offset from UT as a four-digit signed integer for use by programs written in languages in which it is difficult to manipulate bits. The format is +/- HHMM. Note that if no time zone is configured, the value 9999 is returned.</p>
34	<p>Time zone abbreviation in left-justified blank-filled EBCDIC characters. If no time zone is configured, six nulls are returned. If a standard time zone is configured its abbreviation is returned in the task or system language.</p>

Parameter	Result Returned
36, 136	Halt/load time in TIME(6) format. The only difference between the parameters 36 and 136 is that 136 will not display warning 113 (year 2000 changes) when the system option SYSOPS DATEWARN has the value SET.
41	The current time of day, in 60ths of a second, an integer value. Times before noon are counted from midnight and are returned as negative numbers. Times after noon are returned as 12:00 through 12:59 and 1:00 through 11:59.
42	The current time of day, returned in universal time if the system time zone has been set. Note: For TIME(50) through TIME(59), when a timezone is not configured, the times return the same information as the time values on which they are based.
50	The same value as TIME(10), adjusted for Universal Time.
51	The same value as TIME(1), adjusted for Universal Time.
52	The same value as TIME(11), adjusted for Universal Time.
55, 155	The same value as TIME(15), adjusted for Universal Time. The only difference between the parameters 55 and 155 is that 155 will not display warning 113 (year 2000 changes) when the system option SYSOPS DATEWARN has the value SET.
56, 156	The same value as TIME(6), adjusted for Universal Time. The only difference between the parameters 56 and 156 is that 156 will not display warning 113 (year 2000 changes) when the system option SYSOPS DATEWARN has the value SET.
57	The same value as TIME(7), adjusted for Universal Time.
59	The same value as TIME(9), adjusted for Universal Time.

Parameter	Result Returned
60	<p>The time, date, and time zone in the following format:</p> <p>[47:01] Not used</p> <p>[46:07] Time zone number or zero</p> <p>[39:16] Julian date (JULIANDATE-70000)</p> <p>This Julian date is in YYYYDDD format where the value changes from 099365 to 100001 at midnight on December 31, 1999.</p> <p>[23:24] Time of day in hundredths (.01) of a second</p> <p>To retrieve the year, use the formula:</p> <p><code>YEAR:= (TIME (60).[39:16]) DIV 1000 + 1970;</code></p> <p>To retrieve the day (range 1 through 366), use the formula:</p> <p><code>DAY := TIME (60).[39:16] MOD 1000;</code></p>
61	<p>The current time of day. Represented in the same way as in TIME(41), but the value is in units of 2.4 microseconds.</p>
62	<p>The current time of day, returned in universal time if the system time zone has been set, and the value is in units of 2.4 microseconds.</p>
110	<p>The current date in EBCDIC characters in the Julian date format of 8"YYYYDDD." This value changes from 099365 to 100001 at midnight on December 31, 1999.</p>
150	<p>The Universal date in EBCDIC characters in the Julian date format of 8"YYYYDDD." This value changes from 099365 to 100001 at midnight on December 31, 1999.</p>

TRANSLATE Function

— TRANSLATE — (—<string expression>— , —<translate table>—) —|

The TRANSLATE function returns a string of the same length as the string expression; each character of the string expression is translated according to the designated translate table.

The translate table can be a translate table identifier declared in the program or one of the intrinsic translate tables. The use of a subscripted variable as a translate table is not allowed in the TRANSLATE function.

Examples of the TRANSLATE Function

```
TRANSLATE(S,HEXTOEBCDIC)
```

```
TRANSLATE(TAKE(S,10),MYTT)
```

UNLOCK Statement

The UNLOCK statement returns an integer value. For more information, refer to “UNLOCK Statement” in Section 4.

UNREADYCL Function

The UNREADYCL function enables you to disallow linkages to a connection library until the READYCL is called again.

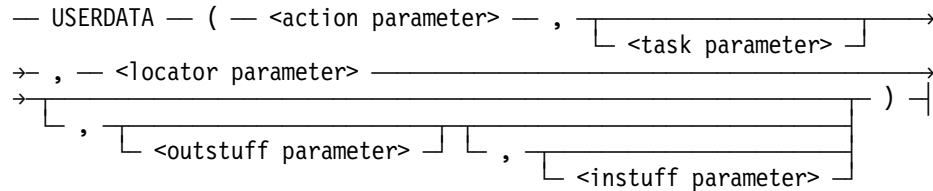
— UNREADYCL — (—<connection library identifier>—) —————|
 └<thiscl intrinsic>┘

The values returned by the UNREADYCL function can be interpreted as follows:

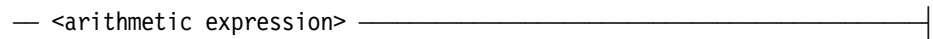
Value	Meaning
0	No error.
1	The LIB attribute is not a library. This is a compiler error.
3	The connection library was not previously readied in a READYCL statement.

USERDATA Function

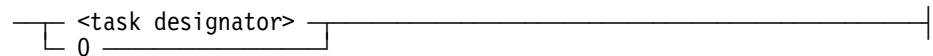
The USERDATA function provides a programmatic interface to the USERDATA operating system procedure. A description of the parameters, results, and error codes of the function is found in the discussion of MAKEUSER in the *Security Administration Guide*.



<action parameter>



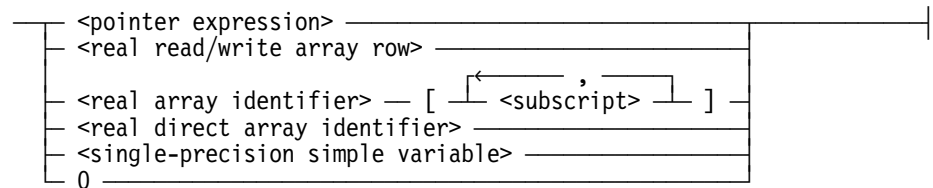
<task parameter>



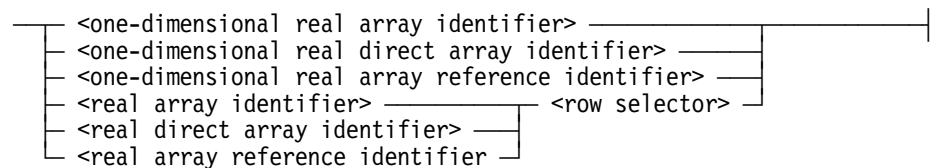
<locator parameter>



<outstuff parameter>



<real read/write array row>



<real direct array identifier>

A direct array identifier that is declared of type REAL.

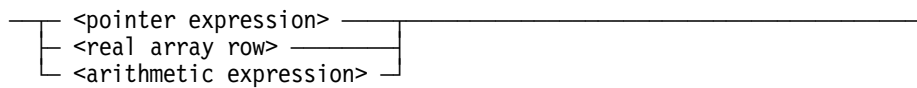
<one-dimensional real direct array identifier>

A direct array identifier that is declared of type REAL and with one dimension.

<single-precision simple variable>

A simple variable whose identifier is declared of type INTEGER or REAL.

<instuff parameter>

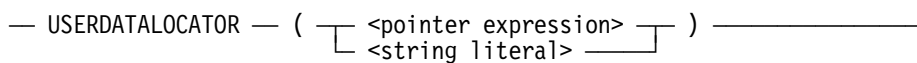


<real array row>

An array row whose identifier is declared of type REAL.

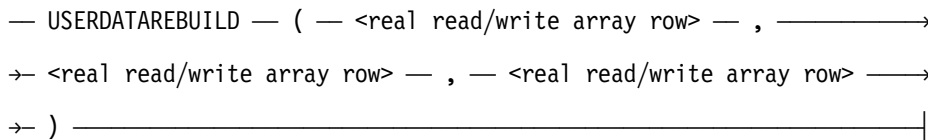
USERDATALOCATOR Function

The USERDATALOCATOR function returns a locator word by which the corresponding portion of the user entry can be addressed. These locators are interpreted in turn by the USERDATA and USERDATAREBUILD operating system procedures and by the SYSTEM/MAKEUSER program. A description of the parameter, results, and error codes of the function is found in the discussion of MAKEUSER in the *Security Administration Guide*.



USERDATAREBUILD Function

The USERDATAREBUILD function constructs or modifies a USERFILE entry. This function is used by the SYSTEM/MAKEUSER program. The operating system procedure USERDATAREBUILD performs the same task as the function. The parameters, results, and error codes of the function are described in the discussion of MAKEUSER in the *Security Administration Guide*.



VALUE Function

```

— VALUE — ( —<mnemonic file attribute value>————— ) —|
              |—————|
              |<arithmetic-valued file attribute name>|
              |<Boolean-valued file attribute name>|
              |<pointer-valued file attribute name>|
              |<translate-table-valued file attribute name>|
              |<mnemonic task attribute value>|
              |<arithmetic-valued task attribute name>|
              |<Boolean-valued task attribute name>|
              |<event-valued task attribute name>|
              |<file-valued task attribute name>|
              |<pointer-valued task attribute name>|
              |<task-valued task attribute name>|
              |<mnemonic library attribute value>|
  
```

<mnemonic task attribute value>

A valid mnemonic name for a value of a task attribute.

If a mnemonic file attribute value, mnemonic library attribute value, or mnemonic task attribute value is specified, the VALUE function returns the integer value that corresponds to that mnemonic attribute value. If an attribute name is specified, the VALUE function returns the attribute number that corresponds to that attribute name.

For more information on file attribute names and mnemonic file attribute values, refer to the *File Attributes Programming Reference Manual*. For more information on task attributes and their values, refer to the *Task Attributes Programming Reference Manual*.

Examples of the VALUE Function

```
F.KIND := VALUE(DISK)
```

```
F.INTMODE := VALUE(EBCDIC)
```

```
F.STATUS := VALUE(TERMINATED)
```

WAIT Statement

Depending on the form used, the WAIT statement returns no value, a Boolean value, or an integer value. For more information, refer to “WAIT Statement” in Section 4, “Statements.”

WAITANDRESET Statement

The WAITANDRESET statement returns an integer value. For more information, refer to “WAITANDRESET Statement” in Section 4, “Statements.”

WRITE Statement

The WRITE statement returns a Boolean value. For more information, refer to “WRITE Statement” in Section 4, “Statements.”

Section 6

Compiling Programs

This section presents information outside of the ALGOL language. This information is necessary to compile and run an ALGOL program. The section describes the files used by the ALGOL compiler when it compiles a program, the format of a source record, and compiler control options.

Files Used by the Compiler

When the ALGOL compiler compiles a program, it requires, at minimum, one input file that contains the source code to be compiled. If the compile is successful, the compiler produces, at minimum, one output file that contains executable object code.

Through the use of compiler control options, the compiler can be directed to use additional input files and to produce additional output files. Among the optional input files that can be used are files containing source code that is to be merged or inserted into the required source code file, and files containing information needed to perform separate compilation and binding. Among the optional output files that can be produced are a printer listing of the program, a cross-reference file, an updated source file, an error message file, and a file containing information needed for future separate compilation.

Table 6–1 lists the logical input and output files used by the ALGOL compiler. Each file is listed with values for the INTNAME, KIND, INTMODE, MAXRECSIZE, BLOCKSIZE, and FILETYPE attributes. Some or all of the attributes for these files can be changed, using compiler file equation, when the compiler is initiated. For information on file attributes refer to the File Attributes Programming Reference Manual. For information about compiler file equation, see the Work Flow Language (WFL) Programming Reference Manual.

Table 6-1. Compiler Input and Output Files

COMPILER INPUT FILES							
Description	INTNAME	Initiation	KIND	INTMODE	MAXRECSIZE	BLOCKSIZE	DEPENDENT SPECS
PRIMARY INPUT FILE	CARD	WFL	READER	EBCDIC	Taken from physical file	Taken from physical file	TRUE
		CANDE	DISK				
OPTIONAL SECONDARY INPUT FILE	SOURCE*	WFL AND CANDE	DISK	EBCDIC	Taken from physical file	Taken from physical file	TRUE
OPTIONAL SOURCE FILES INPUT BY \$ INCLUDE COMPILER CONTROL OPTIONS	(INCLUDEd files)	WFL AND CANDE	DISK	EBCDIC	Taken from physical file	Taken from physical file	TRUE
OPTIONAL OBJECT FILE INPUT	HOST	WFL AND CANDE	DISK	SINGLE	30 words	270 words	TRUE
OPTIONAL INFO FILE	INFO	WFL AND CANDE	DISK	SINGLE	Taken from physical file	Taken from physical file	TRUE

COMPILER OUTPUT FILES							
Description	INTNAME	Initiation	KIND	INTMODE	MAXRECSIZE	BLOCKSIZE	FILETYPE
OBJECT CODE FILE	CODE	WFL AND CANDE	DISK	SINGLE	30 words	270 words	——
OPTIONAL UPDATED SYMBOLIC FILE	NEW-SOURCE*	WFL AND CANDE	DISK	EBCDIC	15 words	420 words	——
OPTIONAL LINE PRINTER LISTING	LINE	WFL AND CANDE	PRINTER	EBCDIC	22 words	22 words	——
OPTIONAL ERROR MESSAGE FILE	ERRORS*	WFL	PRINTER	EBCDIC	12 words	12 words	
		CANDE	REMOTE				
OPTIONAL CROSS-REFERENCE FILE	XREFFILE	WFL AND CANDE	DISK	EBCDIC	510 words	510 words	0
OPTIONAL INFO FILE	INFO	WFL AND CANDE	DISK	SINGLE	256 words	2560 words	0

* The following synonyms can be used:

File Name	Synonym
SOURCE	TAPE
NEWSOURCE	NEWTAPE
ERRORS	ERRORFILE

Input Files

The input files used by the compiler consist of the following:

- CARD, the required source code file
- SOURCE or TAPE, a source code file that can be merged with CARD
- INCLUDE files, source code files that can be inserted into CARD or TAPE
- HOST, information used for separate compilation and binding
- INFO, information used for separate compilation

The EXTMODE attribute (the character type of the physical file) of these input files can be EBCDIC or ASCII. The MAXRECSIZE attribute of these input files must be large enough to accommodate at least 72 characters. Because the values of the MAXRECSIZE and BLOCKSIZE attributes for these files are taken from the physical file (since DEPENDENTSPECS = TRUE), these two attributes do not require explicit assignment.

The DEPENDENTSPECS file attribute is set to TRUE for all compiler input files. To set file attributes that are different from the present file (for example, BLOCKSIZE), you must set DEPENDENTSPECS to FALSE.

CARD File

The CARD file supplies the primary source input to the compiler and must be present for each compilation. If the compiler is initiated from the Work Flow Language (WFL) and compiler file equation is not applied to the CARD file, the file is assumed to be a card reader file. If the compiler is initiated through CANDE and compiler file equation is not applied to the CARD file, the file is assumed to be a disk file.

SOURCE File

The synonym TAPE can be used in place of SOURCE.

The SOURCE file supplies secondary source input to the compiler. Its presence is optional. If you do not file-equate either SOURCE or TAPE, the compiler attempts to open a file with the name TAPE, and if it does not find one, it attempts to open a file named SOURCE. The file is assumed to be a disk file regardless of whether the compiler is initiated from WFL or CANDE.

When this file is present and the MERGE option is TRUE, records from the TAPE file are merged with those of the CARD file on the basis of sequence numbers. If a record from the CARD file and a record from the TAPE file have the same sequence number, the record from the CARD file is compiled and the TAPE record is ignored. For more information on the MERGE option, see "MERGE Option" later in this section.

INCLUDE Files

INCLUDE files provide source input to the compiler in addition to that supplied by the CARD and SOURCE (or TAPE) files. An INCLUDE file is used only if an INCLUDE compiler control option appears in the source input being compiled. For more information on the INCLUDE option, see "INCLUDE Option" later in this section.

HOST File

The HOST file provides the compiler with information that allows it to separately compile and bind to a host program only procedures that are being changed. This process is called a SEPCOMP and the HOST file is used if the SEPCOMP compiler control option is TRUE. The HOST file is a special object code file, created by a previous compile when the MAKEHOST compiler control option was TRUE. The HOST file contains information about the outer block environment of the program and the environments of selected procedures. For more information on the SEPCOMP and MAKEHOST options, see "SEPCOMP Option" and "MAKEHOST Option" later in this section.

INFO File

The INFO file contains the contents of variables and tables used in the compiler, saved from a previous compile. This file is used for separate compilation of procedures. It is created by the DUMPINFO compiler control option and is used as an input file by the LOADINFO compiler control option. For more information on the DUMPINFO and LOADINFO options, see "DUMPINFO Option" and "LOADINFO Option" later in this section.

Output Files

The output files produced by the compiler consist of the following:

- CODE: the object code file
- NEWSOURCE or NEWTAPE: the updated source code file
- LINE: the printer listing of the program
- ERRORS or ERRORFILE: the error message file
- XREFFILE: cross-reference information
- INFO: information used for separate compilation

CODE File

The CODE file is produced unconditionally and contains the executable object code produced by the compiler. This file is either stored permanently, executed and then discarded, or discarded, depending on the specifications in the WFL or CANDE COMPILE statement that initiated the compiler, and on whether or not syntax errors occurred during the compilation. For more information on the WFL and CANDE COMPILE statements, refer to the *Work Flow Language (WFL) Programming Reference Manual* and the *CANDE Operations Reference Manual*.

The object code file title can be specified using file equation. If the compiler is invoked from WFL, the <code file title> is specified in the WFL COMPILE statement. If the compiler is invoked from CANDE, the code file title is determined from the <object file name> or <source file title> specified in the CANDE COMPILE statement or the <work file title>. When a procedure is put into a separate code file, the last node of the file name is replaced by the procedure name.

NEWSOURCE File

The synonym NEWSOURCE can be used in place of NEWTAPE.

The NEWSOURCE file is produced only if the NEW compiler control option is TRUE. The NEWSOURCE file is an updated source file that consists of the source input from the CARD file, the SOURCE (or TAPE) file, and (if the INCLNEW compiler control option is TRUE) the included files that were actually compiled. If either NEWSOURCE or NEWTAPE are not file equated, a disk file is created with the name *NEWSOURCE*. For more information on the NEW and INCLNEW options, see "New Option" and "INCLNEW Option" later in this section.

LINE File

The LINE file is produced if either of the compiler control options LIST or TIME is TRUE. If compiler file equation is not applied to the LINE file, it is a printer file.

The contents and format of the LINE file depend on the values of the following compiler control options:

CODE	MAP
FORMAT	PAGE
LISTDELETED	SEGS
LISTINCL	SINGLE
LISTOMITTED	STACK
LISTP	TIME

The LINE file always contains at least the following information:

- The title of the source file CARD used as input to the compiler
- Code segmentation information
- Error messages and error count, if syntax errors occur
- Summary information about the compile, such as the number of lines read and the size of the object code file

For more information about the compiler control options that affect the LINE file, see “Option Descriptions” later in this section.

ERRORS File

The synonym ERRORFILE can be used in place of ERRORS.

The ERRORS file is produced only if the ERRLIST compiler control option is TRUE. If either ERRORS or ERRORFILE are not equated, the file is created with the title *ERRORS*. It is a printer file if the compile is initiated through WFL, and a remote file if the compile is initiated through CANDE. If no syntax errors occur during compilation, no ERRORS file is produced, regardless of the value of the ERRLIST option. For more information on the ERRLIST option, see “ERRLIST Option” later in this section.

For every syntax error that occurs during compilation, two records are written to the ERRORFILE file. The first record is a copy of the source record that contains the error. The second record contains the sequence number of that source record (if the source record was sequenced) and the error message.

XREFFILE File

When either of the compiler control options XREF or XREFFILES is TRUE, the compiler saves raw cross-reference information in the XREFFILE file. The contents of the file are affected by the compiler control options XDECS and XREFS. Before this information can be printed or read by SYSTEM/INTERACTIVEXREF or the Editor, it must be analyzed by SYSTEM/XREFANALYZER. The XREFFILE file is given the file name *XREF/<code file name>*, where *<code file name>* is the file name of the object code file being produced. For more information about the cross-reference options, see "Option Descriptions" later in this section.

INFO File

The INFO file contains the contents of variables and tables used in the compiler. It is intended for use in separate compilations of procedures. This file is created by the DUMPINFO compiler control option, and it is used as an input file by the LOADINFO compiler control option.

Source Record Format

The records of a source code file read by the ALGOL compiler can be any size greater than or equal to 72 characters.

Assume that the character positions (called columns) of a source record are numbered from 1 to n , where n is the size of the record in characters. The compiler divides each source record as follows:

- The data in columns 1 through 72 is assumed to be ALGOL source language as defined in this manual. Any characters that appear beyond column 72 are not compiled as source language constructs.
- Characters in columns 73 through 80 of a record are treated as a sequence number, which is optional.
- Any information beyond column 80 is ignored.

The column in which an ALGOL construct begins is not significant, unless the source record is a compiler control record or the construct continues beyond column 72. For more information on compiler control records, refer to “Compiler Control Records” later in this section. The data in columns 1 through 72 is treated as a continuous stream from record to record. In other words, no delimiters are implied at the end of a record, and string literals, identifiers, and all other valid ALGOL constructs can be continued from column 72 of one record to column 1 of the succeeding record.

Compiler Control Options

Compiler control options provide the programmer with the means to control many aspects of the compilation of an ALGOL program. These options can instruct the compiler to use optional input files or to produce optional output files. In addition to other things, the options can do all of the following:

- Affect the contents of the printer listing
- Designate the target computer for which code is to be produced
- Set the default character type
- Cause the compiler to perform separate compilation
- Invoke the Binder
- Invoke debugging features
- Exclude source records
- Set a limit on the number of syntax errors the program can get
- Resequence source records
- Check that sequence numbers are in order
- Control code file segmentation

The TASKSTRING attribute can be used to pass compiler control options to the compiler. It is a string-valued task attribute of up to 254 EBCDIC characters. The use of the dollar (\$) sign is required to begin the string image. The percent (%) sign is treated as a comment indicator and the information that follows is ignored. SET, RESET, and POP can be used with the compiler control options in TASKSTRING just as they would be used in standard compiler control images. The options DELETE, VOIDT, PAGE, and VOID cannot be passed to the compiler through TASKSTRING, or appear in any files that are included by an INCLUDE option in the TASKSTRING.

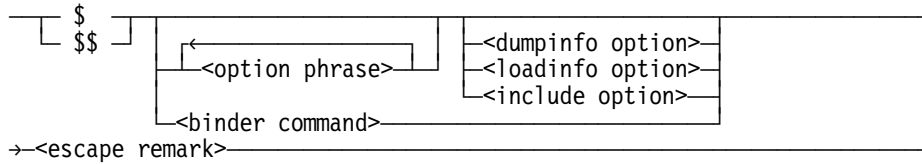
If the INCLUDE option is used in TASKSTRING, it must be the last option in the string. The included files must consist of valid compiler control options only. The options are processed before any records of the actual symbolics. To pass compiler control options to the compiler with this attribute, enter the following as a compiler attribute assignment: *COMPILER TASKSTRING="<compiler control record>*". The following is an example of the use of the compiler attribute assignment:

```
C; COMPILER TASKSTRING="$SET LIST"
```

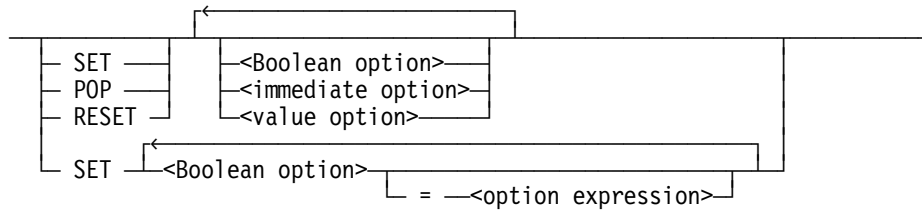
Compiler Control Records

Compiler control options are included in an ALGOL program by using special source records called compiler control records. A compiler control record can contain more than one option. All options on a compiler control record must follow the currency sign (\$). At least one space must follow each option. No option can continue past column 72 of a compiler control record.

<compiler control record>



<option phrase>



<Boolean option>

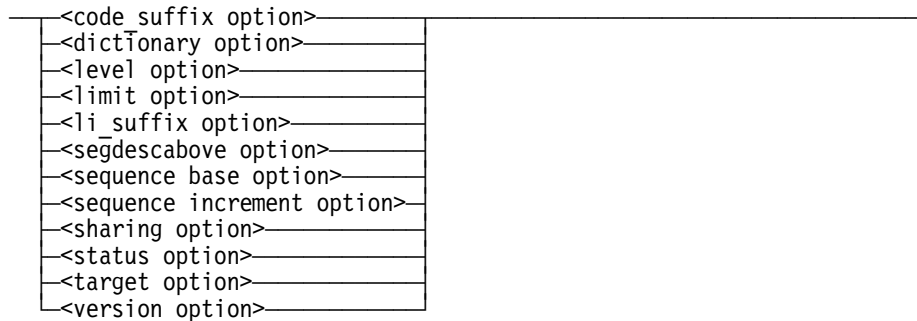
A <Boolean option> can be any of the following:

- | | | |
|-----------------------|------------------------|---------------------|
| <ASCII option> | <makehost option> | <strings option> |
| <autobind option> | <map option> | <TADS option> |
| <check option> | <MCP option> | <time option> |
| <code option> | <merge option> | <user option> |
| <delete option> | <new option> | <void option> |
| <errlist option> | <newseqerr option> | <voidt option> |
| <errorlist option> | <nobindinfo option> | <waitimport> option |
| <format option> | <nostackarrays option> | <warnsupr option> |
| <inclnew option> | <noxreflist option> | <writeafter option> |
| <inclseq option> | <omit option> | <xdecs option> |
| <installation option> | <optimize option> | <xref option> |
| <intrinsic option> | <rangecheck option> | <xreffiles option> |
| <library option> | <segs option> | <xrefs option> |
| <lineinfo option> | <sepcomp option> | |
| <list option> | <seq option> | |
| <listdeleted option> | <sequence option> | |
| <listdollar option> | <seqerr option> | |
| <listincl option> | <single option> | |
| <listomitted option> | <stack option> | |
| <listp option> | <statistics option> | |

<immediate option>



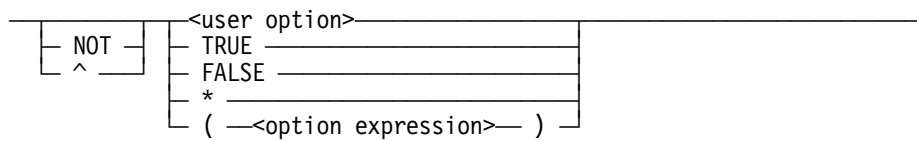
<value option>



<option expression>



<option primary>



<binder command>



Compiler control records are submitted to the ALGOL compiler as part of the source input and are distinguished from other constructs by the dollar sign (\$) that must begin every compiler control record.

Compiler control records with the \$ in column 1 but not in column 2 affect only the current compilation and are not saved in the NEWSOURCE file, if there is one. Compiler control records with the \$\$ in columns 1 and 2, or a blank in column 1 and a \$ in column 2, are considered permanent compiler control records and are saved in the NEWSOURCE file, if there is one.

A compiler control record can contain the following:

- Boolean options
- Value options
- Immediate options
- Special options
- Binder commands

A Boolean option is one that is either enabled (TRUE) or disabled (FALSE). When enabled, it causes the compiler to apply an associated function to all subsequent processing until the option is disabled.

A value option causes the compiler to store a value associated with a given function.

An immediate option causes the compiler to perform immediately a function that is not applied to subsequent processing.

The special options (the DUMPINFO option, the LOADINFO option, and the INCLUDE option), like the immediate options, cause the compiler to perform immediately a function that is not applied to subsequent processing. However, they are not grouped with the immediate options because of special syntactic requirements.

Binder commands are passed directly to the Binder program. For more information on Binder statements and control options, refer to the *Binder Programming Reference Manual*.

The keywords SET, RESET, and POP affect the value of Boolean options. Each Boolean option has an associated stack in which the current value and up to 46 previous values of the option are saved. The management of this stack of values is as follows:

1. If the first keyword to the left of a Boolean option in a compiler control record is SET or RESET, the current value of the option is pushed onto the stack and the option is assigned a value of TRUE or FALSE, respectively. In other words, the option is assigned a new value, and the previous value is saved in the stack.
2. If the first keyword to the left of a Boolean option in a compiler control record is POP, the current value of the option is discarded and the previous value is removed from the top of the stack and assigned to the option.

3. If a Boolean option is not preceded in a compiler control record by any keyword, then the following actions occur:
 - a. All resettable standard Boolean options are assigned a value of FALSE and all previous values in their stacks are discarded.
 - b. The Boolean options appearing in the compiler control record with no preceding keyword are assigned a value of TRUE.

The following is a list of the resettable standard Boolean options that are affected when a Boolean option appears in a compiler control record without a preceding keyword:

ASCII	LISTDOLLAR	STACK
AUTOBIND (if SEPCOMP is FALSE)	LISTINCL	STATISTICS
CHECK	LISTP	STRINGS
CODE	MAKEHOST (if the first syntactic item has not been seen)	TIME
DELETE	MAP	VOID
ERRLIST	NEW	VOIDT
ERRORLIST	NEWSEQERR	WARNSUPR
FORMAT	NOXREFLIST	WRITEAFTER
INCLNEW	OMIT	XREF
INCLSEQ	OPTIMIZE	XREFFILES
INSTALLATION	SEGS	XREFS
INTRINSICS	SEQ	\$
LIST	SEQUENCE	
LISTDELETED	SEQERR	

A compiler control record that consists of only an initial \$ (a \$ followed by all blanks) has no effect unless it is in the CARD file, the MERGE option is TRUE, and a record is present in the secondary input file TAPE that has the same sequence number as this compiler control record. When these three conditions are met, then that TAPE record is ignored.

In an <option expression>, the Boolean operators have the same precedence as they do in Boolean expressions; that is, *NOT* and *^* are equivalent and have the highest precedence, followed by *AND*, *OR*, *IMP*, and *EQV*, in that order. The <option primary> "*" represents the current value of the Boolean option whose value is being assigned. For example, the following compiler control record causes records from both the CARD and TAPE files to be ignored if either the OMIT option is TRUE or the user option DEBUGCODE is TRUE.

```
$ SET OMIT = * OR DEBUGCODE
```

Examples

The following compiler control record assigns a value of TRUE to the LIST option, MERGE option, and NEW option, and assigns a value of FALSE to the SINGLE option. The previous value of each of these options is saved in the corresponding stack.

```
$ SET LIST MERGE NEW RESET SINGLE
```

The following compiler control record invokes the immediate option PAGE, causing the compiler to skip the printer listing to the top of the next page (if the LIST option is TRUE). Because the PAGE option is not a Boolean option, the action described above for Boolean options not preceded by a keyword does not occur.

```
$ PAGE
```

The following compiler control record assigns a value of TRUE to the LISTP option, assigns the value 135000 to the sequence base option, and returns the LIST option to its most recent previous value.

```
$ SET LISTP 135000 POP LIST
```

The following compiler control record assigns FALSE to all resettable standard Boolean options and discards all previous values in their stacks, assigns the value TRUE to the LISTP option, and returns the USER option MYOPTION to its most recent previous value. The stack for MYOPTION is unaffected by the purge of the stacks for the standard resettable Boolean options, because it is not a standard option.

```
$ LISTP POP MYOPTION
```

Option Descriptions

The remainder of this section describes the individual compiler control options.

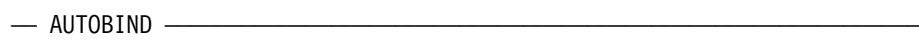
ASCII Option



(Type: Boolean, Default value: FALSE)

STRINGS = ASCII is a synonym that can be used for the ASCII option. When TRUE, the ASCII option sets the default character type to ASCII. For more information, refer to "Default Character Type" in Appendix C, "Data Representation."

AUTOBIND Option



(Type: Boolean, Default value: FALSE)

If TRUE, the AUTOBIND option causes the processes of compilation and program binding to be combined into one job. During compilation, the compiler produces a set of instructions to be passed to the Binder program. In most cases, these Binder instructions are sufficient. If additional Binder instructions are required, the ALGOL <binder command> syntax can be used.

The AUTOBIND option can be assigned a value at any time during compilation. However, for the following reasons, it should be assigned a value only once, at the beginning of compilation:

- The value of the AUTOBIND option is significant only at the end of compilation. For example, if four procedures are being compiled, the first three with the AUTOBIND option FALSE and the last with the AUTOBIND option TRUE, the Binder attempts to bind all four procedures to the specified host.
- When the AUTOBIND option is FALSE, compile-and-go on a separate procedure is not executed. If the AUTOBIND option is TRUE throughout compilation, execution of the resulting program takes place after binding.

In ALGOL, an outer block or a separate procedure compiled at lexical (lex) level two can serve as a host for binding. Separate ALGOL procedures compiled at lex level three (the default level) or higher can be bound into a host. Any number of separate procedures, but only one host, can be compiled in one job. The host must be the last program unit compiled. If an appropriate host file is compiled with the AUTOBIND option equal to TRUE, it is assumed to be the host for binding. This assumption cannot be overridden by file equation or by use of the HOST option. If no eligible host is being compiled, a host must be specified, either by file equation of the compiler file HOST or by use of the HOST option.

The object code file of any procedure compiled at lex level three or higher with the AUTOBIND option equal to TRUE is marked as nonexecutable. To be executed, the procedure must be bound into a host file by the Binder program or invoked by a PROCESS or CALL statement.

Object code files of any procedures compiled at lex level three or higher are purged after being bound into a host by the AUTOBIND option. To be retained, such a code file must be referenced specifically in either a BIND option or an EXTERNAL option.

If the TADS option is TRUE when the AUTOBIND option is assigned the value TRUE, then the AUTOBIND option is left equal to FALSE and a warning message is given.

The process of binding might be permitted by the compiler, even if there are syntax errors in the compilation. This can occur, for example, when the host file is compiled successfully, even though none of the separate procedures succeeded. In another example, there is no host file in the compilation, but at least one of the separate procedures compiles.

Note that Binder uses a default code file title to search for external procedures if no title is provided in a BIND option. The following is an example of searching for external procedures:

```
BEGIN
  PROCEDURE P1; EXTERNAL;
  PROCEDURE P2; EXTERNAL;
  REAL R;
  PROCEDURE P3;
  BEGIN
  ...
  END;

...
  P1;
  P2;
  P3;
END.
```

This program can be compiled successfully if separate code files have been generated for the external procedures P1 and P2, and the titles match the default titles that Binder expects.

A <bind option>, for example *\$BIND P1, P2 FROM <codefile name>*, needs to be added to this program if the external procedures P1 and P2 are compiled into one multiprocedure code file, or if they do not use Binder's default naming convention.

The default or set value of the LIBRARY option affects the separately compiled external procedures. Refer to the LIBRARY option later in this section for further information.

BEGINSEGMENT Option

— BEGINSEGMENT —————|

(Type: immediate)

The BEGINSEGMENT option and the ENDSEGMENT option allow the programmer to control code file segmentation. Procedures encountered between a BEGINSEGMENT option and an ENDSEGMENT option are placed in the same code segment, which is called a user segment because it is user-controlled instead of compiler-controlled.

The BEGINSEGMENT option must appear before the declaration of the first procedure to be included in the user segment. The ENDSEGMENT option must appear after the last source record of the last procedure to be included in the user segment.

A procedure cannot be split across user segments. The first procedure in the user segment must be one for which the compiler would normally generate a segment; that is, it must have local declarations. External procedures cannot be declared in a user segment.

Declarations of global items other than procedures within a user segment can result in those items not being initialized. This could cause the program to get a fault at run time. Declarations of global items other than procedures should be placed outside user segments.

User segments can be nested; that is, a BEGINSEGMENT option can appear in a user segment. In this case, an ENDSEGMENT option applies to the user segment currently being compiled.

If a BEGINSEGMENT option appears before the beginning of a separately compiled procedure, an ENDSEGMENT option is assumed at the end of the procedure, even if none appears. The driver procedure created for procedures compiled at lexical level three is always in a different code segment.

The segment information in the printer listing is modified for user segments. User segments are numbered consecutively in a program, beginning with 1; that is, the first user segment is USERSEGMENT1, the second user segment is USERSEGMENT2, and so forth. The code segment number of each user segment is printed at its beginning; the length of each user segment is printed at its end. Procedures or blocks whose segmentation is overridden by user segmentation are printed out as being in that user segment.

Forward procedure declarations are not affected by user segmentation.

If more than one BEGINSEGMENT option appears before a procedure, the warning message *EXTRA BEGINSEGMENT IGNORED* is printed. If an ENDSEGMENT option appears and there has been no BEGINSEGMENT option, the warning message *EXTRA ENDSEGMENT IGNORED* is printed.

The BEGINSEGMENT option and ENDSEGMENT option allow the programmer to reduce presence-bit overhead by grouping frequently called procedures and infrequently called procedures in separate segments.

BIND Option

— BIND —<text>—————|

(Type: Binder command)

During autobinding, the BIND option is passed directly to the Binder program for analysis. The format and function of this option are the same as those of the Binder BIND statement and are described in the *Binder Programming Reference Manual*.

BINDER Option

— BINDER —<text>—————|

(Type: Binder command)

During autobinding, the text in the BINDER option is passed directly to the Binder program for analysis. The format and function of <text> are the same as those of the Binder control options and are described in the *Binder Programming Reference Manual*.

BINDER_MATCH Option

<binder_match option>
 — BINDER_MATCH [=] (—<string>— , —<string>—) —————|

(Type: Immediate)

Setting the BINDER_MATCH option adds the two strings to the object file. When two BINDER_MATCH options have identical first strings, Binder verifies that the second strings are also identical. If they are not, Binder prints an error message and the bind is aborted. Applications can use this option to verify that different object files were compiled with the same set of options. Strings must match exactly, including the use of upper- and lowercase letters.

Do not start the first string with a star (*), as this is reserved for system use only.

Multiple BINDER_MATCH options can be specified. Each is stored separately and Binder tests each separately. Having two BINDER_MATCH options in the same file with the same first string but different second strings causes a compile-time error.

CHECK Option

— CHECK —————|

(Type: Boolean, Default value: FALSE)

When TRUE, the CHECK option causes an error to be given if the sequence number on a record of the SOURCE or NEWSOURCE file is not strictly greater than the sequence number of the preceding record. If a sequence error occurs in the SOURCE file, the word *SEQERR*, followed by the sequence number of the previous source record, is printed at the right side of the source record on the printer listing. If a sequence error occurs in the NEWTAPE file, the message *NEWTAPE SEQ ERROR*, followed by the sequence number of the previous source record, is printed on the listing and the message *NEWTAPE SEQ ERR* is displayed on the Operator Display Terminal (ODT). In the ERRORS file, the sequence number of the record that caused the sequence error and the sequence number of the previous source record appear on the line following the source record.

If the NEW option is FALSE and resequencing is occurring, the old sequence number is the sequence number that is checked.

CODE Option

— CODE —————|

(Type: Boolean, Default value: FALSE)

If both the LIST option and the CODE option are TRUE, the printer listing includes the compiler-generated object code. If the LIST option is TRUE but the CODE option is FALSE, the printer listing does not include the object code. The value of the CODE option is ignored if the LIST option is FALSE.

CODE_SUFFIX Option

— CODE_SUFFIX [=] <EBCDIC string> —————|

(Type: Value, Default value: none or procedure name)

The CODE_SUFFIX option is used to replace the last node of a code file name with the <EBCDIC string>. The size of the string literal cannot exceed 36 characters. The result of this option must be a valid code file title or a compile-time attribute error occurs. The CODE_SUFFIX option must appear before the procedure name to take effect. If multiple CODE_SUFFIX options appear, the last valid use of the option before the procedure name is the one used by the compiler.

The CODE_SUFFIX option is most useful when the LIBRARY option is FALSE and the source file contains multiple separate procedures. If there are multiple separate procedures in a source file, the CODE_SUFFIX option must be specified for each separate procedure for which it is to be used. The option value does not persist from one procedure to the next.

DONTBIND Option

— DONTBIND —<text>_____

During autobinding, the DONTBIND option is passed directly to the Binder program for analysis. The DONTBIND option directs the Binder not to bind a specified subprogram (procedure). In addition, the DONTBIND option can be used to suppress the binding of all external references declared by a program. The format and function of the DONTBIND option are the same as those of the Binder DONTBIND statement. For information about the DONTBIND statement, refer to the *Binder Programming Reference Manual*.

DUMPINFO Option

— DUMPINFO _____
 └─<file specification>┘

<file specification>

—<title>_____┘
 └─<internal file name>┘
 └─<name and title>┘

<title>

A quoted string containing a file title.

<internal file name>

—<identifier>_____

<name and title>

—<internal file name>— = —<title>_____

(Type: special)

The DUMPINFO option is described with the LOADINFO option in this section.

ENDSEGMENT Option

— ENDSEGMENT —————|

(Type: immediate)

The ENDSEGMENT option is described with the BEGINSEGMENT option in this section.

ERRLIST Option

┌ ERRLIST —————|
└ ERRORLIST ┘

(Type: Boolean, Default value: TRUE for CANDE-originated compiles, FALSE otherwise)

ERRORLIST is a synonym that can be used for the ERRLIST option. The ERRORLIST synonym should not be used if code is to be run through SYSTEM/PATCH.

When TRUE, the ERRLIST option causes syntax error information to be written to the ERRORS file. When a syntax error is detected in the source input, the source record that contains the error, an error message, and the syntactical item where the error occurred are written on two lines in the ERRORS file. This option is provided primarily for use when the compiler is invoked at a terminal by CANDE, but it can be used regardless of the manner in which the compiler is invoked. When the compiler is invoked from CANDE, the default value of the ERRLIST option is TRUE, and the ERRORS file is automatically equated to the remote device from which the compiler was invoked.

EXTERNAL Option

— EXTERNAL <text> —————|

(Type: Binder command)

During autobinding, the EXTERNAL option is passed directly to the Binder program for analysis. The format and function of this option are the same as those of the Binder EXTERNAL statement and are described in the *Binder Programming Reference Manual*.

FORMAT Option

— FORMAT —————|

(Type: Boolean, Default value: FALSE)

If both the LIST option and the FORMAT option are TRUE, then to aid readability of the printer listing, several blank lines are inserted after each procedure. If the LIST option is TRUE but the FORMAT option is FALSE, no blank lines are inserted after procedures. If the LIST option is FALSE, the value of the FORMAT option is ignored.

GO TO Option

— GO TO <sequence number> _____|

<sequence number>

— /8\-<digit> _____|

(Type: immediate)

The GO TO option is used to reposition the secondary source input file TAPE. This option is intended for use with disk files and does not work on tape files.

The <sequence number> construct specifies a sequence number appearing on a record in the TAPE file. The GO TO option causes the TAPE file to be repositioned so that the next record from this file used by the compiler is the first record with a sequence number greater than or equal to the specified sequence number. The TAPE file must be properly sequenced in ascending order; that is, the sequence number on each record in the file must be strictly greater than the sequence number on the preceding record. The specified sequence number can be greater than or less than the sequence number of the record on which the option appears.

This option cannot appear within a DEFINE declaration or in included source input.

HOST Option

— HOST —<text> _____|

(Type: Binder command)

During autobinding, the HOST option is passed directly to the Binder program for analysis. The format and function of this option are the same as those of the Binder HOST statement and are described in the *Binder Programming Reference Manual*.

INCLNEW Option

— INCLNEW _____|

(Type: Boolean, Default value: FALSE)

If both the NEW option and the INCLNEW option are TRUE, included source input is written to the NEWTAPE file. If the NEW option is TRUE but the INCLNEW option is FALSE, included source input is not written to the NEWTAPE file. If the NEW option is FALSE, the value of the INCLNEW option is ignored.

The <start specification> construct specifies the record of the included file at which inclusion is to start. If the <sequence number> form is specified, inclusion begins with the first record with a sequence number greater than or equal to the specified sequence number. If the asterisk (*) form of <start specification> is used, inclusion begins at the point where it left off the last time inclusion took place from this file at the same level of nesting. If the <start specification> construct is not used, inclusion begins with the first record of the file.

The <stop specification> construct specifies the record after which inclusion is to stop. If the <stop specification> construct is not used, inclusion ends after the last record of the file.

Source files suitable for use by the INCLUDE option can be produced by the compiler by using the NEW option.

Files declared globally in Work Flow Language (WFL) jobs should not be used as INCLUDE files.

The INCLUDE option must be the last option appearing on a compiler control record.

An include file is available that enables you to access the POSIX (Portable Operating System Interface) functionality implemented in the MCP support library. For more information on this include file and POSIX functions, see the *ALGOL and MCP Interfaces to POSIX Features Programming Reference Manual*.

Examples

The following example instructs the compiler to accept as input all records from the file indicated by the internal name FILE8, with a sequence number greater than or equal to 00001000 and less than or equal to 09000000.

```
$ INCLUDE FILE8 00001000 - 09000000
```

The following example instructs the compiler to accept as input a portion of the file accessed by the last INCLUDE option at this level of nesting. The records to be included are all records that follow the last record included by that preceding INCLUDE option. If, for example, the preceding INCLUDE option was the one in the first example above, the file that is accessed is FILE8, and the records that are included are all the records with a sequence number greater than 09000000.

```
$ INCLUDE *
```

The following example instructs the compiler to accept as input a portion of the file with the title *SOURCE/XYZ*. The included records are all records of the file with a sequence number less than or equal to 00000900.

```
$ INCLUDE "SOURCE/XYZ." - 900
```

The following example instructs the compiler to accept as input all records of either the file to which the internal name INCL was file-equated or, if the INCL file was not file-equated, the file *SYMBOL/ALGOL/INCLUDE1*.

```
$ INCLUDE INCL = "SYMBOL/ALGOL/INCLUDE1."
```

INITIALIZE Option

— INITIALIZE —<text>—————|

(Type: Binder command)

During autobinding, the INITIALIZE option is passed directly to the Binder program for analysis. The format and function of this option are the same as those of the Binder INITIALIZE statement and are described in the *Binder Programming Reference Manual*.

INSTALLATION Option

— INSTALLATION —————|
 └<installation number list>┘

<installation number list>

┌—————┐
└<installation number>┘ ————|
 └ —<installation number> ┘

<installation number>

—<unsigned integer>—————|

(Type: Boolean, Default value: FALSE)

When TRUE, the INSTALLATION option causes the compiler to recognize one or more groups of installation intrinsics so that they can be referenced in an ALGOL program. This option must be assigned a value before the first syntactic item in a program. Assigning a value to this option at any other time has no effect.

An installation number must be an unsigned integer between 1 and 2047, inclusive. Each installation number in an installation number list must be strictly greater than the preceding installation number in that list. Installation numbers larger than 2047 are treated as if they were equal to 2047.

An INSTALLATION option with no installation number list is equivalent to one with an installation number list of 100 through 2047.

INTRINSICS Option

— INTRINSICS —————|

(Type: Boolean, Default value: FALSE)

When TRUE, the INTRINSICS option causes separately compiled procedures to be compiled at lexical (lex) level two and allows a <global part> construct to appear before the procedures. These procedures can then be used as installation intrinsics. A <global part> is not normally allowed when compiling separate procedures at lex level two.

The title of the object code file generated for a procedure when the INTRINSICS option is TRUE is the same as if the procedure were compiled at lex level three. Thus, the separate procedures being compiled can be bound into the intrinsics. When the Binder program is used to bind procedures into the intrinsics, the INTRINSICS option must be assigned the value TRUE before the first source statement.

The default or set value of the LIBRARY option affects the code generation of the separately compiled procedures. Refer to the LIBRARY option later in this section for further information.

LARGE_LINEINFO Option

— LARGE_LINEINFO —————|

(Type: Boolean, Default value: FALSE)

When TRUE, the LARGE_LINEINFO option causes LINEINFO data to be written to a temporary file as it is being generated during the compilation. At the end of the compilation, the LINEINFO data is copied from the temporary file to the end of the code file. This option allows the LINEINFO option to be set during the compilation of an extremely large program. Otherwise, the compilation would be terminated because the code file had become too large.

The LARGE_LINEINFO option can only be set if the LINEINFO option is also set. The LARGE_LINEINFO option must appear before the first syntactic item in the program. This option cannot be set for separate procedures and is only meaningful for the first <program unit> being compiled.

LEVEL Option

— LEVEL —<outer level>—————|

<outer level>

—<unsigned integer>—————|

(Type:value, Default value: 2 for programs, 3 for separately compiled procedures)

The LEVEL option allows the programmer to override the lexical (lex) levels assigned by the compiler. This feature is needed when compiling separate procedures for binding to a host program. The <outer level> construct specifies the lex level at which compilation is to begin.

The LEVEL option must appear before the first syntactic item in a program.

LIBRARY Option

— LIBRARY —————|

(Type: Boolean, Default value: TRUE for CANDE-originated compiles and when the SEPCOMP option is TRUE, FALSE otherwise)

When compiling multiple separate procedures (such as intrinsics), assigning TRUE to the LIBRARY option improves the efficiency of the binding. When TRUE, this option causes all object code from this compilation to be put in one file, which is marked as a multiprocedure code file. If the LIBRARY option is FALSE, each separate procedure produces its own object code file.

The LIBRARY option must appear before the first syntactic item in a program.

The LIBRARY option is unrelated to the library facility described in Section 8, "Library Facility."

LIMIT Option or ERRORLIMIT Option

— [LIMIT] [ERRORLIMIT] [=] <error limit> —————|

<error limit>

—<unsigned integer> —————|

(Type: value, Default value: 10 for CANDE-originated compiles, 150 otherwise)

The LIMIT option allows the programmer to specify the number of compile-time errors that can occur before the compilation is terminated because of excessive errors.

A limit of 0 indicates that the program is not to be terminated for excessive errors. If, when the LIMIT option is assigned a value, the number of syntax errors already equals or exceeds that value, then the program is immediately terminated.

LINEINFO Option

— LINEINFO —————|

(Type: Boolean, Default value: TRUE for CANDE-originated compiles, FALSE otherwise)

When TRUE, the LINEINFO option causes the compiler to associate sequence number information with the object code. This information is then displayed in the event of a run-time error. A larger code file is generated if the LINEINFO option is TRUE than if it is FALSE.

LIST Option

— LIST —————|

(Type: Boolean, Default value: FALSE for CANDE-originated compiles, TRUE otherwise)

When TRUE, the LIST option causes source input from the CARD and TAPE files and other information to be printed on the compiler LINE file.

When a value is assigned to the LIST option, the same value is assigned to the SEGS option.

LISTDELETED Option

— LISTDELETED —————|

(Type: Boolean, Default value: FALSE)

When both the LIST option and the LISTDELETED option are TRUE, the printer listing includes records from the secondary input file TAPE that are replaced, voided, or deleted during the compilation. The word *REPLACED* appears to the right of the source records replaced by a record from the primary input file, CARD; the word *VOIDT* appears if the record is voided from the TAPE file by the VOIDT option; and the word *DELETED* appears if the record is deleted by a compiler control record that consists of a dollar sign (\$) followed by all blanks.

If the LIST option is TRUE but the LISTDELETED option is FALSE, records from the TAPE file that are replaced, voided, or deleted are not written to the printer listing. If the LIST option is FALSE, the value of the LISTDELETED option is ignored.

LISTDOLLAR Option

— LISTDOLLAR —————|

(Type: Boolean, Default value: FALSE)

If both the LIST and the LISTDOLLAR option are TRUE, the printer listing includes all compiler control records. If the LIST option is TRUE but the LISTDOLLAR option is FALSE, only compiler records with a double dollar sign (\$\$) appear in the printer listing. If the LIST option is FALSE, the value of the LISTDOLLAR option is ignored.

LISTINCL Option

— LISTINCL —————|

(Type: Boolean, Default value: FALSE)

When both the LIST option and the LISTINCL option are TRUE, source records included by using the INCLUDE option are written to the printer listing. If the LIST option is TRUE but the LISTINCL option is FALSE, the included records are not written to the printer listing. If the LIST option is FALSE, the value of the LISTINCL option is ignored.

LISTOMITTED Option

— LISTOMITTED —————|

(Type: Boolean, Default value: TRUE)

When both the LIST option and the LISTOMITTED option are TRUE, source records omitted by the OMIT option are written to the printer listing. In the listing, the word *OMIT* appears next to the sequence number of each omitted record. If the LIST option is TRUE but the LISTOMITTED option is FALSE, omitted records are not written to the printer listing. If the LIST option is FALSE, the value of the LISTOMITTED option is ignored.

LISTP Option

— LISTP —————|

(Type: Boolean, Default value: FALSE)

When TRUE, the LISTP option causes records from the primary source input file, CARD, to be written to the printer listing. Because these records are also written when the LIST option is TRUE, the LISTP option is effective only when the LIST option is FALSE.

LI_SUFFIX Option

— LI_SUFFIX = <EBCDIC string>

(Type: Value, Default value: none)

The LI_SUFFIX option appends the EBCDIC string to the sequence number in the LINEINFO information. The size of the string literal cannot exceed 36 characters. This option is meaningful only when the LINEINFO option is TRUE.

The value of the LI_SUFFIX option is applied to all sequence numbers until the option is RESET or the option is given a different value. If the option is RESET, no string is associated with the remaining sequence numbers.

If a program failure occurs and the LINEINFO compiler option was set when the program was compiled, a series of sequence numbers are displayed identifying the sequence number at which the program failed. If the LI_SUFFIX compiler option was used, the string associated with the program is displayed after that sequence number. This action permits easy identification of the source file or files associated with the displayed sequence numbers.

LOADINFO Option

— LOADINFO <file specification>

(Type: special)

The DUMPINFO option and the LOADINFO option make it possible for the contents of certain simple variables and arrays of the compiler to be saved in a disk file and subsequently reloaded for separate compilation.

The <file specification> construct specifies the file to be created by the DUMPINFO option or loaded by the LOADINFO option. If the <title> form is used, the quoted string specifies the TITLE attribute of the file. The title cannot include embedded quotation marks ("). The <internal file name> form provides an internal file name that can be associated with an actual file by file equation. The <name and title> form provides both an internal file name available for file equation and a title to be used if the internal file name is not file-equated. If the <file specification> construct is not used, the default INFO file or the file that is file-equated to the INFO file is used.

These options are used in conjunction with separate compilation of procedures. Typically, all global declarations are compiled, and then the DUMPINFO option is used to dump information about the global declarations from the compiler to the file INFO. When the separate procedures are to be compiled, the file INFO, containing information about all of the global declarations, is read in by the LOADINFO option before the procedures are compiled. When the XREF option or XREFFILES option is set, the cross-reference information available at the time of a DUMPINFO is saved. If the option is set when a LOADINFO is done, that cross-reference information will be loaded and available for use with any new cross-reference information created in the separate compilation of procedures. For example, consider the following programs:

Program 1

```
BEGIN
  <global declarations>
  $ DUMPINFO
END.
```

Program 2

```
%%% LOAD THE GLOBALS
[
  $ LOADINFO
  <additional global declarations>
]
<separate PROCEDURE declarations>
```

Each time a loadinfo operation is done, the old information in the affected variables and tables of the compiler is discarded. Thus, compiling different portions of the same program, even if they are in different environments, can be done in the same compilation.

The loadinfo operation changes all items in the INFO file to globals and all procedures already compiled to forward PROCEDURE declarations. Thus, an INFO file created by a DUMPINFO operation that is done immediately before a PROCEDURE declaration in a normal compilation is suitable for loading global declarations when that procedure is to be compiled separately.

When two or more items with the same identifier are declared at different lexical (lex) levels, a separate compilation can access only the last declaration seen before the loadinfo operation occurred.

If the release level of the compiler that performs the dumpinfo operation to create an INFO file and the release level of the compiler that performs the loadinfo operation on that file are not the same, a syntax error is given, and the compilation is discontinued. If a loadinfo operation is attempted and the file specified as the INFO file is not in fact an INFO file, or the INFO file was created by a compiler for a different language, a syntax error is given and the compilation is discontinued.

The DUMPINFO option and LOADINFO option must be the last options appearing on a compiler control record.

MAKEHOST Option

— MAKEHOST — ($\overbrace{\hspace{1.5cm}}$, $\overbrace{\hspace{1.5cm}}$) _____|

<environment>

$\overbrace{\hspace{1.5cm}}$ OF $\overbrace{\hspace{1.5cm}}$ _____|

(Type: Boolean, Default value: FALSE)

Given only the source code and object code of a host program to be changed and the patches to change it, the SEPCOMP facility of the compiler can separately compile and bind to the host program only the procedures that are being changed. This method, which is particularly useful for large programs, requires that information not normally collected and saved during the compilation of the host program be saved. When TRUE, the MAKEHOST option causes this information to be saved when compiling a program or a procedure at lexical (lex) level two.

If the MAKEHOST option is TRUE, information is saved in the object code file of the program about the symbolic file used or created by the compilation, the sequence ranges of all procedures declared in the outer block of the program, and the global declarations. The saving of the outer block environment enables lex-level-three procedures to be compiled separately within this environment.

The information saved about the items declared in an environment describes all of the items in that environment. There is no information about the relative order of the declarations and therefore which items are visible from which procedures. Thus, a SEPCOMP of a patch is not necessarily equivalent to a full compile of that patch. For example, consider the following host program:

```
BEGIN
  REAL X;
  PROCEDURE P;
    BEGIN
      REAL R;
      R := X + 5;
    END P;
  REAL ARRAY A[0:4];
  P;
END.
```

If a patch replaces the statement $R := X + 5;$ by the statement $R := X + A[2];$ a full compile with the patch fails with a syntax error on the assignment to R , because array A has not yet been declared. A SEPCOMP of the patch, however, is successful, because the environment information saved for the outer block describes A as well as X and P .

Additional environments can be saved, if desired, so that procedures at lex levels greater than three can be replaced. The list of environments can extend across several source records. Environments must be fully qualified through the outermost level of a PROCEDURE declaration, except that for a program that is a procedure, the name of that procedure must not appear. A procedure that is specified as an environment must contain a local declaration. If a specified environment does not contain a local declaration or if it is never found during the course of compilation, the compiler gives a syntax error containing the name of the environment. Environments can appear in any order, without regard to the actual block structure of the host program.

Source records that are inserted into a program using the INCLUDE option are not included in the environment information saved by the MAKEHOST option. This information is not saved because sequence numbers on included records can duplicate sequence numbers occurring in the rest of the program.

The information necessary to make a program into a host program includes the information saved for the Binder program when the NOBINDINFO option is FALSE; therefore, an error is given if both the NOBINDINFO option and the MAKEHOST option are TRUE.

When a host program is being created, the NEW option should be set to TRUE if any changes are made to the host program. The default title of the source file is saved in the host file for use during a SEPCOMP. If the host is being created as the result of a SEPCOMP, the title of the SOURCE file is saved. If the compilation is not a SEPCOMP but the NEW option is TRUE, the title of the NEWSOURCE file is saved. If NEW is FALSE but MERGE is TRUE, the title of the SOURCE file is saved. Otherwise, the title of the CARD file is saved.

The MAKEHOST option must appear before the first syntactic item in a program.

Examples

When the following program is compiled, information about the global environment is saved in the object code file for the program. If a patch is made to the body of procedure INNER, then during the SEPCOMP process, all of procedure P1 is recompiled and bound to the host program. If, however, the MAKEHOST option was *\$ SET MAKEHOST (P1)* then the local environment of P1 is also saved in the object code file of the program. A SEPCOMP of a patch to the body of procedure INNER would cause only INNER to be recompiled and bound to the host program.

```
BEGIN
  ARRAY A[0:9];
  PROCEDURE P1;
  BEGIN
    BOOLEAN B;
    PROCEDURE INNER;
    BEGIN
      REAL R;
      IF B THEN
        R := * + A[2];
      ...
    END INNER;
    ...
  END P1;
  PROCEDURE P2;
  BEGIN
    ...
  END P2;
  ...
END.
```

In the following example, the second compiler control option overrides the first, saving the environment of procedures PASSONE, PASSTWO, and WRAPUP OF PASSTWO in addition to the global environment.

```
$SET MAKEHOST
$SET MAKEHOST (PASSONE, PASSTWO, WRAPUP OF PASSTWO)
```

MCP Option

— MCP —————|

(Type: Boolean, Default value: FALSE)

When TRUE, the MCP option causes all value arrays, translate tables, truth sets, and constant pools to be allocated at lexical (lex) level two.

The MCP option cannot be assigned a value after the appearance of the first syntactical item in a program.

MERGE Option

— MERGE —————|

(Type: Boolean, Default value: FALSE)

When TRUE, the MERGE option causes the primary source input file, CARD, to be merged with a secondary source input file, TAPE, to form the input to the compiler. If matching sequence numbers occur, the record from CARD overrides the record from TAPE. If the MERGE option is FALSE, only primary source input is used, and the TAPE file is ignored.

The total input to the compiler when the MERGE option is TRUE consists of all records from the CARD file, all records from the TAPE file that do not have sequence numbers identical to those on records in the CARD file, and all records inserted by INCLUDE options. Records in the CARD file also override INCLUDE options in the TAPE file if matching sequence numbers are encountered.

NEW Option

— NEW —————|

(Type: Boolean, Default value: FALSE)

When the NEW option is TRUE, the source input to the compiler from the CARD file is written to the updated source output file NEWTAPE. If the MERGE option is also TRUE, then the merged source input from the CARD and TAPE files is written to NEWTAPE. The format of the file written to NEWTAPE is such that it can later be used as input to the compiler.

Records included in the source input by the INCLUDE option are written to the NEWTAPE file only if the INCLNEW option is also TRUE. Compiler control records are written to the NEWTAPE file only if the initial dollar sign (\$) does not appear in column 1.

The NEWTAPE file is created whether or not syntax errors occur in the source input.

If the MAKEHOST option is TRUE and the first syntactic item has been compiled, any attempt to assign a value to the NEW option results in a syntax error.

NEWSEQERR Option

— NEWSEQERR —————|

(Type: Boolean, Default value: FALSE)

When TRUE, the NEWSEQERR option causes an error to be given if the sequence number on a record of the NEWTAPE file is not strictly greater than the sequence number of the preceding record. If sequence errors occur and the NEWSEQERR option is TRUE, the NEWTAPE file is not locked, the message *NEWTAPE NOT LOCKED* is displayed on the Operator Display Terminal (ODT), and the message *NEWTAPE NOT LOCKED <number of errors> NEWTAPE SEQUENCE ERRORS* is printed on the printer listing. The NEWSEQERR option is effective even if the CHECK option is FALSE.

NOBINDINFO Option

— NOBINDINFO —————|

(Type: Boolean, Default value: FALSE)

When TRUE, the NOBINDINFO option prevents information needed by the Binder program from being written to the object code file. The resulting object code file can be executed, but it cannot be used as an input file to the Binder program. Object code files that do not contain binding information (bindinfo) are smaller than object code files that contain bindinfo.

The Binder program cannot bind object code files that contain the timing code necessary for statistics; therefore, if the NOBINDINFO option is FALSE when the STATISTICS option is assigned the value TRUE, the NOBINDINFO option is assigned the value TRUE and a warning message is given. If the STATISTICS option is TRUE when the NOBINDINFO option is assigned the value FALSE, a syntax error is given.

If the MAKEHOST option is TRUE when the NOBINDINFO option is assigned the value TRUE, a syntax error is given.

NOSTACKARRAYS Option

— NOSTACKARRAYS —————|

(Type: Boolean, Default value: FALSE)

When TRUE, the NOSTACKARRAYS option prevents arrays from being allocated within the stack.

When the NOSTACKARRAYS option is FALSE, the data from certain arrays is allocated within the stack. Such arrays are referred to as in-stack arrays, and can be accessed slightly faster than an array whose data area is allocated in memory.

NOXREFLIST Option

— NOXREFLIST —————|

(Type: Boolean, Default value: FALSE)

When TRUE, the NOXREFLIST option prevents the SYSTEM/XREFANALYZER program from being initiated by the compiler when cross-reference information is being saved (that is, when either the XREF option or the XREFFILES option is TRUE). Instead, the file XREF/<code file name>, where <code file name> is the name of the object code file generated by the compiler, remains on disk. SYSTEM/XREFANALYZER can be run later using the file XREF/<code file name> as input. The NOXREFLIST option has no effect if both the XREF option and the XREFFILES option are FALSE.

For more information on cross-referencing, refer to the description of the XREF option later in this section.

OMIT Option

— OMIT —————|

(Type: Boolean, Default value: FALSE)

When TRUE, the OMIT option causes records from the CARD file (and, if the MERGE option is TRUE, from the TAPE file) to be ignored (not compiled). If both the LIST option and the LISTOMITTED option are TRUE, then in the printer listing, the word OMIT appears next to the sequence number of each omitted record. When the OMIT option is TRUE, compiler control records with the initial dollar sign (\$) in either column 1 or column 2 are recognized, but compiler control records with the \$ in columns 3 through 72, inclusive, are ignored.

Example

In the following example, neither NOLIBRARY nor LINKBYTITLE is set. The OMIT option is FALSE until \$SET OMIT=NOT LINKBYTITLE is encountered, which is when it is set to TRUE. As a result, the compiler control record \$POP OMIT % NOT LINKBYTITLE is ignored, causing the line containing ";" to be omitted.

```
BEGIN
  $SET OMIT=NOLIBRARY
LIBRARY
  $SET OMIT=LINKBYTITLE
  LIBA (LIBACCESS=BYFUNCTION, FUNCTIONNAME="MYLIB.")
  $POP OMIT % LINKBYTITLE
  $SET OMIT=NOTLINKBYTITLE
  LIBA (LIBACCESS=BYTITLE, TITLE="SYSTEM/MYLIB.")
  $POP OMIT % NOT LINKBYTITLE
  ;
  $POP OMIT % NOLIBRARY
DISPLAY ("Example of OMIT option.");
END.
```

OPTIMIZE Option

— OPTIMIZE _____|

(Type: Boolean, Default value: FALSE)

When TRUE, the OPTIMIZE option causes additional analysis of Boolean expressions to be performed, and object code is generated to permit early termination of the expression evaluation. Any portion of the Boolean expression that could cause side effects is always evaluated.

PAGE Option

— PAGE _____|

(Type: immediate)

When the LIST option is TRUE and the PAGE option appears, the printer listing is spaced to the top of the next page.

PARAMCHECK Option

— PARAMCHECK —————|

When TRUE, the PARAMCHECK option causes the compiler to report, as a syntax error, the redeclaration of a formal parameter of a procedure, within the outer block of the procedure. This permits the user to see where a formal parameter has been rendered inaccessible to the procedure.

PURGE Option

— PURGE <text>—————|

(Type: Binder command)

During autobinding, the PURGE option is passed directly to the Binder program for analysis. The format and function of this option are the same as those of the Binder PURGE statement and are described in the *Binder Programming Reference Manual*.

SEGDESCABOVE Option

— SEGDESCABOVE [<unsigned integer>]—————|

(Type: value, Default value: none)

The SEGDESCABOVE option is used when compiling large programs that may have difficulty in addressing the segment dictionary.

When a host program is compiled, this option causes all code segment descriptors to be allocated starting at the word in the D1 stack specified by the unsigned integer. The unsigned integer must be in the range 4 to 4095, inclusive. If the option is used after the first syntactic item has been compiled, the given value is added to the current size of the D1 stack. The Binder program preserves the segdescabove specification. Care should be taken when using this option, because unused D1 stack locations below the code segment descriptors occupy save memory when the program is running.

This option is intended to be used when compiling host files and is ignored when separate procedures are compiled.

SEGS Option

— SEGS —————|

(Type: Boolean, Default value: FALSE for CANDE-originated compiles, TRUE otherwise)

If both the LIST option and the SEGS option are TRUE, the printer listing will contain beginning and ending segment messages. Assigning a value to the LIST option assigns the same value to the SEGS option. However, to suppress the segment messages, the SEGS option can be assigned the value FALSE even though the LIST option is TRUE. When the value of the LIST option is FALSE, the value of the SEGS option is ignored.

SEPCOMP Option

— SEPCOMP —————|
 └<title>┘

(Type: Boolean, Default value: FALSE)

When TRUE, the SEPCOMP option invokes the automatic separate compilation and binding facility, called the SEPCOMP facility.

The title of the host program can be specified either by using the <title> syntax of the SEPCOMP option or by file-equating the HOST file of the compiler. The <title> specification takes precedence over file equation. The <title> cannot include embedded quotation marks ("). The title of the default source file is stored in the host program, but this title can be overridden by file equation of the compiler file TAPE.

Compiler control records with blank sequence numbers are accepted following the compiler control record that assigns TRUE to the SEPCOMP option and before the first patch record. A patch record is a source record with a nonblank sequence number; at least one patch record is required in a SEPCOMP. Sequence number errors among patch records are not allowed. The SEPCOMP option examines the patch records, decides which procedures of the host program must be recompiled, and generates Binder input for binding these procedures to the host program. The SEPCOMP option always tries to compile procedures at the highest possible lexical (lex) level. Therefore, the number of extra environments specified when making a host program affects the choices available to the SEPCOMP option.

When TRUE, the SEPCOMP option assigns TRUE to both the AUTOBIND option and the LIBRARY option, causing all procedures to be compiled into one multiprocedure code file (a temporary file used by the Binder program). Explicitly assigning FALSE to the AUTOBIND option prevents the Binder from being called and causes the object code file to be locked on disk if the LIBRARY option is TRUE. Explicitly assigning FALSE to the LIBRARY option causes each procedure compiled to be put in a separate, permanent object code file. Binding still occurs, but at a somewhat slower rate.

If procedures are put in separate code files, the titles of the code files are determined in the standard way, with the procedure name replacing the last identifier from the title on the compile statement that invoked the compiler. Procedures compiled at lex level four and higher have the name of their environment in the code file name also. In the following example, when two lex-level-four procedures are compiled having the same name but different environments, two code files are produced (titled *A/PASSONE/Q* and *A/PASSTWO/Q*) in addition to the new host file titled *A/HOST*, assuming that *PASSONE* and *PASSTWO* were specified as extra environments when *A/HOST* was made.

```
% PATCH CARD TO Q OF PASSONE      <sequence number>
% PATCH CARD TO Q OF PASSTWO      <sequence number>
```

The special information associated with a host program is always copied by the Binder to the object code file of the new program so it can be used as a host. This information is not updated by either the Binder or the compiler during the SEPCOMP process. Following a SEPCOMP, this information can inaccurately reflect the actual structure and content of the host program with which it is associated.

Because the arrangement of data in a bound code file differs from that of an unbound code file, binding to a bound host is faster than binding to an unbound host. For this reason, assigning TRUE to the AUTOBIND option when compiling a host program can be advantageous, because it causes the Binder to be invoked.

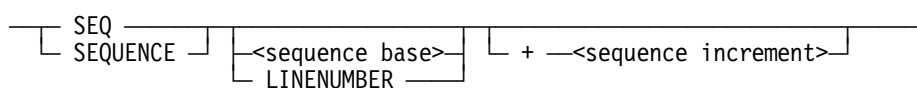
If the release level of the compiler that creates the host code file and the release level of the compiler that is attempting a SEPCOMP to that file are not the same, a syntax error is given and the compilation is discontinued. If the code file specified as the host program was not compiled with the MAKEHOST option equal to TRUE, or if the host program was compiled by a compiler for a different language, a syntax error is given and the compilation is discontinued.

The SEPCOMP option automatically assigns values to several other compiler control options in order to simplify operation. The MERGE option is unavailable for use while the SEPCOMP option is TRUE. Assigning TRUE to the MERGE option before assigning TRUE to the SEPCOMP option is not allowed because it destroys the default file equation of the source file to be used for the patches.

If the TADS option is TRUE when the SEPCOMP option is assigned the value TRUE, the value of the SEPCOMP option is left equal to FALSE and a warning message is given. The SEPCOMP option cannot be assigned a value after the first syntactic item of a program has been compiled. Multiple SEPCOMP option settings are not allowed because, when first assigned TRUE, the SEPCOMP option initiates preprocessing of the source input from the CARD file.

For more information on the SEPCOMP facility, refer to "MAKEHOST Option" earlier in this section.

SEQ Option



<sequence base>

<sequence increment>



(Type: Boolean, Default value: FALSE)

SEQUENCE is a synonym that can be used for the SEQ option. The SEQUENCE synonym should not be used if the code is to be run through SYSTEM/PATCH.

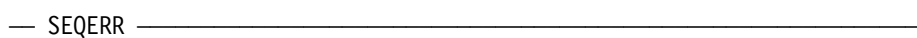
When TRUE, the SEQ option causes the printer listing and the updated source file, NEWSOURCE, to contain new sequence numbers. These new sequence numbers are determined by the current values of the sequence increment and the sequence base or LINENUMBER.

The SEQ option is effective only when the LIST option or NEW option is TRUE. The sequence numbers that appear on the records in these files when the SEQ option is FALSE are identical to the sequence numbers on the corresponding records in the input files.

The sequence base specifies the sequence number that is to be assigned to the next record. LINENUMBER assigns the corresponding record sequence number from the input file to the next record. LINENUMBER is especially useful when creating xreffiles for a NEWSOURCE file, which contains INCLUDE files with INCLNEW set to TRUE.

The value of the sequence increment is used to increment the value of the sequence base or LINENUMBER after each record is resequenced.

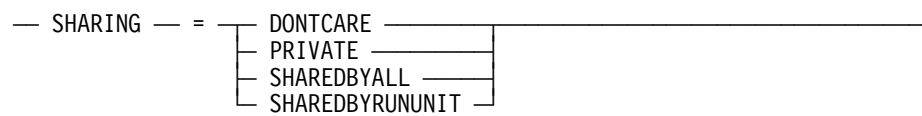
SEQERR Option



(Type: Boolean, Default value: FALSE)

When TRUE, the SEQERR option causes an error to be given if the sequence number on a record of the TAPE file is not strictly greater than the sequence number of the preceding record. If sequence errors occur and the SEQERR option is TRUE, the object code file is not locked, the message *CODE FILE NOT LOCKED* is displayed on the Operator Display Terminal (ODT), and the message *CODE FILE NOT LOCKED <number of errors> TAPE SEQUENCE ERRORS* is printed on the printer listing. The SEQERR option is effective even when the CHECK option is FALSE.

SHARING Option



(Type: value, Default value: DONTCARE)

The SHARING option is used in a library program to specify how other programs are to share the library.

DONTCARE

If the SHARING option has the value DONTCARE, the operating system determines the sharing, and it is unknown to all users invoking the library. DONTCARE is the default value of the SHARING option.

PRIVATE

If the SHARING option has the value PRIVATE, a separate instance of the library is started for each invocation of the library. Any changes made to global items in the library by a block that invoked the library apply only to that user of the library.

SHARED BY ALL

If the SHARING option has the value SHARED BY ALL, all invocations of the library share the same instance of the library. Any changes made to global items in the library by a block that has invoked the library apply to all users of that library.

SHARED BY RUN UNIT

A run unit consists of a program and all libraries that are initiated either directly or indirectly by that program. A program, in this context, does not include either a library that is not frozen or any tasks that are initiated by the program (that is, a process family is not a run unit). If the SHARING option has the value SHARED BY RUN UNIT, all invocations of a library within a run unit share the same instance of the library.

Note that a library is its own run unit until it freezes. For example, program P initiates library A and, before library A freezes, it in turn initiates library B. Now library B is in library A's run unit, not in program P's run unit. Had library A initiated library B after freezing, both library A and library B would be in program P's run unit.

The SHARING option must appear before the first syntactic item in the program.

SINGLE Option

— SINGLE —————|

(Type: Boolean, Default value: TRUE if the compiler was compiled with the DOUBLESPEACE compiler-generation option equal to FALSE, FALSE otherwise)

When TRUE, the SINGLE option causes the printer listing to be single-spaced. When FALSE, the SINGLE option causes the printer listing to be double-spaced.

STACK Option

— [STACK
MAP] —————|

(Type: Boolean, Default value: FALSE)

MAP is a synonym that can be used for the STACK option.

If both the LIST option and the STACK option are TRUE, the printer listing includes the relative stack addresses, in the form of address couples, for all program variables. If the LIST option is TRUE but the STACK option is FALSE, the printer listing does not contain these address couples. The value of the STACK option is ignored if the LIST option is FALSE.

STATISTICS Option

— STATISTICS —————|
| ([SET
| RESET] , [LABELS
| PBITS]) |

(Type: Boolean, Default value: FALSE)

When TRUE, the STATISTICS option causes timing statistics to be gathered. The option is examined at the beginning of each procedure or block and, if it is TRUE at that time, timing statistics are gathered for that procedure or block. Although the value of the option can be altered at any time, only its value at the beginning of procedures and blocks is significant in determining whether timings are made.

The Binder program cannot bind object code files that contain the timing code necessary for statistics; therefore, if the NOBINDINFO option is FALSE when the STATISTICS option is assigned the value TRUE, the NOBINDINFO option is assigned the value TRUE and a warning message is given. If the STATISTICS option is TRUE when the NOBINDINFO option is assigned the value FALSE, a syntax error is given.

If statistics are taken for a procedure or block, the frequency of execution of that procedure or block is measured, along with the length of time spent in that procedure or block. When the program is completed for any reason, including both normal and abnormal termination, the statistics of the executing task are printed out to the TASKFILE.

On the statistics output listing, an asterisk (*) indicates that doubt exists about the timing for the specific procedure whose name precedes the asterisk. In addition, timings are invalid for any procedure or block that is resumed by a bad GO TO.

Statistics on presence bit actions can be accumulated by specifying SET PBITS as the statistics option. The PBITS option can be SET and RESET in various places in the program.

For each block, the statistics report contains the first 25 characters of the identifiers, the processor time accumulated for the block, the number of times the block is executed, and the average time spent per entry. If PBITS is invoked, the report also contains the number of initial and other presence bit operations.

Using the STATISTICS option, especially when PBITS is specified, can significantly increase the amount of code generated for each specified block. This, in turn, can cause a *SEGMENT TOO LARGE* error. To avoid this error, reduce the amount of code enclosed by each BEGINSEGMENT/ENDSEGMENT pair. Or, if the program segmentation is not user-controlled, force part of the block into a new segment by enclosing that part of the code in a block. For more information on program segmentation, refer to the BEGINSEGMENT option earlier in this section.

For any procedure or block that has statistics gathered, the timings can be broken down to the label level within that procedure or block by using the LABELS syntax. The word LABELS can be preceded by SET or RESET; if both are omitted, SET is assumed. For example, the following begins timing of label breakpoints:

```
$ SET STATISTICS (LABELS)
```

The following ends timing of label breakpoints:

```
$ SET STATISTICS (RESET LABELS)
```

The words SET or RESET inside the parentheses affect only the LABELS specification.

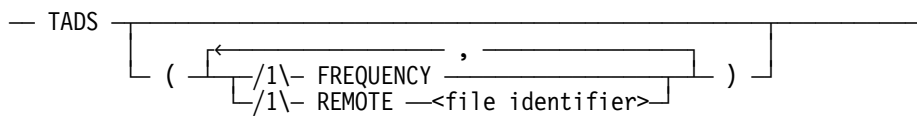
STOP Option

```
— STOP —<text>—————|
```

(Type: Binder command)

During autobinding, the STOP option is passed directly to the Binder program for analysis. The format and function of this option are the same as those of the Binder STOP statement and are described in the *Binder Programming Reference Manual*.

TADS Option



(Type: Boolean, Default value: FALSE)

When the TADS option is TRUE, special debugging code and tables are generated as part of the object code file. The tables are generated to support the symbolic debugging environment of the ALGOL Test and Debug System (TADS). For more information on ALGOL TADS, refer to the *ALGOL Test and Debug System (TADS) Programming Guide*.

The FREQUENCY option causes additional code and tables to be generated for coverage and frequency analysis. This option must be specified if either the TADS *FREQUENCY* command or the TADS *COVERAGE* command is to be used to determine the number of times individual statements have been executed or which statements have not been executed.

The REMOTE option allows TADS to share a remote file with the program being tested. Sharing a file might be necessary, because only one remote input file can be open for each station. A remote file shared with TADS cannot be explicitly opened by the program or an OPEN ERROR results and a *FILE NOT CLOSED* error message is displayed. If specified, the <file identifier> must correspond to a file identifier declared in a FILE declaration occurring later in the outer block of the program. The file must have the attributes KIND equal to REMOTE, UNITS equal to CHARACTERS, and FILEUSE equal to IO. The MAXRECSIZE attribute must not be less than 72. The file must not be declared to be a direct file.

If the AUTOBIND option, the MAKEHOST option, or the SEPCOMP option is TRUE when the TADS option is assigned the value TRUE, then the TADS option is assigned the value FALSE and a warning message is given. Programs compiled with the TADS option equal to TRUE cannot assign TRUE to the MAKEHOST option, the AUTOBIND option, or the SEPCOMP option, and they cannot be used as input files to the Binder program.

The interaction between the outer lexicographical level and the TADS option is as follows:

- If the LEVEL option has not been encountered when the TADS option is set to TRUE, the outer lexicographical level of the program is set to 2 and the code file produced contains TADS code.
- If the LEVEL option is not equal to 2 when the TADS option is set to TRUE, the TADS option is assigned the value FALSE and the warning message *TADS CANNOT BE SET WITH A LEVEL > 2* is given. The code file produced does not contain TADS code.
- If the TADS option is TRUE when the LEVEL option is assigned a value not equal to 2, the warning message *A LEVEL > 2 IS INVALID WHEN TADS IS SET* is given, the LEVEL option is ignored, and the code file produced contains TADS code.

The TADS option must appear before the first syntactic item in a program.

TARGET Option

The TARGET option generates code suited for a specific computer system or group of systems and should be used to specify all machines on which the code file needs to run. The generated code file can be used on all the machines specified in the TARGET option.

```
— TARGET — = —<primary identifier>—————→
→ |—————, —————|
  | ( —<secondary identifier>— ) |
```

<primary identifier>

```
—<target identifier>—————|
```

<secondary identifier>

```
—<target identifier>—————|
```

(Type: Value, Default value: installation-defined)

See the COMPILERTARGET system command in the *System Commands Operations Reference Manual* for a complete list of target identifier values that are allowed.

The code file generated is optimized for the machine or group of machines identified by the primary identifier, subject to the compatibility constraints of the machine or group of machines identified by the secondary identifier. That is, the code file is optimized for the machine or machines listed as the primary identifier, but no operator is generated that is not supported by all the machines listed as secondary identifiers.

The TARGET option must appear before the first record that is not a compiler control record in the source program.

The default TARGET option is installation defined. For information on the way an installation defines the default target computer family, refer to the *System Commands Operations Reference Manual*.

TIME Option

— TIME _____|

(Type: Boolean, Default value: FALSE)

When TRUE, the TIME option causes trailer information, such as the number of errors, the number of code segments, and the compilation time, to be printed on the printer listing. Because this trailer information is also printed when the LIST option is TRUE, the value of the TIME option is effective only when the LIST option is FALSE.

USE Option

— USE —<text>_____|

(Type: Binder command)

During autobinding, the USE option is passed directly to the Binder program for analysis. The format and function of this option are the same as those of the Binder USE statement and are described in the *Binder Programming Reference Manual*.

USER Option

—<identifier>_____|

(Type: Boolean, Default value: FALSE)

If an identifier on a compiler control record is not recognized as one of the predefined options, it is considered to be a user option. A user option can be manipulated exactly like any other Boolean option; that is, it can be assigned values by using the SET, RESET, and POP keywords. In addition, it can be used in option expressions to assign values to standard Boolean options or to other user options.

VERSION Option

— $\left[\begin{array}{l} \langle \text{replace version} \rangle \\ \langle \text{append version} \rangle \end{array} \right]$ —————→

<replace version>

— VERSION — $\langle \text{version increment} \rangle$ — . — $\langle \text{cycle increment} \rangle$ —————→
 → $\left[\begin{array}{l} . \langle \text{patch number} \rangle \end{array} \right]$

<version increment>

— $\left[\begin{array}{l} \langle /1 \rangle \\ \langle \text{digit} \rangle \end{array} \right]$ —————→

<cycle increment>

— $\left[\begin{array}{l} \langle /2 \rangle \\ \langle \text{digit} \rangle \end{array} \right]$ —————→

<patch number>

— $\left[\begin{array}{l} \langle /3 \rangle \\ \langle \text{digit} \rangle \end{array} \right]$ —————→

<append version>

— VERSION — + — $\langle \text{version increment} \rangle$ — . — + — $\langle \text{cycle increment} \rangle$ —————→
 → $\left[\begin{array}{l} . \langle \text{patch number} \rangle \end{array} \right]$

(Type: value, Default value: 00.000.0000)

The VERSION option allows the user to specify an initial version number for a program, replace an existing version number, or append to an existing version number.

If the NEW option is TRUE, a VERSION option appears in the TAPE file, and the CARD file contains a <replace version> or <append version>, then the VERSION option record in the NEWTAPE file is updated with the version, cycle, and patch number in the last VERSION option record in the CARD file. The sequence number of the VERSION option in the CARD file must be less than the sequence number of the VERSION option in the TAPE file.

The functions COMPILETIME(20), COMPILETIME(21), and COMPILETIME(22) allow a programmer to access the current version, cycle, and patch numbers, respectively. For more information, see "COMPILETIME Function" in Section 5, "Expressions and Functions."

Examples

The following example sets the current version to 25.010.0010.

```
$ VERSION 25.010.0010
```

The following example increments the current version by 01 in the version increment, by 001 in the cycle increment, and assigns 0010 to the patch number. For example, if the existing version is 25.010.0005 and this VERSION option is compiled, the resulting version is 26.011.0010.

```
$ VERSION +01.+001.010
```

VOID Option

— VOID _____|

(Type: Boolean, Default value: FALSE)

When TRUE, the VOID option causes all source input other than compiler control records from both the TAPE and CARD files to be ignored by the compiler until the VOID option is assigned the value FALSE. The ignored source input is neither listed nor included in the updated source file, regardless of the values of the LIST option and the NEW option. Once the VOID option is assigned TRUE, it can be assigned FALSE only by a compiler control record in the CARD file.

VOIDT Option

— VOIDT _____|
 DELETE

(Type: Boolean, Default value: FALSE)

DELETE is a synonym that can be used for the VOIDT option.

If the MERGE option is TRUE and the VOIDT option is TRUE, all source input from the TAPE file is ignored by the compiler until the VOIDT option is assigned the value FALSE. Therefore, while the VOIDT option is TRUE, only primary source input from the file CARD is compiled. The ignored input is neither listed nor included in the updated source file, regardless of the values of the LIST option and the NEW option. Once the VOIDT option is assigned TRUE, it can be assigned FALSE only by a compiler control record in the CARD file. If the MERGE option is FALSE, the value of the VOIDT option is ignored.

WAITIMPORT Option

— WAITIMPORT —————|

(Type: Boolean, Default value: FALSE)

When TRUE, an IMPORTED EVENT that is declared in a CONNECTION BLOCK can be used as an event designator in an event list for a WAIT or WAITANDRESET statement. A value cannot be assigned to the WAITIMPORT option after the first syntactical item in a program.

WARNSUPR Option

— WARNSUPR —————|

(Type: Boolean, Default value: FALSE)

When TRUE, the WARNSUPR option prevents warning messages from being given.

WRITEAFTER Option

— WRITEAFTER —————|

(Type: Boolean, Default value: FALSE)

The WRITEAFTER option provides the ability to designate whether carriage control is performed before or after a write operation. The option can be assigned values repeatedly in order to select before-write or after-write carriage control for individual files and I/O statements.

Normally in ALGOL, carriage control is performed following a write operation.

Carriage control is done before a particular write operation if the WRITEAFTER option is TRUE when the I/O statement is compiled or if the I/O statement explicitly references a file whose declaration was compiled when the WRITEAFTER was TRUE.

For disk files, notwithstanding any value of the WRITEAFTER option, the carriage control action is always taken before a nonbinary write operation.

The WRITEAFTER option does not apply to direct files or direct I/O statements.

XDECS Option

— XDECS _____|

(Type: Boolean, Default value: TRUE if either the XREF option or the XREFFILES option is TRUE, FALSE otherwise)

When the compiler is saving cross-reference information because the XREF option or the XREFFILES option is TRUE, only identifiers declared while the XDECS option is TRUE are included in the cross-reference information. This option is assigned TRUE when the XREF option or the XREFFILES option is assigned TRUE and can be assigned a value as many times as desired. If the XDECS option is assigned the value TRUE and both the XREF option and the XREFFILES option are FALSE, the XDECS option is assigned FALSE and a syntax error is given.

For more information on cross-referencing, refer to “XREF Option” later in this section.

XREF Option

— XREF _____|
 └<linewidth>┘

<linewidth>

An integer between 72 and 160.

(Type: Boolean, Default value: FALSE)

Depending on the values of several related compiler control options, the compiler optionally generates cross-reference information containing an alphabetized list of identifiers that appear in the program and, for each identifier, the type of the item named by that identifier, the sequence number of the source input record on which the identifier is declared, the sequence numbers of the input records on which the identifier is referenced, and other relevant information. The following factors can be controlled through the use of compiler control options:

- Whether or not the compiler saves cross-reference information as it is processing the source input
- Which identifiers and which references to these identifiers are cross-referenced
- Whether or not the SYSTEM/XREFANALYZER program is initiated automatically by the compiler
- If SYSTEM/XREFANALYZER is automatically initiated, whether it is to produce printed output, disk files suitable for input to SYSTEM/INTERACTIVEXREF and the Editor, or both

The compiler saves cross-reference information if either the XREF option or the XREFFILES option or both options are TRUE. This information is discarded if any syntax errors occur during compilation. If used, these options should be assigned TRUE before any source input has been processed. Once assigned TRUE, these options cannot be assigned FALSE.

The identifiers and their references to be cross-referenced can be selected by the XDECS option and the XREFS option, respectively. Neither of these options can be used if cross-reference information is not being saved by the compiler (that is, if both the XREF option and the XREFFILES option are FALSE).

When the compiler is saving cross-reference information, this information is written to a disk file in raw form. Before this information can be printed or read by SYSTEM/INTERACTIVEXREF and the Editor, it must be analyzed by SYSTEM/XREFANALYZER. If the NOXREFLIST option is FALSE, the compiler automatically initiates SYSTEM/XREFANALYZER to process the raw cross-reference file. If the NOXREFLIST option is TRUE, SYSTEM/XREFANALYZER is not initiated, and the compiler's raw file is left on disk with the title "XREF/<code file name>", where <code file name> is the name of the object code file produced by the compiler; this file can be analyzed at a later time by running SYSTEM/XREFANALYZER directly.

If the compiler initiates SYSTEM/XREFANALYZER (that is, if the NOXREFLIST option is FALSE), the program produces either a printed listing or a pair of disk files suitable for the Editor and SYSTEM/INTERACTIVEXREF. If the XREF option is TRUE, a listing is produced. If the XREFFILES option is TRUE, the pair of disk files is produced; these files are titled XREFFILES/<code file name>/XREFS and XREFFILES/<code file name>/XDECS. If both the XREF option and the XREFFILES option are TRUE, both a listing and the disk files are produced.

If USERBACKUPNAME is set for the compiler's LINE file and the compiler calls XREFANALYZER to generate a crossreference listing, the file name of the LINE file used by XREFANALYZER is the file name of the compiler's LINE file with the node /XREFLIST appended.

The line width, in characters, of the listing produced by SYSTEM/XREFANALYZER can be specified by the <linewidth> construct. If not specified, the line width is 132.

User options are included in the cross-reference information.

XREFFILES Option

— XREFFILES —————|

(Type: Boolean, Default value: FALSE)

When TRUE, the XREFFILES option causes cross-reference information to be saved by the compiler and causes the SYSTEM/XREFANALYZER program, if the NOXREFLIST option is FALSE, to produce files that can be used by the Editor and SYSTEM/INTERACTIVEXREF. These files have the titles XREFFILES/<code file name>/XDECS and XREFFILES/<code file name>/XREFS, where <code file name> is the name of the object code file that the compiler is generating.

For more information on cross-referencing, refer to “XREF Option” earlier in this section.

XREFS Option

— XREFS —————|

(Type: Boolean, Default value: TRUE if either the XREF option or the XREFFILES option is TRUE, FALSE otherwise)

When the compiler is saving cross-reference information because the XREF option or the XREFFILES option is TRUE, only identifier references that are encountered while the XREFS option is TRUE are included in the cross-reference information. This option is assigned TRUE when the XREF option or the XREFFILES option is assigned TRUE, and it can be assigned a value as many times as desired. If the XREFS option is assigned the value TRUE when both the XREF option and the XREFFILES option are FALSE, then the XREFS option is assigned FALSE and a syntax error is given.

For more information on cross-referencing, refer to “XREF Option” earlier in this section.

Section 7

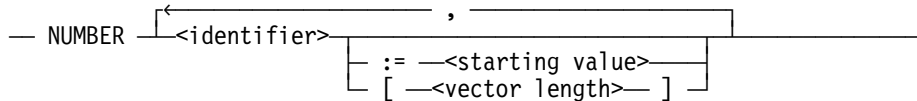
Compile-Time Facility

The compile-time facility is used to conditionally and iteratively compile ALGOL source data. This section describes the declaration and use of compile-time variables, compile-time identifiers, compile-time statements, an extension to the DEFINE declaration, and compiler control options that pertain to the compile-time facility.

The compile-time facility is available in DCALGOL, BDMSALGOL, and DMALGOL.

Compile-Time Variable

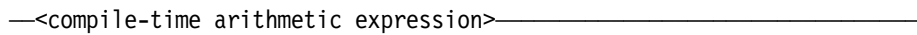
<compile-time variable declaration>



<number identifier>

An <identifier> that is associated with a compile-time variable in a compile-time variable declaration.

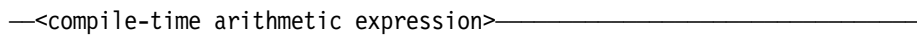
<starting value>



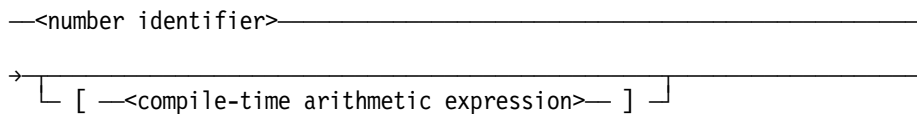
<compile-time arithmetic expression>

Any <arithmetic expression> that can be fully evaluated at compile time. A compile-time arithmetic expression consists of constants and <compile-time variable>s.

<vector length>



<compile-time variable>



An identifier declared in a compile-time variable declaration is a number variable, or an arithmetic compile-time variable. A compile-time variable represents a single-precision arithmetic value. It can be used wherever an arithmetic value is allowed and represents the value most recently assigned to it. The value of a compile-time variable can be changed at any time during compilation by using the compile-time 'LET statement. A compile-time variable can be declared with a starting value; if a starting value is not explicitly declared, the starting value is zero.

If a compile-time variable is declared with a vector length, a vector, or array, of compile-time variables is created. When an identifier declared in this way is used, it must be subscripted by a compile-time arithmetic expression with a value in the range 0 through (vector length - 1). The value of the vector length must be in the range 1 to 1023.

Compile-Time Identifier

<compile-time identifier>

—<identifier>— ' —<number identifier>—————|

A compile-time identifier can appear anywhere an identifier can be used, including declarations. No blank characters can appear between the identifier and the apostrophe ('). The created identifier consists of the identifier, followed by an apostrophe, followed by numeric characters corresponding to the value of the number identifier with leading zeros suppressed.

When a DEFINE declaration contains another DEFINE declaration, the formal parameters to the outer DEFINE cannot be used as <identifier> or <number identifier> components of the compile-time identifier.

Compile-Time Statements

<compile-time statement>

—<compile-time begin statement>	—————
—<compile-time define statement>	
—<compile-time for statement>	
—<compile-time if statement>	
—<compile-time invoke statement>	
—<compile-time let statement>	
—<compile-time thru statement>	
—<compile-time while statement>	

Compile-time statements begin with an apostrophe ('), which distinguishes them from other ALGOL constructs. They are recognized at a very primitive level in the compiler and can, therefore, appear almost anywhere (for example, between any two ALGOL language components).

The compile-time statements are intended to provide a method for altering the normal flow of compilation, primarily by conditional and iterative compilation.

Compile-time statements are terminated in the same manner as ALGOL statements.

Note that through the use of compile-time variables, a compile-time arithmetic or Boolean expression can be fully evaluated at compilation time and yet not have the same value each time it is evaluated.

BEGIN Statement

<compile-time begin statement>

— 'BEGIN —<compile-time text>— 'END —<end remark>—————|

<compile-time text>

Any ALGOL source text, including any complete compile-time statement.

A 'BEGIN statement delimits a portion of ALGOL source text. It is normally used in conjunction with 'FOR statements, 'IF statements, and 'THRU statements. If the 'BEGIN statement is executed, the compiler processes all the delimited text; otherwise, the compiler skips (ignores) the text. Anything following the 'END up to the first special character, END, ELSE, 'END, 'ELSE, or UNTIL is considered to be an <end remark> and is ignored.

DEFINE Statement

<compile-time define statement>

— 'DEFINE —<identifier>— = —<compile-time statement>—————|

<compile-time define identifier>

An <identifier> that is associated with a compile-time statement in a compile-time define statement.

The 'DEFINE statement declares an identifier to represent a compile-time statement.

The compile-time statement is processed when it is referenced by the identifier in a subsequent 'INVOKE statement.

FOR Statement

<compile-time for statement>

```

— 'FOR —<number identifier>— := _____→
→<compile-time arithmetic expression>— STEP _____→
→<compile-time arithmetic expression>— UNTIL _____→
→<compile-time arithmetic expression>— DO —<compile-time statement>—|
    
```

The 'FOR statement provides iterative compilation of ALGOL source input. The value of the compile-time arithmetic expression following STEP can be positive or negative but must not be equal to zero.

The action of this statement is similar to that of the noncompile-time FOR statement. One exception is that the compile-time arithmetic expressions following STEP and UNTIL are evaluated only once, at the beginning of the 'FOR statement, and are not reevaluated, even though their compile-time components can change value.

IF Statement

<compile-time if statement>

```

— 'IF —<compile-time Boolean expression>— THEN _____→
→<compile-time statement>—┐
└ 'ELSE —<compile-time statement>—┘
    
```

<compile-time Boolean expression>

Any <Boolean expression> that can be fully evaluated at compilation time. A compile-time Boolean expression consists of constants and <compile-time variable>s.

The 'IF statement provides conditional compilation of ALGOL source input.

If the value of the compile-time Boolean expression is TRUE, then the compile-time statement following THEN is processed; if it is FALSE, the compile-time statement following 'ELSE is processed, if present. In either case, compilation continues with the statement following the 'IF statement.

A <compile-time Boolean expression> can yield unexpected results if it is accessed when it has no valid value. This condition can be found in an IF statement that is nested in the ELSE part of another IF statement. Even though the outer IF evaluates to TRUE, the nested IF is also evaluated. This evaluation can produce conflicting results. If you have discovered this condition, enclose the IF statement in a 'BEGIN/'END pair. Once you have done this, the expression is evaluated only when the enclosing IF statement evaluates to FALSE.

INVOKE Statement

<compile-time invoke statement>

— 'INVOKE —<compile-time define identifier>—————|

The 'INVOKE statement causes the compile-time statement previously associated with the compile-time define identifier in a 'DEFINE statement to be processed.

LET Statement

<compile-time let statement>

— 'LET —<compile-time variable>— := —————→

→<compile-time arithmetic expression>—————|

The 'LET statement is used to modify the value of a compile-time variable. If the compile-time variable was declared using the <vector length> construct, it must be subscripted by a compile-time arithmetic expression with a value in the range 0 to (vector length - 1).

THRU Statement

<compile-time thru statement>

— 'THRU —<compile-time arithmetic expression>— DO —————→

→<compile-time statement>—————|

The 'THRU statement provides iterative compilation of ALGOL source input. The compile-time arithmetic expression must have a value greater than or equal to zero.

The compile-time statement following DO is processed <compile-time arithmetic expression> times. If this value is zero, the 'THRU statement is skipped.

WHILE Statement

<compile-time while statement>

— 'WHILE —<compile-time Boolean expression>— DO —————→

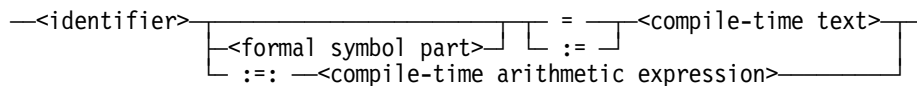
→<compile-time statement>—————|

The 'WHILE statement provides iterative compilation of ALGOL source input. The compile-time Boolean expression is evaluated at the beginning of the statement. If it is TRUE, the compile-time statement is processed, the compile-time Boolean expression is evaluated again, and this sequence is repeated. Whenever the compile-time Boolean expression is FALSE, the 'WHILE statement is finished, and compilation continues with the following statement.

Extension to the Define Declaration

The following extension to the DEFINE declaration is available when using the compile-time facility.

<definition>



If a DEFINE identifier is declared using the assignment operator (:=), the compile-time text is evaluated in the same manner as using the (=) syntax. This is not the preferred usage and is provided for compatibility reasons. If a DEFINE identifier is declared using the swap operator (:=:), then the compile-time arithmetic expression is evaluated once at the time of, and in the scope of, the DEFINE declaration. Otherwise, the compile-time items in the compile-time text are evaluated on each invocation of the define identifier.

Compile-time text does not include the compile-time DEFINE statement, when used as the definition in a DEFINE declaration. A compile-time DEFINE statement cannot be embedded within an ordinary DEFINE declaration.

When a DEFINE declaration contains another DEFINE declaration, the formal parameters to the outer DEFINE cannot be used as <identifier> or <number identifier> components of the compile-time identifier.

Compile-Time Compiler Control Options

The following compiler control options are available only in the compile-time facility. For more information on compiler control options and the printer listing file of the compiler, see Section 6, "Compiling Programs."

CTLIST Option

— CTLIST —————|

(Type: Boolean, Default value: FALSE)

If both the LIST option and the CTLIST option are TRUE, all input records processed are written to the printer listing. In particular, during iterative compile-time statements, input records that are compiled repeatedly are listed repeatedly. These input records are identified by an asterisk (*) just to the left of the sequence number. If the LIST option is TRUE but the CTLIST option is FALSE, input records are written to the printer listing only the first time they are compiled. The value of the CTLIST option is ignored if the LIST option is FALSE.

CTMON Option

— CTMON —————|

(Type: Boolean, Default value: FALSE)

If both the LIST option and the CTMON option are TRUE, all assignments to compile-time variables are monitored and written to the printer listing. The current value of a compile-time variable when it is referenced and the new value when it is changed are listed. If the LIST option is TRUE and the CTMON option is FALSE, this monitor information does not appear in the printer listing. The value of the CTMON option is ignored if the LIST option is FALSE.

CTTRACE Option

— CTRACE —————|

(Type: Boolean, Default value: FALSE)

If both the LIST option and the CTRACE option are TRUE, values of certain expressions that are components of compile-time statements are written to the printer listing. If the LIST option is TRUE and the CTRACE option is FALSE, this information does not appear in the printer listing. The value of the CTRACE option is ignored if the LIST option is FALSE.

LISTSKIP Option

— LISTSKIP —————|

(Type: Boolean, Default value: TRUE)

If both the LIST option and the LISTSKIP option are TRUE, all records are written to the printer listing whether or not they are skipped. Skipped records are denoted in the listing by the word SKIP to the right of the sequence number. If the LIST option is TRUE and the LISTSKIP option is FALSE, source records that are skipped by the compile-time facility are not written to the printer listing. The value of the LISTSKIP option is ignored if the LIST option is FALSE.

Section 8

Library Facility

The library facility is a feature that can be used to structure processes. A library is a program containing one or more library objects that can be accessed by other programs, which are referred to as calling programs. A library object is an object that is shared by a library and one or more user programs and is exported from the library using an export declaration. Exported procedures are called entry points into the library. Unlike a regular program, which is always entered at the beginning, a library can be entered at any entry point.

Libraries provide all the benefits of procedures plus the added advantages that they can be reused and shared by a number of programs. Consolidating logically related functions into a library can make programming easier and program structure more visible. A call on a procedure in a library is equivalent to a call on a procedure in the calling program.

Libraries offer the following improvements over binding:

- Interlanguage communication is significantly improved.
- Standard packages of functions, such as plotting and statistics, need not be copied into any calling programs.

Libraries offer the following improvements over installation intrinsics:

- A library can have its own global files, databases, transaction bases, and so on.
- Libraries can contain initialization and termination code.
- Libraries can themselves call other libraries.
- Individual users can create their own libraries without possessing special privileges.
- Libraries can be written in more languages than can the installation intrinsics.
- More than one version of a library can be in use at a time.

For information about connection libraries, see the *Task Management Programming Guide*.

Operating the Components of the Library Facility

The components of the library facility can be used to perform a variety of tasks. Following are descriptions of the library facility components and the features they offer.

Library Programs

A library program specifies entry points (procedures) for use by calling programs. It makes use of EXPORT declarations and FREEZE statements. A procedure in a library is specified to be an entry point by appearing in an EXPORT declaration. A library program becomes a library after execution of a FREEZE statement.

Calling Programs

A calling program calls entry points provided by a library. It uses LIBRARY declarations and PROCEDURE declarations that contain the <library entry point specification> construct.

A library can itself function as a calling program and can call other libraries. A library cannot reference itself, but circular references are allowed with the following restrictions:

- A circular linkage can be made only if all libraries are frozen and at least one of the libraries was frozen TEMPORARY or PERMANENT at the time it linked to its neighbor in the circle.
- A circular reference of indirectly provided objects is not allowed. An indirectly provided object is an object imported from another library and then exported.

Library Directories and Templates

The directory and the template are data structures built by the compilers. They contain the information used by the operating system to match entry points in a library with entry points declared in a calling program.

When a program exports entry points and contains a FREEZE statement, the object code file for that program contains a library directory. One directory exists for each block that contains an EXPORT declaration. After a library freezes (executes a FREEZE statement), only one directory is in effect until the library program finishes executing.

A library directory contains a description of all the entry points in the library. This description includes the following information:

- The name of the entry point
- The type of the entry point
- A description of the entry point parameters
- Information on how the entry point is provided. For more information, see "Linkage Provisions" later in this section.

When a program declares a library and entry points in that library, the object code file for the program contains a library template that describes the library and the declared entry points. One template exists for each library declared in the calling program. A template contains the following information:

- A description of the attributes of the library
- A description of all the entry points of the library that are declared by the program. Each description includes the following information:
 - The name of the entry point
 - The type of the entry point
 - A description of the entry point parameters

The SETACTUALNAME function can be used to change the name of an entry point in an unlinked library template. For more information, refer to “SETACTUALNAME Function” in Section 5, “Expressions and Functions.”

Library Initiation

On the first call on an entry point of a library or at an explicit linkage request, the calling program is suspended. The description of the entry point in the library template of the calling program is compared to the description of the entry point with the same name in the library directory associated with the referenced library.

If the entry point does not exist in the library, or if the two entry point descriptions are not compatible, a run-time error is given and the calling program is terminated. If the entry point exists and the two entry point descriptions are compatible, the operating system automatically initiates the library program (if it has not already been initiated). The library program runs normally until it executes a FREEZE statement, which makes the entry points available. All of the entry points of the library that are declared in the calling program are linked to the calling program, and the calling program resumes execution.

If a calling program declares an entry point that does not exist in the library, no error is generated when the library is initiated; however, a subsequent call on that entry point causes a *MISSING ENTRY POINT* run-time error, and the calling program is terminated.

A library can be specified to be permanent or temporary. A permanent library remains available until it is terminated either by the Operator Display Terminal (ODT) commands DS (Discontinue) or THAW, or by execution of a CANCEL statement. A temporary library remains available as long as users of the library remain. A temporary library that is no longer in use unfreezes and resumes running as a regular program.

The PERMANENT or TEMPORARY specifications of the FREEZE statement control the duration of a library. Any running program that executes a FREEZE statement becomes a library. When a library is initiated by explicitly running the library program instead of by calling an entry point, the FREEZE statement should specify PERMANENT. If TEMPORARY is specified, the library immediately unfreezes because it has no users. After a library unfreezes, it must not execute another FREEZE statement in an attempt to become a library again.

The CONTROL specification of the FREEZE statement controls the nature of the freeze. The program is set up as a permanent library, but after the freeze operation has been performed, control is transferred to the specified procedure, known as the control procedure. The control procedure must be untyped and must have no parameters.

Once the control procedure is in control, the library can keep track of the number of its users through the task attribute LIBRARYUSERS. The library can unfreeze itself by changing the task attribute STATUS to VALUE(GOINGAWAY). After this change, the library is equivalent to a thawing library. When the control procedure is exited, the library unfreezes if there are no users. If there are users, the library becomes an ordinary library, and a warning message is issued.

Because a library program initially runs as a regular program, the flow of execution can be such that the execution of a FREEZE statement is conditional and can occur anywhere in the program.

If a calling program causes a library program to be initiated and the library program terminates without executing a FREEZE statement (for example, because it was not actually a library program and, thus, had no FREEZE statement), the attempted linkage to the library entry points cannot be made, and the calling program is terminated.

Linkage to a library can be requested explicitly by using the LINKLIBRARY function. For more information, see "LINKLIBRARY Function" in Section 5, "Expressions and Functions."

Linkage Provisions

Entry points declared in a calling program are linked to corresponding entry points provided by a library in one of three ways:

- Directly
- Indirectly
- Dynamically

The library program specifies the form of linkage. Indirect and dynamic linkages allow linkage to be established to libraries other than the library specified by the calling program. The calling program can control the library invocation to which it is linked only by specifying the object code file title or the function name of the library or, for dynamic linkage, by specifying the LIBPARAMETER library attribute. Depending on the value of the LIBACCESS library attribute, the TITLE attribute or FUNCTIONNAME attribute is used to specify the object code file title of the library. For more information about library attributes, see "Library Attributes" later in this section.

Direct linkage occurs when the library program contains the procedure that is named in the EXPORT declaration of the library.

Indirect linkage occurs when the library program exports a procedure that is declared as an entry point of another library. The operating system then attempts to link the calling program to this second library, which can provide the entry point directly, indirectly, or dynamically.

Dynamic linkage allows a library program to determine at link time which library task the calling program will be linked to. The library program must provide a selection procedure that accepts the value of the LIBPARAMETER library attribute as a parameter. Based on the value of LIBPARAMETER, the selection procedure selects and initiates a library task. The selection procedure must also accept, as a second parameter, a procedure. This procedure, which is provided by the operating system to verify that the library task is valid and complete, must be called before the selection procedure is exited. The operating system calls the selection procedure at link time. For a more detailed explanation and examples of libraries that provide dynamic linkage, see "Library Examples" later in this section.

The restrictions on the complexity of indirect and dynamic linkages are as follows:

- Eventually, a library must provide the entry point directly.
- The chain of referenced libraries must never become circular.

Discontinuing Linkage

A program can delink from a library program by using either the CANCEL statement or the DELINKLIBRARY function.

The CANCEL statement causes the library program to unfreeze and resume running as a regular program. The CANCEL statement can be used only on PRIVATE and SHARED BY RUN UNIT libraries.

The DELINKLIBRARY function affects only the linkage the program executing the DELINKLIBRARY function and the specified library. Any other programs linked to the specified library are not affected. For more information, see "DELINKLIBRARY Function" in Section 5, "Expressions and Functions."

Error Handling

Any fault caused and ignored by a procedure in a library that is invoked by a calling program is treated as a fault in the calling program. If ignored by the calling program, this fault causes the calling program to be terminated but has no effect on the status of the library.

If a library program faults or is otherwise terminated before executing a FREEZE statement, then all calling programs that are waiting to link to that library program are also terminated.

If a library is terminated while calling programs are linked to it, those calling programs are also terminated.

The first call on an entry point in a library causes library linkage to be made. In this phase, an attempt is made to locate and establish links to all entry points declared by the calling program. If an entry point declared in the calling program does not exist in the library, its linkage cannot be established, and any subsequent calls to that entry point result in a *MISSING ENTRY POINT* error. This error continues to occur whenever a calling program links to that instance of the library and calls that entry point. Thus, it is advisable to remove that instance of the library (by either a THAW or DS (Discontinue) ODT command) and initiate a correct version of the library. For more information, refer to the *System Commands Operations Reference Manual* for a description of these commands.

Creating Libraries

A library program is created by using the EXPORT declaration to declare procedures to be exported as entry points, and by using the FREEZE statement. The duration of a library program following initiation is controlled by the TEMPORARY or PERMANENT specification of the FREEZE statement. The allowed sharing of a library program is controlled by the SHARING compiler control option.

Users of a library can be restricted through the normal file access features provided by the system. The allowed simultaneous usage of a library can be specified by the creator of the library at compilation time through the SHARING compiler control option. The library sharing can be PRIVATE, SHARED BY ALL, SHARED BY RUN UNIT, or DONT CARE.

PRIVATE

A separate instance of the library is started for each invocation of the library. Any changes made to global items in the library by the program unit (block, procedure, or external task) invoking the library apply only to that particular calling program.

SHARED BY ALL

All invocations of the library share the same instance of the library. Any changes made to global items in the library by a program unit that has invoked the library apply to all users of that library.

SHARED BY RUN UNIT

A run unit consists of a program and all libraries that are called, either directly or indirectly, by that program. A program, in this context, excludes both a library that is not frozen and any tasks that are initiated by the program (that is, a process family is not a run unit). All invocations of a library within a run unit share the same instance of the library.

Note that a library is its own run unit until it freezes. For example, program P initiates library A and, before library A freezes, it in turn initiates library B. Now library B is in library A run unit, not in program P run unit. Had library A initiated library B after freezing, both library A and library B would be in program P run unit.

DONT CARE

The operating system determines the sharing. This determination is unknown to all users invoking the library. The default value of the SHARING compiler control option is DONT CARE.

Referencing Libraries

To use a library, the calling program does the following:

- Declares the library in a LIBRARY declaration, specifying the attributes of the library
- Declares the entry points of the library in PROCEDURE declarations with library entry point specification parts

When an entry point is invoked or at an explicit linkage request, the operating system automatically creates the library linkage. If the library program has not already been initiated, the operating system initiates it; then, when the library is frozen, the operating system links the library to the calling program. The operating system attempts to make linkages to all entry points referenced in a library at the time that the library is first invoked.

The LINKLIBRARY function can be used to determine whether or not the calling program is currently linked to, or is capable of being linked to, a particular library program. If the calling program is not currently linked but is capable of being linked, the linkage is performed. During the linkage process, an attempt is made to link to every entry point exported from the library whose name matches an entry point declared in the calling program. Only those names that match are checked for correct function type, number of parameters, and parameter types. Therefore, the LINKLIBRARY function does not check that every entry point declared in the calling program is also exported from the library. For more information, see "LINKLIBRARY Function" in Section 5, "Expressions and Functions."

The CANCEL statement and the DELINKLIBRARY function can be used to terminate the linkage between a calling program and a library. The CANCEL statement causes the library to unfreeze and resume running as a regular program regardless of whether it is temporary or permanent. Only PRIVATE libraries or SHARED BY RUN UNIT libraries can be canceled. The DELINKLIBRARY function has no effect on any other users of the library. For more information, see "DELINKLIBRARY Function" in Section 5, "Expressions and Functions."

The SETACTUALNAME function determines whether or not the name of a particular library entry point can be changed in the template to a particular character string and, if possible, makes the change. The name of an entry point of a linked library cannot be modified. Therefore, a linked library must be delinked before the SETACTUALNAME function can be called to change the name of any of its entry points. For more information, see "SETACTUALNAME Function" in Section 5, "Expressions and Functions."

Library Attributes

Libraries, like files, have attributes that can be assigned values and tested programmatically.

The calling program can change library attributes dynamically; however, since the operating system ignores any changes made to library attributes of linked libraries, these changes must not be made while the program is linked to the library. Any library attribute changes must be made before the calling program has linked to the library or after the library has been delinked from the program.

The pointer-valued and string-valued attributes can be used both as string or pointer primary attributes and as string or pointer expressions. For example, the following attributes are available:

```
LIB.TITLE := <EBCDIC literal>;  
  
LIB.INTNAME := <string variable>;  
  
REPLACE POINTER BY LIB.TITLE;  
  
REPLACE LIB.TITLE BY <EBCDIC literal>;  
  
REPLACE LIB.INTNAME BY <string variable>;  
  
REPLACE LIB.FUNCTIONNAME BY <pointer> FOR <length>;
```

The following paragraphs describe the library attributes. The first line of each description tells whether the attribute can be read or written or both, and gives the type and the default value, if any.

FUNCTIONNAME

(Read/Write, EBCDIC pointer-valued)

FUNCTIONNAME specifies the system function name used to find the target object code file for the library. For more information, refer to the LIBACCESS attribute in this section.

INTERFACENAME

(Read/Write, EBCDIC string-valued)

INTERFACENAME identifies a particular connection library within a program.

INTNAME

(Read/Write, EBCDIC pointer-valued)

INTNAME specifies the internal identifier for the library.

LIBACCESS

(Read/Write, mnemonic-valued, Default value: BYTITLE)

LIBACCESS specifies the way a library object code file is accessed when a library is called. LIBACCESS has one of the following mnemonic values:

- **BYTITLE**
The TITLE attribute of the library is used to locate the object code file.
- **BYFUNCTION**
The value of the FUNCTIONNAME attribute of the library is used to access the operating system library function table, and the object code file associated with that FUNCTIONNAME is used.
- **BYINITIATOR**
The object code file that initially caused the library to freeze is used. Note that BYINITIATOR causes a circular linkage between two libraries.

LIBPARAMETER

(Read/Write, EBCDIC string-valued)

LIBPARAMETER is used to transmit information from the calling program to the selection procedures of libraries that provide entry points dynamically.

TITLE

(Read/Write, EBCDIC pointer-valued)

TITLE specifies the object code file title of the library. For more information, refer to "LIBACCESS" earlier in this section.

Example 1 of Library Attributes

The following program shows the use of the LIBACCESS, TITLE, and FUNCTIONNAME library attributes.

```
BEGIN
  % LIBRARY1, DECLARED BELOW, IS NOT A SUPPORT LIBRARY.  ITS LIBACCESS
  % ATTRIBUTE, SET TO BYTITLE, INDICATES THAT THE TITLE ATTRIBUTE
  % IS USED TO LOCATE THE LIBRARY'S CODE FILE.

  LIBRARY LIBRARY1(TITLE="OBJECT/LIBRARY1.",LIBACCESS=BYTITLE);
  REAL PROCEDURE PROC1;
    LIBRARY LIBRARY1;
  REAL PROCEDURE PROC2;
    LIBRARY LIBRARY1;
  REAL PROCEDURE PROC3;
    LIBRARY LIBRARY1;

  % THE LIBRARY DECLARED BELOW IS A SUPPORT LIBRARY.  ITS LIBACCESS
  % ATTRIBUTE, SET TO BYFUNCTION, INDICATES THAT THE FUNCTIONNAME
  % ATTRIBUTE OF THE LIBRARY IS LOOKED UP IN THE MCP LIBRARY FUNCTION
  % TABLE, AND THE CODE FILE ASSOCIATED WITH THE FUNCTIONNAME,
  % SYSTEMLIB, IS USED.

  LIBRARY LIBRARY2(FUNCTIONNAME="SYSTEMLIB.",LIBACCESS=BYFUNCTION);
  PROCEDURE SYSTEMLIBPROC;
    LIBRARY LIBRARY2;
  % EXECUTABLE STATEMENTS FOLLOW.
END.
```

Example 2 of Library Attributes

Family substitution can occur when the library is linked by title. In this situation, it is desirable to link by function to an initialized library or to control the family substitution as shown in the following program sample.

```
% THE FOLLOWING SEQUENCE LINKS A LIBRARY TO A CODE FILE ON "DISK"
% WHEN FAMILY SUBSTITUTION KEEPS THE LIBRARY FROM LINKING.

REPLACE POINTER(SAVFAM[0],8) BY MYSELF.FAMILY;
REPLACE MYSELF.FAMILY BY ".";
LINKLIBRARY(LIB1);
REPLACE MYSELF.FAMILY BY POINTER(SAVEFAM[0],8);
```

For information on the FAMILY attribute, see the *Task Attributes Programming Reference Manual*.

Entry Point Type Matching

Library entry points can be either typed or untyped, and they can have parameters. Type matching is performed on entry points during library linkage. If the description of an entry point in the template does not match the description of the entry point in the directory, the linkage is not made, and the calling program is terminated. Matching is based on several factors: the procedure type, the number of parameters, the parameter types, and the ways in which the parameters are passed. Parameters are passed as call-by-value, call-by-reference, or call-by-name.

An ALGOL library entry point can be any of the following:

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

The parameters of an ALGOL library entry point can be any of the following:

- Boolean variable, array, or direct array
- Double variable, array, or direct array
- Real variable, array, or direct array
- Integer variable, array, or direct array
- Complex variable or array
- EBCDIC string variable or array
- ASCII string variable or array
- Hexadecimal string variable or array
- EBCDIC character array or direct array
- ASCII character array or direct array
- Hexadecimal character array or direct array
- Event variable or array
- Task variable or array
- File or direct file

- Pointer
- A fully-specified procedure (declared using FORMAL) with the above restrictions on its possible parameters and type

Parameter Passing

If a library program declares a parameter to be call-by-name, the calling program can declare the parameter to be call-by-name, call-by-reference, or call-by-value. If a library program declares a parameter to be call-by-reference, the calling program can declare the parameter to be call-by-name, call-by-reference, or call-by-value. If a library program declares a parameter to be call-by-value, the calling program can declare the parameter only to be call-by-value. For pointer parameters, the parameter passing mode of the library program and the client program must match exactly.

Table 8–1 shows the parameter passing rules. In the table, the legal combinations of parameter passing modes are marked with a P for pointer parameters and an X for all other parameter types.

Table 8–1. Parameter Passing Rules

Library Program	Calling Program		
	Name	Reference	Value
Name	X	X	X
Reference	X	P	X
Value			P

In ALGOL programs, parameters are declared to be either call-by-value, call-by-reference, or call-by-name. In ALGOL library programs, parameters to entry points that are declared to be call-by-value are described in the directory as call-by-value. Parameters declared to be call-by-reference are described in the directory as call-by-reference. Parameters declared to be call-by-name are described in the directory as call-by-reference, except for formal procedures and Boolean, complex, double, integer, and real variables, which are described in the directory as call-by-name.

In ALGOL calling programs, parameters to entry points that are declared to be call-by-value are described in the template as call-by-value. Parameters declared to be call-by-reference are described in the template as call-by-reference. Parameters declared to be call-by-name are described in the template as call-by-reference, except for Boolean, complex, double, integer, and real variables, which are described in the template as call-by-name.

An array parameter in ALGOL that is declared with any of its lower bounds as an asterisk (*) lower bound is described in the template or directory as N+1 parameters, where N is the number of dimensions of the formal array. The first parameter is the array itself, followed by the N lower bounds described as call-by-value integer variables.

Library Examples

This topic provides examples of libraries and calling programs that call these libraries.

Library: OBJECT/FILEMANAGER/LIB

The following library program illustrates dynamic linkage. This library provides a set of file management routines. The users of this library assign the title of the file to be used to the LIBPARAMETER attribute. LIBPARAMETER is then used at link time to determine to which library task the user is to be linked.

```

$ SHARING = PRIVATE

BEGIN
  TASK ARRAY LIBTASKS[0:10];      % PROVIDES UP TO 11 DIFFERENT LIBRARY
                                  % TASKS
  STRING ARRAY FILETITLES[0:10]; % LIBPARAMETER FOR EACH OF THE TASKS

  PROCEDURE FILEMANAGER(TASKINDEX);
  VALUE TASKINDEX;
  INTEGER TASKINDEX;

      BEGIN
        PROCEDURE READFILE;
          BEGIN
            .
            .
            .
          END READFILE;
        PROCEDURE WRITEFILE;
          BEGIN
            .
            .
          END WRITEFILE;

        EXPORT READFILE,WRITEFILE;
        FREEZE(TEMPORARY);
        FILETITLES[TASKINDEX] := ".";
        END FILEMANAGER;

  PROCEDURE SELECTION(USERSFILE,MCPCHECK);
  VALUE USERSFILE;
  EBCDIC STRING USERSFILE;
  PROCEDURE MCPCHECK(T); TASK T; FORMAL;

```

```
BEGIN
INTEGER TASKINDEX;
BOOLEAN FOUND;
% LOOK AT ALL THE FILETITLES, CHECKING TO SEE IF A LIBRARY TASK
% HAS ALREADY BEEN INITIATED FOR FILE TITLE USERSFILE.

WHILE NOT FOUND AND (TASKINDEX LEQ 10) DO
  BEGIN
  IF FILETITLES[TASKINDEX] = USERSFILE THEN
    FOUND := TRUE
  ELSE
    TASKINDEX := *+1;
  END;

IF NOT FOUND THEN
  BEGIN
  % A LIBRARY TASK DOES NOT EXIST FOR THIS FILE TITLE.
  WHILE NOT FOUND DO % FIND AN UNUSED TASK
  BEGIN
  TASKINDEX := 0;
  WHILE NOT FOUND AND (TASKINDEX LEQ 10) DO
    IF LIBTASKS[TASKINDEX].STATUS LEQ 0 THEN
      FOUND := TRUE
    ELSE
      TASKINDEX := *+1;
  IF NOT FOUND THEN
    % WAIT A SECOND AND MAYBE A LIBRARY TASK WILL GO TO EOT.
    WAIT((1));
  END;

  PROCESS FILEMANAGER(TASKINDEX) [LIBTASKS[TASKINDEX]];
  WHILE LIBTASKS[TASKINDEX].STATUS NEQ VALUE(FROZEN) DO
    WAIT((1));
  FILETITLES[TASKINDEX] := USERSFILE;
  END;

  MCPCHECK(LIBTASKS[TASKINDEX]);
  END SELECTION;

PROCEDURE READFILE;
  BY CALLING SELECTION;
PROCEDURE WRITEFILE;
  BY CALLING SELECTION;

EXPORT READFILE,WRITEFILE;
FREEZE(TEMPORARY);
END.
```

At library linkage time, the procedure SELECTION is invoked. SELECTION accepts two parameters, USERSFILE and MCPCHECK.

USERSFILE is passed the value of the LIBPARAMETER attribute, which was assigned a value by the calling program. The SELECTION procedure checks to see if a library task has been initiated for the file specified by USERSFILE. If it has, then the calling program is linked to that task. If no library task exists for that file, then a new library task is initiated and the calling program is linked to it.

Only one call is made on the SELECTION procedure per linkage; that is, all links to entry points in this library are resolved during linkage. Therefore, any changes made to any library attributes after linkage is made are ignored. The attributes can be changed if the library is delinked.

MPCHECK is a procedure that is provided by the operating system and must be called before exiting the SELECTION procedure. The parameter to MPCHECK is the task variable of the library task to which the calling program is to be linked. MPCHECK verifies that the task is valid and complete. The actual library linkage is not performed until SELECTION has been exited.

Calling Program #1

The following calling program invokes the dynamic library OBJECT/FILEMANAGER/LIB previously described. In this example, the user has set the LIBPARAMETER attribute to MYFILE, which is the name of the file to be accessed. At library linkage time, which occurs during the call on READFILE, the library procedure SELECTION is invoked. All links to the library's entry points are resolved during linkage. Changes to the library attributes are ignored after linkage is made. However, the program can delink from the library and change the library attributes before any relinking.

```
BEGIN
  LIBRARY L (TITLE="OBJECT/FILEMANAGER/LIB.");
  PROCEDURE READFILE;
    LIBRARY L;
  PROCEDURE WRITEFILE;
    LIBRARY L;

  L.LIBPARAMETER := "MYFILE";

  READFILE;                                % LINKAGE IS MADE

  CANCEL(L);                                % LINKAGE IS BROKEN

  L.LIBPARAMETER := "OTHERFILE"; % LIBPARAMETER CAN BE CHANGED
                                     % BECAUSE THE LIBRARY HAS BEEN
                                     % CANCELED.

  WRITEFILE;                                % LINKAGE IS MADE AGAIN AND NEW
                                     % VALUE OF LIBPARAMETER IS USED

END.
```

Library: OBJECT/SAMPLE/LIBRARY

The following ALGOL 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, ";
        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,8) BY
    " - SAMPLE LIBRARY STARTED",
    " " FOR 94;
  DATEANDTIME(MSG,60);
  DISPLAY(MSG);
  FREEZE(TEMPORARY);
  REPLACE POINTER(MSG,8)+19 BY "ENDED ";
  DATEANDTIME(MSG,60);
  DISPLAY(MSG);
END.

```

In this library program, two procedures are exported, making them entry points that can be called by calling programs. The two procedures, FACT and DATEANDTIME, are contained within the library program, so they are provided directly.

In the EXPORT declaration, the procedure DATEANDTIME is given the name DAYTIME in an AS clause. In the directory built for this library, the name of this entry point will be DAYTIME. Calling programs must use the name DAYTIME to call this entry point.

Library: OBJECT/SAMPLE/DYNAMICLIB

The following ALGOL library, compiled as OBJECT/SAMPLE/DYNAMICLIB, illustrates dynamic and indirect library linkage. This library references the library OBJECT/SAMPLE/LIBRARY previously described.

```

BEGIN
  TASK LIB1TASK,LIB2TASK;
  LIBRARY SAMLIB(TITLE="OBJECT/SAMPLE/LIBRARY.");

  % ENTRY POINT PROVIDED INDIRECTLY
  INTEGER PROCEDURE FACT(N);
  INTEGER N;
  LIBRARY SAMLIB;

  % POSSIBLY CALLED BY THE SELECTION PROCEDURE
  PROCEDURE DYNLIB1;
  BEGIN % PRINTS DATE WITH TIME
  LIBRARY SAMLIB(TITLE="OBJECT/SAMPLE/LIBRARY.");
  % ENTRY POINT PROVIDED INDIRECTLY
  PROCEDURE DAYTIME(TOARRAY,WHERE);
  ARRAY TOARRAY[*];
  INTEGER WHERE;
  LIBRARY SAMLIB;

  EXPORT DAYTIME;
  FREEZE(TEMPORARY);
  END; % OF DYNLIB1

```

```
% POSSIBLY CALLED BY THE SELECTION PROCEDURE
PROCEDURE DYNLIB2;
  BEGIN % PRINTS DATE WITHOUT TIME.
    % ENTRY POINT PROVIDED DIRECTLY
    PROCEDURE DAYTIME(TOARRAY,WHERE);
      ARRAY TOARRAY[*];
      INTEGER WHERE;

      BEGIN
        REAL T;
        T := TIME(7);
        REPLACE POINTER(TOARRAY,8) + WHERE
          BY T.[35:6] FOR 2 DIGITS, "-",
            T.[29:6] FOR 2 DIGITS, "-",
            T.[47:12] FOR 4 DIGITS;
        END; % OF DAYTIME

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

% THE SELECTION PROCEDURE
PROCEDURE THESELECTIONPROC(LIBPARSTR,NAMINGPROC);
  VALUE LIBPARSTR;
  EBCDIC STRING LIBPARSTR;
  PROCEDURE NAMINGPROC(LIBTASK); TASK LIBTASK; FORMAL;

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

```

% ENTRY POINT PROVIDED DYNAMICALLY
PROCEDURE DAYTIME(TOARRAY,WHERE);
ARRAY TOARRAY[*];
INTEGER WHERE;
    BY CALLING THESELECTIONPROC;

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

```

Calling Program #2

The following calling program invokes OBJECT/SAMPLE/DYNAMICLIB, the library described previously.

```

BEGIN
    LIBRARY MYLIB(TITLE="OBJECT/SAMPLE/DYNAMICLIB.");
    INTEGER PROCEDURE FAKTORIAL(N);
    INTEGER N;
        LIBRARY MYLIB(ACTUALNAME="FACT");

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

    REAL T;
    ARRAY DATIME[0:120];

    MYLIB.LIBPARAMETER := "WITH TIME";
    REPLACE POINTER(DATIME[0],8) BY
        " 13 FACTORIAL IS ",
        FAKTORIAL(13) FOR 12 DIGITS,
        " - ";
    DAYTIME(DATIME[*],40);
    DISPLAY(DATIME[0]);
END.

```

In this program, the declaration of the library entry point FAKTORIAL specifies that the ACTUALNAME of the entry point is FACT. In the template built for the library MYLIB, the name of this entry point is FACT, so for linkage to occur, the directory of the library OBJECT/SAMPLE/DYNAMICLIB must contain an entry point named FACT. However, within the program, the entry point is referred to as FAKTORIAL.

Library: MCPSUPPORT

The following subsections describe several procedures that are available in the MCPSUPPORT library. Refer to the *Master Control Program (MCP) System Interfaces Programming Reference Manual* for descriptions of additional MCPSUPPORT library procedures.

CONVERTDATEANDTIME Procedure

You can use the CONVERTDATEANDTIME procedure to convert universal times and dates to local times and dates and to convert local times and dates to universal times and dates. To use this procedure, use the following declaration:

```
INTEGER PROCEDURE CONVERTDATEANDTIME (WHICH, UD, UT, LD, LT, TZOFF);
      VALUE WHICH,          TZOFF;
      INTEGER WHICH, UD, UT, LD, LT, TZOFF;
LIBRARY MCPSUPPORT;
```

The parameters for CONVERTDATEANDTIME are:

Parameter Meaning

WHICH	Value of 1 to convert local date and time to universal date and time. Value of 2 to convert universal date and time to local date and time.
UD	Universal Julian date in form yyyyddd. The year (yyy) is 1900 less than the actual year. For example, 099356 represents December 31, 1999, and 100001 January 1, 2000. When the WHICH parameter equals 1, the procedure returns the universal date in the UD parameter. When the WHICH parameter equals 2, you must supply the universal date to be converted in the UD parameter.
UT	Universal time of day in microseconds. When the WHICH parameter equals 1 the procedure returns the universal time in the UT parameter. When the WHICH parameter equals 2 you must supply the universal time to be converted in the UT parameter.
LD	Local Julian date in form yyyyddd. The year (yyy) is 1900 less than the actual year. For example, 099356 represents December 31, 1999, and 100001 January 1, 2000. When the WHICH parameter equals 2 the procedure returns the local date in the LD parameter. When the WHICH parameter equals 1, you must supply the local date to be converted in the LD parameter.

Parameter Meaning

LT	Local time of day in microseconds. When the WHICH parameter equals 2 the procedure returns the local time in the LT parameter. When the WHICH parameter equals 1 you must supply the local time to be converted in the LT parameter.
TZOFF	Offset of universal time from local time. TZOFF is expressed in minutes. A negative value indicates that local time is earlier than universal time. A positive value indicates that local time is later than universal time. TZOFF must be in the range $-24*60$ through $+24*60$.

The CONVERTDATEANDTIME procedure returns a zero (0) value if the conversion is successful. It returns a negative value if one or more of the parameters are out of bounds (if the input time parameter, in LT or UT, is less than 0 or greater than $24*60*60*1000000$).

TIMEZONENAME Procedure

The TIMEZONENAME procedure returns the text name or abbreviation for a given time zone number. To use the TIMEZONENAME procedure use the following declaration:

```
INTEGER PROCEDURE TIMEZONENAME(WHICH, TZ, LANGL, LANGP, TZL, TZP);
                                VALUE WHICH, TZ, LANGL, LANGP,      TZP;
                                INTEGER WHICH, TZ, LANGL,          TZL;
                                POINTER                                LANGP,      TZP;
LIBRARY MCPSUPPORT;
```

The parameters for TIMEZONENAME are:

Parameter Meaning

WHICH	If equal to 1, the procedure returns the time zone name. If equal to 2, the procedure returns the time zone abbreviation.
TZ	Time zone number in the range 1 through 88.
LANGL	Zero or length of MLS language in LANGP parameter. If LANGL is zero, then TIMEZONENAME uses the task language or the SYSTEMELANGUAGE.
LANGP	If LANGL parameter is greater than 0, then pointer is to MLS language in which the time zone name or abbreviation should be returned.
TZL	Length of time zone name or abbreviation returned by the procedure.
TZP	Pointer to array where the procedure returns the time zone name or abbreviation.

TIMEZONENAME returns the same error values as the MESSAGESEARCHER statement.

TIMEZONEOFFSET Procedure

The TIMEZONEOFFSET procedure returns the offset of universal time from local time for a given time zone number. To use the TIMEZONEOFFSET procedure, use the following declaration:

```
INTEGER PROCEDURE TIMEZONEOFFSET (TZ_NUMBER);
                                VALUE  TZ_NUMBER;
                                INTEGER TZ_NUMBER;
LIBRARY MCPSUPPORT;
```

The parameter to TIMEZONEOFFSET is:

Parameter	Meaning
TZ_NUMBER	Time zone number in the range 1 to 88

TIMEZONEOFFSET returns the number of minutes difference, positive or negative, between the given time zone and universal time. The result is negative if the time of day in the time zone is earlier than universal time. The result is positive if the time of day in the time zone is later than universal time. The result is 0 (zero) if the time of day in the time zone is equal to universal time. If the TZ_NUMBER parameter is not valid then TIMEZONEOFFSET returns 9999.

Using the EVENT_STATUS Entry Point

The following example provides a possible structure on which to build an ALGOL program using the EVENT_STATUS entry point of the MCPSUPPORT library.

The EVENT_STATUS function is exported from the operating system as an unprotected REAL function that, when given an event, returns the following information. All undefined bits are zero.

Bits	Meaning
[42:39]	Lock owner, if event is unavailable
[3: 2]	Event usage: 0 = Normal event 1 = Event reserved for use by operating system 2 = Interrupt attached to event 3 = Event reserved for use by operating system
[1: 1]	Event unavailable (procured)
[0: 1]	Event happened (caused)

Using the EVENT_STATUS function is the only safe means of finding the owner of an event. The value returned in the Lock Owner field is the same kind of value that is returned by the PROCESSID function. In fact, if the task that owns the lock referenced the PROCESSID function, the value returned would be the same as the value in the Lock Owner field. UNSAFE NEWP programs that manipulate events directly should be avoided since different systems use varying event formats.

```

BEGIN
  TASK TASKA, TASKB;
  REAL STACKA, STACKB, EVSTATUS;
  EVENT XYZ_LOCK;

%THE FOLLOWING DECLARATIONS MAKE IT POSSIBLE TO USE THE EVENT_STATUS
%FUNCTION IN AN ALGOL PROGRAM. NOTE THAT THE LIBACCESS ATTRIBUTE IS
%EQUAL TO BYFUNCTION.

LIBRARY MCPSUPPORT (LIBACCESS = BYFUNCTION);

REAL PROCEDURE EVENT_STATUS (EV);
  EVENT EV;
  LIBRARY MCPSUPPORT;

% THE FIELDS FOR EVENT_STATUS RESULT ARE:

DEFINE  ES_OWNERSTACKF = [42:39]#,
        ES_EVENTUSAGEF = [3:2]#,
        ES_UNAVAILABLEF = [1:1]#,
        ES_HAPPENEDF = [0:1]#;
PROCEDURE A;
BEGIN
  STACKA := PROCESSID;

  PROCURE(XYZ_LOCK);

  LIBERATE(XYZ_LOCK);
END;

PROCEDURE B;
BEGIN
  STACKB := PROCESSID;

  PROCURE(XYZ_LOCK);

  LIBERATE(XYZ_LOCK);
END;

```

```
PROCESS A [TASKA];
PROCESS B [TASKB];

EVSTATUS := EVENT_STATUS(XYZ_LOCK);
IF EVSTATUS.ES_OWNERSTACKF = STACKA THEN
  BEGIN
    % XYZ_LOCK WAS OWNED BY TASKA AT THE TIME
    % THE EVENT_STATUS CALL WAS MADE.
  END
ELSE
  IF EVSTATUS.ES_OWNERSTACKF = STACKB THEN
    BEGIN
      % XYZ_LOCK WAS OWNED BY TASKB AT THE TIME
      % THE EVENT_STATUS CALL WAS MADE.
    END;
  END;

WHILE (TASKA.STATUS > VALUE(TERMINATED))
  AND (TASKB.STATUS > VALUE(TERMINATED)) DO
  WAITANDRESET(MYSELF.EXCEPTIONEVENT);
END.
```


Section 9

Internationalization

Internationalization refers to the Unisys software, firmware, and hardware features that enable you to develop and run application systems that can be customized to meet the needs of a specific language, culture, or business environment. The internationalization features provide support for several character sets, different international business and cultural conventions, extensions to data communications protocols, and the ability to use one or more natural languages concurrently.

This section describes the internationalization features you can use to customize an application for the language and conventions of a particular locality. Using these features to write or modify an application is termed *localization*. The MultiLingual System (MLS) environment enables you to localize your applications. Some of the localization methods included in the MLS environment include translating messages to another language, choosing a particular character set to be used for data processing, and defining date, time, number, and currency formats for a particular business application.

In addition to the information described in this section, refer to the *MultiLingual System (MLS) Administration, Operations, and Programming Guide* for information. The *MLS Guide* provides definitions for, and detailed information about, the CCS versions, character sets, languages, and conventions provided by Unisys. It also provides procedures for setting system values for the internationalization features.

Accessing the Internationalization Features

You can use the following two methods separately or together to localize your programs. Both of these methods are fully described later in this section.

- ALGOL provides language syntax that eases localization of your program. For example, picture clauses can be used to specify currency symbols.
- The system provides a system library, CENTRALSUPPORT, that contains procedures for localizing a program. The procedures can be accessed through calls. When a call occurs, input parameters describe the type of information that is needed or the action that is to be performed. Output parameters are returned with the result of the procedure call. For example, a program can call the procedure CNV_FORMATTIME to format the time according to the language and convention specified in a program.

A program must designate that it uses internationalization features; otherwise, it will not access them. A program is not affected by the features described in this section unless a program specifically invokes them. Any programs that already exist and do not make use of internationalization features are not affected by the features.

Using the Ccsversion, Language, and Convention Default Settings

The program can choose the specific ccsversion, language, and convention default settings that it needs by setting the input parameters. The system also has default settings for the internationalization features. The default settings can be accessed by the program. See “Understanding the Hierarchy for Default Settings” later in this section for information on the available levels and on the features supported at each level.

The current default settings for the system can be determined by using one of the following two methods:

- The program calls the CENTRALSTATUS procedure in the CENTRALSUPPORT library.
- A system administrator, a privileged user, or a user who is allowed to use the system console can use MARC menus and screens or the SYSTEMOPTIONS system command. Refer to the *MLS Guide* or the *Menu-Assisted Resource Control (MARC) Operations Guide* for the instructions to display the default ccsversion, language, or convention with MARC.

The system default settings provided by Unisys are as follows:

Feature	Default
Ccsversion	ASeriesNative
Language	English
Convention	ASeriesNative

Before you can change the default settings for localization, you must consider the feature and the level at which the feature is defined. As an example, the `ccsversion` can be changed only at the system operations level. A program can avoid making specific settings by taking advantage of the default settings. For example, if the system-defined `ccsversion` is France, the language is French, and the convention is France Listing, the program can use those default settings without coding the settings within the program. A program can use the default settings with the use of predefined values as input parameters. These parameters tell the procedure to use the current default setting. See “Input Parameters” later in this section for specific information on those parameters.

Understanding the Hierarchy for Default Settings

The default settings for the internationalization features can be established at the following levels:

Task	Established at task initiation
Session	Handled by MARC or CANDE commands or by programs which support sessioning
Usercode	Established in the USERDATAFILE file
System	Established with a system or MARC command

There is a priority associated with these levels. A setting at the task level overrides a setting at the session level. A setting at the session level overrides a setting at the usercode level. A setting at the usercode level overrides a setting at the system level, and so on. A default language and convention can be established at any level, but the default `ccsversion` can be established only at the system level.

Two task attributes enable you to change the language, the convention, or both. These attributes are the `LANGUAGE` and the `CONVENTION` task attributes. By using these attributes, you have the option to select from among multiple languages and conventions when running a program. Information on the use of task attributes is provided in the *Task Attributes Programming Reference Manual*.

The `LANGUAGE` task attribute establishes the language used by a program at run time. The `CONVENTION` task attribute establishes the convention used by a program at run time. For example, an international bank might have a program to print bank statements for customers in different countries. This program could have a general routine to format dates, times, currency, and numerics according to the selected conventions. To print a bank statement for a French customer, this program could set the `CONVENTION` task attribute to France Bureautique and process the general routine. For a customer in Sweden, the program could set the `CONVENTION` task attribute to Sweden and process the general routine.

As you code your program you can use the defaults in both the source code and the calls to the `CENTRALSUPPORT` library, or you can use the settings of your choice. The task level and system level are probably the most useful levels for your program. Because the language and convention features have task attributes defined, you can access or set these task attributes in your program.

Understanding the Components of the MLS Environment

The following four components of the MLS environment support different languages and cultures:

- Coded character sets
- Ccsversions
- Languages
- Conventions

The following paragraphs describe the function of each of these components.

Coded Character Sets and Ccsversions

A coded character set is a set of rules that establishes a character set and the one-to-one relationship between the characters of the set and their code values. The same character set can exist with different encodings. For example, the LATIN1-based character set can be encoded in an International Organization for Standardization (ISO) format or an EBCDIC format. Coded character sets are defined in the *MLS Guide*. A coded character set name or number is given to each unique coded character set definition.

A coded character set name or number can also be used to set the INTMODE or EXTMODE file attribute values for a file. For more information on how to use the INTMODE and EXTMODE file attributes, see the *File Attributes Programming Reference Manual*.

A ccsversion is a collection of information necessary to apply a coded character set in a given country, language, or line of business. This information includes the processing requirements such as data classes, lowercase-to-uppercase mapping, ordering of characters, and the presentation set and escapement rules necessary for output. A ccsversion name and number is given to each unique group of information. A ccsversion name or number can also be used to set the CCSVERSION file attribute for a file. For more information, refer to the *File Attributes Programming Reference Manual*.

Each enterprise server includes a data file, SYSTEM/CCSFILE, containing all coded character sets and ccsversions that are supported on the system. You cannot choose a coded character set directly, but by choosing a ccsversion, you implicitly designate the default coded character set for your system.

Data can be entered and manipulated in only one coded character set and ccsversion at a time. Although there are many ccsversions which can be accessed, there is only one ccsversion active for the entire system at one time. This is called the system default ccsversion. See "Using the Ccsversion, Language, and Convention Default Settings" and "Understanding the Hierarchy for Default Settings" earlier in this section.

Several ways exist to determine which coded character sets and ccsversions are available on your system.

- Look in the *MLS Guide*. Your system might have a subset of the ones defined in that guide.
- Use the MARC menus and screens or the system command SYSTEMOPTIONS. Refer to the *MLS Guide* or the *System Commands Operations Reference Manual*.
- Call the CCSVSN_NAMES_NUMS procedure.

You might want to refer to the *MLS Guide* for a complete understanding of ccsversions and the relationship of a coded character set and a ccsversion.

All coded character set and ccsversion information on your system can be accessed by calling CENTRALSUPPORT library procedures. To call these CENTRALSUPPORT library procedures, an ALGOL program should first include a file provided by Unisys that declares all the library procedures. This file is called *SYMBOL/INTL/ALGOL/PROPERTIES. The procedures can then be used in the program. The file also contains DEFINES for some input parameters and DEFINES for the results returned from the procedures. The method for including *SYMBOL/INTL/ALGOL/PROPERTIES in your ALGOL program is shown in each of the CENTRALSUPPORT library procedures detailed later in this section. If the file is not included, any library procedure that is used must be declared individually in the program.

Many of the procedures require the specification of a coded character set or ccsversion as an input parameter. A program can choose a specific coded character set or ccsversion by calling the procedures using the name or number of the coded character set or the ccsversion as an input parameter. For example, by calling the VSNORDERING_INFO procedure with the ccsversion name ASeriesNative, then calling the VSNORDERING_INFO procedure again with the ccsversion name SWISS your program could access data in the ASeriesNative ccsversion and then access data in the SWISS ccsversion. A program can also use the system default setting by using predefined values as input parameters. See "Input Parameters" later in this section for specific information about those parameters.

Mapping Tables

A mapping table is used to map one group of characters to another group of characters or another representation of the original characters. Many CENTRALSUPPORT library procedures store coded character set and ccsversion information in ALGOL translate tables as a way of defining, processing, and mapping data. For example, a translate table can exist to translate lowercase characters to uppercase characters.

The internationalization procedures provide you with access to translate tables that apply to data specified in coded character sets or specified ccsversions. These translate tables are as follows:

- Mapping data from one coded character set to another coded character set
- Mapping data from lowercase to uppercase characters
- Mapping data from uppercase to lowercase characters
- Mapping data from alternative numeric digits defined in a ccsversion to numeric digits defined in U.S. EBCDIC
- Mapping data from numeric digits in U.S. EBCDIC to alternative numeric digits defined in a ccsversion
- Mapping characters to their escapement values

You must use procedures from the CENTRALSUPPORT library to access these translate tables or to process data using these tables. For example, you use the CCSTOCCS_TRANS_TEXT procedure to translate data from one coded character set to another coded character set. You use the VSNTRANS_TEXT procedure to map lowercase data to uppercase.

See the *MLS Guide* for definitions of mapping tables for each coded character set and ccsversion.

Data Classes

A data class is a group of characters sharing common attributes such as alphabetic, upon which membership tests can be made. A truth set is a method of storing the declared set of characters upon which membership tests can be made. Many CENTRALSUPPORT library procedures store ccsversion information in ALGOL truth set tables as a way to define ccsversion data classes.

The internationalization features provide you with access to truth sets that apply to a ccsversion. These truth sets are as follows:

- Ccsversion alphabetic
- Ccsversion numeric
- Ccsversion graphics
- Ccsversion spaces
- Ccsversion lowercase
- Ccsversion uppercase

The alphabetic truth set contains those characters that are considered to be alphabetic for a specified ccsversion. The numeric truth set contains those characters that are considered to be numbers for a specified ccsversion, and so on.

You can use procedures from the CENTRALSUPPORT library to access these truth sets or to process data using these truth sets. For example, if a program manipulates an employee identification number such as 555962364, it might then need to verify that the text is all numeric. The program can call the VSNINSPECT_TEXT CENTRALSUPPORT library procedure to compare the text to the numeric truth set. This procedure returns the information that the text is or is not all numeric.

Refer to the *MLS Guide* for definitions of ccsversions and data classes.

Text Comparisons

A text comparison can be required for sorting text, or for comparing relationships between two pieces of text.

The traditional method for handling text comparisons is based on a strict binary comparison of the character values. The binary method of comparison is not meaningful when used for sorting text if the binary ordering of the coded characters does not match the ordering sequence of the alphabet of the language. This situation is the case for most coded character sets.

Because the binary method is not sufficient for all usage requirements, Unisys supports the definitions of two other levels of ordering.

The first level is called Ordering. Each character gets an ordering sequence value (OSV). An OSV is an integer from 0 through 255, assigned to each code position in a character set. The OSV signifies a relative ordering value of a character. An OSV of 0 indicates that the character comes before a character with an OSV of 1. More than one character can be assigned the same OSV.

The second level is called Collating. Each character gets an OSV and a priority sequence value (PSV). A PSV is an integer from 1 to 15 that is assigned to each code position in a character set. The PSV is a relative priority value within each OSV. Each character with a unique OSV has a PSV of 1; however, if 2 characters have the same OSV, they will have different PSVs for additional differentiation.

In comparing two strings of data, a comparison which uses only 1 level, the Ordering level, is called an equivalent comparison. A comparison which utilizes both levels, Ordering and Collating, is called a logical comparison.

You can use the following three types of text comparisons by calling the procedures of the CENTRALSUPPORT library.

Order Type	Explanation
Binary	A comparison based on the hexadecimal code values of the characters
Equivalent	A comparison based on the ordering sequence values (OSVs) of the characters
Logical	A comparison based on the OSVs plus the priority sequence values (PSVs) of the characters

In addition to the three types of ordering, Unisys also supports the following two types of character substitution.

Substitution	Explanation
Many to One	A predetermined string of up to three characters can be ordered as if it were one character, assigning it a single OSV and PSV pair. Even if a character is part of a predetermined string of characters that are ordered as a single value, the character still has an OSV and a PSV pair assigned to it to allow for cases in which the character appears in other strings or individually. For example, in Spanish, the letter pair <i>ch</i> is ordered as if it were a single letter, different from either <i>c</i> or <i>h</i> , and ordering between <i>c</i> and <i>d</i> .
One to Many	A single character can generate a string of two or three OSV and PSV pairs. For example, the β (the German sharp S) character is ordered as though it were <i>ss</i> .

For a description of how the characters in each ccsversion are ordered and how text comparisons work, see the *MLS Guide*.

Providing Support for Natural Languages

The natural language feature enables users of an application program to communicate with the computer system in their natural language. A natural language is a human language in contrast to a computer programming language.

A program must be written in the subset of the standard EBCDIC character set defined by the ALGOL language. Only the contents of string literals, data items with variable character data, or comments can be in a character set other than that subset.

If a program interacts with a user, has a user interface with screens or forms, displays messages or accepts user input, then those aspects of the program should be in the natural language of the user. For example, French would be the natural language of a person from France. Refer to the *MLS Guide* for a list of user interfaces, including screens or forms, that can be localized. The following text explains how to develop an ALGOL application program which supports messages in the natural language of the user.

Although you can use many natural languages, one is defined as the default for the entire system. You can override this default and access texts in another language, as long as they have been translated and identified to the system.

Creating Messages for an Application Program

In the MLS environment, the messages handled by your application program are grouped into the following categories:

Message Category	Explanation
Output message	A message that an application program displays to the user. Some examples of output messages are error messages and prompts for input. An output message is localized so that it can be displayed in the language of the user.
Input message	A message received by an interactive program either from a user or from another program in response to a prompt for input. The input message might be in a language that the program cannot recognize. In this case, the message must be translated so that it can be understood by the program.

If you develop input and output messages within an output message array, you make the localization process easier. When messages are in an output message array, a translator can use the MSGTRANS utility to localize the messages into one or more natural languages. The MSGTRANS utility finds all output message arrays in a program and presents them to a translator for translation. If messages are not in output message arrays, a translator searches the source file for each message and then translates the message.

An output message array contains output messages to be used by the MLS environment. See "OUTPUTMESSAGE ARRAY Declaration" in Section 3, "Declarations."

Creating Multilingual Messages for Translation

The following are guidelines for creating messages that can be multilingual:

- Put all output messages in output message arrays.
- Allow more space for translated messages. Because the English language is more compact than many other natural languages, a message in English generally becomes about 33 percent longer after it is translated into another language. For example, if a program can display an 80-character message, an English message should be only 60 characters long so that the translated message can expand by one-third and not exceed the maximum display size.
- Accept or display any messages in the application program using the `MLSACCEPT` statement or the `MLSDISPLAY` statement. For more information, see Section 4, "Statements."
- Use complete sentences for messages because phrases are difficult to translate accurately.
- Do not use abbreviations because they also are difficult to translate.

Providing Support for Business and Cultural Conventions

The business and cultural features enable users of an application program to display and receive data according to local conventions. A convention consists of formatting instructions for date, time, numeric, currency, and page size.

Unisys provides standard convention definitions for many formatting styles. For example, some of the conventions are Denmark, Italy, Turkey, and UnitedKingdom1. These convention definitions contain information to create formats for time, date, numbers, currency, and page size required by a particular locality.

Each enterprise server includes a data file named `*SYSTEM/CONVENTIONS` that contains all the convention definitions supported on the system. Although you can use many conventions, only one default convention is defined as the default for the entire system. This convention is called the system default convention. You can override this default and you can access all the conventions by calling the `CENTRALSUPPORT` library procedures. To display the names of the conventions available on the host computer, use the `CNV_NAMES` `CENTRALSUPPORT` library procedure. The *MLS Guide* provides complete information on all of the conventions supplied to you.

If none of the conventions provided to you meet your needs, you can define a new convention. You must use a template to define a convention. A template is a group of predefined control characters that describe the components for date, time, numeric, or currency. For example, the data item `02251990` and the template `!0o!/!dd!/!yyyy!` produce the formatted date, `02/25/1990`. To use some of the `CENTRALSUPPORT` library procedures, you must understand how templates are defined. The *MLS Guide* describes how to define and use a template.

Using the Date and Time Features

Date and time features are provided by the ALGOL programming language and by several CENTRALSUPPORT library procedures.

The ALGOL programming language has date and time features for standard use. The TIME function provides system date and time with various formats, for example YYDDD, MMDDYY, or HHMMSS. For more information, see "TIME function" in Section 5, "Expressions and Functions."

Date and time punctuation can be specified in the PICTURE clause through string literals. For example:

```
PICTURE PDATE(99"/"99"/"99),
PICTURE PTIME(99":"99":"99),
```

For more information, see "PICTURE Declaration" in Section 3, "Declarations."

The ALGOL date and time features do not use the convention features for internationalization. The ALGOL internationalization features are limited in internationalizing programs. The user can call the CENTRALSUPPORT library procedures to format date and time items. The following types of procedures are available to format the date and time:

Procedure Type	Description
Convention	You supply the convention name and the value for the date or time. The procedure returns the date or time value in the format used by the convention. All the conventions are described in the <i>MLS Guide</i> .
Template	You supply the following: the format that you want for the date or time in a template parameter; the value for the date or time. You must use predefined control characters to create the template. These control characters are described in the <i>MLS Guide</i> .
System	The system supplies the date and time. There is a procedure that formats the system date, the system time, or both according to a convention and a procedure that formats the system date, the system time, or both according to a template that you supply.

You can use the `CNV_SYSTEMDATETIME` procedure to display the system date and time according to the convention and language you choose. If you designate the `ASeriesNative` convention and the `ENGLISH` language, the date and time can be displayed as follows:

```
9:25 AM Monday, July 4, 1988
```

If you designate the `France Listing` convention and the `French` language, the same date and time can be displayed as follows:

```
9h25, lundi 4 juillet 1988
```

Using the Numeric and Currency Features

Numeric and currency features are provided by the ALGOL programming language and by several `CENTRALSUPPORT` library procedures.

The ALGOL programming language has numeric and currency features for standard use. ALGOL also provides the introduction codes that can be used to specify the thousands separator, the decimal sign, and a single character currency symbol in the `PICTURE` clause. For example:

```
PICTURE PAMOUNT(C/ N, U#, FFFDZZ9I99)
```

declares `"/` as the thousands separator, `,` as the decimal sign, and `#` as the currency symbol for `PAMOUNT`.

For more information, see `"PICTURE Declaration"` in Section 3, `"Declarations."`

ALGOL provides a `DECIMALPOINTISCOMMA` option for messages in output message arrays. This enables the user to replace commas with periods and periods with commas in message parameters. For more information, see `"OUTPUTMESSAGE ARRAY Declaration"` in Section 3, `"Declarations."`

In addition to using the features in ALGOL, you can call the `CENTRALSUPPORT` library procedures to inquire about numeric symbols or to format currency amounts. All numeric or currency symbols can be retrieved with a `CENTRALSUPPORT` library call. Monetary amounts in real number form can be formatted according to different conventions.

You can use the `CNV_CURRENCYEDIT` procedure to format a monetary value according to the convention you choose. If you designate the `Greece` convention, the monetary amount `12345.67` is formatted as `DR.12 345,67`.

Using the Page Size Formatting Features

The CNV_FORMSIZE CENTRALSUPPORT procedure enables you to retrieve default lines-per-page and characters-per-line values for a specified convention.

For example, the Netherlands convention definition specifies 70 lines as the default page length and 82 characters as the default page width, while the Zimbabwe convention definition specifies 66 lines as the default page length and 132 characters as the default page width.

Summary of CENTRALSUPPORT Library Procedures

The CENTRALSUPPORT library procedures are integer-valued functions. The procedures return values in the output parameters and the functional result. The results returned by each procedure can be checked using standard programming practices. The results are useful in deciding if an error has occurred. The error results and their meanings are presented in the description of each procedure and at the end of this section.

The CENTRALSUPPORT library procedures are called by application programs and system software. An application program can call the CENTRALSUPPORT library procedures to perform the following tasks:

- Determine the default settings on the host computer.
- Identify and validate character sets and ccsversion.
- Obtain information about a coded character set or a ccsversion.
- Check data against ccsversion data classes.
- Translate data using ccsversion mapping tables.
- Compare data.
- Manipulate text.
- Translate characters using ccsversion escapement.
- Obtain information about the convention contents.
- Add, modify and delete conventions.
- Format date and time.
- Format monetary and numeric data.
- Obtain default characteristics for hard copy output.

Table 9–1 lists the CENTRALSUPPORT library procedures according to the tasks the procedures perform and describes the purpose of each procedure.

Table 9–1. Functional Grouping of CENTRALSUPPORT Library Procedures

Procedure Name	Purpose
Identifying the Available Coded Character Sets and Ccsversions	
CCSVSN_NAMES_NUMS	Returns the names and numbers of all coded character sets or all ccsversions available on the host computer. The names and numbers are listed in two arrays. These arrays are ordered so that the names in the names array correspond to the numbers in the numbers array.
CENTRALSTATUS	Obtains the values of the default settings for internationalization features on the host computer. This procedure returns the names of the default ccsversion, language, and convention. It also returns the number of the default ccsversion.
VALIDATE_NAME_RETURN_NUM	Verifies that a designated coded character set or ccsversion name is valid on the host computer. If the coded character set or ccsversion is valid, the procedure returns the corresponding number for the designated coded character set or ccsversion.
VALIDATE_NUM_RETURN_NAME	Verifies that the designated coded character set or ccsversion number is valid on the host computer. If the coded character set or ccsversion is valid, the procedure returns the name of the coded character set or ccsversion.
Obtaining Coded Character Set and Ccsversion Information	
CCSINFO	Provides basic information about a designated coded character set, including the number of bits per character (8, 16, or mixed), the coding format (for example, ISO or EBCDIC), and the space character.
VSNINFO	Returns the following information for a designated ccsversion: <ul style="list-style-type: none"> • The number of the base character set to which the ccsversion applies • The escapement information • The characters in the spaces data class for the ccsversion • The array sizes required by the other ccsversion elements

Table 9–1. Functional Grouping of CENTRALSUPPORT Library Procedures

Mapping Data from One Coded Character Set to Another	
CCSTOCCS_TRANS_TABLE	Returns a translation table used to translate data appearing in one designated character set to another designated character set. The result is a one-to-one mapping of the characters. You also can use the translate table directly with the REPLACE syntax.
CCSTOCCS_TRANS_TABLE_ALT	Returns a coded character set to coded character set translate table. The procedure provides an alternative to CCSTOCCS_TRANS_TABLE.
CCSTOCCS_TRANS_TEXT	Translates data from one character set to another character set by using a translate table. Characters are translated using a one-to-one mapping between the two character sets.
CCSTOCCS_TRANS_TEXT_COMPLEX	Translates data from one character set to another character set by using an advanced character mapping operation. The procedure supports the complex translation requirements of mixed, multibyte, and 16-bit coded character sets, as well as 8-bit coded character sets.
Processing Data According to a Ccsversion	
VSNINSPECT_TEXT	Compares the input text to a designated ccsversion truth set to determine whether the characters in the text are in the truth set. This procedure can be used to determine if characters are in one of the following data classes: <ul style="list-style-type: none"> • Alphabetic • Lowercase • Numeric • Presentation • Spaces • Uppercase

Table 9-1. Functional Grouping of CENTRALSUPPORT Library Procedures

<p>VSNTRANSTABLE</p>	<p>Returns a translation table for a designated ccsversion. The type of translation table requested depends on the task to be performed.</p> <p>Translation tables can be requested to perform the following tasks:</p> <ul style="list-style-type: none"> • Translate lowercase letters to uppercase letters. • Translate uppercase letters to lowercase letters. • Translate any digits 0 through 9 to any alternate digits (that is, one-to-one mapping of 0 through 9 to another representation for those digits). • Translate alternate digits to 0 through 9. • Translate characters to their character escapement direction value.
<p>VSNTRANS_TEXT</p>	<p>Translates data using a designated ccsversion translate table. The type of table used determines the type of translation done. All translation is a one-to-one mapping of the characters.</p> <p>This procedure can be used to perform the following types of translations:</p> <ul style="list-style-type: none"> • Lowercase to uppercase characters • Uppercase to lowercase characters • The digits 0 through 9 to alternate digits • Alternate digits to the digits 0 through 9 • Characters to their character escapement direction value
<p>VSNTRUTHSET</p>	<p>Returns a truth set for the designated ccsversion. The truth set contains the characters in a given data class for the ccsversion. Truth sets are available for the following data classes:</p> <ul style="list-style-type: none"> • Alphabetic • Lowercase • Numeric • Presentation • Space • Uppercase

Table 9–1. Functional Grouping of CENTRALSUPPORT Library Procedures

Comparing and Sorting Text	
COMPARE_TEXT_USING_ORDER_INFO	<p>Compares two strings by using the ordering information returned by the VSNORDERING_INFO procedure. One of the following comparison methods can be chosen:</p> <ul style="list-style-type: none"> • Equivalent comparison This method is based on the ordering sequence values of the characters. • Logical comparison This method is based on the ordering sequence values and the priority sequence values of the characters.
VSNCOMPARE_TEXT	<p>Compares two strings by using one of three comparison methods for a designated ccsversion. You can select one of the following comparison methods:</p> <ul style="list-style-type: none"> • Binary comparison, which is based on the binary values of the characters • Equivalent comparison, which is based on the Ordering Sequence Values (OSVs) of the characters • Logical comparison, which is based on the OSVs and Priority Sequence Values (PSVs) of the characters
VSNGETORDERINGFOR_ONE_TEXT	<p>Returns the ordering information for the input text. The ordering information determines how the input text is collated. It includes the ordering and priority sequence values of the characters and any substitution of characters to be made when the input text is sorted.</p> <p>One of the following types of ordering information can be chosen:</p> <ul style="list-style-type: none"> • Equivalent ordering information, which comprises only the ordering sequence values • Logical ordering information, which comprises the ordering sequence values followed by the priority sequence values
VSNORDERING_INFO	<p>Returns the ordering information for a designated ccsversion. The ordering information determines the way in which data is collated for the ccsversion. It includes the ordering and priority sequence values of the characters and any substitution of characters to be made when the designated ccsversion ordering is applied to a string of text.</p>

Table 9–1. Functional Grouping of CENTRALSUPPORT Library Procedures

Positioning Characters	
VSNEscapeMENT	Rearranges the input text according to the escapement rules of the ccsversion.
Determining the Available Natural Languages	
MCP_BOUND_LANGUAGES	Returns the names of the languages that are currently bound to the MCP.
Accessing CENTRALSUPPORT Library Messages	
GET_CS_MSG	Returns the text of a CENTRALSUPPORT message associated with the designated error number.
Identifying the Available Convention Definitions	
CENTRALSTATUS	Obtains the values of the default settings for internationalization features on the host computer. This procedure returns the names of the default ccsversion, language, and convention. It also returns the number of the default ccsversion.
CNV_NAMES	Returns the names of the conventions available on the host computer.
CNV_VALIDATENAME	Indicates whether a designated convention name is currently defined on the host computer.
Obtaining Information About Conventions	
CNV_INFO	Returns a description of all the elements defined in a designated convention.

Table 9–1. Functional Grouping of CENTRALSUPPORT Library Procedures

<p>CNV_SYMBOLS</p>	<p>Returns the numeric and monetary symbols defined for a designated convention. The symbols in the convention are</p> <ul style="list-style-type: none"> • Numeric positive symbol • Numeric negative symbol • Numeric thousands separator symbol • Numeric decimal symbol • Numeric left enclosure symbol • Numeric right enclosure symbol • Numeric grouping specifications • International currency notation • Monetary positive symbol • Monetary negative symbol • Currency symbol • Monetary thousands separator symbol • Monetary decimal symbol • Monetary left enclosure symbol • Monetary right enclosure symbol • Monetary grouping specifications
<p>CNV_TEMPLATE</p>	<p>Returns the requested template for a designated convention. The template can be obtained for the following:</p> <ul style="list-style-type: none"> • Long date format • Short date format • Numeric date format • Long time format • Numeric time format • Monetary format • Numeric format
<p>Formatting Dates According to a Convention</p>	
<p>CNV_CONVERTDATE</p>	<p>Converts a formatted numeric date passed as a parameter to the procedure to the format YYYYMMDD. The numeric date must be formatted according to the numeric date format template defined in the designated convention.</p>
<p>CNV_DISPLAYMODEL</p>	<p>Returns either the date or time display model defined for a designated convention. The components for the model are translated to the designated language.</p>
<p>CNV_FORMATDATE</p>	<p>Formats a numeric date passed as a parameter to the procedure according to a designated convention and language. The date can be formatted using the long, short, or numeric date format defined in the convention.</p>

Table 9-1. Functional Grouping of CENTRALSUPPORT Library Procedures

CNV_FORMATDATETMP	Formats a numeric date passed as a parameter to the procedure according to a template and language passed as parameters of the procedure.
CNV_SYSTEMDATETIME	<p>Returns the system date and/or time formatted according to the designated convention and language. The following types of templates can be chosen:</p> <ul style="list-style-type: none"> • Long date and long time • Long date and numeric time • Short date and long time • Short date and numeric time • Numeric date and long time • Numeric date and numeric time • Long date only • Short date only • Long time only • Numeric time only
CNV_SYSTEMDATETIMETMP	Returns the system date and/or time in the designated language, formatted according to a template passed as a parameter to this procedure.
Formatting Times According to a Convention	
CNV_CONVERTTIME	Converts a formatted numeric time passed as a parameter to the procedure to the format HHMMSS. The numeric time must be formatted according to the numeric time format template defined in the designated convention.
CNV_DISPLAYMODEL	Returns either the date or time display model defined for a designated convention. The components for the model are translated to the designated language.
CNV_FORMATTIME	Formats a time passed as a parameter to the procedure according to a designated convention and language. The time can be formatted using the long or numeric time format defined in the convention.
CNV_FORMATTIMETMP	Formats a time passed as a parameter to the procedure according to a template and language passed as parameters of the procedure.

Table 9–1. Functional Grouping of CENTRALSUPPORT Library Procedures

<p>CNV_SYSTEMDATETIME</p>	<p>Returns the system date and/or time formatted according to the designated convention and language. The following types of templates can be chosen:</p> <ul style="list-style-type: none"> • Long date and long time • Long date and numeric time • Short date and long time • Short date and numeric time • Numeric date and long time • Numeric date and numeric time • Long date only • Short date only • Long time only • Numeric time only
<p>CNV_SYSTEMDATETIMETMP</p>	<p>Returns the system date and/or time in the designated language, formatted according to a template passed as a parameter to this procedure.</p>
<p>Formatting Numeric Data According to a Convention</p>	
<p>CNV_CONVERTNUMERIC</p>	<p>Converts a string containing digits and numeric symbols to a real number.</p>
<p>Formatting Monetary Data According to a Convention</p>	
<p>CNV_CONVERTCURRENCY</p>	<p>Converts a string containing digits and monetary symbols to a real number.</p>
<p>CNV_CURRENCYEDIT</p>	<p>Formats a monetary value passed as a parameter to the procedure according to the monetary formatting template defined in the designated convention.</p>
<p>CNV_CURRENCYEDIT_DOUBLE</p>	<p>Formats a double-precision integer that represents a monetary value according to the monetary formatting template defined in the designated convention.</p>
<p>CNV_CURRENCYEDITTMP</p>	<p>Formats a monetary value passed as a parameter to the procedure according to a template also passed as a parameter.</p>
<p>CNV_CURRENCYEDITTMP_DOUBLE</p>	<p>Formats the double-precision integer that represents a monetary value according to the template parameter.</p>
<p>Determining the Default Page Length and Width</p>	
<p>CNV_FORMSIZE</p>	<p>Returns the default lines-per-page and characters-per-line values from a designated convention for formatting printer output.</p>

Table 9-1. Functional Grouping of CENTRALSUPPORT Library Procedures

Adding, Modifying, and Deleting Conventions	
CNV_ADD	Adds a new convention to the *SYSTEM/CONVENTIONS file. The new definition goes into effect immediately.
CNV_DELETE	Deletes an existing convention from the SYSTEM/CONVENTIONS file. The convention is deleted after the next halt/load. Only conventions that have been created by a user can be deleted. The standard conventions that are provided cannot be deleted.
CNV_MODIFY	Modifies an existing convention in the *SYSTEM/CONVENTIONS file. The modified set becomes effective after the next halt/load. Only conventions that have been created by a user can be modified. The standard conventions that are provided cannot be modified.

Library Calls

The procedures in the CENTRALSUPPORT library can be accessed by using the following steps:

1. Use the INCLUDE compiler control option to include the file **SYMBOL/INTL/ALGOL/PROPERTIES* in a program. By using the INCLUDE option you can also include only portions of the file.
2. Call the procedure.

An example of the call syntax necessary to invoke the CENTRALSUPPORT library is provided in the description of each procedure later in this section.

For general information about using library procedures, refer to Section 8, "Library Facility."

For information about the INCLUDE compiler control option, refer to Section 6, "Compiling Programs."

Parameter Categories

The CENTRALSUPPORT library procedures return output parameters and procedure result values. The parameter types are further described on the following pages.

Input Parameters

In many cases, the input parameter requires the program to supply the ccsversion name or number, the language name, or the convention name. This information can be obtained in the following ways:

- The *MLS Guide* describes all the possible ccsversions, languages, and conventions provided to you. However, the system on which the program is running might have only a subset of these. There can also be customized conventions that are not listed in the *MLS Guide*. These can be identified by the next two options.
- A system administrator, a privileged user, or a person allowed to use the system console can use MARC menus and screens to list the options that exist on the system. The SYSTEMOPTIONS command can be used. For more information on the SYSTEMOPTIONS command, see the *System Commands Operations Reference Manual*. The *MLS Guide* provides additional instructions needed to obtain this information.
- Procedures can be called in the CENTRALSUPPORT library that can return this information. If an application is being written to be used on another system, these library procedures can be used to verify that the ccsversion, language, or convention specified by the user is valid on that system.

For any procedure that accepts a ccsversion number as an input parameter, you can specify a -2 (or the constant value CCSVSNNOTSPECIFIEDV) as input to indicate that the system default value should be used. For any procedure that accepts a ccsversion name as an input parameter, you can specify all blanks or all zeros as inputs to indicate that the system default value should be used. For any procedure that accepts a language or convention name as an input parameter, you can specify all blanks or all zeros as inputs to indicate that the task attribute should be used. If the task attribute is not available, the CENTRALSUPPORT library searches down the hierarchy until a usable value is found.

Input Parameters with Type Values

Many of the CENTRALSUPPORT procedures have a parameter that indicates the type of information to be applied or returned by the procedure. The values in these parameters are referred to as type values. The values are common across all procedures of the library. The values used in the convention (CNV) procedures are common across all CNV procedures. The values used in the coded character set and ccsversion (VSN) procedures are common across all CCS and VSN procedures. The INCLUDE file contains DEFINES that map these constant integer type values to an identifier that can be used in your program.

For example, the TYP parameter is used in a number of procedures to indicate the type of date or time formatting to be used. The type value indicates the type of format to be used. For example, a value of 3 indicates the long time format. The DEFINE is CS_LTIMEV.

Output Parameters

These parameters contain the results of the procedure. For example, the DEST parameter of the CCSTOCCS_TRANS_TEXT procedure contains the translated text produced by the procedure. In the examples for each procedure described later in this section, not all output variables are printed. Representative output is shown.

Result

All the library procedures return an integer that indicates whether an error occurred during the execution of the procedure. A returned value of CS_DATAOKV (1) or CS_FALSEV (1001) means that no error occurred. However, the CNV_VALIDATENAME and VSNCOMPARE_TEXT procedures are exceptions to this rule. For these procedures, the returned value can be 0 (zero), 1, or another value. A returned value of 0 means that no error occurred and the condition is FALSE. A returned value of 1 means that no error occurred and the condition is TRUE. Any other value means that an error occurred.

Each procedure lists the values that can be returned by that procedure. The meanings of these values are explained at the end of this section. These results can be used to call the GET_CS_MSG procedure and to display the error that occurred, or error routines can be coded to handle the possible errors.

Refer to "GET_CS_MSG" later in this section for more information about using that procedure.

The INCLUDE file contains DEFINES that map each integer result into a define identifier to be used by your program.

Procedure Descriptions

The following pages describe the procedures in the CENTRALSUPPORT library that an ALGOL program can access.

The procedures are listed in alphabetical order. Each description includes a general overview of the procedure, an example showing how to call the procedure, and a description of the parameters used in the example.

These procedures are found in the CENTRALSUPPORT library specification file **SYMBOL/INTL/ALGOL/PROPERTIES*, provided to you. The file contains library name specifications, library subroutine specifications, library parameter specifications, library name declarations for type values, and error message values.

CCSINFO

This procedure provides basic information about a designated coded character set, including the number of bits per character (8, 16, or mixed) and the coding format (for example, ISO or EBCDIC).

This procedure can be used to determine the code format for a file that has its INTMODE set to a character set name. See the *MLS Guide* for a list of coded character sets available.

Example

This example calls the procedure CCSINFO to get information about the Latin1EBCDIC coded character set. The *MLS Guide* contains the information that the coded character set number for Latin1EBCDIC is 12. This number can also be retrieved by calling VALIDATE_NAME_RETURN_NUM with the name Latin1EBCDIC.

```
BEGIN

  $ INCLUDE INTL=" SYMBOL/INTL/ALGOL/PROPERTIES." 1000000 - 4999999

  FILE OUT(KIND=DISK,UNITS=CHARACTERS,FILEUSE=IO,FILETYPE=0,NEWFILE,
    MAXRECSIZE=80,TITLE="OUT/ALGOL/CCSINFO.");
  ARRAY CCS_ARY[0:4]
    ,LINEOUT[0:14];
  INTEGER CCS_NUM
    ,RESULT;

  CCS_NUM := 12;                                % Latin1EBCDIC CCS
  RESULT := CCSINFO(CCS_NUM, CCS_ARY);
  WRITE(OUT,<"RESULT = ",J4>,RESULT);
  IF RESULT EQL CS_DATAOKV THEN
    BEGIN
      REPLACE LINEOUT BY " " FOR 80;
      WRITE(OUT,80,LINEOUT);
      WRITE(OUT,<"Field Meaning          Location          Value">);
      WRITE(OUT,<"-----          -----          -----">);
```

```

WRITE(OUT,<"# of bytes per character      [0]      ",J3>,
      CCS_ARY[0]);
WRITE(OUT,<"Repertoire Number            [1]      ",J3>,
      CCS_ARY[1]);
WRITE(OUT,<"Encoding Number              [2]      ",J3>,
      CCS_ARY[2]);
WRITE(OUT,<"Coding Format                  [3]      ",J3>,
      CCS_ARY[3]);
WRITE(OUT,<"Space Character (hexadecimal) [4]      ",H12>,
      CCS_ARY[4]);
END;
CLOSE(OUT,LOCK);
END.

```

Output

RESULT = 1

Field Meaning	Location	Value
-----	-----	-----
# of bytes per character	[0]	1
Repertoire number	[1]	26
Encoding number	[2]	1
Coding format	[3]	3
Space character (hexadecimal)	[4]	404040404040

In this call, the parameters have the following meanings:

CCS_NUM is an integer passed to the procedure. This integer contains the number of the coded character set about which information is requested. The coded character set number is the same as the INTMODE or EXTMODE attribute value of a file.

CCS_ARY is a real array returned by the procedure. This real array contains information about a coded character set from the character set and ccsversion file. The layout for return information in CCS_ARY is as follows:

Word[0] Number of bytes per character, (if the value is 3, the character set is mixed multibyte with both single and double-byte characters)

Word[1] Repertoire number (not used by new coded character sets)

Word[2] Encoding number (not used by new coded character sets)

Word[3] Code format
 % 1 = ISO
 % 2 = ASERIESEBCDIC
 % 3 = STDEBCDIC
 % 4 = BTOS
 % 5 = PC
 % 6 = EBCDICMB
 % 7 = ISOMB

% 8 = PCMB
% 9 = EUCMB
% 10 = 2200MB
% 13 = DOUBLEBYTE
% 14 = UCSBMP
% 15 = UCSBMPNT

Word[4] A full word of space characters
% If the Ccs is 8-bit, the space character is 8-bit.
% If the Ccs is 16-bit, the space character is 16-bit.
% If the Ccs is mixed, the space character is
% 8-bit and the SubCcs records also identify
% their 8-bit or 16-bit space characters.

Word[5]+ When there are extension facilities
(Code format is ISO (word[3] = 1))

Word[5].[47:08] Extension facilities
% 0 = NONE
% 1 = UNISYS01
% 255 = non-UNISYS01

For extension facilities is NONE (word[5].[47:08] = 0)

Word[5].[39:40] Not used

When extension facilities are NONE, the entire word 5
is empty as are any subsequent words in the output array.

For UNISYS01 extension facilities (word[5].[47:08] = 1)

Word[5].[39:01] Defined G1 set
% 0 = undefined G1 set
% 1 = defined G1 set

If defined G1 set (word[5].[39:01] = 1)

Word[5].[38:09] Number of characters in G1 set (94 or 96)

Word[5].[29:01] Registration
% 0 = unregistered
% 1 = registered

If registered (word[5].[29:01] = 1)

Word[5].[28:13] Not used

Word[5].[15:16] Number of escape sequence characters
% characters in the escape
% sequence start at word 6

For non-UNISYS01 extension facilities (word[5].[47:08] = 255)

Word[5].[39:24] Not used

Word[5].[15:16] Number of escape sequence characters
% characters in the escape
% sequence start at word 6

Word[5]+ When there are SubCcs records
(Number of bytes per character is mixed (word[0] = 3))

Word[5] Number of SubCcs records
% each SubCcs record is 4 words long

Word[6] SubCcs library number

Word[7] SubCcs bytes per character

Word[8] SubCcs invocation
% 0 = NONE
% 1 = SS2 (single shift 2)
% 2 = SS3 (single shift 3)
% 3 = ESB (EBCDIC single bracket)
% 4 = ASB (ASCII single bracket)
% 5 = LS (lock shift)

Word[9] SubCcs space character (a full word)

The contents of words 6-9 are repeated starting at word[10] for every SubCcs which is a part of this coded character set. That is, word[5] is followed by by four words of information for every SubCcs record. The number of SubCcs records is specified in word[5].

The recommended array size for CcsInfo is 5 words unless there are escape sequences or SubCcs records, in which case the recommended size is 30 words.

Notes:

- *Extension facilities are the escape sequences in an ISO environment that are used to specify the code sets (such as C0, C1, G0, G1) from which ISO builds coded character sets. The escape sequences are defined in ISO standard 2022, titled Information Processing–ISO 7-Bit and 8-Bit Coded Character Sets – Code-extension Techniques.*
- *A mixed multibyte coded character set requires at least one 8-bit CCS, at least one 16-bit CCS, and a set of subCCS records to define the characteristics of each CCS.*

RESULT is an integer returned by the procedure. It indicates whether or not an error occurred during the execution of the procedure. An explanation of the error result values and messages can be found at the end of this section. The values returned by this procedure include:

CS_ARRAY_TOO_SMALLV (3001)	CS_FILE_ACCESS_ERRORV (1000)
CS_BAD_ARRAY_DESCRIPTIONV (3000)	CS_NO_NUM_FOUNDV (3003)
CS_DATAOKV (1)	CS_SOFTERRV (1002)
CS_FAULTV (1001)	

CCSTOCCS_TRANS_TABLE

This procedure returns a translation table used to translate data appearing in one designated coded character set to another designated coded character set. The result is a one-to-one mapping of the characters.

When an application program performs a high volume of translations, this procedure can be used in combination with the TRANS_TEXT_USING_TTABLE procedure instead of calling the CCSTOCCS_TRANS_TEXT procedure. The translation table from CCSTOCCS_TRANS_TABLE can be retained in working memory. The translation table is then passed as a parameter to TRANS_TEXT_USING_TTABLE. Because the table is stored in working memory it does not have to be retrieved each time it is to be used.

Example

This example reads input from a text file. The data in this input file is in an international coded character set, but that coded character set is unknown until the EXTMODE of the file is interrogated. The EXTMODE contains the coded character set number of the file. For this example, assume the data in the input file is encoded in Latin1EBCDIC, with a coded character set number of 12. The data is translated from the Latin1EBCDIC coded character set to the Latin1ISO coded character set and then displayed to the terminal (remote file). The *MLS Guide* can be used to determine that the coded character set number for Latin1ISO is 13. This number can also be retrieved by calling VALIDATE_NAME_RETURN_NUM with the name Latin1ISO.

```

BEGIN

$ INCLUDE INTL="SYMBOL/INTL/ALGOL/PROPERTIES." 10000000 - 49999999

FILE OUT(KIND=DISK,UNITS=CHARACTERS,FILEUSE=IO,FILETYPE=0,NEWFILE,
        MAXRECSIZE=80,TITLE="OUT/ALGOL/CCSTOCCS_T_TABLE.");
FILE INFILE(TITLE="ALGOL/CCSTOCCS_T_TABLE/INFILE.",FILEUSE=IN,
        DEPENDENTSPECS=TRUE);
ARRAY TTABLE_ARY[0:63]
        ,REC[0:14]
        ,TRANSREC[0:14];

INTEGER CCSNUMFROM
        ,CCSNUMTO
        ,RESULT;

OPEN(INFILE);
CCSNUMFROM:= INFILE.EXTMODE; %capture the ccs number
CCSNUMTO:= 13; %Latin1ISO
RESULT:= CCSTOCCS_TRANS_TABLE(CCSNUMFROM, CCSNUMTO, TTABLE_ARY);
WRITE(OUT,<"RESULT = ",J4>,RESULT);
IF RESULT EQL CS_DATAOKV THEN
    BEGIN
        WHILE NOT READ(INFILE, 15, REC) DO
            BEGIN
                REPLACE POINTER(TRANSREC) BY POINTER(REC) FOR 90 WITH
                    TTABLE_ARY[0];
            
```

```
        WRITE(OUT, 15, TRANSREC);
        END;
    END
ELSE
    WRITE(OUT, <"ERROR IN CREATING TRANSLATE TABLE">);
CLOSE(OUT, LOCK);
END.
```

Output

```
RESULT = 1
```

In this call, the parameters have the following meanings:

CCSNUMFROM is an integer passed to the procedure. This integer contains the number of the coded character set that you are translating from.

CCSNUMTO is an integer passed to the procedure. This integer contains the number of the coded character set that you are translating to.

TTABLE_ARY is a 64-word ALGOL translate table array in which the translate table is returned. For more information, see "TRANSLATETABLE Declaration" in Section 3, "Declarations."

RESULT is an integer returned by the procedure. This integer indicates whether or not an error occurred during the execution of the procedure. An explanation of the error result values and messages can be found at the end of this section. Values returned by this procedure include:

CS_ARRAY_TOO_SMALLV (3001)	CS_DATAOKV (1)
CS_BAD_ARRAY_DESCRIPTIONV (3000)	CS_FAULTV (1001)
CS_BAD_DATA_LEN (3002)	CS_FILE_ACCESS_ERRORV (1000)
CS_COMPLEX_TRAN_REQV (4004)	CS_SOFTERRV (1002)
CS_DATA_NOT_FOUNDV (4002)	

CCSTOCCS_TRANS_TABLE_ALT

This procedure returns a coded character set to coded character set translate table. This procedure offers an alternative to the ALGOL translate table.

Example

This example gets the Latin1EBCDIC coded character set to Latin1ISO coded character set translate table. The variable EXPAND is set to CS_EXPAND_HEXV. Therefore, an index into TTABLE_ARY represents the hexadecimal code value of a source character, and the character at that index in the array is the destination character. The *MLS Guide* can be used to determine that the coded character set number for Latin1EBCDIC is 12 and the Latin1ISO is 13. These numbers can also be retrieved by calling VALIDATE_NAME_RETURN_NUM with the coded character set names.

```
BEGIN
  $ INCLUDE INTL="SYMBOL/INTL/ALGOL/PROPERTIES." 10000000 - 49999999

  FILE OUT(KIND=DISK,UNITS=CHARACTERS,FILEUSE=IO,FILETYPE=0,NEWFILE,
           MAXRECSIZE=80,TITLE="OUT/ALGOL/CCSTOCCS_T_TABLE/ALT.");
  EBCDIC ARRAY TTABLE_ARY[0:255];
  INTEGER CCSNUMFROM
           ,CCSNUMTO
           ,RESULT;

  CCSNUMFROM := 12;
  CCSNUMTO := 13;
  RESULT := CCSTOCCS_TRANS_TABLE_ALT(CCSNUMFROM,CCSNUMTO,CS_EXPAND_HEXV,
                                     TTABLE_ARY);
  WRITE(OUT,<"RESULT = ",J4>,RESULT);
  CLOSE(OUT,LOCK);
  END.
```

Output

```
RESULT = 1
```

In this call, the parameters have the following meanings:

CCSNUMFROM is an integer passed to the procedure. It contains the number of the coded character that you are translating from.

CCSNUMTO is an integer passed to the procedure. It contains the number of the coded character set that you are translating to.

Internationalization

CS_EXPAND_HEXV is an integer that represents the value of the translate table form. This integer is one of two values:

Type Value	Value Name	Description
0	CS_EXPAND_HEXV	Do not expand
1	CS_EXPAND_HEXTOEBCDICV	Expand

TTABLE_ARY is an EBCDIC array returned from the procedure. TTABLE_ARY contains a translate table in one of two forms, depending on whether CS_EXPAND_HEXV or CS_EXPAND_HEXTOEBCDICV has been chosen. For either choice, the arrays are character oriented. If CS_EXPAND_HEXV is chosen, then the return array is 256 characters. The index into the array 0 through 255 represents the hex code of a source character, and the character at that index in the array is the destination character.

For example, in a standard EBCDIC to ASCII translate table, the 0 through 255 index represents the EBCDIC characters, while the ASCII codes are stored in the array at the proper index.

If CS_EXPAND_HEXTOEBCDICV is chosen, then the return array still represents 256 characters, but it is represented by 512 bytes. The destination characters are expanded from hex to their representation in EBCDIC.

For example, if 00, 01, 02 in the FROM set translates to 04, 6F, 83 in the TO set, then the first six array elements in the TTABLE_ARY will be F0F4 F6C6 F8F3.

The recommended array size is 256 characters if CS_EXPAND_HEXV is chosen. If CS_EXPAND_HEXTOEBCDICV is chosen, the recommended array size is 512.

RESULT is an integer returned by the procedure. This integer indicates whether an error occurred during the execution of the procedure. An explanation of the error result values and messages can be found at the end of this section. Values returned by this procedure include:

CS_ARRAY_TOO_SMALLV (3001)	CS_DATA_NOT_FOUNDV (4002)
CS_BAD_ARRAY_DESCRIPTIONV (3000)	CS_DATAOKV (1)
CS_BAD_DATA_LEN (3002)	CS_FAULTV (1001)
CS_BAD_TYPE_CODEV (3006)	CS_FILE_ACCESS_ERRORV (1000)
CS_COMPLEX_TRAN_REQV (4004)	CS_SOFTERRV (1002)

CCSTOCCS_TRANS_TEXT

This procedure translates data specified in one coded character set to another coded character set by using a translation table. Characters are translated using a one-to-one mapping between two character sets.

For example, a program might translate text from the LATIN1EBCDIC coded character set to the LATIN1ISO coded character set. Refer to the *MLS Guide* for a list of the coded character set numbers that are available as inputs to this procedure.

Although there are many character set numbers, there is not a mapping table between every combination of coded character sets. The procedure returns an error indicating the data was not found if you pass two valid coded character set numbers for a table that does not exist. See the *MLS Guide* for the mapping tables available as input to this procedure.

Example

This example takes a string encoded in the Latin1EBCDIC coded character set and translates it to the Latin1ISO coded character set. The *MLS Guide* can be used to determine that the coded character set number for Latin1EBCDIC is 12 and for Latin1ISO is 13. These numbers can also be retrieved by calling VALIDATE_NAME_RETURN_NUM with the coded character set names.

```
BEGIN

$ INCLUDE INTL="SYMBOL/INTL/ALGOL/PROPERTIES." 10000000 - 49999999

DEFINE SOURCE_START = 0 #
    ,DEST_START = 0 #
    ,TRANS_LEN = SIZE(SOURCE) #;

FILE OUT(KIND=DISK,UNITS=CHARACTERS,FILEUSE=IO,FILETYPE=0,NEWFILE,
    MAXRECSIZE=80,TITLE="OUT/ALGOL/CCSTOCCS_T_TEXT.");
EBCDIC ARRAY SOURCE[0:6]
    ,DEST[0:6];
INTEGER CCSNUMFROM
    ,CCSNUMTO
    ,RESULT;

CCSNUMFROM := 12;                %Latin1EBCDIC
CCSNUMTO := 13;                  %Latin1ISO
REPLACE SOURCE[0] BY "pañuelo" FOR TRANS_LEN;

RESULT := CCSTOCCS_TRANS_TEXT(CCSNUMFROM, CCSNUMTO, SOURCE,
    SOURCE_START, DEST, DEST_START, TRANS_LEN);
WRITE(OUT, <"RESULT = ", J4>, RESULT);
IF RESULT EQL CS_DATAOKV THEN
    WRITE(OUT, <"DEST = ", A7>, DEST);
CLOSE(OUT, LOCK);
END.
```

Output

```
RESULT = 1
DEST   = pañuelo
```

In this call, the parameters have the following meanings:

CCSNUMFROM is an integer passed to the procedure. This integer contains the number of the coded character set that you are translating from. The coded character sets and their numbers are described in the *MLS Guide*.

CCSNUMTO is an integer passed to the procedure. This integer contains the number of the coded character set that you are translating to. The coded character sets and their numbers are described in the *MLS Guide*.

SOURCE is an EBCDIC array containing the text to translate.

SOURCE_START is an integer that specifies the offset (0 relative) where the translation starts.

DEST is an EBCDIC array in which the translated text is returned.

DEST_START is an integer that specifies the offset (0 relative) where the output text is stored.

TRANS_LEN is an integer that specifies the number of characters in SOURCE to translate, beginning at SOURCE_START.

RESULT is an integer returned by the procedure. This integer indicates whether an error occurred during the execution of the procedure. An explanation of the error result values and messages can be found at the end of this section. The values returned by this procedure include:

CS_ARRAY_TOO_SMALLV (3001)	CS_DATAOKV (1)
CS_BAD_ARRAY_DESCRIPTIONV (3000)	CS_FAULTV (1001)
CS_BAD_DATA_LEN (3002)	CS_FILE_ACCESS_ERRORV (1000)
CS_COMPLEX_TRAN_REQV (4004)	CS_SOFTERRV (1002)
CS_DATA_NOT_FOUNDV (4002)	

CCSTOCCS_TRANS_TEXT_COMPLEX

The CCSTOCCS_TRANS_TEXT_COMPLEX procedure is an advanced character mapping operation that supports the complex translation requirements of mixed, multibyte and 16-bit coded character sets, as well as the standard 8-bit coded character sets.

Unlike the standard 8-bit mapping procedures of the CENTRALSUPPORT library, the multibyte character mappings handled by this procedure can be done only within the CCSTOCCS_TRANS_TEXT_COMPLEX procedure in the CENTRALSUPPORT library. No multibyte translate tables are available to calling programs.

The CCSTOCCS_TRANS_TEXT_COMPLEX procedure maintains a STATE array so that mappings can be spread over multiple calls. The procedure also includes an OPTION parameter that controls how error recovery is handled and how mappings are terminated.

Example

The following example takes a string encoded in the UCS2NT coded character set and translates the string to the ASERIESEBCDIC coded character set. Refer to the *MultiLingual System (MLS) Administration, Operations, and Programming Guide* to determine the coded character set numbers. For instance, the coded character set number for UCS2NT is 84 and the coded character set number for ASERIESEBCDIC is 4. These numbers can also be retrieved by calling VALIDATE_NAME_RETURN_NUM with the coded character set names.

The string Complex is represented by hexadecimal codes 43006F006D0070006C0065007800 in the UCS2NT coded character set. When translated to the ASERIESEBCDIC coded character set, the string displays as Complex in the output file.

```
BEGIN

$ INCLUDE INTL="SYMBOL/INTL/ALGOL/PROPERTIES." 10000000-49999999

FILE OUT(KIND=DISK,UNITS=CHARACTERS,FILEUSE=IO,FILETYPE=0,NEWFILE,
        MAXRECSIZE=80,TITLE="OUT/ALGOL/CCSTOCCS_COMPLEX.");
EBCDIC ARRAY SOURCE[0:19]
           ,DEST[0:19];
REAL ARRAY STATE[0:9];
INTEGER CCSNUMFROM
        ,CCSNUMTO
        ,SOURCE_START
        ,SOURCE_BYTES
        ,DEST_START
        ,DEST_BYTES
        ,OPTION
        ,RESULT;

CCSNUMFROM := 84; % UCS2NT
CCSNUMTO := 4; % ASERIESEBCDIC
SOURCE_START := 0;
SOURCE_BYTES := 14;
```

```
DEST_START := 0;
DEST_BYTES := 10;
OPTION := CS_OPT_INITIAL_COMPLETEV;

REPLACE SOURCE[0] BY 48"43006F006D0070006C0065007800";

RESULT := CCSTOCCS_TRANS_TEXT_COMPLEX(CCSNUMFROM, CCSNUMTO, SOURCE,
    SOURCE_START, SOURCE_BYTES, DEST, DEST_START, DEST_BYTES,
    STATE, OPTION);
WRITE(OUT, <"RESULT = ", J4>, RESULT);
IF RESULT EQL CS_DATAOKV THEN
    WRITE(OUT, <"DEST = ", A7>, DEST);
CLOSE(OUT, LOCK);
END.
```

Note: *SOURCE_INX* and *DEST_INX* are not used twice to offset into the arrays. The pointer version of the Complex procedure ignores the *SOURCE_INX* and *DEST_INX* parameters on input to the procedure. On output to the procedure, the *SOURCE_INX* parameter is incremented by the amount of source consumed and the *DEST_INX* parameter is incremented by the amount of data mapped into the destination.

Output

```
RESULT = 1
DEST = Complex
```

In the preceding call, the parameters have the following meanings:

- *CCSNUMFROM* is the library number of the source CCS.
- *CCSNUMTO* is the library number of the destination CCS.
- *SOURCE* contains the characters to be mapped. The maximum is 64,000 words.
- *SOURCE_START* specifies the byte offset (0 relative) where the mapping in *SOURCE* starts. *SOURCE_START* is ignored on input by the *_PTR* version. The output marks the offset to where the mapping ended.
- *SOURCE_BYTES* is the number of bytes to be mapped beginning at the *SOURCE_INX*. The value of *SOURCE_BYTES* + *SOURCE_INX* must be equal to the value of *LEQ_SIZE* (*SOURCE*).
- *DEST* contains the mapped text. The size of this record might need to be larger than the record in *SOURCE-TEXT*. Using a record that is twice the size of the source always works.
- *DEST_START* specifies the byte offset (0 relative) where the mapped text is stored in *DEST*. *DEST_START* is ignored on input by the *_PTR* version. The output marks the offset where the mapped text ends.
- *DEST_BYTES* is the maximum number of bytes beginning at *DEST_INX* that can be filled with destination characters. The value of *DEST_BYTES* + *DEST_INX* must be equal to the value of *LEQ_SIZE* (*DEST*).

- STATE is a real array that is ten words long. STATE is owned, but untouched by, the caller. The following descriptions are for words 0-9:

Word	Description
Word 0	Indices to CENTRALSUPPORT tables
Word 1	Previous-operation status information
Word 2	Partial source character
Word 3	Special case optimization information
Words 4-9	Reserved

- OPTION indicates whether this is a continuation call. Following are the values and their meanings:

Value	Meaning
0	CS_OPT_INITIAL_COMPLETEV is the first call for a combination of source and destination coded character sets. The source data stream is complete and the data is mapped in one call.
5	CS_OPT_COMPLETEV is the beginning of a new source data stream, but NOT the first call for this combination of source and destination coded character sets. The source data stream is complete and the data is mapped in one call.
2	CS_OPT_INITIAL_HEADV is the first call for a combination of source and destination coded character sets. The source data is incomplete or the destination is not large enough to hold all of the data in one call.
7	CS_OPT_HEADV is the beginning of a new source data stream; it is not the first call for this combination of source and destination coded character sets. The source data stream is incomplete or the destination is not large enough to hold all of the data in one call.
3	CS_OPT_MIDDLEV is the second or higher call for a source data stream; it is not the last call.
1	CS_OPT_TAILV is the second or higher call for a source data stream and it is the last call.

The CCSTOCCS_TRANS_TEXT_COMPLEX procedure can return the following values:

CS_ARRAY_TOO_SMALLV (3001)	CS_FILE_ACCESS_ERRORV (1000)
CS_BAD_ARRAY_DESCRIPTIONV (3000)	CS_FAULTV (1001)
CS_BAD_DATA_LEN (3002)	CS_INCOMPLETE_CHARV (2005)
CS_DATA_NOT_FOUNDV (4002)	CS_INCOMPLETE_DATAV (2004)
CS_DATAOKV (1)	CS_SOFTERRV (1002)

CCSVSN_NAMES_NUMS

This procedure returns a list of coded character set names and numbers or a list of ccsversion names and numbers that are available on the system. The user can specify the type of list as either a ccsversion or a coded character set. The names and numbers are listed in two arrays. These arrays are ordered so that the names in the names array correspond to the numbers in the numbers array.

This procedure might be used to create a menu that lists the ccsversions on the host computer from which a user can choose. It might also be used to verify that the ccsversion to be used by a program is available on the host computer.

Example

This example returns a list of available ccsversion names and numbers on a system. This is an arbitrary list of three ccsversions and might not be the same on every system.

```
BEGIN

  $ INCLUDE INTL="SYMBOL/INTL/ALGOL/PROPERTIES." 10000000 - 49999999

  DEFINE MAX_NAME_LEN = 17 #;

  FILE OUT(KIND=DISK,UNITS=CHARACTERS,FILEUSE=IO,FILETYPE=0,NEWFILE,
           MAXRECSIZE=80,TITLE="OUT/ALGOL/CCSVSNAMESNUMS.");
  EBCDIC ARRAY NAMES_ARY[0:(20*MAX_NAME_LEN) - 1]; %holds 20 names
  INTEGER ARRAY NUMS_ARY[0:19]; %holds 20 numbers
  ARRAY LINEOUT[0:14];
  INTEGER TOTAL
    ,I
    ,RESULT;

  RESULT := CCSVSN_NAMES_NUMS(CS_CCSVERSIONV, TOTAL, NAMES_ARY,
                              NUMS_ARY);
  WRITE(OUT,<"RESULT = ",J4>,RESULT);
  IF RESULT EQL CS_DATAOKV THEN

BEGIN
  REPLACE LINEOUT BY " " FOR 80;
  WRITE(OUT, 80, LINEOUT);
  WRITE(OUT, <"CCSVSN NAME", T24,"CCSVSN NUMBER">);
  WRITE(OUT, <"-----", T24,"-----">);
  FOR I:= 0 STEP 1 UNTIL TOTAL-1 DO
    WRITE(OUT, <A17,T24,X8,J4>, NAMES_ARY[I*17], NUMS_ARY[I]);
  END;
  CLOSE(OUT,LOCK);
END.
```


Output

RESULT = 1

CCSVERSION NAME	CCSVERSION NUMBER
-----	-----
ASERIESNATIVE	0
SWISS	64
SWEDISH1	99

In this call, the parameters have the following meanings:

CS_CCSVERSIONV represents an integer that requests the names and numbers of the ccsversions. This is one of two types of information that can be returned in the output parameters:

Value	Value Name	Meaning
0	CS_CHARACTERSETV	Return the names and numbers of the coded character sets
1	CS_CCSVERSIONV	Return the names and numbers of the ccsversions

TOTAL is an integer returned by the procedure that contains the number of coded character set or ccsversion entries that exist.

NAMES_ARY is an EBCDIC array returned by the procedure. Each entry contains the name of a coded character set or ccsversion defined in the SYSTEM/CCSFILE. Each name takes 17 EBCDIC characters. The size of NAMES_ARY should be 17 times the number of convention names you might have on the system. The MLS Guide names all the coded character sets and ccsversions.

NUMS_ARY is an array of integers returned by the procedure that contains the coded character set or ccsversion numbers. NUMS_ARY contains all the coded character set or ccsversion numbers defined in file SYSTEM/CCSFILE. Each element in NUMS_ARY corresponds to an element in NAMES_ARY. Each number uses one element of NUMS_ARY. The MLS Guide gives all the numbers for the coded character sets and ccsversions.

RESULT is an integer returned by the procedure. It indicates whether or not an error occurred during the execution of the procedure. An explanation of the error result values and messages can be found at the end of this section. The values returned by this procedure include:

CS_ARRAY_TOO_SMALLV (3001)	CS_DATAOKV (1)
CS_BAD_ARRAY_DESCRIPTIONV (3000)	CS_FAULTV (1001)
CS_BAD_TYPE_CODEV (3006)	CS_SOFTERRV (1002)

CENTRALSTATUS

This procedure returns the name and number of the system default ccsversion, the name of the system default convention, and the name of the system default language.

This procedure might be used to provide a means for application users to inquire about the default settings on the host computer.

Example

This example returns the current values for the system default ccsversion, language, and convention. These are arbitrary system values and might not be the same on every system.

```
BEGIN

$ INCLUDE INTL="SYMBOL/INTL/ALGOL/PROPERTIES." 10000000 - 49999999

FILE OUT(KIND=DISK,UNITS=CHARACTERS,FILEUSE=IO,FILETYPE=0,NEWFILE,
        MAXRECSIZE=80,TITLE="OUT/ALGOL/CENTRALSTATUS.");
EBCDIC ARRAY SYS_INFO[0:50];
INTEGER ARRAY CONTROL_INFO[0:7];
ARRAY LINEOUT[0:14];
INTEGER RESULT;

RESULT := CENTRALSTATUS(SYS_INFO, CONTROL_INFO);
WRITE(OUT,<"RESULT = ",J4>,RESULT);
IF RESULT EQL CS_DATAOKV THEN
  BEGIN
    REPLACE LINEOUT BY " " FOR 80;
    WRITE(OUT,80,LINEOUT);
    WRITE(OUT,<"SYSTEM DEFAULTS">);
    WRITE(OUT,<"-----">);
    WRITE(OUT,<"CCSVERSION:",T18,A17>,SYS_INFO[0]);
    WRITE(OUT,<"LANGUAGE:",T18,A17>,SYS_INFO[17]);
    WRITE(OUT,<"CONVENTION:",T18,A17>,SYS_INFO[34]);
    WRITE(OUT,80,LINEOUT);
    WRITE(OUT,<"CONTROL ARRAY">);
    WRITE(OUT,<"FIELD MEANING",T40,"LOCATION",T52,"VALUE">);
    WRITE(OUT,<"-----",T40,"-----",T52,"-----">);
    WRITE(OUT,<"SYSTEM DEFAULT CCSVERSION NUMBER",T43,"[0]",T54,J4>,
          CONTROL_INFO[0]);
```

```

        END;
    CLOSE(OUT, LOCK);
END.
```

Output

```
RESULT = 1
```

```
SYSTEM DEFAULTS
```

```

-----
CCSVERSION:    ASERIESNATIVE
LANGUAGE:      ENGLISH
CONVENTION:    ASERIESNATIVE
```

CONTROL ARRAY FIELD MEANING	LOCATION	VALUE
-----	-----	-----
SYSTEM DEFAULT CCSVERSION NUMBER	[0]	0

In this call, the parameters have the following meanings:

SYS_INFO is an EBCDIC array returned by the procedure. It contains the system default ccsversion, language, and convention name in that order. Each name is 17 characters long. Names shorter than 17 characters are padded on the right with blanks. SYS_INFO should 51 characters to hold 3 names of 17 characters each.

CONTROL_INFO is an integer array returned by the procedure. It should be 8 words long. It contains the following information:

Location	Information
Word[0]	System default ccsversion number
Word[1] through [7]	Reserved

RESULT is an integer returned by the procedure. It indicates whether or not an error occurred during the execution of the procedure. An explanation of the error result values and messages can be found at the end of this section. The values returned by this procedure include:

CS_ARRAY_TOO_SMALLV (3001)	CS_FAULTV (1001)
CS_BAD_ARRAY_DESCRIPTIONV (3000)	CS_SOFTERRV (1002)
CS_DATAOKV (1)	

CNV_ADD

CNV_ADD adds a new convention to the *SYSTEM/CONVENTIONS file. The new definition is available immediately.

Example

This example adds a new convention definition named Utopia to the SYSTEM/CONVENTIONS file.

```
BEGIN

$ INCLUDE INTL="SYMBOL/INTL/ALGOL/PROPERTIES." 10000000 - 49999999

FILE OUT(KIND=DISK,UNITS=CHARACTERS,MYUSE=IO,FILETYPE=0,NEWFILE,
        MAXRECSIZE=80,TITLE="OUT/ALGOL/CNVADD.");
DEFINE MAX_NAME_LEN      = 17#;      % Maximum length of convention
                                % name
ARRAY ADD_ARY[0:62];
EBCDIC ARRAY CNV_NAME[0:MAX_NAME_LEN - 1];
INTEGER RESULT;

REPLACE CNV_NAME BY "UTOPIA";
ADD_ARY[0] := 12;                % Maximum digits
ADD_ARY[1] := 2;                 % # of fractional digits
ADD_ARY[2] := 2;                 % # of international frac digits
REPLACE POINTER(ADD_ARY[3]) BY "!UTA!";
REPLACE POINTER(ADD_ARY[5]) BY "!W!,!N!!D!,!Y!";
REPLACE POINTER(ADD_ARY[13]) BY "!1W!,!1N!!D!,!Y!";
REPLACE POINTER(ADD_ARY[21]) BY "!00!/!0D!/!Y!";
REPLACE POINTER(ADD_ARY[25]) BY "!0T!:!0M!:!0S!.!PPPP!";
REPLACE POINTER(ADD_ARY[33]) BY "!0T!:!0M!:!0S!";
REPLACE POINTER(ADD_ARY[37]) BY "!C[$]N[-]T[, :0,3]D[.]#!";
REPLACE POINTER(ADD_ARY[45]) BY "!N[-]T[, :0,3]D[.]#!";
ADD_ARY[61] := 66;
ADD_ARY[62] := 132;
RESULT :=CNV_ADD(CNV_NAME, ADD_ARY);
WRITE(OUT,<"RESULT = ",J4>,RESULT);
CLOSE(OUT,LOCK);
END.
```

Output

RESULT = 1

In this call, the parameters have the following meanings:

CNV_NAME is an EBCDIC array passed to the procedure. It contains the name of the convention to be added to the conventions file. If this parameter contains all blanks or zeros, the procedure uses the hierarchy to determine the convention to be used. Refer to the MLS Guide for the list of convention names and an explanation of the hierarchy.

ADD_ARY is a real array passed to the procedure. It contains the convention definition to be added to conventions in memory and to the *SYSTEM/CONVENTIONS file in use. For the procedure to work, all fields in ADD_ARY must contain data, or an appropriate error result is returned. Data in ADD_ARY is passed in fields as follows:

Field Meaning	Word
Maximum integer digits	0
Maximum decimal digits	1
Maximum international decimal digits	2
International currency notation	3
Long date template	5
Short date template	13
Numeric date template	21
Long time template	25
Numeric time template	33
Monetary template	37
Numeric template	45
Lines per page	61
Characters per line	62

Internationalization

The international currency notation field contains the international currency notation defined for the convention. The international currency notation is surrounded by a pair of matching delimiters that are not part of the notation. Any blanks inside the delimiters are significant and are treated as any other character. For example, the international currency notation for the ASERIESNATIVE convention is "USD "; the trailing blank is significant and the quotation marks are delimiters.

RESULT is an integer returned by the procedure. It indicates whether or not an error occurred during the execution of the procedure. An explanation of the error result values and messages can be found at the end of this section. The values returned by this procedure include:

CS_BAD_ALT_FRAC_DIGITSV (3017)	CS_BAD_NDATETEMPV (3020)
CS_BAD_ARRAY_DESCRIPTIONV (3000)	CS_BAD_NTIMETEMPV (3022)
CS_BAD_CPLV (3028)	CS_BAD_NUMTEMPV (3024)
CS_BAD_DATA_LEN (3002)	CS_BAD_SDATETEMPV (3019)
CS_BAD_FRACDIGITSV (3016)	CS_CNV_EXISTS_ERRV (3014)
CS_BAD_LDATETEMPV (3018)	CS_CNVFILE_NOTPRESENTV (3037)
CS_BAD_LPPV (3027)	CS_CONVENTION_NOT_FOUNDV (2002)
CS_BAD_LTIMETEMPV (3021)	CS_DATAOKV (1)
CS_BAD_LPPV (3027)	CS_FAULTV (1001)
CS_BAD_MAXDIGITSV (3015)	CS_FILE_ACCESS_ERRORV (1000)
CS_BAD_MONTEMPV (3023)	CS_SOFTERRV (1002)

CNV_CONVERTCURRENCY_STAR

This procedure converts a string containing digits and monetary symbols to a real number.

Example

This example converts an EBCDIC string containing digits and monetary symbols to a real number. The ASERIESNATIVE ccsversion is used.

```

BEGIN

$ INCLUDE INTL="*SYMBOL/INTL/ALGOL/PROPERTIES." 10000000 - 49999999

FILE OUT(KIND=DISK,UNITS=CHARACTERS,MYUSE=IO,FILETYPE=0,NEWFILE,
        MAXRECSIZE=80,TITLE="OUT/ALGOL/CNVCONVERTCURSTAR.");
DEFINE MAX_NAME_LEN      = 17#;      % Maximum length of convention
                                % name
EBCDIC ARRAY CNV_NAME[0:MAX_NAME_LEN - 1], CC_ARY[0:9];
INTEGER VSN_NUM, CC_ARY_SIZE, CNV_NAME_SIZE, RESULT;
REAL AMT;

VSN_NUM := 0;
REPLACE CC_ARY BY "$12,345.67";
REPLACE CNV_NAME BY "ASERIESNATIVE";
CC_ARY_SIZE := SIZE(CC_ARY);
CNV_NAME_SIZE := SIZE(CNV_NAME);
RESULT := CNV_CONVERTCURRENCY_STAR(VSN_NUM, CC_ARY, CC_ARY_SIZE,
                                CNV_NAME, CNV_NAME_SIZE, AMT);

WRITE(OUT,<"RESULT = ",J4>,RESULT);
IF RESULT EQL CS_DATAOKV THEN
    WRITE(OUT,<"AMT      = ",F8.2>,AMT);
CLOSE(OUT,LOCK);
END.

```

Output

```
RESULT = 1
AMT    = 12345.67
```

In this call, the parameters have the following meanings:

VSN_NUM is an integer passed to the procedure. It contains the number of the ccsversion that was in effect when the input string was created. If the input string contains alternate digits, then they will be translated to their corresponding 0-9 numbers. Refer to the MLS Guide for a list of the ccsversion numbers. The values allowed for VSN_NUM and the meanings of the values are as follows:

Value	Meaning
Greater than or equal to 0	Use the specified ccsversion number.
-2	Use the system default ccsversion. If the system default ccsversion is not available, an error is returned.

CC_ARY is an EBCDIC array that contains the input string of digits and monetary symbols.

CC_ARY_SIZE is an integer that contains the total number of bytes in CC_ARY.

CNV_NAME is an EBCDIC array that is passed to the procedure. It contains the name of the convention to be used to format the input string. If this parameter contains all blanks or zeros, the procedure uses the hierarchy to determine the convention to be used. Refer to the MLS Guide for the list of convention names and an explanation of the hierarchy.

CNV_NAME_SIZE is an integer that contains the total number of bytes in CNV_NAME.

AMT is a real number that is returned by the procedure. It contains the converted real number.

RESULT is an integer returned by the procedure. It indicates whether or not an error occurred during the execution of the procedure. An explanation of the error result values and messages can be found at the end of this section. The values returned by this procedure include:

CS_ARRAY_TOO_SMALLV (3001)	CS_FILE_ACCESS_ERRORV (1000)
CS_BAD_ARRAY_DESCRIPTIONV (3000)	CS_DATA_NOT_FOUNDV (4002)
CS_BAD_DATA_LEN (3002)	CS_DATAOKV (1)
CS_BAD_INPUTVALV (3035)	CS_FAULTV (1001)
CS_CONVENTION_NOT_FOUNDV (2002)	CS_SOFTERRV (1002)

CNV_CONVERTDATE_STAR

This procedure converts a formatted numeric date passed as a parameter to the procedure to the format YYYYMMDD. The numeric date must be formatted according to the numeric date format template defined in the designated convention set.

Example

This example converts a date from ASERIESNATIVE numeric form to standard input form. The ASERIESNATIVE ccsversion is used.

```
BEGIN

$ INCLUDE INTL="*SYMBOL/INTL/ALGOL/PROPERTIES." 10000000 - 49999999

FILE OUT(KIND=DISK,UNITS=CHARACTERS,MYUSE=IO,FILETYPE=0,NEWFILE,
        MAXRECSIZE=80,TITLE="OUT/ALGOL/CNVCONVERTDATSTAR.");
DEFINE MAX_NAME_LEN      = 17#;      % Maximum length of convention
                                % name
EBCDIC ARRAY CNV_NAME[0:MAX_NAME_LEN - 1], CD_ARY[0:9], DATE_ARY[0:7];
INTEGER VSN_NUM, CD_ARY_SIZE, CNV_NAME_SIZE, DATE_ARY_SIZE, RESULT;

VSN_NUM := 0;
REPLACE CD_ARY BY "09/15/1990";
REPLACE CNV_NAME BY "ASERIESNATIVE";
CD_ARY_SIZE := SIZE(CD_ARY);
CNV_NAME_SIZE := SIZE(CNV_NAME);
DATE_ARY_SIZE := SIZE(DATE_ARY);
RESULT := CNV_CONVERTDATE_STAR(VSN_NUM, CD_ARY, CD_ARY_SIZE,
                                CNV_NAME, CNV_NAME_SIZE,
                                DATE_ARY, DATE_ARY_SIZE);

WRITE(OUT,<"RESULT = ",J4>,RESULT);
IF RESULT EQL CS_DATAOKV THEN
    WRITE(OUT,<"DATE_ARY = ",C8>,DATE_ARY);
CLOSE(OUT,LOCK);
END.
```

Output

```
RESULT    = 1
DATE_ARY  = 19900915
```

In this call, the parameters have the following meanings:

VSN_NUM is an integer passed to the procedure. It contains the ccsversion number that was in effect when date was formatted. If the input string contains alternate digits, then they will be translated to their corresponding 0-9 numbers. Refer to the MLS Guide for a list of the ccsversion numbers. The values allowed for VSN_NUM and the meanings of the values are as follows:

Value	Meaning
Greater than or equal to 0	Use the specified ccsversion number.
-2	Use the system default ccsversion. If the system default ccsversion is not available, an error is returned.

CD_ARY is an EBCDIC array passed to the procedure. It contains the formatted date.

CD_ARY_SIZE is an integer that contains the total number of bytes in CD_ARY.

CNV_NAME is an EBCDIC array passed to the procedure. It contains the name of the convention from which the date formatting templates are retrieved.

CNV_NAME_SIZE is an integer that contains the total number of bytes in CNV_NAME.

DATE_ARY is an EBCDIC array returned by the procedure. It contains the converted date in the form YYYYMMDD.

DATE_ARY_SIZE is an integer that contains the total number of bytes in DATE_ARY.

RESULT is an integer returned by the procedure. It indicates whether or not an error occurred during the execution of the procedure. An explanation of the error result values and messages can be found at the end of this section. The values returned by this procedure include:

CS_ARRAY_TOO_SMALLV (3001)	CS_BAD_YEAR_YYV (3045)
CS_BAD_ARRAY_DESCRIPTIONV (3000)	CS_BAD_YEARV (3046)
CS_BAD_DATA_LEN (3002)	CS_CONVENTION_NOT_FOUNDV (2002)
CS_BADDATEINPUTV (3012)	CS_DATA_NOT_FOUNDV (4002)
CS_BAD_DATESEPARATORV (3044)	CS_DATAOKV (1)
CS_BAD_DAYV (3048)	CS_FAULTV (1001)
CS_BAD_MONTHV (3047)	CS_FILE_ACCESS_ERRORV (1000)
CS_BAD_TEMPCHARV (3011)	CS_NO_DATEINPUTV (3049)
CS_BAD_TYPE_CODEV (3006)	CS_SOFTERRV (1002)

CNV_CONVERTNUMERIC_STAR

This procedure converts a string containing digits and numeric symbols to a real number.

Example

This example converts an EBCDIC string containing digits and numeric symbols to a real number. The ASERIESNATIVE ccsversion is used.

```
BEGIN

$ INCLUDE INTL="*SYMBOL/INTL/ALGOL/PROPERTIES." 10000000 - 49999999

FILE OUT(KIND=DISK,UNITS=CHARACTERS,MYUSE=IO,FILETYPE=0,NEWFILE,
        MAXRECSIZE=80,TITLE="OUT/ALGOL/CNV_CONVERTNUMSTAR.");
DEFINE MAX_NAME_LEN      = 17#;      % Maximum length of convention
                                % name
EBCDIC ARRAY CNV_NAME[0:MAX_NAME_LEN - 1], CC_ARY[0:9];
INTEGER VSN_NUM, CC_ARY_SIZE, CNV_NAME_SIZE, RESULT;
REAL AMT;

VSN_NUM := 0;
REPLACE CC_ARY BY "12,345.67";
REPLACE CNV_NAME BY "ASERIESNATIVE";
CC_ARY_SIZE := SIZE(CC_ARY);
CNV_NAME_SIZE := SIZE(CNV_NAME);
RESULT := CNV_CONVERTNUMERIC_STAR(VSN_NUM, CC_ARY, CC_ARY_SIZE,
                                CNV_NAME, CNV_NAME_SIZE, AMT);
WRITE(OUT,<"RESULT = ",J4>,RESULT);
IF RESULT EQL CS_DATAOKV THEN
    WRITE(OUT,<"AMT      = ",F8.2>,AMT);
CLOSE(OUT,LOCK);
END.
```

Output

```
RESULT = 1
AMT    = 12345.67
```

In this call, the parameters have the following meanings:

VSN_NUM is an integer that is passed to the procedure. It contains the number of the ccsversion that was in effect when the input string was created. If the input string contains alternate digits, then they will be translated to their corresponding 0-9 numbers. Refer to the MLS Guide for a list of the ccsversion numbers. The values allowed for VSN_NUM and the meanings of the values are as follows:

Value	Meaning
Greater than or equal to 0	Use the specified ccsversion number.
-2	Use the system default ccsversion. If the system default ccsversion is not available, an error is returned.

CC_ARY is an EBCDIC array that is passed to the procedure. It contains the string of digits and numeric symbols to be converted.

CC_ARY_SIZE is an integer that contains the total number of bytes in CC_ARY.

CNV_NAME is an EBCDIC array that is passed to the procedure. It contains the name of the convention to be used to format the input string.

CNV_NAME_SIZE is an integer that contains the total number of bytes in CNV_NAME.

AMT is a result returned by the procedure. It contains the converted real number.

RESULT is an integer returned by the procedure. It indicates whether or not an error occurred during the execution of the procedure. An explanation of the error result values and messages can be found at the end of this section. The values returned by this procedure include:

CS_ARRAY_TOO_SMALLV (3001)	CS_DATA_NOT_FOUNDV (4002)
CS_BAD_ARRAY_DESCRIPTIONV (3000)	CS_DATAOKV (1)
CS_BAD_DATA_LEN (3002)	CS_FAULTV (1001)
CS_BAD_INPUTVALV (3035)	CS_FILE_ACCESS_ERRORV (1000)
CS_CONVENTION_NOT_FOUNDV (2002)	CS_SOFTERRV (1002)

CNV_CONVERTTIME_STAR

This procedure converts a formatted numeric time passed as a parameter to the procedure to the format HHMMSS. The numeric time must be formatted according to the numeric time format template defined in the designated convention.

Example

This example converts a time from ASERIESNATIVE numeric form to standard input form. The ASERIESNATIVE ccsversion is used.

```
BEGIN

$ INCLUDE INTL="*SYMBOL/INTL/ALGOL/PROPERTIES." 10000000 - 49999999

FILE OUT(KIND=DISK,UNITS=CHARACTERS,MYUSE=IO,FILETYPE=0,NEWFILE,
         MAXRECSIZE=80,TITLE="OUT/ALGOL/CNVCONVERTTIMSTAR.");
DEFINE MAX_NAME_LEN      = 17#;      % Maximum length of convention
                                % name
EBCDIC ARRAY CNV_NAME[0:MAX_NAME_LEN - 1], CT_ARY[0:7], TIME_ARY[0:5];
INTEGER VSN_NUM, CT_ARY_SIZE, CNV_NAME_SIZE, TIME_ARY_SIZE, RESULT;

VSN_NUM := 0;
REPLACE CT_ARY BY "11:49:58";
REPLACE CNV_NAME BY "ASERIESNATIVE";
CT_ARY_SIZE := SIZE(CT_ARY);
CNV_NAME_SIZE := SIZE(CNV_NAME);
TIME_ARY_SIZE := SIZE(TIME_ARY);
RESULT := CNV_CONVERTTIME_STAR(VSN_NUM, CT_ARY, CT_ARY_SIZE,
                              CNV_NAME, CNV_NAME_SIZE,
                              TIME_ARY, TIME_ARY_SIZE);

WRITE(OUT,<"RESULT = ",J4>,RESULT);
IF RESULT EQL CS_DATAOKV THEN
  WRITE(OUT,<"TIME_ARY = ",C6>,TIME_ARY);
CLOSE(OUT,LOCK);
END.
```

Output

```
RESULT    = 1
TIME_ARY  = 114958
```

In this call, the parameters have the following meanings:

VSN_NUM is an integer passed to the procedure. It contains the ccsversion number that was in effect when the time was formatted. If the input string contains alternate digits, then they will be translated to their corresponding 0-9 numbers. Refer to the MLS Guide for a list of the ccsversion numbers. The values allowed for VSN_NUM and the meanings of the values are as follows:

Value	Meaning
Greater than or equal to 0	Use the specified ccsversion number.
-2	Use the system default ccsversion. If the system default ccsversion is not available, an error is returned.

CT_ARY is an EBCDIC array passed to the procedure. It contains the formatted time.

CT_ARY_SIZE is an integer that contains the total number of bytes in CT_ARY.

CNV_NAME is an EBCDIC array passed to the procedure. It contains the name of the convention that was used to format the input time. If this parameter contains all blanks or zeros, the procedure uses the hierarchy to determine the convention to be used. Refer to the MLS Guide for the list of convention names and an explanation of the hierarchy.

CNV_NAME_SIZE is an integer that contains the total number of bytes in CNV_NAME.

TIME_ARY is an EBCDIC array returned by the procedure. It contains the converted time in the form HHMMSS.

TIME_ARY_SIZE is an integer that contains the total number of bytes in TIME_ARY.

Internationalization

RESULT is an integer returned by the procedure. It indicates whether or not an error occurred during the execution of the procedure. An explanation of the error result values and messages can be found at the end of this section. The values returned by this procedure include:

CS_ARRAY_TOO_SMALLV (3001)	CS_BAD_12HOURV (3052)
CS_BAD_ARRAY_DESCRIPTIONV (3000)	CS_CONVENTION_NOT_FOUNDV (2002)
CS_BAD_DATA_LEN (3002)	CS_DATA_NOT_FOUNDV (4002)
CS_BAD_HOURV (3051)	CS_DATAOKV (1)
CS_BAD_MINUTEV (3053)	CS_FAULTV (1001)
CS_BAD_PSECONDV (3055)	CS_FILE_ACCESS_ERRORV (1000)
CS_BAD_SECONDV (3054)	CS_NO_TIMEINPUTV (3056)
CS_BADTIMEINPUTV (3013)	CS_SOFTERRV (1002)
CS_BAD_TIMESEPARATORV (3050)	

CNV_CURRENCYEDIT

This procedure converts a real number that represents a currency value to a formatted monetary string. The string contains the currency value with formatting symbols included. The formatting performed by the procedure is done according to the convention specified. The MLS Guide describes all the conventions and the type of currency formatting associated with each convention.

The procedure might be used to print a report with the numeric and currency formats for the Costa Rican convention (for example CRC 89.99) or for the Norway convention (for example, NKR 89.99).

Example

The following example converts a real number and edits monetary symbols from convention Denmark into an EBCDIC string.

```
BEGIN

$ INCLUDE INTL="*SYMBOL/INTL/ALGOL/PROPERTIES." 10000000 - 49999999

FILE OUT (KIND=DISK,UNITS=CHARACTERS,MYUSE=IO,FILETYPE=0,NEWFILE,
          MAXRECSIZE=80,TITLE="OUT/ALGOL/CNVCUREDIT.");
DEFINE MAX_NAME_LEN      = 17#;      % Maximum length of convention
                               % name
EBCDIC ARRAY CNV_NAME[0:MAX_NAME_LEN - 1], CE_ARY[0:29];
INTEGER CE_LEN, RESULT;
REAL AMT;

AMT := 12345.67;
REPLACE CNV_NAME BY "DENMARK";
RESULT := CNV_CURRENCYEDIT(AMT, CNV_NAME, CE_LEN, CE_ARY);
WRITE(OUT,<"RESULT = ",J4>,RESULT);
IF RESULT EQL CS_DATAOKV THEN
  WRITE(OUT,<"CE_ARY = ",C30>,CE_ARY);
CLOSE(OUT,LOCK);
END.
```

Output

```
RESULT = 1  
CE_ARY = Kr.12.345,67
```

In this call, the parameters have the following meanings:

AMT is a real number passed to the procedure. It contains the value of the currency.

CNV_NAME is an EBCDIC array passed to the procedure. It contains the name of the convention to be used to format the currency value. If this parameter contains all blanks or zeros, the procedure uses the hierarchy to determine the convention to be used. Refer to the MLS Guide for the list of convention names and an explanation of the hierarchy.

CE_LEN is an integer returned by the procedure. It contains the number of characters in CE_ARY.

CE_ARY is an EBCDIC array returned by the procedure. It contains the formatted value.

RESULT is an integer returned by the procedure. It indicates whether or not an error occurred during the execution of the procedure. An explanation of the error result values and messages can be found at the end of this section. The values returned by this procedure include:

CS_BAD_ARRAY_DESCRIPTIONV (3000)	CS_DATAOKV (1)
CS_BAD_DATA_LEN (3002)	CS_FAULTV (1001)
CS_CONVENTION_NOT_FOUNDV (2002)	CS_SOFTERRV (1002)

CNV_CURRENCYEDIT_DOUBLE

This procedure converts a double-precision integer that represents a currency value to a formatted monetary string. The string contains the currency value with formatting symbols included. The formatting performed by the procedure is done according to the convention specified. The MLS Guide describes all the conventions and the type of currency formatting associated with each convention.

The procedure might be used to print a report with the numeric and currency formats for the Costa Rican convention (for example, CRC 89.99) or for the Norway convention (for example, NKR 89.99).

Example

The following example converts a double-precision integer and edits monetary symbols from convention Denmark into an EBCDIC string.

```
BEGIN

  $ INCLUDE INTL="SYMBOL/INTL/ALGOL/PROPERTIES." 10000000 - 49999999

  FILE OUT(KIND=DISK,UNITS=CHARACTERS,MYUSE=IO,FILETYPE=0, NEWFILE,
           MAXRECSIZE=80,TITLE="OUT/ALGOL/CUREDITDOUB.");
  DEFINE MAX_NAME_LEN = 17#; % Maximum length of convention name
  EBCDIC ARRAY CNV_NAME[0:MAX_NAME_LEN - 1],
  CE_ARY[0:29];
  INTEGER CE_LEN, RESULT, PRECISION;
  DOUBLE AMT;

  AMT := 1234567 @@ 0;
  PRECISION := 2;
  REPLACE CNV_NAME BY "DENMARK";
  RESULT := CNV_CURRENCYEDIT_DOUBLE
            (AMT, PRECISION, CNV_NAME, CE_LEN, CE_ARY);
  WRITE(OUT,<"RESULT = ",J4>,RESULT);
  IF RESULT EQL CS_DATAOKV THEN
    WRITE(OUT,<"CE_ARY = ",C30>,CE_ARY);
  CLOSE(OUT,LOCK);
END.
```

Output

```
RESULT = 1
CE_ARY = Kr.12.345,67
```

In this call, the parameters have the following meanings:

AMT is a double-precision integer passed to the procedure. This integer contains the value of the currency.

CNV_NAME is an EBCDIC array passed to the procedure. This array contains the name of the convention to be used to format the currency value. If this parameter contains all blanks or zeros, the procedure uses the hierarchy to determine the convention to be used. Refer to the MLS Guide for the list of convention names and an explanation of the hierarchy.

CE_LEN is an integer returned by the procedure. This integer contains the number of characters in CE_ARY.

CE_ARY is an EBCDIC array returned by the procedure. This array contains the formatted value.

PRECISION is an integer passed to the procedure. This integer specifies the number of digits in AMT to be placed after the decimal symbol.

RESULT is an integer returned by the procedure. This integer indicates whether an error occurred during the execution of the procedure. An explanation of the error result values and messages can be found at the end of this section. The values returned by this procedure include:

CS_BAD_ARRAY_DESCRIPTIONV (3000)	CS_CONVENTION_NOT_FOUNDV (2002)
CS_BAD_AMTV (3009)	CS_DATAOKV (1)
CS_BAD_DATA_LENv (3002)	CS_FAULTV (1001)
CS_BAD_PRECISIONV (3038)	CS_SOFTERRV (1002)

CNV_CURRENCYEDITTMP

This procedure receives a real number and formats it to represent a currency value according to the template specified in the TEMP_ARY parameter. The template can be retrieved for any convention from the CNV_TEMPLATE procedure. It can also be created by the user. CE_ARY contains the formatted currency value. The MLS Guide describes the symbols used to create a template.

Example

This example converts a real number and edits in numeric symbols which are passed in as a monetary template. The currency symbol is retrieved from the monetary template in the ASERIESNATIVE convention and edited into the EBCDIC string.

```
BEGIN

$ INCLUDE INTL="*SYMBOL/INTL/ALGOL/PROPERTIES." 10000000 - 49999999

FILE OUT(KIND=DISK,UNITS=CHARACTERS,MYUSE=IO,FILETYPE=0,NEWFILE,
        MAXRECSIZE=80,TITLE="OUT/ALGOL/CNVCUREDITTMP.");
DEFINE MAX_NAME_LEN      = 17#;      % Maximum length of convention
                                % name
EBCDIC ARRAY CNV_NAME[0:MAX_NAME_LEN - 1], CE_ARY[0:29],
            TMP_ARY[0:47];
INTEGER CE_LEN, RESULT;
REAL AMT;

AMT := 12345.67;
REPLACE CNV_NAME BY "ASERIESNATIVE";
REPLACE TMP_ARY BY "!CN[-]T[, :0,3]D[.]#!";
RESULT := CNV_CURRENCYEDITTMP(AMT, TMP_ARY, CNV_NAME, CE_LEN, CE_ARY);
WRITE(OUT,<"RESULT = ",J4>,RESULT);
IF RESULT EQL CS_DATAOKV THEN
    WRITE(OUT,<"CE_ARY = ",C30>,CE_ARY);
CLOSE(OUT,LOCK);
END.
```

Output

```
RESULT = 1  
CE_ARY = $12,345.67
```

In this call, the parameters have the following meanings:

AMT is a real number passed to the procedure. It contains the value of the currency.

TMP_ARY is an EBCDIC array passed to the procedure. It contains the formatting template used to format the currency value.

CNV_NAME is an EBCDIC array passed to the procedure. It contains the name of the convention to be used. When a caller-supplied monetary template (in TMP_ARY) contains one or more control characters in simple form (control character without a symbol definition enclosed in square brackets ([]) following it), symbols associated with those control characters are retrieved from the monetary template in the convention specified by CNV_NAME.

For example, if a caller-provided monetary template is !N[-]CT[,:0,3]D[.]#! and CNV_NAME contains Sweden, then the local currency symbol is retrieved from the monetary template in Sweden convention and the amount 12345.67 is edited into a formatted string as Kr.12,134.67.

CE_LEN is an integer returned by the procedure. It contains the length of the formatted value in the output array.

CE_ARY is an EBCDIC array returned by the procedure. It contains the formatted monetary value.

RESULT is an integer returned by the procedure. It indicates whether or not an error occurred during the execution of the procedure. An explanation of the error result values and messages can be found at the end of this section. The values returned by this procedure include:

CS_BAD_ARRAY_DESCRIPTIONV (3000)	CS_FAULTV (1001)
CS_BAD_DATA_LEN (3002)	CS_SOFTERRV (1002)
CS_DATAOKV (1)	

CNV_CURRENCYEDITTMP_DOUBLE

This procedure receives a double-precision integer and formats it to represent a currency value according to the template specified in the TEMP_ARY parameter. The template can be retrieved for any convention from the CNV_TEMPLATE procedure. The template can also be created by the user. CE_ARY contains the formatted currency value. The MLS Guide describes the symbols used to create a template.

Example

This example converts a double-precision integer and edits in numeric symbols which are passed in as a monetary template. The currency symbol is retrieved from the monetary template in the ASERIESNATIVE convention and edited into the EBCDIC string.

```
BEGIN

  $ INCLUDE INTL="SYMBOL/INTL/ALGOL/PROPERTIES." 10000000 - 49999999

  FILE OUT(KIND=DISK,UNITS=CHARACTERS,MYUSE=IO,FILETYPE=0,
  NEWFILE, MAXRECSIZE=80,TITLE="OUT/ALGOL/CUREDITTMPDOUB.");
  DEFINE MAX_NAME_LEN = 17#; % Maximum length of convention name
  EBCDIC ARRAY CNV_NAME[0:MAX_NAME_LEN - 1],
  CE_ARY[0:29], TMP_ARY[0:47];
  INTEGER CE_LEN, RESULT, PRECISION;
  DOUBLE AMT;

  AMT := 1234567 @@ 0;
  PRECISION := 2;
  REPLACE CNV_NAME BY "ASERIESNATIVE";
  REPLACE TMP_ARY BY "!N[-]CT[, :0,3]D[.]#!";
  RESULT := CNV_CURRENCYEDITTMP_DOUBLE
            (AMT, PRECISION, TMP_ARY, CNV_NAME, CE_LEN, CE_ARY);

  WRITE(OUT,<"RESULT = ",J4>,RESULT);
  IF RESULT EQL CS_DATAOKV THEN
    WRITE(OUT,<"CE_ARY = ",C30>,CE_ARY);
  CLOSE(OUT,LOCK);
END.
```

Output

```
RESULT = 1  
CE_ARY = $12,345.67
```

In this call, the parameters have the following meanings:

AMT is a double-precision integer passed to the procedure. This integer contains the value of the currency.

TMP_ARY is an EBCDIC array passed to the procedure. This array contains the formatting template used to format the currency value.

CNV_NAME is an EBCDIC array passed to the procedure. This array contains the name of the convention to be used. When a caller-supplied monetary template (in TMP_ARY) contains one or more control characters in simple form (control character without a symbol definition enclosed in square brackets ([]) following it), symbols associated with those control characters are retrieved from the monetary template in the convention specified by CNV_NAME.

For example, if a caller-provided monetary template is !N[-]CT[,:0,3]D[.]#! and CNV_NAME contains Sweden, then the local currency symbol is retrieved from the monetary template in Sweden convention and the amount 12345.67 is edited into a formatted string as Kr.12,134.67.

CE_LEN is an integer returned by the procedure. This integer contains the length of the formatted value in the output array.

CE_ARY is an EBCDIC array returned by the procedure. This array contains the formatted monetary value.

PRECISION is an integer passed to the procedure. This integer specifies the number of digits in AMT to be placed after the decimal symbol.

RESULT is an integer returned by the procedure. This integer indicates whether an error occurred during the execution of the procedure. An explanation of the error result values and messages can be found at the end of this section. The values returned by this procedure include:

CS_BAD_ARRAY_DESCRIPTIONV (3000)	CS_DATAOKV (1)
CS_BAD_AMTV (3009)	CS_FAULTV (1001)
CS_BAD_DATA_LEN (3002)	CS_SOFTERRV (1002)
CS_BAD_PRECISIONV (3038)	

CNV_DELETE

This procedure deletes an existing convention definition from the *SYSTEM/CONVENTIONS file. The convention is deleted after the next halt/load.

Note that this procedure can be used to delete only conventions that have been created by a user. Convention sets provided to you cannot be deleted.

Example

This example deletes a convention named Utopia from the conventions file.

```
BEGIN

$ INCLUDE INTL="*SYMBOL/INTL/ALGOL/PROPERTIES." 10000000 - 49999999

FILE OUT(KIND=DISK,UNITS=CHARACTERS,MYUSE=IO,FILETYPE=0,NEWFILE,
        MAXRECSIZE=80,TITLE="OUT/ALGOL/CNVDELETE.");
DEFINE MAX_NAME_LEN      = 17#;    % Maximum length of convention
                                % name
EBCDIC ARRAY CNV_NAME[0:MAX_NAME_LEN - 1];
INTEGER RESULT;

REPLACE CNV_NAME BY "UTOPIA";
RESULT := CNV_DELETE(CNV_NAME);
WRITE(OUT,<"RESULT = ",J4>,RESULT);
CLOSE(OUT,LOCK);
END.
```

Output

```
RESULT = 1
```

In this call, the parameters have the following meanings:

CNV_NAME is an EBCDIC array passed to the procedure. It contains the name of the convention to be deleted from the conventions file.

RESULT is an integer returned by the procedure. It indicates whether or not an error occurred during the execution of the procedure. An explanation of the error result values and messages can be found at the end of this section. The values returned by this procedure include:

CS_BAD_ARRAY_DESCRIPTIONV (3000)	CS_DATAOKV (1)
CS_BAD_DATA_LEN (3002)	CS_FAULTV (1001)
CS_CNV_NOTAVAILV (3036)	CS_FILE_ACCESS_ERRORV (1000)
CS_CNVFILE_NOTPRESENTV (3037)	CS_SOFTERRV (1002)

CNV_DISPLAYMODEL

This procedure returns a display model for either the numeric date or the numeric time, whichever one is requested in the TYP parameter. A display model is a format that can be displayed to the user to show the form of the requested input. For example, YY/DD/MM is a display model that shows a user that the date must be entered in that form. The procedure creates the display model according to the convention and language specified.

Example

This example obtains a date display model from the ASERIESNATIVE convention. The display model is translated to English and returned in DM_ARY.

```
BEGIN

$ INCLUDE INTL="*SYMBOL/INTL/ALGOL/PROPERTIES." 10000000 - 49999999

FILE OUT(KIND=DISK,UNITS=CHARACTERS,MYUSE=IO,FILETYPE=0,NEWFILE,
        MAXRECSIZE=80,TITLE="OUT/ALGOL/CNVDSMODEL.");
DEFINE MAX_NAME_LEN      = 17#;    % Maximum length of convention
                                % name
EBCDIC ARRAY CNV_NAME[0:MAX_NAME_LEN - 1], DM_ARY[0:9],
            LANG_NAME[0:MAX_NAME_LEN - 1];
INTEGER DM_LEN, RESULT;

REPLACE CNV_NAME BY "ASERIESNATIVE";
REPLACE LANG_NAME BY "ENGLISH";
RESULT := CNV_DISPLAYMODEL(CS_DATE_DISPLAYMODELV, CNV_NAME, LANG_NAME,
                          DM_LEN, DM_ARY);
WRITE(OUT,<"RESULT = ",J4>,RESULT);
IF RESULT EQL CS_DATAOKV THEN
    WRITE(OUT,<"DM_ARY = ",C10>,DM_ARY);
CLOSE(OUT,LOCK);
END.
```

Output

RESULT = 1
 DM_ARY = mm/dd/yyyy

In this call, the parameters have the following meanings:

CS_DATE_DISPLAYMODELV represents an integer that requests that the display model for the numeric date be returned. It is one of two display model options:

Value	Value Name	Meaning
0	CS_DATE_DISPLAYMODELV	The display model is for a numeric date.
1	CS_TIME_DISPLAYMODELV	The display model is for a numeric time.

CNV_NAME is an EBCDIC array passed to the procedure. It contains the name of the convention from which the date or time model is retrieved. If this parameter contains all blanks or zeros, the procedure uses the hierarchy to determine the convention to be used. Refer to the MLS Guide for the list of convention names and the explanation of the hierarchy.

LANG_NAME is an EBCDIC array passed to the procedure. It contains the language name to be used in formatting the date or time. If this parameter contains all blanks or zeros, the procedure uses the hierarchy to determine the language to be used. Refer to the MLS Guide for information about determining the valid language names and an explanation of the hierarchy.

DM_LEN is an integer that contains the number of characters in DM_ARY.

DM_ARY is an EBCDIC array returned to the procedure. It contains the requested display model.

RESULT is an integer returned by the procedure. It indicates whether or not an error occurred during the execution of the procedure. An explanation of the error result values and messages can be found at the end of this section. The values returned by this procedure include:

CS_ARRAY_TOO_SMALLV (3001)	CS_FAULTV (1001)
CS_BAD_ARRAY_DESCRIPTIONV (3000)	CS_DATAOKV (1)
CS_BAD_DATA_LEN (3002)	CS_LANGUAGE_NOT_FOUNDV (2001)
CS_BAD_TYPE_CODEV (3006)	CS_SOFTERRV (1002)
CS_CONVENTION_NOT_FOUNDV (2002)	

CNV_FORMATDATE

This procedure formats a date, specified in DATE_ARY, according to the convention specified in CNV_NAME. The formatted date is returned in FD_ARY in the language specified by LANG_NAME.

Example

This example formats the date in numeric form using the Netherlands convention. The system default language is specified.

```
BEGIN

$ INCLUDE INTL="*SYMBOL/INTL/ALGOL/PROPERTIES." 10000000 - 49999999

FILE OUT(KIND=DISK,UNITS=CHARACTERS,MYUSE=IO,FILETYPE=0,NEWFILE,
        MAXRECSIZE=80,TITLE="OUT/ALGOL/CNVFMTDATE.");
DEFINE MAX_NAME_LEN      = 17#;    % Maximum length of convention
                                % name
EBCDIC ARRAY CNV_NAME[0:MAX_NAME_LEN - 1], DATE_ARY[0:7], FD_ARY[0:9],
            LANG_NAME[0:MAX_NAME_LEN - 1];
INTEGER FD_LEN, RESULT;

REPLACE DATE_ARY BY "17760704";
REPLACE CNV_NAME BY "NETHERLANDS";
RESULT := CNV_FORMATDATE(CS_NDATEV, DATE_ARY, CNV_NAME, LANG_NAME,
                        FD_LEN, FD_ARY);
WRITE(OUT,<"RESULT = ",J4>,RESULT);
IF RESULT EQL CS_DATAOKV THEN
    WRITE(OUT,<"FD_ARY = ",C10>,FD_ARY);
CLOSE(OUT,LOCK);
END.
```

Output

```
RESULT = 1
FD_ARY = 4.7.76
```

In this call, the parameters have the following meanings:

CS_NDATEV represents an integer that requests a numeric date format to be returned. It is one of three date format options:

Value	Value Name	Meaning
0	CS_LDATEV	Use the long date format.
1	CS_SDATEV	Use the short date format.
2	CS_NDATEV	Use the numeric date format.

DATE_ARY is an EBCDIC array passed to the procedure. It contains the date to be formatted. The date must be in the form YYYYMMDD, left justified. The fields of the array have fixed positions. Blanks or zeros must be used in any fields that are omitted.

CNV_NAME is an EBCDIC array passed to the procedure. It contains the name of the convention to be used to edit the date value. The maximum length of each name is 17 characters. If this parameter contains all blanks or zeros, the procedure uses the hierarchy to determine the convention to be used. Refer to the MLS Guide for the list of convention names and the explanation of the hierarchy.

LANG_NAME is an EBCDIC array passed to the procedure. It contains the language to be used in formatting the date. The maximum length of each name is 17 characters. If this parameter contains all blanks or zeros, the procedure uses the hierarchy to determine the language to be used. Refer to the MLS Guide for information about determining the valid language names on the system and an explanation of the hierarchy.

FD_LEN is an integer returned by the procedure. It contains the length of the formatted date.

FD_ARY is an EBCDIC array returned by the procedure that contains the formatted date.

RESULT is an integer returned by the procedure. It indicates whether or not an error occurred during the execution of the procedure. An explanation of the error result values and messages can be found at the end of this section. The values returned by this procedure include:

CS_ARRAY_TOO_SMALLV (3001)	CS_BAD_YEAR_YV (3045)
CS_BAD_ARRAY_DESCRIPTIONV (3000)	CS_CONVENTION_NOT_FOUNDV (2002)
CS_BAD_DATA_LEN (3002)	CS_DATAOKV (1)
CS_BADDATEINPUTV (3012)	CS_FAULTV (1001)
CS_BAD_DAYOFYEARV (3058)	CS_FIELD_TRUNCATEDV (2003)
CS_BAD_DAYV (3048)	CS_LANGUAGE_NOT_FOUNDV (2001)
CS_BAD_MONTHV (3047)	CS_MISSING_MONTH_COMPONENTV (3057)
CS_BAD_TYPE_CODEV (3006)	CS_MISSING_DATE_COMPONENTV (3059)
CS_BAD_YEARV (3046)	CS_SOFTERRV (1002)

CNV_FORMATDATETMP

This procedure formats a date, specified in DATE_ARY, according to a template, specified in TMP_ARY. The template can be retrieved for any convention from the CNV_TEMPLATE procedure. It can also be created by the user. The formatted date is returned in the language specified in LANG_NAME.

Example

This example formats a date using a template provided by the calling program. The formatted date is translated into English and returned in FD_ARY. The date consists of an unabridged name of a day of the week, an abbreviated month name, a numeric day of the month, a day of the month suffix "rd", and a numeric year.

```
BEGIN

$ INCLUDE INTL="*SYMBOL/INTL/ALGOL/PROPERTIES." 10000000 - 49999999

FILE OUT(KIND=DISK,UNITS=CHARACTERS,MYUSE=IO,FILETYPE=0,NEWFILE,
        MAXRECSIZE=80,TITLE="OUT/ALGOL/CNVFMTDATETMP.");
DEFINE MAX_NAME_LEN      = 17#;      % Maximum length of convention
                                % name
EBCDIC ARRAY TMP_ARY[0:47], DATE_ARY[0:7], FD_ARY[0:44],
            LANG_NAME[0:MAX_NAME_LEN - 1];
INTEGER FD_LEN, RESULT;

REPLACE DATE_ARY BY "19901003";
REPLACE TMP_ARY BY "!W!,!1N!.!DE!,!YYYY!";
REPLACE LANG_NAME BY "ENGLISH";
RESULT := CNV_FORMATDATETMP(DATE_ARY, TMP_ARY, LANG_NAME,
                            FD_LEN, FD_ARY);
WRITE(OUT,<"RESULT = ",J4>,RESULT);
IF RESULT EQL CS_DATAOKV THEN
    WRITE(OUT,<"FD_ARY = ",C45>,FD_ARY);
CLOSE(OUT,LOCK);
END.
```

Output

```

RESULT = 1
FD_ARY = Wednesday, Oct. 3rd, 1990

```

In this call, the parameters have the following meanings:

DATE_ARY is an EBCDIC array passed to the procedure. It contains the date to be formatted. The date must be in the form YYYYMMDD. The fields of the array have fixed positions. Blanks or zeroes must be used in any fields that are omitted.

TMP_ARY is an EBCDIC array passed to the procedure. It contains the template used to format the date. The template must use the control characters described in the MLS Guide.

LANG_NAME is an EBCDIC array passed to the procedure. It contains the language to be used in formatting the date. Refer to the MLS Guide for information about determining the valid language names on the system on which the user program is running.

FD_LEN is an integer returned by the procedure. It contains the length of the formatted date.

FD_ARY is an EBCDIC array returned by the procedure. It contains the formatted date.

RESULT is an integer returned by the procedure. It indicates whether or not an error occurred during the execution of the procedure. An explanation of the error result values and messages can be found at the end of this section. The values returned by this procedure include:

CS_ARRAY_TOO_SMALLV (3001)	CS_BAD_YEARV (3046)
CS_BAD_ARRAY_DESCRIPTIONV (3000)	CS_BAD_YEAR_YV (3045)
CS_BAD_DATA_LEN (3002)	CS_DATAOKV (1)
CS_BADDATEINPUTV (3012)	CS_FAULTV (1001)
CS_BAD_DAYOFYEARV (3058)	CS_FIELD_TRUNCATEDV (2003)
CS_BAD_DAYV (3048)	CS_LANGUAGE_NOT_FOUNDV (2001)
CS_BAD_MONTHV (3047)	CS_MISSING_DATE_COMPONENTV (3059)
CS_BAD_TEMPCHARV (3011)	CS_MISSING_MONTH_COMPONENTV (3057)
CS_BAD_TEMPL (3030)	CS_SOFTERRV (1002)

CNV_FORMATTIME

This procedure formats a user-supplied time and identifies the kind of template to be retrieved from the named convention and used to format time. The formatted time is returned in the user-specified language.

Example

This example formats the time in numeric form using the Belgium convention. The formatted time is returned in FT_ARY.

```
BEGIN

$ INCLUDE INTL="*SYMBOL/INTL/ALGOL/PROPERTIES." 10000000 - 49999999

FILE OUT(KIND=DISK,UNITS=CHARACTERS,MYUSE=IO,FILETYPE=0,NEWFILE,
        MAXRECSIZE=80,TITLE="OUT/ALGOL/CNVFMTTIME.");
DEFINE MAX_NAME_LEN      = 17#;    % Maximum length of convention
                                % name
EBCDIC ARRAY CNV_NAME[0:MAX_NAME_LEN - 1], FT_ARY[0:7],
             LANG_NAME[0:MAX_NAME_LEN - 1], TIME_ARY[0:9];
INTEGER FT_LEN, RESULT;

REPLACE TIME_ARY BY "114958";
REPLACE LANG_NAME BY "ENGLISH";
REPLACE CNV_NAME BY "BELGIUM";
RESULT := CNV_FORMATTIME(CS_NTIMEV, TIME_ARY, CNV_NAME, LANG_NAME,
                        FT_LEN, FT_ARY);
WRITE(OUT,<"RESULT = ",J4>,RESULT);
IF RESULT EQL CS_DATAOKV THEN
    WRITE(OUT,<"FT_ARY = ",C8>,FT_ARY);
CLOSE(OUT,LOCK);
END.
```

Output

```
RESULT = 1
FT_ARY = 11:49:58
```

In this call, the parameters have the following meanings:

CS_NTIMEV represents an integer that requests the numeric time format be returned. This integer is one of two options:

Value	Value Name	Meaning
3	CS_LTIMEV	Use the long time format.
4	CS_NTIMEV	Use the numeric time format.

TIME_ARY is an EBCDIC array passed to the procedure. It contains the time to be formatted in the form HHMMSS, left justified. The fields of the array have fixed positions. Blanks or zeros must be used in any fields that are omitted.

CNV_NAME is an EBCDIC array passed to the procedure. It contains the name of the convention to be used to edit the time value. If this parameter contains all blanks or zeros, the procedure uses the hierarchy to determine the convention to be used. Refer to the MLS Guide for the list of convention names and the explanation of the hierarchy.

LANG_NAME is an EBCDIC array passed to the procedure. It contains the language to be used in formatting the time. If this parameter contains all blanks or zeros, the procedure uses the hierarchy to determine the language to be used. Refer to the MLS Guide for information about determining the valid language names on the system being used and the explanation of the hierarchy.

FT_LEN is an integer that contains the length of FT_ARY.

FT_ARY is an EBCDIC array returned by the procedure. It contains the time value formatted according to the specified convention and language.

RESULT is an integer returned by the procedure. It indicates whether or not an error occurred during the execution of the procedure. An explanation of the error result values and messages can be found at the end of this section. The values returned by this procedure include:

CS_ARRAY_TOO_SMALLV (3001)	CS_BAD_12HOURV (3052)
CS_BAD_ARRAY_DESCRIPTIONV (3000)	CS_BAD_TYPE_CODEV (3006)
CS_BAD_DATA_LEN (3002)	CS_CONVENTION_NOT_FOUNDV (2002)
CS_BAD_HOURV (3051)	CS_DATAOKV (1)
CS_BAD_MINUTEV (3053)	CS_FAULTV (1001)
CS_BAD_PSECONDV (3055)	CS_FIELD_TRUNCATEDV (2003)
CS_BAD_SECONDV (3054)	CS_LANGUAGE_NOT_FOUNDV (2001)
CS_BAD_TEMPCHARV (3011)	CS_SOFTERRV (1002)
CS_BADTIMEINPUTV (3013)	

CNV_FORMATTIMETMP

This procedure formats a time value, specified in TIME_ARY, according to a template, specified by TMP_ARY. The template can be retrieved for any convention from the CNV_TEMPLATE procedure. It can also be created by the user. The formatted time is returned in the language specified in LANG_NAME.

Example

This example formats a caller-supplied time, using a template also passed in by the calling program. Alphabetic time components are translated into English and returned in FT_ARY.

```
BEGIN

$ INCLUDE INTL="*SYMBOL/INTL/ALGOL/PROPERTIES." 10000000 - 49999999

FILE OUT(KIND=DISK,UNITS=CHARACTERS,MYUSE=IO,FILETYPE=0,NEWFILE,
        MAXRECSIZE=80,TITLE="OUT/ALGOL/CNVFMTTIMETMP.");
DEFINE MAX_NAME_LEN      = 17#;      % Maximum length of convention
                                % name
EBCDIC ARRAY TMP_ARY[0:47], FT_ARY[0:44], TIME_ARY[0:9],
            LANG_NAME[0:MAX_NAME_LEN - 1];
INTEGER FT_LEN, RESULT;

REPLACE TIME_ARY BY "114958";
REPLACE TMP_ARY BY "!T!!!M!!K!!S!!R!";
REPLACE LANG_NAME BY "ENGLISH";
RESULT := CNV_FORMATTIMETMP(TIME_ARY, TMP_ARY, LANG_NAME,
                            FT_LEN, FT_ARY);
WRITE(OUT,<"RESULT = ",J4>,RESULT);
IF RESULT EQL CS_DATAOKV THEN
    WRITE(OUT,<"FT_ARY = ",C45>,FT_ARY);
CLOSE(OUT,LOCK);
END.
```

Output

```

RESULT = 1
FT_ARY = 11 hours 49 minutes 58 seconds

```

In this call, the parameters have the following meanings:

TIME_ARY is an EBCDIC array passed to the procedure. It contains the time to be formatted in the form HHMMSSPPPP. The partial seconds field, PPPP, is optional. The fields of the array have fixed positions. Blanks or zeros must be used in any fields that are omitted.

TMP_ARY is an EBCDIC array passed to the procedure in which the template to be used to format the time is specified. Refer to the MLS Guide for information about creating a template.

LANG_NAME is an EBCDIC array passed to the procedure. It contains the language to be used in formatting the time. If this parameter contains all blanks or zeros, the procedure uses the hierarchy to determine the language to be used. Refer to the MLS Guide for information about determining the valid language names on the system that is used by the program and the explanation of the hierarchy.

FT_LEN is an integer returned by the procedure. It contains the length of the formatted time.

FT_ARY is an EBCDIC array returned by the procedure. It contains the time value formatted according to the specified template and language.

RESULT is an integer returned by the procedure. This integer indicates whether an error occurred during the execution of the procedure. An explanation of the error result values and messages can be found at the end of this section. The values returned by this procedure include:

CS_ARRAY_TOO_SMALLV (3001)	CS_BAD_TEMPLENV (3030)
CS_BAD_ARRAY_DESCRIPTIONV (3000)	CS_BADTIMEINPUTV (3013)
CS_BAD_DATA_LEN (3002)	CS_BAD_12HOURV (3052)
CS_BAD_HOURV (3051)	CS_DATAOKV (1)
CS_BAD_MINUTEV (3053)	CS_FAULTV (1001)
CS_BAD_PSECONDV (3055)	CS_FIELD_TRUNCATEDV (2003)
CS_BAD_SECONDSV (3054)	CS_LANGUAGE_NOT_FOUNDV (2001)
CS_BAD_TEMPCHARV (3011)	CS_SOFTERRV (1002)

CNV_FORMSIZE

This procedure returns the default lines per page and default characters per line values from the specified convention. Each convention provides values for these items to be used by default with printed output.

Example

This example obtains default lines per page and characters per line from the Denmark convention.

```
BEGIN

$ INCLUDE INTL="*SYMBOL/INTL/ALGOL/PROPERTIES." 10000000 - 49999999

FILE OUT(KIND=DISK,UNITS=CHARACTERS,MYUSE=IO,FILETYPE=0,NEWFILE,
        MAXRECSIZE=80,TITLE="OUT/ALGOL/CNVFORMSIZE.");
DEFINE MAX_NAME_LEN      = 17#;      % Maximum length of convention
                                % name
EBCDIC ARRAY CNV_NAME[0:MAX_NAME_LEN - 1];
INTEGER LPP, CPL, RESULT;

REPLACE CNV_NAME BY "DENMARK";
RESULT := CNV_FORMSIZE(CNV_NAME, LPP, CPL);
WRITE(OUT,<"RESULT = ",J4>,RESULT);
IF RESULT EQL CS_DATAOKV THEN
    BEGIN
        WRITE(OUT,<"Lines per Page      = ",J2>,LPP);
        WRITE(OUT,<"Characters per Line = ",J2>,CPL);
    END;
CLOSE(OUT,LOCK);
END.
```

Output

```
RESULT = 1  
Lines per Page      = 70  
Characters per Line = 82
```

In this call, the parameters have the following meanings:

CNV_NAME is an EBCDIC array passed to the procedure. It contains the name of the convention to be used to specify the printer form default sizes. If this parameter contains all blanks or zeros, the procedure uses the hierarchy to determine the convention to be used. Refer to the MLS Guide for the list of convention names and the explanation of the hierarchy.

LPP is an integer returned by the procedure. It contains the number of lines allowed per page for the specified convention.

CPL is an integer returned by the procedure. It contains the number of characters allowed per line for the specified convention.

RESULT is an integer returned by the procedure. It indicates whether or not an error occurred during the execution of the procedure. An explanation of the error result values and messages can be found at the end of this section. The values returned by this procedure include:

CS_BAD_ARRAY_DESCRIPTIONV (3000)	CS_FAULTV (1001)
CS_CONVENTION_NOT_FOUNDV (2002)	CS_SOFTERRV (1002)
CS_DATAOKV (1)	

CNV_INFO

This procedure returns the definition of the specified convention. The definition can be retrieved from memory or from the *SYSTEM/CONVENTIONS file.

Example

This example retrieves the UnitedKingdom1 convention definition from the *SYSTEM/CONVENTIONS file and returns it in INFO_ARY.

```

BEGIN

$ INCLUDE INTL="*SYMBOL/INTL/ALGOL/PROPERTIES." 10000000 - 49999999

FILE OUT(KIND=DISK,UNITS=CHARACTERS,MYUSE=IO,FILETYPE=0,NEWFILE,
        MAXRECSIZE=80,TITLE="OUT/ALGOL/CNVINFO.");
DEFINE MAX_NAME_LEN      = 17#;      % Maximum length of convention
                                % name
ARRAY LINEOUT[0:14], INFO_ARY[0:62], ALTCURRENCY[0:2],
        LONGDATETMP[0:9];
EBCDIC ARRAY CNV_NAME[0:MAX_NAME_LEN - 1];
INTEGER INFO_LEN, RESULT;

REPLACE CNV_NAME BY "UNITEDKINGDOM1";
RESULT := CNV_INFO(CS_CNVFILE_INFOV, CNV_NAME, INFO_LEN, INFO_ARY);
WRITE(OUT,<"RESULT = ",J4>,RESULT);
IF RESULT EQL CS_DATAOKV THEN
    BEGIN
    REPLACE LINEOUT BY " " FOR 80;
    WRITE(OUT,80,LINEOUT);
    WRITE(OUT,<"Field Meaning                                Location  Value">);
    WRITE(OUT,<"-----"----->);
    WRITE(OUT,<"Max. Integer Digits                                [0]        ",J3>,
          INFO_ARY[0]);
    WRITE(OUT,<"Max. Fractional Digits                               [1]        ",J3>,
          INFO_ARY[1]);
    WRITE(OUT,<"Max. International Fractional Digits                 [2]        ",J3>,
          INFO_ARY[2]);
    WRITE(OUT,<"International Currency Notation                       [3:4]      ",C12>,
          POINTER(INFO_ARY[3]));
    WRITE(OUT,<"Long Date Template                                     [5:12]     ",C24>,
          POINTER(INFO_ARY[5]));
    WRITE(OUT,<"Short Date Template                                    [13:20]    ",C24>,
          POINTER(INFO_ARY[13]));
    WRITE(OUT,<"Numeric Date Template                                  [21:24]    ",C24>,
          POINTER(INFO_ARY[21]));
    WRITE(OUT,<"Long Time Template                                     [25:32]    ",C24>,
          POINTER(INFO_ARY[25]));
    WRITE(OUT,<"Numeric Time Template                                  [33:36]    ",C24>,
          POINTER(INFO_ARY[33]));
    WRITE(OUT,<"Monetary Template                                     [37:44]    ",C24>,
          POINTER(INFO_ARY[37]));
    
```

```

WRITE(OUT,<"Numeric Template           [45:52]  ",C24>,
      POINTER(INFO_ARY[45]));
WRITE(OUT,<"Reserved                   [53:60]  ",C24>,
      POINTER(INFO_ARY[53]));
WRITE(OUT,<"Lines per Page              [61]     ",J3>,
      INFO_ARY[61]);
WRITE(OUT,<"Characters per Line         [62]     ",J3>,
      INFO_ARY[62]);
END;
CLOSE(OUT,LOCK);
END.

```

Output

RESULT = 1

Field Meaning	Location	Value
Max. Integer Digits	[0]	12
Max. Fractional Digits	[1]	2
Max. International Fractional Digits	[2]	2
International Currency Notation	[3:4]	"\$ "
Long Date Template	[5:12]	!W!,!DE!!N!,!YYYY!
Short Date Template	[13:20]	!D!!1N!!0Y!
Numeric Date Template	[21:24]	!D!.!00!.!0Y!
Long Time Template	[25:32]	!H!:!0M!:!0S!.!PP!!A!
Numeric Time Template	[33:36]	!H!:!0M!
Monetary Template	[37:44]	!C[\$]T[, :0,3]D[.]#N[-]!
Numeric Template	[45:52]	!T[, :0,3]D[.]#N[-]!
Reserved	[53:60]	
Lines per Page	[61]	70
Characters per Line	[62]	82

In this call, the parameters have the following meanings:

CS_CNVFILE_INFOV represents an integer that requests that convention information be retrieved from the file in use, *SYSTEM/CONVENTIONS. This integer is one of two options:

Value	Value Name	Type of Information Returned
0	CS_CNVMEM_INFOV	Convention information is retrieved from the memory (current working set).
1	CS_CNVFILE_INFOV	Convention information is retrieved from the file in use (*SYSTEM/CONVENTIONS).

Two copies of convention information reside on the system. One copy is in memory and is stored in the CENTRALSUPPORT library during initialization. The other copy is on disk and is stored in the *SYSTEM/CONVENTIONS file.

Internationalization

CNV_NAME is an EBCDIC array that contains the name of the convention set for which a definition is requested. If this parameter contains all blanks or zeros, the procedure uses the hierarchy to determine the convention to be used. Refer to the MLS Guide for the list of convention names and the explanation of the hierarchy.

INFO_LEN is an integer that contains the number of characters in INFO_ARY.

INFO_ARY is a real array passed to the procedure. It contains the convention definition to be added to conventions in memory and to the *SYSTEM/CONVENTIONS file in use. For the procedure to work, all fields in INFO_ARY must contain data, or an appropriate error result is returned. Data in INFO_ARY is passed in fields as follows:

Field Meaning	Word
Maximum integer digits	0
Maximum decimal digits	1
Maximum international decimal digits	2
International currency notation	3
Long date template	5
Short date template	13
Numeric date template	21
Long time template	25
Numeric time template	33
Monetary template	37
Numeric template	45
Lines per page	61
Characters per line	62

The international currency notation field contains the international currency notation defined for the convention. The international currency notation is surrounded by a pair of matching delimiters that are not part of the symbol. Any blanks inside the delimiters are significant and are treated as any other character. For example, the international currency notation for the ASERIESNATIVE convention is "USD "; the trailing blank is significant and the quotation marks are delimiters.

RESULT is an integer returned by the procedure. It indicates whether or not an error occurred during the execution of the procedure. An explanation of the error result values and messages can be found at the end of this section. The values returned by this procedure include:

CS_ARRAY_TOO_SMALLV (3001)	CS_DATAOKV (1)
CS_BAD_ARRAY_DESCRIPTIONV (3000)	CS_FAULTV (1001)
CS_BAD_DATA_LEN (3002)	CS_INCOMPLETE_DATAV (2004)
CS_CONVENTION_NOT_FOUNDV (2002)	CS_SOFTERRV (1002)

CNV_MODIFY

This procedure modifies an existing convention in the *SYSTEM/CONVENTIONS file. The modified set becomes effective after the next halt/load.

Note that this procedure can be used to modify only conventions that have been created by a user. The conventions provided to you cannot be modified.

Example

This example modifies the short date and monetary templates in a convention named Utopia.

```
BEGIN
$ INCLUDE INTL="*SYMBOL/INTL/ALGOL/PROPERTIES." 10000000 - 49999999

FILE OUT(KIND=DISK,UNITS=CHARACTERS,MYUSE=IO,FILETYPE=0,NEWFILE,
        MAXRECSIZE=80,TITLE="OUT/ALGOL/CNVMODIFY.");
DEFINE MAX_NAME_LEN      = 17#;      % Maximum length of convention
                                % name
DEFINE
    SHORTDATETEMP        = 5#,      % MODMASK bit position
    MONETARYTEMP         = 9#;      % MODMASK bit position
ARRAY MOD_ARY[0:62];
EBCDIC ARRAY CNV_NAME[0:MAX_NAME_LEN - 1];
INTEGER RESULT;
REAL MODMASK;

REPLACE CNV_NAME BY "UTOPIA";
MODMASK := 0 & 1 [SHORTDATETEMP:1] % set bits to show what fields
          & 1 [MONETARYTEMP:1]; % contain data in MOD_ARY
REPLACE POINTER(MOD_ARY[13]) BY "!1W!,!N!!DE!,!YYYY!";
REPLACE POINTER(MOD_ARY[37]) BY "!N[-]C[UTA]T[.:0,3]D[.,]#!";
RESULT := CNV_MODIFY(CNV_NAME, MODMASK, MOD_ARY);
WRITE(OUT,<"RESULT = ",J4>,RESULT);
CLOSE(OUT,LOCK);
END.
```

Output

RESULT = 1

In this call, the parameters have the following meanings:

CNV_NAME is an EBCDIC array that is passed to the procedure. It contains the name of the convention to be modified. If this parameter contains all blanks or zeros, the procedure uses the hierarchy to determine the convention to be used. Refer to the MLS Guide for the list of convention names and the explanation of the hierarchy.

MODMASK is a real number that is passed to the procedure. It provides a mask that indicates the fields to be modified in the specified convention.

Any or all bits in positions 0 through 12 can be set to indicate which fields should be modified. Each bit is associated with a unique field in MOD_ARY. If the bit is equal to 1, then the data in the corresponding field in MOD_ARY is validated and stored in the designated convention. Bit 0 in MODMASK corresponds to the first field in MOD_ARY, bit 1 to field 2, and so on.

MOD_ARY is a real array passed to the procedure. It contains the convention definition to be added to conventions in memory and to the *SYSTEM/CONVENTIONS file in use. For the procedure to work, all fields in MOD_ARY must contain data, or an appropriate error result is returned. Data in MOD_ARY is passed in fields as follows:

Field Meaning	Word
Maximum integer digits	0
Maximum decimal digits	1
Maximum international decimal digits	2
International currency notation	3
Long date template	5
Short date template	13
Numeric date template	21
Long time template	25
Numeric time template	33
Monetary template	37
Numeric template	45
Lines per page	61
Characters per line	62

The international currency notation field contains the international currency notation defined for the convention. The international currency notation is surrounded by a pair of matching delimiters that are not part of the notation. Any blanks inside the delimiters are significant and are treated as any other character. For example, the international currency

notation for the ASERIESNATIVE convention is "USD "; the trailing blank is significant and the quotation marks are delimiters.

RESULT is an integer returned by the procedure. It indicates whether or not an error occurred during the execution of the procedure. An explanation of the error result values and messages can be found at the end of this section. The values returned by this procedure include:

CS_BAD_ALT_FRAC_DIGITSV (3017)	CS_BAD_NTIMETEMPV (3022)
CS_BAD_ARRAY_DESCRIPTIONV (3000)	CS_BAD_NUMTEMPV (3024)
CS_BAD_CPLV (3028)	CS_BAD_SDATETEMPV (3019)
CS_BAD_DATA_LEN (3002)	CS_CNVFILE_NOTPRESENTV (3037)
CS_BAD_FRACDIGITSV (3016)	CS_CNV_NOTAVAILV (3036)
CS_BAD_LDATETEMPV (3018)	CS_CONVENTION_NOT_FOUNDV (2002)
CS_BAD_LPPV (3027)	CS_DATAOKV (1)
CS_BAD_LTIMETEMPV (3021)	CS_DEL_PERMANENT_CNV_ERRV (3040)
CS_BAD_MAXDIGITSV (3015)	CS_FAULTV (1001)
CS_BAD_MONTEMPV (3023)	CS_FILE_ACCESS_ERRORV (1000)
CS_BAD_NDATETEMPV (3020)	CS_SOFTERRV (1002)

CNV_NAMES

This procedure returns a list of convention names and the total number of conventions that are available on the system. This list includes any conventions that are defined.

Example

This example obtains the names of conventions currently available on the system. The result shows an arbitrary list of selected convention names.

```
BEGIN

$ INCLUDE INTL="*SYMBOL/INTL/ALGOL/PROPERTIES." 10000000 - 49999999

FILE OUT(KIND=DISK,UNITS=CHARACTERS,MYUSE=IO,FILETYPE=0,NEWFILE,
        MAXRECSIZE=80,TITLE="OUT/ALGOL/CNVNAMES.");
DEFINE MAX_NUM_NAMES      = 45#,      % Maximum number of names
        MAX_NAME_LEN      = 17#,      % Maximum length of convention
                                         % name
ARRAY LINEOUT[0:14];
EBCDIC ARRAY NAMES_ARY[0:MAX_NUM_NAMES*MAX_NAME_LEN - 1];
INTEGER I, RESULT, TOTAL;

RESULT := CNV_NAMES(TOTAL, NAMES_ARY);
WRITE(OUT,<"RESULT = ",J4>,RESULT);
IF RESULT EQL CS_DATAOKV THEN
    BEGIN
        REPLACE LINEOUT BY " " FOR 80;
        WRITE(OUT,80,LINEOUT);
        WRITE(OUT,<"Convention Names">);
        WRITE(OUT,<"-----">);
        WHILE I <= TOTAL DO
            BEGIN
                WRITE(OUT,<C17>,NAMES_ARY[I*17]);
                I := * + 1;
            END;
        END;
        CLOSE(OUT,LOCK);
    END.
```

Output

```
RESULT = 1
```

```
Convention Names
```

```
-----
```

```
ASERIESNATIVE
```

```
Netherlands
```

```
Denmark
```

```
UnitedKingdom1
```

```
Turkey
```

```
Norway
```

```
Sweden
```

```
Greece
```

```
FranceListing
```

```
FranceBureautique
```

```
EuropeanStandard
```

In this call, the parameters have the following meanings:

TOTAL is an integer returned by the procedure. It contains the total number of conventions that reside on the system being used.

NAMES_ARY is an EBCDIC array returned by the procedure. It contains the convention definition names. Each name uses one element of NAMES_ARY. Each name can be up to 17 characters long and is left justified in the field. Any parts of the array that are not used are filled with blanks.

RESULT is an integer returned by the procedure. It indicates whether or not an error occurred during the execution of the procedure. An explanation of the error result values and messages can be found at the end of this section. The values returned by this procedure include:

```
CS_ARRAY_TOO_SMALLV (3001)
```

```
CS_DATAOKV (1)
```

```
CS_FAULTV (1001)
```

CNV_SYMBOLS

This procedure returns a list of numeric and monetary symbols defined for a specified convention.

SYMLEN_ARY and SYM_ARY are parallel arrays. Each entry in SYMLEN_ARY specifies the length of the symbol (in characters) in the corresponding entry in SYM_ARY. If an entry in SYMLEN_ARY is 0 (zero), it indicates that the symbol is not defined and the corresponding entry in SYM_ARY is filled with blanks. If an entry in SYMLEN_ARY is not 0 (zero), but the corresponding entry in SYM_ARY is all blanks, then the number of blanks specified by the SYMLEN_ARY entry constitutes the symbol.

The monetary and numeric symbols defined in the monetary and numeric templates for a convention are returned in fixed-length fields in SYM_ARY. Each field is 12 bytes long, except where noted.

The following table shows the symbols that are returned in SYM_ARY and the offset of the field in which the symbol is returned for the monetary symbols:

Monetary Symbol	Offset in Bytes
International currency notation	0
National currency symbol	12
Thousands separator	24
Decimal symbol	36
Positive sign	48
Negative sign	60
Left enclosure	72
Right enclosure	84

The following table shows the symbols that are returned and the offset of the field in which the symbol is returned for the numeric symbols:

Numeric Symbol	Offset in Bytes
Thousands separator	96
Decimal symbol	108
Positive sign	120
Negative sign	132
Left enclosure	144
Right enclosure	156
Monetary grouping	168
Numeric grouping	192

The monetary and numeric groupings each occupy two adjacent fields (24 bytes) in SYM_ARRAY. The monetary and numeric groupings, when present, are character strings consisting of unsigned integers separated by commas. The integers specify the number of digits in each group. They appear exactly as they are declared in the monetary and numeric templates, including embedded commas.

The following table shows the offset in bytes of the fields in the record SYMLEN_ARRAY, which contain the symbol lengths for the monetary and numeric symbols.

Offset	Contains Length of
0	International currency notation
1	National currency symbol
2	Monetary thousands separator
3	Monetary decimal symbol
4	Monetary positive symbol
5	Monetary negative symbol
6	Monetary left enclosure symbol
7	Monetary right enclosure symbol
8	Numeric thousands separator

Offset	Contains Length of
9	Numeric decimal symbol
10	Numeric positive symbol
11	Numeric negative symbol
12	Numeric left enclosure symbol
13	Numeric right enclosure symbol
14	Monetary grouping
15	Numeric grouping

Example

This example obtains monetary and numeric symbols, monetary and numeric grouping specifications, and the international currency notation defined for the Norway convention.

```

BEGIN

$ INCLUDE INTL="*SYMBOL/INTL/ALGOL/PROPERTIES." 10000000 - 49999999

FILE OUT(KIND=DISK,UNITS=CHARACTERS,MYUSE=IO,FILETYPE=0,NEWFILE,
        MAXRECSIZE=80,TITLE="OUT/ALGOL/CNVSYMBOLS.");
DEFINE MAX_NAME_LEN      = 17#;      % Maximum length of convention
                                % name

ARRAY LINEOUT[0:14];
EBCDIC ARRAY CNV_NAME[0:MAX_NAME_LEN - 1], SYM_ARY[0:215];
INTEGER ARRAY SYMLEN_ARY[0:16];
INTEGER RESULT, TOTAL;

REPLACE CNV_NAME BY "NORWAY";
RESULT := CNV_SYMBOLS(CNV_NAME, TOTAL, SYMLEN_ARY, SYM_ARY);
WRITE(OUT,<"RESULT = ",J4>,RESULT);
IF RESULT EQL CS_DATAOKV THEN
  BEGIN
  REPLACE LINEOUT BY " " FOR 80;
  WRITE(OUT,80,LINEOUT);
  WRITE(OUT,<"Field Meaning                               Symbols Length  ",
        "Convention Symbols">);
  WRITE(OUT,<"-----                               -----  ",
        "-----">);
  WRITE(OUT,<"International Currency Notation: ",J4,"          ",
        " ",C12>,SYMLEN_ARY[0],SYM_ARY[0]);
  WRITE(OUT,<"National Currency Symbol:      ",J4,"          ",
        " ",C12>,SYMLEN_ARY[1],SYM_ARY[1*12]);

  WRITE(OUT,<"Monetary Thousands Separator:   ",J4,"          ",
        " ",C12>,SYMLEN_ARY[2],SYM_ARY[2*12]);
  WRITE(OUT,<"Monetary Decimal Symbol:         ",J4,"          ",
        " ",C12>,SYMLEN_ARY[3],SYM_ARY[3*12]);
  WRITE(OUT,<"Monetary Positive Symbol:        ",J4,"          ",

```



```

        " ",C12>,SYMLEN_ARY[4],SYM_ARY[4*12]);
WRITE(OUT,<"Monetary Negative Symbol: ",J4," ",
        " ",C12>,SYMLEN_ARY[5],SYM_ARY[5*12]);
WRITE(OUT,<"Monetary Left Enclosure Symbol: ",J4," ",
        " ",C12>,SYMLEN_ARY[6],SYM_ARY[6*12]);
WRITE(OUT,<"Monetary Right Enclosure Symbol: ",J4," ",
        " ",C12>,SYMLEN_ARY[7],SYM_ARY[7*12]);
WRITE(OUT,<"Numeric Thousands Separator: ",J4," ",
        " ",C12>,SYMLEN_ARY[8],SYM_ARY[8*12]);
WRITE(OUT,<"Numeric Decimal Symbol: ",J4," ",
        " ",C12>,SYMLEN_ARY[9],SYM_ARY[9*12]);
WRITE(OUT,<"Numeric Positive Symbol: ",J4," ",
        " ",C12>,SYMLEN_ARY[10],SYM_ARY[10*12]);
WRITE(OUT,<"Numeric Negative Symbol: ",J4," ",
        " ",C12>,SYMLEN_ARY[11],SYM_ARY[11*12]);
WRITE(OUT,<"Numeric Left Enclosure Symbol: ",J4," ",
        " ",C12>,SYMLEN_ARY[12],SYM_ARY[12*12]);
WRITE(OUT,<"Numeric Right Enclosure Symbol: ",J4," ",
        " ",C12>,SYMLEN_ARY[13],SYM_ARY[13*12]);
WRITE(OUT,<"Monetary Grouping Specification: ",J4," ",
        " ",C24>,SYMLEN_ARY[14],SYM_ARY[14*12]);
WRITE(OUT,<"Numeric Grouping Specification: ",J4," ",
        " ",C24>,SYMLEN_ARY[15],SYM_ARY[16*12]);
END;
CLOSE(OUT,LOCK);
END.

```

Output

RESULT = 1

Field Meaning	Symbols Length	Convention Symbols
International Currency Notation:	3	NKR
National Currency Symbol:	3	KR.
Monetary Thousands Separator:	1	
Monetary Decimal Symbol:	1	,
Monetary Positive Symbol:	0	
Monetary Negative Symbol:	1	-
Monetary Left Enclosure Symbol:	0	
Monetary Right Enclosure Symbol:	0	
Numeric Thousands Separator:	1	
Numeric Decimal Symbol:	1	,
Numeric Positive Symbol:	0	
Numeric Negative Symbol:	1	-
Numeric Left Enclosure Symbol:	0	
Numeric Right Enclosure Symbol:	0	
Monetary Grouping Specification:	1	3
Numeric Grouping Specification:	1	3

In this call, the parameters have the following meanings:

CNV_NAME is an EBCDIC array passed to the procedure. It contains the name of the convention to be used to retrieve the monetary and numeric symbols. If this parameter contains all blanks or zeros, the procedure uses the hierarchy to determine the convention to be used. Refer to the MLS Guide for the list of convention names and the explanation of the hierarchy.

TOTAL is an integer returned by the procedure. It contains the total number of symbols returned.

SYMLEN_ARY is an integer returned by the procedure. It contains the lengths of all symbols being returned in SYMBOLS.

SYM_ARY is an EBCDIC array returned by the procedure. Each element of the array contains blanks or a symbol defined in the monetary and numeric template for the specified convention. The corresponding entry in SYMLEN_ARY contains the length of each symbol.

RESULT is an integer returned by the procedure. It indicates whether or not an error occurred during the execution of the procedure. An explanation of the error result values and messages can be found at the end of this section. The values returned by this procedure include:

CS_ARRAY_TOO_SMALLV (3001)	CS_DATAOKV (1)
CS_BAD_ARRAY_DESCRIPTIONV (3000)	CS_FAULTV (1001)
CS_BAD_DATA_LEN (3002)	CS_INCOMPLETE_DATAV (2004)
CS_CONVENTION_NOT_FOUNDV (2002)	CS_SOFTERRV (1002)

CNV_SYSTEMDATETIME

This procedure obtains the current system date and time and formats them according to the specified convention. Date and time components are translated to the natural language designated in LANG_NAME. The system computes both the date and time from the result of a single TIME(6) function call. Thus, the possibility that the date and time are split across midnight does not exist.

Example

This example formats system date and time according to formatting definitions in the ASERIESNATIVE convention. The form of date and time is specified by CS_LDATENTIMEV (long date and numeric time). Formatted date and time are translated to English and returned in SDT_ARY.

```
BEGIN

$ INCLUDE INTL="*SYMBOL/INTL/ALGOL/PROPERTIES." 10000000 - 49999999

FILE OUT(KIND=DISK,UNITS=CHARACTERS,MYUSE=IO,FILETYPE=0,NEWFILE,
        MAXRECSIZE=80,TITLE="OUT/ALGOL/CNVSYSDATETIME.");
DEFINE MAX_NAME_LEN      = 17#;      % Maximum length of convention
                                % name
EBCDIC ARRAY CNV_NAME[0:MAX_NAME_LEN - 1], SDT_ARY[0:59],
            LANG_NAME[0:MAX_NAME_LEN - 1];
INTEGER DATE_LEN, RESULT, SDT_LEN;

REPLACE CNV_NAME BY "ASERIESNATIVE";
REPLACE LANG_NAME BY "ENGLISH";
RESULT := CNV_SYSTEMDATETIME(CS_LDATENTIMEV, CNV_NAME, LANG_NAME,
                            DATE_LEN, SDT_LEN, SDT_ARY);
WRITE(OUT,<"RESULT = ",J4>,RESULT);
IF RESULT EQL CS_DATAOKV THEN
    WRITE(OUT,<"SDT_ARY = ",C60>,SDT_ARY);
CLOSE(OUT,LOCK);
END.
```

Output

```
RESULT = 1
SDT_ARY = Friday, November 9, 1990 10:31:30
```

In this call, the parameters have the following meanings:

CS_LDATENTIMEV represents an integer that requests the long date and time format to be returned. It is one of eleven options:

Value	Value Name	Meaning
0	CS_LDATEV	Use the long date format.
1	CS_SDATEV	Use the short date format.
2	CS_NDATEV	Use the numeric date format.
3	CS_LTIMEV	Use the long time format.
4	CS_NTIMEV	Use the numeric time format.
5	CS_LDATELTIMEV	Use the long date and long time format.
6	CS_LDATENTIMEV	Use the long date and numeric time format.
7	CS_SDATELTIMEV	Use the short date and long time format.
8	CS_SDATENTIMEV	Use the short date and numeric time format.
9	CS_NDATELTIMEV	Use the numeric date and long time format.
10	CS_NDATENTIMEV	Use the numeric date and numeric time format.

CNV_NAME is an EBCDIC array passed to the procedure. It contains the name of the convention to be used to edit the date and time value. If this parameter contains all blanks or zeros, the procedure uses the hierarchy to determine the convention to be used. Refer to the MLS Guide for the list of convention names and the explanation of the hierarchy.

LANG_NAME is an EBCDIC array passed to the procedure. It contains the language to be used in formatting the date and time value. If this parameter contains all blanks or zeros, the procedure uses the hierarchy to determine the language to be used. Refer to the MLS Guide for information about determining the valid language names on the system being used and the explanation of the hierarchy.

DATE_LEN is an integer returned by the procedure. It specifies the length of the formatted date portion of the date and/or time. If this parameter is 0 (zero), there is no formatted date in the output. If both date and time are presented in the output, the formatted date is separated from the formatted time by a blank. The extra character is reflected in the length of the formatted date.

SDT_LEN is an integer returned by the procedure that specifies the total length of the formatted date and/or time.

SDT_ARY is an EBCDIC array returned by the procedure that contains the formatted date and/or time. The recommended size of this array is 96 characters.

RESULT is an integer returned by the procedure. It indicates whether or not an error occurred during the execution of the procedure. An explanation of the error result values and messages can be found at the end of this section. The values returned by this procedure include:

CS_ARRAY_TOO_SMALLV (3001)	CS_BAD_YEARV (3046)
CS_BAD_ARRAY_DESCRIPTIONV (3000)	CS_BAD_YEAR_YV (3045)
CS_BAD_DAYOFYEARV (3058)	CS_CONVENTION_NOT_FOUNDV (2002)
CS_BAD_DAYV (3048)	CS_DATAOKV (1)
CS_BAD_HOURV (3051)	CS_FAULTV (1001)
CS_BAD_MINUTEV (3053)	CS_FIELD_TRUNCATEDV (2003)
CS_BAD_MONTHV (3047)	CS_INCOMPLETE_DATAV (2004)
CS_BAD_PSECONDV (3055)	CS_LANGUAGE_NOT_FOUNDV (2001)
CS_BAD_SECONDSV (3054)	CS_MISSING_DATE_COMPONENTV (3059)
CS_BAD_12HOURV (3052)	CS_MISSING_MONTH_COMPONENTV (3057)
CS_BAD_TYPE_CODEV (3006)	CS_SOFTERRV (1002)

CNV_SYSTEMDATETIMETMP

This procedure obtains the system date and time and formats them according to a template and language supplied by the program. The template can be retrieved for any convention from the CNV_TEMPLATE procedure. It can also be created by the user. The system obtains the date and time from a single TIME(6) function call to avoid the possibility of splitting the date and time across a day boundary.

Example

This example formats system date and time according to a template provided by the calling program in TEMP_ARY. The formatted date and time are translated to English and returned in SDT_ARY. DTEMP_LEN is set to the length of date template in TEMP_ARY.

```
BEGIN

$ INCLUDE INTL="*SYMBOL/INTL/ALGOL/PROPERTIES." 10000000 - 49999999

FILE OUT(KIND=DISK,UNITS=CHARACTERS,MYUSE=IO,FILETYPE=0,NEWFILE,
        MAXRECSIZE=80,TITLE="OUT/ALGOL/CNVSYSDATETIMETMP.");
DEFINE MAX_NAME_LEN      = 17#;      % Maximum length of convention
                                % name
EBCDIC ARRAY TEMP_ARY[0:47], SDT_ARY[0:59],
            LANG_NAME[0:MAX_NAME_LEN - 1];
INTEGER DATE_LEN, DTEMP_LEN, RESULT, SDT_LEN;
POINTER PTR;

REPLACE PTR:TEMP_ARY BY "!W!,!N!!D!,!YYYY! "; % Long date template
DTEMP_LEN := OFFSET(PTR);
REPLACE PTR BY "!OT!:!OM!:!OS!"; % Numeric time template
REPLACE LANG_NAME BY "ENGLISH";
RESULT := CNV_SYSTEMDATETIMETMP(TEMP_ARY, LANG_NAME, DTEMP_LEN,
                                DATE_LEN, SDT_LEN, SDT_ARY);
WRITE(OUT,<"RESULT = ",J4>,RESULT);
IF RESULT EQL CS_DATAOKV THEN
    WRITE(OUT,<"SDT_ARY = ",C60>,SDT_ARY);
CLOSE(OUT,LOCK);
END.
```

Output

```
RESULT = 1
SDT_ARY = Friday, November 9, 1990 10:31:44
```

In this call, the parameters have the following meanings:

TEMP_ARY is an EBCDIC array passed to the procedure. It contains the template, left justified. If both date and time templates are present, the date template must appear first.

LANG_NAME is an EBCDIC array passed to the procedure. It contains the language to be used in formatting the date and time value. If this parameter contains all blanks or zeros, the procedure uses the hierarchy to determine the language to be used. Refer to the MLS Guide for information about determining the valid language names on the system being used and the explanation of the hierarchy.

DTEMP_LEN is an integer passed to the procedure. It specifies the length of the date template in TEMP_ARY. If DTEMP_LEN is 0 (zero), there is no date template in TEMP_ARY. If both a date and time template are specified, then the date template must appear first in TEMP_ARY. The date and time are formatted if both date and time templates are present. The date is formatted if only the date template is present, and the time is formatted if only the time template is present.

DATE_LEN is an integer returned by the procedure that contains the length of the formatted date portion of date and time. This parameter is 0 (zero) if there is no formatted date in the output. If both the date and time are presented in the output, the formatted date is separated from the formatted time by a blank. The extra character is reflected in the length of the formatted date.

SDT_LEN is an integer returned by the procedure that contains the length of the formatted date and time.

SDT_ARY is an EBCDIC array returned by the procedure. It contains the formatted date and/or time.

Internationalization

RESULT is an integer returned by the procedure. It indicates whether or not an error occurred during the execution of the procedure. An explanation of the error result values and messages can be found at the end of this section. The values returned by this procedure include:

CS_ARRAY_TOO_SMALLV (3001)	CS_BAD_12HOURV (3052)
CS_BAD_ARRAY_DESCRIPTIONV (3000)	CS_BAD_YEARV (3046)
CS_BAD_DATA_LEN (3002)	CS_BAD_YEAR_YV (3045)
CS_BAD_DAYOFYEARV (3058)	CS_DATAOKV (1)
CS_BAD_DAYV (3048)	CS_FAULTV (1001)
CS_BAD_HOURV (3051)	CS_FIELD_TRUNCATEDV (2003)
CS_BAD_MINUTEV (3053)	CS_LANGUAGE_NOT_FOUNDV (2001)
CS_BAD_MONTHV (3047)	CS_MISSING_DATE_COMPONENTV (3059)
CS_BAD_PSECONDV (3055)	CS_MISSING_MONTH_COMPONENTV (3057)
CS_BAD_SECONDSV (3054)	CS_SOFTERRV (1002)
CS_BAD_TEMPCHARV (3011)	

CNV_TEMPLATE

This procedure returns the requested type of formatting template retrieved from the convention that is specified in CNV_NAME.

This procedure might be used to improve the performance of programs. By retrieving and storing a template to be used in many places, the performance of a program can be improved by eliminating the calls to the CENTRALSUPPORT library.

Example

The following retrieves a monetary editing template from the convention Turkey. The template is returned in TMP_ARY.

```
BEGIN

$ INCLUDE INTL="*SYMBOL/INTL/ALGOL/PROPERTIES." 10000000 - 49999999

FILE OUT(KIND=DISK,UNITS=CHARACTERS,MYUSE=IO,FILETYPE=0,NEWFILE,
        MAXRECSIZE=80,TITLE="OUT/ALGOL/CNVTEMPLATE.");
DEFINE MAX_NAME_LEN      = 17#;      % Maximum length of convention
                                % name
EBCDIC ARRAY CNV_NAME[0:MAX_NAME_LEN - 1], TMP_ARY[0:47];
INTEGER TMP_LEN, RESULT;

REPLACE CNV_NAME BY "TURKEY";
RESULT := CNV_TEMPLATE(CS_MONETARY_TEMPV, CNV_NAME, TMP_LEN, TMP_ARY);
WRITE(OUT,<"RESULT = ",J4>,RESULT);
IF RESULT EQL CS_DATAOKV THEN
    WRITE(OUT,<"TMP_ARY = ",C48>,TMP_ARY);
CLOSE(OUT,LOCK);
END.
```

Output

```
RESULT = 1
TMP_ARY = !T[.:0,3]D[,]#N[-]C[T]!
```

In this call, the parameters have the following meanings:

CS_MONETARY_TEMPV represents an integer that requests the type of template to be returned. It is one of seven options:

Value	Value Name	Template to be Retrieved
0	CS_LONGDATE_TEMPV	Long date
1	CS_SHORTDATE_TEMPV	Short date
2	CS_NUMDATE_TEMPV	Numeric date
3	CS_LONGTIME_TEMPV	Long time
4	CS_NUMTIME_TEMPV	Numeric time
5	CS_MONETARY_TEMPV	Monetary
6	CS_NUMERIC_TEMPV	Numeric

CNV_NAME is an EBCDIC array passed to the procedure that contains the name of the convention that is specified. If this parameter contains all blanks or zeros, the procedure uses the hierarchy to determine the convention to be used. Refer to the MLS Guide for the list of convention names and the explanation of the hierarchy.

TMP_LEN is an integer returned from the procedure that contains the number of characters in TMP_ARY.

TMP_ARY is an EBCDIC array returned by the procedure that contains the requested template.

RESULT is an integer returned by the procedure. It indicates whether or not an error occurred during the execution of the procedure. An explanation of the error result values and messages can be found at the end of this section. The values returned by this procedure include:

CS_ARRAY_TOO_SMALLV (3001)	CS_CONVENTION_NOT_FOUNDV (2002)
CS_BAD_ARRAY_DESCRIPTIONV (3000)	CS_DATAOKV (1)
CS_BAD_DATA_LEN (3002)	CS_FAULTV (1001)
CS_BAD_TYPE_CODEV (3006)	CS_SOFTERRV (1002)

CNV_VALIDATENAME

This procedure returns a value in RESULT that indicates whether or not the convention name specified in CNV_NAME is currently defined on the system.

This procedure might be used to ensure that a convention used as an input parameter exists on the system on which the program is running.

Example

This example determines whether or not a convention named Sweden is currently available to CENTRALSUPPORT.

```
BEGIN

$ INCLUDE INTL="*SYMBOL/INTL/ALGOL/PROPERTIES." 10000000 - 49999999

FILE OUT(KIND=DISK,UNITS=CHARACTERS,MYUSE=IO,FILETYPE=0,NEWFILE,
        MAXRECSIZE=80,TITLE="OUT/ALGOL/VALIDATENAME.");
DEFINE MAX_NAME_LEN      = 17#;    % Maximum length of convention
                                % name
EBCDIC ARRAY CNV_NAME[0:MAX_NAME_LEN - 1];
INTEGER RESULT;

REPLACE CNV_NAME BY "SWEDEN";
RESULT := CNV_VALIDATENAME(CNV_NAME);
WRITE(OUT,<"RESULT = ",J4>,RESULT);
CLOSE(OUT,LOCK);
END.
```

Output

```
RESULT = 1
```

In this call, the parameters have the following meanings:

CNV_NAME is an EBCDIC array passed to the procedure. It contains the name of the convention to be checked. If this parameter contains all blanks or nulls, the RESULT parameter returns a value of 0 (zero) or FALSE. Refer to the MLS Guide for the list of convention names and the explanation of the hierarchy.

RESULT is an integer returned by the procedure. It indicates whether or not an error occurred during the execution of the procedure. An explanation of the error result values and messages can be found at the end of this section. The values returned by this procedure include:

Value	Condition-Name	Meaning
0	CS_DATAOKV	The convention name is valid.
1	CS_FALSEV	The convention name is not valid.

COMPARE_TEXT_USING_ORDER_INFO

This procedure compares two strings using the ordering information returned by VSNORDERING_INFO. A starting position can be given for each piece of text, along with the compare relationship to be checked. One of the following methods of comparison can be chosen:

- Equivalent comparison, which is based on the ordering sequence values of characters
- Logical comparison, which is based on the ordering sequence values and the priority sequence values of characters.

Example

This example compares two strings using the CanadaEBCDIC ccsversion. The compare relation is CS_CMPEQLV(=) to determine if one string is equal to another, using a logical comparison. The ORDER_ARY is obtained by calling the VSNORDERING_INFO procedure. This call obtains the ORDER_ARY for the CanadaEBCDIC ccsversion. The MLS Guide can be used to determine that the ccsversion number for CanadaEBCDIC is 74. This number can also be retrieved by calling VALIDATE_NAME_RETURN_NUM with the name CanadaEBCDIC.

```
BEGIN

$ INCLUDE INTL="SYMBOL/INTL/ALGOL/PROPERTIES." 10000000 - 49999999

FILE OUT(KIND=DISK,UNITS=CHARACTERS,FILEUSE=IO,FILETYPE=0,NEWFILE,
        MAXRECSIZE=80,TITLE="OUT/ALGOL/COMP_TEXT_ORD_INF.");
EBCDIC ARRAY TEXT1[0:4]
            ,TEXT2[0:4];
REAL ARRAY ORDER_ARY[0:255];
POINTER P_TEXT1
        ,P_TEXT2;
INTEGER COMPARE_LEN
        ,VSN_NUM
        ,RESULT;

VSN_NUM:= 74;                                % Canada EBCDIC
RESULT := VSNORDERING_INFO(VSN_NUM, ORDER_ARY);
IF RESULT EQL CS_DATAOKV THEN
    BEGIN
        COMPARE_LEN := MIN(SIZE(TEXT1), SIZE(TEXT2));
        P_TEXT1 := TEXT1[0];
        P_TEXT2 := TEXT2[0];
        REPLACE TEXT1 BY "hotel";
        REPLACE TEXT2 BY "hôte1";

        RESULT := COMPARE_TEXT_USING_ORDER_INFO(P_TEXT1, P_TEXT2,
            COMPARE_LEN, CS_CMPEQLV, CS_LOGICALV, ORDER_ARY);
        WRITE(OUT,<"RESULT = ",J4>,RESULT);
    END;
```

```
CLOSE(OUT, LOCK);
END.
```

Output

```
RESULT = 0                                % The strings are not equal
```

In this call, the parameters have the following meanings:

P_TEXT1 is a POINTER array passed to the procedure. It contains the first string to be compared.

P_TEXT2 is a POINTER array passed to the procedure. It contains the other string to be compared.

COMPARE_LEN is an integer passed to the procedure. It designates the number of characters to be compared. If COMPARE_LEN is larger than the number of characters in the strings, then the procedure might be comparing invalid data. The value of COMPARE_LEN should not exceed the bounds of either the PTEXT1 array or the PTEXT2 array. The strings should be of equal size or padded with blanks up to COMPARE_LEN. If all pairs of characters compare equally through the last pair, the strings are considered equal. The first pair of unequal characters to be encountered is compared to determine their relative ordering. The string that contains the character with the higher ordering (higher OSV and PSV) is considered to be the greater string. If, because of substitution, the strings become of unequal size, the comparison proceeds as if the shorter string had been expanded by blanks on the right to make it equal in size to the longer string.

CS_CMPEQLV represents an integer that compares TEXT1 to TEXT2. This is one of 6 compare relation options:

Parameter Value	Value Name	What the Comparison Is to Determine
0	CS_CMPLESSV	TEXT1 is less than TEXT2.
1	CS_CMPLEQV	TEXT1 is less than or equal to TEXT2.
2	CS_CMPEQLV	TEXT1 is equal to TEXT2.
3	CS_CMPGTRV	TEXT1 is greater than TEXT2.
4	CS_CMPGEQV	TEXT1 is greater than or equal to TEXT2.
5	CS_CMPNEQV	TEXT1 is not equal to TEXT2.

CS_LOGICALV represents an integer that specifies a logical comparison. It is one of two options:

Value	Value Name	Description
1	CS_EQUIVALENTV	Perform an equivalent comparison.
2	CS_LOGICALV	Perform a logical comparison.

ORDER_ARY is a REAL array passed to the procedure. It contains the ordering information to be used to compare the strings. Ordering information can be obtained using the VSNORDERING_INFO procedure. It is recommended that the size of this array be 256 words. The layout of ORDER_ARY can be found in the VSNORDERING_INFO procedure later in this section.

RESULT is an integer returned by the procedure. It indicates whether or not an error occurred during the execution of the procedure. An explanation of the error result values and messages can be found at the end of this section. The values returned by this procedure include:

CS_BAD_ARRAY_DESCRIPTIONV (3000)	CS_FALSEV (0)
CS_BAD_DATA_LEN (3002)	CS_FAULTV (1001)
CS_BAD_TYPE_CODEV (3006)	CS_SOFTERRV (1002)
CS_DATAOKV (1)	

GET_CS_MSG

This procedure returns error message text specified by the error number in NUM and in the language designated in LANG_NAME. The desired length of the return message can be specified in the MSG_LEN parameter. If the returned text is shorter than the length specified, the procedure pads the remaining portion of the record with blanks.

An entire message consists of the following three parts:

- The header, which comprises the first 80 characters of the message returned in the MSG parameter. The text in the header provides the error number and a concise text description.
- The short description, which comprises the second 80 characters of the message returned in the MSG parameter. If space is a consideration, the description of the error can be limited to the header and short description.
- The long description, which comprises the remaining characters of the message returned in the MSG parameter. The long description provides a complete explanation of the error that was returned.

Part or all of the message text can be returned. Note that the header part starts at MSG[0], the short description at MSG[80], and the long description at MSG[160]. For example, if MSG_LEN is specified to be equal to 200 characters, then MSG would contain the header message padded with blanks to 80, if necessary, followed by the short description padded with blanks to 160, if necessary, followed by 40 characters of the long description.

The requested message length should be at least 80 characters, equal to one line of text. Anything less results in an incomplete message. It is recommended that you use either a length of 80, 160, or 999. The value of 999 causes the entire message to be returned.

This procedure might be used to retrieve the text of an error message so that it can be displayed by a program.

Example

This example illustrates how to get the message text associated with a CENTRALSUPPORT error message. Assume that the sample call to VALIDATE_NAME_RETURN_NUM returns error number 3004 (CS_NO_NUM_FOUNDV). When an error is returned this example gets the first 160 characters (2 lines) of the message text for that error.

```

BEGIN

$ INCLUDE INTL="SYMBOL/INTL/ALGOL/PROPERTIES." 10000000 - 49999999

FILE OUT(KIND=DISK,UNITS=CHARACTERS,FILEUSE=IO,FILETYPE=0,NEWFILE,
        MAXRECSIZE=80,TITLE="OUT/ALGOL/GET_CS_MSG.");

EBCDIC ARRAY CCSVSNAME[0:17]
            ,LANG[0:17]
            ,MSG[0:159];
INTEGER MSG_LEN
        ,NUM
        ,RESULT1
        ,RESULT2;

REPLACE CCSVSNAME[0] BY "BADNAME" FOR 7;
RESULT1 := VALIDATE_NAME_RETURN_NUM(CS_CHARACTER_SETV,CCSVSNAME, NUM);
WRITE(OUT,<"RESULT FROM VALIDATE_NAME_RETURN_NUM = ",J4>,RESULT1);
IF RESULT1 NEQ CS_DATAOKV THEN
    BEGIN
        MSG_LEN := 160;
        RESULT2 := GET_CS_MSG(RESULT1, LANG, MSG, MSG_LEN);
        WRITE(OUT,<"RESULT FROM GETCSMSG = ",J4>,RESULT2);
        WRITE(OUT,<"MSG = ">);
        WRITE(OUT,<A80>,MSG[0]);
        WRITE(OUT,<A80>,MSG[80]);
        END;
    CLOSE(OUT,LOCK);
END.

```

Output

```
RESULT FROM VALIDATE_NAME_RETURN_NUM = 3004
RESULT FROM GETCSMSG = 1
MSG =
>>> CENTRALSUPPORT INTERFACE ERROR (#3004) <<<
INVALID CHARACTER SET OR CCSVERSION NAME
```

In this call, the parameters have the following meanings:

RESULT1 is an integer passed by reference to the procedure. It specifies an error that was returned by a library procedure. The error numbers and their meanings are listed at the end of this section.

LANG is an EBCDIC array passed to the procedure. It specifies the language to be used for the message text. If this parameter contains all blanks or zeros, the procedure uses the default language hierarchy to determine the language to be used. Refer to the MLS Guide for details about determining the valid language names on the system and for the explanation of the hierarchy.

MSG is an EBCDIC array returned by the procedure. It contains the message text associated with the error number specified.

MSG_LEN is an integer passed to the procedure. For input, it specifies the maximum length of the message to be returned. It also contains an update value as an output parameter. If MSG_LEN is equal to 0 (zero), one line of text (80 characters) is returned. If MSG_LEN is between 1 and 79, then only a partial message is returned. MSG_LEN should not be greater than the size of the MSG array. Recommended values for MSG_LEN are 80, 160, or a large number, such as 999, that returns all of the message. For output, MSG_LEN specifies the actual length of the message returned by the procedure.

RESULT2 is an integer returned by the procedure. It indicates whether or not an error occurred during the execution of the procedure. An explanation of the error result values and messages can be found at the end of this section. The values returned by this procedure include:

CS_ARRAY_TOO_SMALLV (3001)	CS_FAULTV (1001)
CS_BAD_ARRAY_DESCRIPTIONV (3000)	CS_INCOMPLETE_DATAV (2004)
CS_BAD_DATA_LEN (3002)	CS_NO_NUM_FOUNDV (3003)
CS_DATAOKV (1)	CS_SOFTERRV (1002)

MCP_BOUND_LANGUAGES

This procedure returns the names of the languages that are currently bound to the MCP. For information about binding a language to the operating system, see the MLS Guide.

Example

This example returns the languages bound by the operating system. Assume for this example that the bound languages are English and German.

```
BEGIN

$ INCLUDE INTL="SYMBOL/INTL/ALGOL/PROPERTIES." 10000000 - 49999999

FILE OUT(KIND=DISK,UNITS=CHARACTERS,FILEUSE=IO,FILETYPE=0,NEWFILE,
        MAXRECSIZE=80,TITLE="OUT/ALGOL/MCP_BOUND_LANG.");
EBCDIC ARRAY LANGUAGES_ARY[0:84];
ARRAY LINEOUT[0:14];
INTEGER TOTAL
        ,I
        ,RESULT;

RESULT := MCP_BOUND_LANGUAGES(TOTAL, LANGUAGES_ARY);
WRITE(OUT,<"RESULT = ",J4>,RESULT);
IF RESULT EQL CS_DATAOKV THEN
    BEGIN
    REPLACE LINEOUT BY " " FOR 80;
    WRITE(OUT,80,LINEOUT);
    WRITE(OUT,<"LANGUAGES">);
    WRITE(OUT,<"-----">);
    FOR I := 0 STEP 1 UNTIL TOTAL-1 DO
        WRITE(OUT,17,LANGUAGES_ARY[I*17]);
    END;
CLOSE(OUT,LOCK);
END.
```

Output

```
RESULT = 1
```

```
LANGUAGES
```

```
-----
```

```
ENGLISH
```

```
GERMAN
```

In this call, the parameters have the following meanings:

TOTAL is an integer returned by the procedure. It contains the total number of languages bound to the MCP.

LANGUAGES_ARY is an EBCDIC array returned by the procedure. It contains the names of the languages bound to the MCP. The maximum length of each name is 17 characters, and the names are left justified. For any name that is less than 17 characters, the field is filled on the right with blanks. It is recommended that the size of the array be 84 characters; an array of that size holds 5 names.

RESULT is an integer returned by the procedure. It indicates whether or not an error occurred during the execution of the procedure. An explanation of the error result values and messages can be found at the end of this section. The values returned by this procedure include:

CS_ARRAY_TOO_SMALLV (3001)

CS_FAULTV (1001)

CS_BAD_ARRAY_DESCRIPTIONV (3000)

CS_SOFTERRV (1002)

CS_DATAOKV (1)

VALIDATE_NAME_RETURN_NUM

This procedure checks a coded character set or ccsversion name to determine if it resides in the SYSTEM/CCSFILE file. A coded character set or ccsversion in CCSVSN_TYPE is designated to be checked, and a name is supplied in NAME_ARY. The procedure returns the number of the given character set or ccsversion in NUM.

This procedure might be used to obtain the ccsversion number needed as a parameter in other CENTRALSUPPORT library procedures.

Example

This example checks to see if a ccsversion named CanadaGP is valid. Assume for this example that CanadaGP is valid and its associated number is 75.

```
BEGIN

$ INCLUDE INTL="SYMBOL/INTL/ALGOL/PROPERTIES." 10000000 - 49999999

FILE OUT(KIND=DISK,UNITS=CHARACTERS,FILEUSE=IO,FILETYPE=0,NEWFILE,
        MAXRECSIZE=80,TITLE="OUT/ALGOL/VAL_NAME_RET_NUM.");
EBCDIC ARRAY NAME_ARY[0:17];
INTEGER NUM
        ,RESULT;

REPLACE NAME_ARY[0] BY "CANADAGP";
RESULT := VALIDATE_NAME_RETURN_NUM(CS_CCVERSIONV, NAME_ARY, NUM);
WRITE(OUT,<"RESULT = ",J4>,RESULT);
IF RESULT EQL CS_DATAOKV THEN
    WRITE(OUT,<"NUM    = ",J4>,NUM);
CLOSE(OUT,LOCK);
END.
```

Output

```
RESULT = 1
NUM    = 75
```

In this call, the parameters have the following meanings:

CS_CCVERSIONV represents an integer that requests a ccsversion number be returned. It is one of two options:

Value	Value Name	Meaning
0	CS_CHARACTER_SETV	Return a coded character set number.
1	CS_CCVERSIONV	Return a ccsversion number.

NAME_ARY is an EBCDIC array passed to the procedure. It contains the coded character set or ccsversion name for which a number is being requested. The name can be up to 17 characters long. If this parameter contains zeros or blanks and the first parameter is CS_CCVERSIONV, the procedure validates the system defined ccsversion. If there is no system default, the procedure returns an error in RESULT.

NUM is an integer returned by the procedure. It contains the specified coded character set or ccsversion number.

RESULT is an integer returned by the procedure. It indicates whether or not an error occurred during the execution of the procedure. An explanation of the error result values and messages can be found at the end of this section. The values returned by this procedure include:

CS_BAD_ARRAY_DESCRIPTIONV (3000)	CS_FAULTV (1001)
CS_BAD_DATA_LEN (3002)	CS_NO_NAME_FOUNDV (3004)
CS_BAD_TYPE_CODEV (3006)	CS_SOFTERRV (1002)
CS_DATAOKV (1)	

VALIDATE_NUM_RETURN_NAME

This procedure checks the number of the coded character set or ccsversion to determine if it resides in the SYSTEM/CCSFILE file. A coded character set or ccsversion is designated to be checked and a number in NUM is supplied. The procedure returns the name of the given coded character set or ccsversion number. Refer to the MLS Guide for the list of numbers for coded character sets and ccsversions.

This procedure might be used to display to the application user the name of the coded character set or the ccsversion being used.

Example

This example checks to see if a ccsversion number 75 is valid. Assume for this example that 75 is valid and its associated name is CanadaGP.

```
BEGIN

  $ INCLUDE INTL="SYMBOL/INTL/ALGOL/PROPERTIES." 10000000 - 49999999

  FILE OUT(KIND=DISK,UNITS=CHARACTERS,FILEUSE=IO,FILETYPE=0,NEWFILE,
           MAXRECSIZE=80,TITLE="OUT/ALGOL/VAL_NUM_RET_NAME.");
  EBCDIC ARRAY NAME_ARY[0:17];
  INTEGER NUM
           ,RESULT;

  NUM := 75;
  RESULT := VALIDATE_NUM_RETURN_NAME(CS_CCVERSIONV, NUM, NAME_ARY);
  WRITE(OUT,<"RESULT = ",J4>,RESULT);
  IF RESULT EQL CS_DATAOKV THEN
    WRITE(OUT,<"NAME   = ",A18>,NAME_ARY);
  CLOSE(OUT,LOCK);
END.
```

Output

```
RESULT = 1  
NAME   = CANADAGP
```

In this call, the parameters have the following meanings:

CS_CCVERSIONV represents an integer that requests the ccsversion number be returned. It is one of two options:

Value	Value Name	Meaning
0	CS_CHARACTER_SETV	Return a coded character set name.
1	CS_CCVERSIONV	Return a ccsversion name.

NUM is an integer passed to the procedure. It contains the number for the coded character set or ccsversion for which the name is being requested. The value -2 indicates that the system default ccsversion is being validated. Refer to the MLS Guide for more information about the hierarchy.

NAME_ARY is an EBCDIC array returned by the procedure. It contains the coded character set or ccsversion name. The maximum length of the name is 17 characters.

RESULT is an integer returned by the procedure. It indicates whether or not an error occurred during the execution of the procedure. An explanation of the error result values and messages can be found at the end of this section. The values returned by this procedure include:

CS_ARRAY_TOO_SMALLV (3001)	CS_FAULTV (1001)
CS_BAD_ARRAY_DESCRIPTIONV (3000)	CS_NO_NUM_FOUNDV (3003)
CS_BAD_TYPE_CODEV (3006)	CS_SOFTERRV (1002)
CS_DATAOKV (1)	

VSNCOMPARE_TEXT

This procedure compares two strings, TEXT1 and TEXT2, using a binary, equivalent or logical comparison, specified in ORD_TYPE. The starting position for the comparison is specified for each record, along with the compare relationship to be checked.

A binary comparison is based on the hexadecimal code values of the characters. An equivalent comparison is based on the ordering sequence values (OSVs) of the characters. A logical comparison is based on the OSVs plus the priority sequence values (PSVs). These OSVs and PSVs are retrieved from the SYSTEM/CCSFILE based on the ccsversion.

Example

This example compares two strings using the CanadaEBCDIC ccsversion. The compare relation is CS_CMPEQLV(=) to determine if one string is equal to another, using a logical comparison. The MLS Guide can be used to determine that the ccsversion number for CanadaEBCDIC is 74. This number can also be retrieved by calling VALIDATE_NAME_RETURN_NUM with the name CanadaEBCDIC.

```
BEGIN

$ INCLUDE INTL="SYMBOL/INTL/ALGOL/PROPERTIES." 10000000 - 49999999

DEFINE TEXT1_START = 0 #
      ,TEXT2_START = 0 #;

FILE OUT(KIND=DISK,UNITS=CHARACTERS,FILEUSE=IO,FILETYPE=0,NEWFILE,
      MAXRECSIZE=80,TITLE="OUT/ALGOL/VSNCOMPARE_TEXT.");
EBCDIC ARRAY TEXT1[0:4]
      ,TEXT2[0:4];
INTEGER COMPARE_LEN
      ,VSN_NUM
      ,RESULT;

VSN_NUM := 74;
COMPARE_LEN := MIN(SIZE(TEXT1), SIZE(TEXT2));
REPLACE TEXT1 BY "hotel";
REPLACE TEXT2 BY "hôte1";
RESULT := VSNCOMPARE_TEXT(VSN_NUM, TEXT1, TEXT1_START, TEXT2,
      TEXT2_START, COMPARE_LEN, CS_CMPEQLV, CS_LOGICALV);
WRITE(OUT,<"RESULT = ",J4>,RESULT);
CLOSE(OUT,LOCK);
END.
```

Output

RESULT = 0

In this call, the parameters have the following meanings:

VSN_NUM is an integer passed to the procedure. It contains the number of the ccsversion that is used to compare TEXT1 and TEXT2. The number can be obtained by referring to the MLS Guide. The following shows the valid values:

- If the number is greater than or equal to 0 (zero), then the number designates a ccsversion
- If the number is -2, the procedure uses the system default ccsversion. If the system default ccsversion is not available, the procedure returns an error in RESULT.

TEXT1 is an EBCDIC array passed to the procedure. It contains the first text to be compared.

TEXT1_START is an integer passed to the procedure. It contains the byte offset (0 relative) of TEXT1 where the comparison starts.

TEXT2 is a record passed to the procedure. It contains the last text to be compared.

TEXT2_START is an integer passed to the procedure. It contains the byte offset (0 relative) of TEXT2 where the comparison starts.

COMPARE_LEN is an integer passed to the procedure. It designates the number of characters to be compared. If COMPARE_LEN is larger than the number of characters in the strings, then the procedure might be comparing invalid data. The value of COMPARE_LEN should not exceed the bounds of either the TEXT1 array or the TEXT2 array. The strings should be of equal size or padded with blanks up to COMPARE_LEN. If all pairs of characters compare equally through the last pair, the strings are considered equal. The first pair of unequal characters to be encountered is compared to determine their relative ordering. The string that contains the character with the higher ordering (higher OSV and PSV) is considered to be the greater string. If because of substitution the strings become of unequal size, the comparison proceeds as if the shorter string had been expanded by blanks on the right to make it equal in size to the longer string.

CS_CMPEQLV represents an integer that indicates that TEXT1 is equal to TEXT2. It is one of six comparison options:

Value	Value Name	Meaning
0	CS_CMPLSSV	TEXT1 is less than TEXT2.
1	CS_CMPLEQV	TEXT1 is less than or equal to TEXT2.
2	CS_CMPEQLV	TEXT1 is equal to TEXT2.
3	CS_CMPGTRV	TEXT1 is greater than TEXT2.
4	CS_CMPGEQV	TEXT1 is greater than or equal to TEXT2.
5	CS_CMPNEQV	TEXT1 is not equal to TEXT2.

CS_LOGICALV represents an integer that indicates that a logical comparison is to be done. It is one of two options:

Value	Value Name	Meaning
1	CS_EQUIVALENTV	Perform an equivalent comparison.
2	CS_LOGICALV	Perform a logical comparison.

RESULT is an integer returned by the procedure. It indicates whether or not an error occurred during the execution of the procedure. An explanation of the error result values and messages can be found at the end of this section. The values returned by this procedure include:

CS_BAD_TYPE_CODEV (3006)	CS_FILE_ACCESS_ERRORV (1000)
CS_DATAOKV (1)	CS_NO_NUM_FOUNDV (3003)
CS_FALSEV (0)	CS_SOFTERRV (1002)
CS_FAULTV (1001)	

VSNESCAPEMENT

This procedure takes the input text and rearranges it according to the escapement rules of the `ccsversion`. Both the character advance direction and the character escapement direction are used. If the character advance direction is positive, then the starting position for the text is the leftmost position in the `DEST` parameter. If the character advance direction is negative, then the starting position for the text is the rightmost position in the `DEST` parameter. From that point on, the character advance direction value and the character escapement direction values, in combination, control where each character should be placed in relation to the previous character.

Example

This example takes the string `ABCDEFGH` and rearranges it according to the escapement rules of the `ccsversion`. Assume for this example a `ccsversion` number of 999 with a character advance direction (`CAD`) of plus (+, left to right) and with the following character escapements:

Character	Escapement	Meaning
A	+	Left to right
B	-	Right to left
C	-	Right to left
D	-	Right to left
E	+	Left to right
F	+	Left to right
G	blank	Use <code>CAD</code> value

```
BEGIN
```

```
$ INCLUDE INTL="SYMBOL/INTL/ALGOL/PROPERTIES." 10000000 - 49999999
```

```
DEFINE SOURCE_START = 0 #  
      ,DEST_START = 0 #;
```

```
FILE OUT(KIND=DISK,UNITS=CHARACTERS,FILEUSE=IO,FILETYPE=0,NEWFILE,  
      MAXRECSIZE=80,TITLE="OUT/ALGOL/VSNESCAPEMENT.");
```

```
EBCDIC ARRAY SOURCE[0:6]  
      ,DEST[0:6];
```

```
INTEGER VSN_NUM  
      ,TRANS_LEN  
      ,RESULT;
```

```
VSN_NUM := 999;  
REPLACE SOURCE[0] BY "ABCDEFGH" FOR 7;  
TRANS_LEN := 7;  
RESULT := VSNESCAPEMENT(VSN_NUM, SOURCE, SOURCE_START, DEST,  
      TRANS_LEN);  
WRITE(OUT,<"RESULT = ",J4>,RESULT);  
IF RESULT EQL CS_DATAOKV THEN
```

```

        WRITE(OUT,<"DEST  = ",A7>,DEST);
    CLOSE(OUT,LOCK);
END.

```

Output

```

RESULT = 1
DEST   = ADCBEFG

```

In this call, the parameters have the following meanings:

VSN_NUM is an integer passed by reference to the procedure. It specifies the ccsversion to be used. The ccsversion contains the escapement rules. The following are the values allowed for VSN_NUM:

Value	Meaning
Greater than or equal to 0	Specifies a ccsversion. The numbers of the ccsversions are listed in the <i>MLS Guide</i> .
-2	Specifies the system default ccsversion. If the system default ccsversion is not available, an error is returned.

SOURCE is an EBCDIC array passed to the procedure. It contains the text to be arranged according to the escapement rules.

SOURCE_START is an integer passed to the procedure. It specifies the starting position in SOURCE (0 relative) for the arranging of text by escapement rules to begin.

DEST is an EBCDIC array returned by the procedure. It contains the resulting text after the escapement rules have been applied.

TRANS_LEN is an integer passed to the procedure. It specifies the number of characters to arrange according to the escapement rules.

RESULT is an integer returned by the procedure. It indicates whether or not an error occurred during the execution of the procedure. An explanation of the error result values and messages can be found at the end of this section. The values returned by this procedure include:

CS_BAD_ARRAY_DESCRIPTIONV (3000)	CS_FILE_ACCESS_ERRORV (1000)
CS_BAD_DATA_LEN (3002)	CS_NO_NUM_FOUNDV (3003)
CS_DATAOKV (1)	CS_SOFTERRV (1002)
CS_FAULTV (1001)	

VSNGETORDERINGFOR_ONE_TEXT

This procedure returns the ordering information for the input text. The ordering information determines how the input text is collated. It includes the ordering and priority sequence values of the characters and any substitution of characters to be made when the input text is sorted. One of the following types of ordering information can be chosen:

- Equivalent ordering information, which comprises only the ordering sequence values (OSVs)
- Logical ordering information, which comprises the OSVs followed by the priority sequence values (PSVs)

Example

This example obtains the ordering sequence values and priority sequence values for an input text string. The ccsversion is CanadaEBCDIC. The MLS Guide can be used to determine that the ccsversion number for CanadaEBCDIC is 74. This number can also be retrieved by calling VALIDATE_NAME_RETURN_NUM with the name CanadaEBCDIC. This example requests logical ordering information, so that both the OSVs and PSVs are returned. This example allows for maximum substitution, so the parameter MAX_OSVS is equal to (ITEXT_LEN * 3) and TOTAL_STORAGE is equal to INTEGER(MAX_OSVS/2).

```
BEGIN

$ INCLUDE INTL="SYMBOL/INTL/ALGOL/PROPERTIES." 10000000 - 49999999

DEFINE SOURCE_START = 0 #
      ,DEST_START = 0 #;

FILE OUT(KIND=DISK,UNITS=CHARACTERS,FILEUSE=IO,FILETYPE=0,NEWFILE,
      MAXRECSIZE=80,TITLE="OUT/ALGOL/VSNGETORD_ONE_TEX.");
EBCDIC ARRAY SOURCE[0:25]
      ,DEST[0:25];
INTEGER VSN_NUM
      ,ITEXT_LEN
      ,MAX_OSVS
      ,TOTAL_STORAGE
      ,RESULT;

VSN_NUM := 74;
ITEXT_LEN := 5;
MAX_OSVS := ITEXT_LEN * 3;
TOTAL_STORAGE := MAX_OSVS + INTEGER(MAX_OSVS/2);
REPLACE SOURCE[0] BY "ABC»«";
RESULT:=VSNGETORDERINGFOR_ONE_TEXT(VSN_NUM, SOURCE, SOURCE_START,
      ITEXT_LEN, DEST, DEST_START, MAX_OSVS, TOTAL_STORAGE,
      CS_LOGICALV);
WRITE(OUT,<"RESULT = ",J4>,RESULT);
IF RESULT EQL CS_DATAOKV THEN
      WRITE(OUT,<"DEST = ",A25>,DEST);
```

```
CLOSE(OUT, LOCK);
END.
```

Output

```
RESULT = 1
DEST = 414243414541 450000000000 00000111221 100000000000
```

VSN_NUM is an integer passed to the procedure. It contains the number of the ccsversion that is used.

SOURCE is an array passed to the procedure. It contains the text for which the ordering information is requested.

SOURCE_START is an integer passed to the procedure. It contains the offset of the location where the source text is to begin.

ITEXT_LEN is an integer passed to the procedure. It contains the length of the source text.

DEST is a result returned by the procedure. It contains the ordering information for the source text.

DEST_START is an integer returned by the procedure. It designates the starting offset at which the result values are placed.

MAX_OSVS is an integer passed to the procedure. It designates the maximum number of bytes to be used to store the ordering sequence values.

The value of MAX_OSVS should be at least the length of the input text, but it might need to be greater to allow for substitution. The maximum substitution length is 3 bytes; therefore, to allow for substitution for every character, the value of MAX_OSVS is as follows:

$$\langle \text{length of source text in bytes} \rangle * 3$$

If the number of ordering sequence values returned is less than MAX_OSVS, then the array row is padded with the ordering sequence value for blank.

TOTAL_STORAGE is an integer passed by the procedure. It defines the maximum number of bytes needed to store the complete ordering information for the text. If equivalent ordering information is requested, TOTAL_STORAGE is equal to MAX_OSVS. If logical ordering information is requested, space must be provided for the four-bit priority values in addition to the space allowed for the OSVs. Each OSV has one PSV, and one byte can hold two PSVs. Therefore, the space allowed for PSVs is MAXOSVS/2, and the value of TOTAL_STORAGE is as follows:

$$\text{MAXOSVS} + (\text{MAXOSVS}/2)$$

When the ordering information is returned by the procedure, all the OSVs are listed first, followed by all the PSVs.

Internationalization

CS_LOGICALV represents an integer that requests logical ordering information. It is one of two options:

Parameter Value	Value Name	Type of Information Returned
1	CS_EQUIVALENTV	Ordering sequence values only
2	CS_LOGICALV	Priority sequence values only

RESULT is an integer returned by the procedure. It indicates whether or not an error occurred during the execution of the procedure. An explanation of the error result values and messages can be found at the end of this section. The values returned by this procedure include:

CS_ARRAY_TOO_SMALLV (3001)	CS_FALSEV (0)
CS_BAD_ARRAY_DESCRIPTIONV (3000)	CS_FAULTV (1001)
CS_BAD_DATA_LEN (3002)	CS_FILE_ACCESS_ERRORV (1000)
CS_BAD_TEXT_PARAMV (3008)	CS_NO_NUM_FOUNDV (3003)
CS_BAD_TYPE_CODEV (3006)	CS_SOFTERRV (1002)
CS_DATAOKV (1)	

VSNINFO

This procedure returns the following information for a designated ccsversion:

- The number of the base coded character set to which the ccsversion applies
- The escapement information
- The space characters used for the ccsversion
- The array sizes required by the ccsversion translate tables and sets

Example

This example calls the procedure VSNINFO to get information about the system default ccsversion. For this example assume the system default ccsversion in Norway (71).

```
BEGIN

$ INCLUDE INTL="SYMBOL/INTL/ALGOL/PROPERTIES." 10000000 - 49999999

FILE OUT(KIND=DISK,UNITS=CHARACTERS,FILEUSE=IO,FILETYPE=0,NEWFILE,
        MAXRECSIZE=80,TITLE="OUT/ALGOL/VSNINFO.");

ARRAY VSN_ARY[0:7]
      ,LINEOUT[0:14];
INTEGER VSN_NUM
      ,I
      ,RESULT;
FORMAT FORM1(A31,T41,A10,T55,J2)
      ,FORM2("    SPACE CHARACTER[" ,I1,"] (HEX)",T41,A5,J2,A3,T55,H2);

VSN_NUM := CS_VSN_NOT_SPECIFIEDV;
RESULT := VSNINFO(VSN_NUM, VSN_ARY);
WRITE(OUT,<"RESULT = " ,J4>,RESULT);
IF RESULT EQL CS_DATAOKV THEN
  BEGIN
    REPLACE LINEOUT BY " " FOR 80;
    WRITE(OUT,80,LINEOUT);
    WRITE(OUT,<"VSN ARRAY">);
    WRITE(OUT,<"FIELD MEANING",T40,"LOCATION",T53,"VALUE">);
    WRITE(OUT,<"-----",T40,"-----",T53,"-----">);
    WRITE(OUT, FORM1, "BASE CHARACTER SET NUMBER    ",
          "[0]    ", VSN_ARY[0]);
    WRITE(OUT, FORM1, "TEXT LINE ORIENTATION    ",
          "[1]    ", VSN_ARY[1]);
    WRITE(OUT, FORM1, "LINE ADVANCE DIRECTION    ",
          "[2]    ", VSN_ARY[2]);
    WRITE(OUT, FORM1, "CHARACTER ADVANCE DIRECTION    ",
          "[3]    ", VSN_ARY[3]);
    WRITE(OUT, FORM1, "NUMBER OF SPACES CHARACTERS    ",
          "[4].[47:8]" , VSN_ARY[4].[47:8]);
```

```
FOR I := 1 STEP 1 UNTIL VSN_ARY[4].[47:8] DO
    WRITE(OUT, FORM2, I, "[4].[",47-I*8,":8]",VSN_ARY[4].[47-I*8:8]);
    WRITE(OUT, FORM1, "SIZE OF SPACES IN TRUTHSET ",
        "[5].[47:8]", VSN_ARY[5].[47:8]);
    WRITE(OUT, FORM1, "SIZE OF ALPHA TRUTHSET      ",
        "[5].[39:8]", VSN_ARY[5].[39:8]);
    WRITE(OUT, FORM1, "SIZE OF NUMERIC TRUTHSET      ",
        "[5].[31:8]", VSN_ARY[5].[31:8]);
    WRITE(OUT, FORM1, "SIZE OF PRESENTATION TRUTHSET ",
        "[5].[23:8]", VSN_ARY[5].[23:8]);
    WRITE(OUT, FORM1, "SIZE OF LOWER TRUTHSET      ",
        "[5].[15:8]", VSN_ARY[5].[15:8]);
    WRITE(OUT, FORM1, "SIZE OF UPPER TRUTHSET      ",
        "[5].[7:8] ", VSN_ARY[5].[7:8]);
    WRITE(OUT, FORM1, "UNUSED                          ",
        "[6].[47:8]", VSN_ARY[6].[47:8]);
    WRITE(OUT, FORM1, "SIZE OF ESCAPEMENT TRANS TABLE ",
        "[6].[39:8]", VSN_ARY[6].[39:8]);
    WRITE(OUT, FORM1, "SIZE OF LOWTOUP TRANS TABLE  ",
        "[6].[31:8]", VSN_ARY[6].[31:8]);
    WRITE(OUT, FORM1, "SIZE OF UPTOLOW TRANS TABLE  ",
        "[6].[23:8]", VSN_ARY[6].[23:8]);
    WRITE(OUT, FORM1, "SIZE OF NUMTOALTDIG TRANS TABLE",
        "[6].[15:8]", VSN_ARY[6].[15:8]);
    WRITE(OUT, FORM1, "SIZE OF ALTDIGTONUM TRANS TABLE",
        "[6].[7:8] ", VSN_ARY[6].[7:8]);
    WRITE(OUT, FORM1, "SIZE OF OSV TRANS TABLE      ",
        "[7].[47:8]", VSN_ARY[7].[47:8]);
    WRITE(OUT, FORM1, "SIZE OF PSV TRANS TABLE      ",
        "[7].[39:8]", VSN_ARY[7].[39:8]);
    WRITE(OUT, FORM1, "SIZE OF SUBSTCHAR TRANS TABLE ",
        "[7].[31:8]", VSN_ARY[7].[31:8]);
    WRITE(OUT, FORM1, "SIZE OF SUBSTSEQ TRANS TABLE  ",
        "[7].[23:8]", VSN_ARY[7].[23:8]);
    WRITE(OUT, FORM1, "SIZE OF SUBSTOSV TRANS TABLE  ",
        "[7].[15:8]", VSN_ARY[7].[15:8]);
    WRITE(OUT, FORM1, "SIZE OF SUBSTPSV TRANS TABLE  ",
        "[7].[7:8] ", VSN_ARY[7].[7:8]);
    END;
    CLOSE(OUT, LOCK);
END.
```


Output

RESULT = 1

VSN ARRAY FIELD MEANING	LOCATION	VALUE
-----	-----	-----
BASE CHARACTER SET NUMBER	[0]	4
TEXT LINE ORIENTATION	[1]	0
LINE ADVANCE DIRECTION	[2]	0
CHARACTER ADVANCE DIRECTION	[3]	0
NUMBER OF SPACES CHARACTERS	[4].[47:8]	1
SPACE CHARACTER[1] (HEX)	[4].[39:8]	40
SIZE OF SPACES IN TRUTHSET	[5].[47:8]	8
SIZE OF ALPHA TRUTHSET	[5].[39:8]	8
SIZE OF NUMERIC TRUTHSET	[5].[31:8]	8
SIZE OF PRESENTATION TRUTHSET	[5].[23:8]	8
SIZE OF LOWER TRUTHSET	[5].[15:8]	8
SIZE OF UPPER TRUTHSET	[5].[7:8]	8
UNUSED	[6].[47:8]	0
SIZE OF ESCAPEMENT TRANS TABLE	[6].[39:8]	64
SIZE OF LOWTOUP TRANS TABLE	[6].[31:8]	64
SIZE OF UPTOLOW TRANS TABLE	[6].[23:8]	64
SIZE OF NUMTOALTDIG TRANS TABLE	[6].[15:8]	0
SIZE OF ALTDIGTONUM TRANS TABLE	[6].[7:8]	0
SIZE OF OSV TRANS TABLE	[7].[47:8]	64
SIZE OF PSV TRANS TABLE	[7].[39:8]	64
SIZE OF SUBSTCHAR TRANS TABLE	[7].[31:8]	0
SIZE OF SUBSTSEQ TRANS TABLE	[7].[23:8]	0
SIZE OF SUBSTOSV TRANS TABLE	[7].[15:8]	0
SIZE OF SUBSTPSV TRANS TABLE	[7].[7:8]	0

In this call, the parameters have the following meanings:

VSN_NUM is an integer passed to the procedure. It contains the ccsversion number for which information is requested.

VSN_ARY is an array returned by the procedure. It contains the ccsversion information, as follows:

Location	Information Contained
WORD 0	Base coded character set number
WORD 1	Text line orientation
WORD 2	Line advance direction
WORD 3	Character advance direction
WORD 4	Space characters
	The first byte is the number of space characters; the actual characters follow

Internationalization

Location	Information Contained
WORD 5.[47:08]	Size of SPACES truth set
WORD 5.[39:08]	Size of ALPHA truth set
WORD 5.[31:08]	Size of NUMERIC truth set
WORD 5.[23:08]	Size of PRESENTATION truth set
WORD 5.[15:08]	Size of LOWER truth set
WORD 5.[07:08]	Size of UPPER truth set
WORD 6.[47:08]	Unused
WORD 6.[39:08]	Size of ESCAPEMENT translation table
WORD 6.[31:08]	Size of LOWTOUP translation table
WORD 6.[23:08]	Size of UPTOLOW translation table
WORD 6.[15:08]	Size of NUMTOALTDIG translation table
WORD 6.[07:08]	Size of ALTDIGTONUM translation table
WORD 7.[47:08]	Size of OSV translation table
WORD 7.[39:08]	Size of PSV translation table
WORD 7.[31:08]	Size of SUBSTCHAR translation table
WORD 7.[23:08]	Size of SUBSTSEQ translation table
WORD 7.[15:08]	Size of SUBSTOSV translation table
WORD 7.[07:08]	Size of SUBSTPSV translation table

It is recommended that the size of VSN_ARY be 8 words.

RESULT is an integer returned by the procedure. It indicates whether or not an error occurred during the execution of the procedure. An explanation of the error result values and messages can be found at the end of this section. The values returned by this procedure include:

CS_ARRAY_TOO_SMALLV (3001)	CS_FILE_ACCESS_ERRORV (1000)
CS_BAD_ARRAY_DESCRIPTIONV (3000)	CS_NO_NUM_FOUNDV (3003)
CS_DATAOKV (1)	
CS_FAULTV (1001)	CS_SOFTERRV (1002)

VSNINSPECT_TEXT

This procedure searches specified text for characters that are present or not present in a requested truth set. The SCANNED_CHARS parameter is an integer that represents the number of characters that were searched when the criteria specified in the FLAG parameter were met. If SCANNED_CHARS is equal to INSPECT_LEN, then all the characters were searched, but none met the criteria. Otherwise, adding the TEXT_START value to the SCANNED_CHARS value gives the location of the character, from the start of the array, that met the search criteria.

Example

This example examines a record that contains two fields, a name and a phone number. The name is verified to contain only alphabetic characters as defined by the France ccsversion. The MLS Guide can be used to determine that the ccsversion number for France is 35. This number can also be retrieved by calling VALIDATE_NAME_RETURN_NUM with the name France.

```
BEGIN

$ INCLUDE INTL="SYMBOL/INTL/ALGOL/PROPERTIES." 10000000 - 49999999

DEFINE IDNUMLEN = 10 #
      ,NAMELEN = 30 #
      ,TEXT_START = 0 #
      ,INSPECT_LEN = IDNUMLEN #;

FILE OUT(KIND=DISK,UNITS=CHARACTERS,FILEUSE=IO,FILETYPE=0,NEWFILE,
      MAXRECSIZE=80,TITLE="OUT/ALGOL/VSNINSPECT_TEXT.");
EBCDIC ARRAY TEXT[0:NAMELEN+IDNUMLEN];
INTEGER SCANNED_CHARS
      ,RESULT
      ,VSN_NUM;

REPLACE TEXT[0] BY "7775961089John Alan Smith           ";
VSN_NUM := 35;
RESULT := VSNINSPECT_TEXT(VSN_NUM, TEXT, TEXT_START, INSPECT_LEN,
      CS_NUMERICSV, CS_NOTINTSETV, SCANNED_CHARS);
WRITE(OUT,<"RESULT = ",J4>,RESULT);
IF RESULT EQL CS_DATAOKV THEN
```

```
BEGIN
WRITE(OUT,<"SCANNED CHARACTERS = ",J4>,SCANNED_CHARS);
IF SCANNED_CHARS NEQ IDNUMLEN THEN
WRITE(OUT,<"ERROR - INVALID ID, BAD CHARACTER FOUND">);
END;
CLOSE(OUT,LOCK);
END.
```

Output

```
RESULT = 1
SCANNED CHARACTERS = 10
```

In this call, the parameters have the following meanings:

VSN_NUM is an integer passed to the procedure. It specifies the ccsversion to be used. The ccsversion contains the rules for applying a truth set. The following are the values allowed for VSN_NUM:

Value	Meaning
Greater than or equal to 0	Specifies a ccsversion. The numbers of the ccsversions are listed in the <i>MLS Guide</i> .
-2	Specifies the system default ccsversion. If the system default ccsversion is not available, an error is returned.

TEXT is an EBCDIC array passed to the procedure. The array is then searched using the requested truth set.

TEXT_START is an integer passed to the procedure. It contains the byte offset (0 relative) in TEXT where the search starts.

INSPECT_LEN is an integer passed to the procedure. It specifies the number of characters to be searched beginning at TEXT_START. In other words, it specifies that maximum length of the search.

CS_NUMERICSV represents an integer that indicates that a numeric truth set be used for the search. It is one of six options:

Value	Sample Data Name	Meaning
12	CS_ALPHAV	Alphabetic truth set. It identifies the characters defined as alphabetic in the specified ccsversion.
13	CS_NUMERICSV	Numeric truth set. It identifies the characters defined as numeric in the specified ccsversion.
14	CS_PRESENTATIONV	Presentation truth set. It identifies the characters in the ccsversion that can be represented on a presentation device, for example a printer.
15	CS_SPACESV	Spaces truth set. It identifies the characters defined as space in the specified ccsversion.

Value	Sample Data Name	Meaning
16	CS_LOWERCASEV	Lowercase truth set. It identifies the characters defined as lowercase alphabetic in the ccsversion.
17	CS_UPPERCASEV	Uppercase truth set. It identifies the characters defined as uppercase alphabetic in the ccsversion.

A ccsversion is not required to have a definition for each of these truth sets. Some of the truth sets are optional. Options 16 and 17 are not required. A result of CS_DATA_NOT_FOUNDV (4002) might be returned if the truth set has not been defined for the ccsversion. The input text remains unchanged.

CS_NOTINTSETV represents an integer that indicates the type of search to be performed. It is one of two options:

Value	Sample Data Name	Meaning
0	CS_NOTINTSETV	Search TEXT until a character is found that is not in the requested truth set.
1	CS_INTSETV	Search TEXT until a character is found that is in the requested truth set.

SCANNED_CHARS is an integer returned by the procedure. It contains the number of characters that were scanned when the search criteria were met.

RESULT is an integer returned by the procedure. It indicates whether or not an error occurred during the execution of the procedure. An explanation of the error result values and messages can be found at the end of this section. The values returned by this procedure include:

CS_ARRAY_TOO_SMALLV (3001)	CS_DATA_NOT_FOUNDV (4002)
CS_BAD_ARRAY_DESCRIPTIONV (3000)	CS_FAULTV (1001)
CS_BAD_FLAGV (3007)	CS_FILE_ACCESS_ERRORV (1000)
CS_BAD_TYPE_CODEV (3006)	CS_NO_NUM_FOUNDV (3003)
CS_DATAOKV (1)	CS_SOFTERRV (1002)

VSNORDERING_INFO

This procedure returns the ordering information for a designated ccsversion. The ordering information determines the way in which data is collated for the ccsversion. It includes the ordering and priority sequence values of the characters and any substitution of characters to be made when the designated ccsversion ordering is applied to a string of text.

The table obtained with this procedure might be used with the COMPARE_TEXT_USING_ORDER_INFO procedure. Because the table is stored in working memory, it does not have to be retrieved each time it is used. Using this combination of procedures in place of a general call to the VSNCOMPARE_TEXT procedure can improve the performance of an application program that performs a high volume of translations.

Example

This example gets the ordering information for the CanadaEBCDIC ccsversion. The ordering information is returned in ORDER_ARY. The MLS Guide can be used to determine that the ccsversion number for CanadaEBCDIC is 74. This number can also be retrieved by calling VALIDATE_NAME_RETURN_NUM with the name CanadaEBCDIC.

```
BEGIN

$ INCLUDE INTL="SYMBOL/INTL/ALGOL/PROPERTIES." 10000000 - 49999999

FILE OUT(KIND=DISK,UNITS=CHARACTERS,FILEUSE=IO,FILETYPE=0,NEWFILE,
        MAXRECSIZE=80,TITLE="OUT/ALGOL/VSNORDERING_INFO.");
ARRAY ORDER_ARY[0:255];
INTEGER VSN_NUM
        ,RESULT;

VSN_NUM := 74;
RESULT := VSNORDERING_INFO(VSN_NUM, ORDER_ARY);
WRITE(OUT,<"RESULT = ",J4>, RESULT);
CLOSE(OUT,LOCK);
END.
```

Output

RESULT = 1

In this call, the parameters have the following meanings:

VSN_NUM is an integer passed to the procedure. It contains the number of the ccsversion for which ordering information is requested.

ORDER_ARY is an array returned from the procedure. It contains the ordering information for the ccsversion. The first two words of the array are size words. Word 1 is not used at this time. Word 0 contains the following information:

Word	Offset	Information Contained
Word 0	[47:08]	Size of ordering ttable in words (always 64)
Word 0	[39:08]	Size of priority ttable in words (always 64)
Word 0	[31:08]	Size of substitution characters truth set in words (may be greater than or equal to 0 words)
Word 0	[23:08]	Size of substitution sequences array in words (may be greater than or equal to 0 words)
Word 0	[15:08]	Size of substitution ordering array in words (may be greater than or equal to 0 words)
Word 0	[07:08]	Size of substitution priority array in words (may be greater than or equal to 0 words)

ORDER_ARY also contains the following components in order:

1. Ordering translate table
64-word ALGOL translate table. Translates each character to its ordering sequence value.
2. Priority translate table
64-word ALGOL translate table. Translates each character to its priority sequence value.
3. Substitution characters truth set
An ALGOL truth set that contains all the characters that are part of any substitution sequence, many to one or one to many.
4. Substitution sequences array
A real array. Each word contains one of the substitution sequences, right justified. For example, if ch is a substitution sequence, then it is stored as 0000008388.

5. Substitution ordering array

A real array, parallel to substitution sequences array. For each substitution sequence, the ordering sequence values for that sequence are stored in the corresponding word along with the length (left-justified). The format of the word is [47:08] number of ordering sequence values. This must be greater than or equal to 1, or less than or equal to 3. The next [47:08] bytes contain the OSVs left-justified. For example, suppose a with e is substituted by the OSVs 129 and 133. The substitution ordering word will appear as 028185000000. The elements of the substitution ordering word have the following meanings:

02	Two OSVs
81	The OSV 129
85	The OSV 133

6. Substitution priority array

A real array. Each word contains the priority sequence values corresponding to the substitution ordering array. The word format of this array is exactly the same as the word format of the substitution ordering array.

It is recommended that the size of this array be 256 words.

RESULT is an integer returned by the procedure. It indicates whether or not an error occurred during the execution of the procedure. An explanation of the error result values and messages can be found at the end of this section. The values returned by this procedure include:

CS_ARRAY_TOO_SMALLV (3001)	CS_FILE_ACCESS_ERRORV (1000)
CS_BAD_ARRAY_DESCRIPTIONV (3000)	CS_NO_NUM_FOUNDV (3003)
CS_DATAOKV (1)	CS_SOFTERRV (1002)
CS_FAULTV (1001)	

VSNTRANSTABLE

This procedure returns a translation table for a designated ccsversion. The type of translation table requested depends on the task to be performed.

Translation tables can be used to perform the following tasks:

- Translate lowercase letters to uppercase letters.
- Translate uppercase letters to lowercase letters.
- Translate any digits 0 through 9 to any alternative digits (that is, one-to-one mapping of 0 through 9 to another representation for those digits).
- Translate alternative digits to 0 through 9.
- Determine the escapement direction for each character.

The translation table from VSNTRANSTABLE can be retained in working memory. This translation table can then be used with the REPLACE syntax. This improves performance over a call to VSNTRANS_TEXT. This process can improve performance of an application program that performs a high volume of translations. Because the table is stored in working memory, it does not have to be retrieved each time it is to be used.

Example

This example gets the uppercase to lowercase translate table for the CanadaEBCDIC ccsversion. The translate table is returned in the TTABLE_ARY. The MLS Guide can be used to determine that the ccsversion number for CanadaEBCDIC is 74. This number can also be retrieved by calling VALIDATE_NAME_RETURN_NUM with the name CanadaEBCDIC.

```
BEGIN

$ INCLUDE INTL="SYMBOL/INTL/ALGOL/PROPERTIES." 10000000 - 49999999

FILE OUT(KIND=DISK,UNITS=CHARACTERS,FILEUSE=IO,FILETYPE=0,NEWFILE,
        MAXRECSIZE=80,TITLE="OUT/ALGOL/VSNTRANS_TABLE.");
ARRAY TTABLE_ARY[0:63];
INTEGER VSN_NUM
        ,RESULT;

VSN_NUM := 74;
RESULT := VSNTRANSTABLE(VSN_NUM, CS_UPTOLOWCASEV, TTABLE_ARY);
WRITE(OUT,<"RESULT = ",J4>, RESULT);
CLOSE(OUT,LOCK);

END.
```

Output

RESULT = 1

In this call, the parameters have the following meanings:

VSN_NUM is an integer passed to the procedure. It designates the number of the ccsversion from which the translation table is to be retrieved.

CS_UPTOLOWCASEV represents an integer that requests the translation of uppercase characters to lowercase. It is one of five options:

Parameter Value	Value Name	Translation Task
5	CS_NUMTOALTDIGV	Numeric characters to alternative digits
6	CS_ALTDIGTONUMV	Alternative digits to numeric characters
7	CS_LOWTOUPCASEV	Lowercase characters to uppercase characters
8	CS_UPTOLOWCASEV	Uppercase characters to lowercase characters
9	CS_ESCMENTPERCHARV	Escapement direction for each character

A ccsversion is not required to have a definition for each of these tables. Some of the tables are optional. Options 5, 6, 7, and 8 are not required. A result of CS_DATA_NOT_FOUNDV (4002) might be returned if the table has not been defined for the ccsversion.

TTABLE_ARY is an array returned from the procedure. It contains the translate table. The size of this array must be at least 64 words.

RESULT is an integer returned by the procedure. It indicates whether or not an error occurred during the execution of the procedure. An explanation of the error result values and messages can be found at the end of this section. The values returned by this procedure include:

CS_ARRAY_TOO_SMALLV (3001)	CS_FAULTV (1001)
CS_BAD_ARRAY_DESCRIPTIONV (3000)	CS_FILE_ACCESS_ERRORV (1000)
CS_BAD_TYPE_CODEV (3006)	CS_NO_NUM_FOUNDV (3003)
CS_DATAOKV (1)	CS_SOFTERRV (1002)
CS_DATA_NOT_FOUNDV (4002)	

VSNTRANS_TEXT

This procedure applies a translate table, specified by TTABLE_TYPE, to the source text and places the result into DEST. The same array can be used for both the source and destination text.

There are five translate table types available. These types include the following:

- Numeric to alternate digits
- Alternate digits to numeric
- Lowercase to uppercase
- Uppercase to lowercase
- Escapement direction for each character

Example

This example translates a string in lowercase letters to uppercase letters using the CanadaEBCDIC ccsversion. The MLS Guide can be used to determine that the ccsversion number for CanadaEBCDIC is 74. This number can also be retrieved by calling VALIDATE_NAME_RETURN_NUM with the name CanadaEBCDIC.

```
BEGIN

$ INCLUDE INTL="SYMBOL/INTL/ALGOL/PROPERTIES." 10000000 - 49999999

DEFINE SOURCE_START = 0 #
      ,DEST_START = 0 #;

FILE OUT(KIND=DISK,UNITS=CHARACTERS,FILEUSE=IO,FILETYPE=0,NEWFILE,
      MAXRECSIZE=80,TITLE="OUT/ALGOL/VSNTRANS_TEXT.");
EBCDIC ARRAY SOURCE[0:6]
      ,DEST[0:6];
INTEGER VSN_NUM
      ,TRANS_LEN
      ,RESULT;

REPLACE SOURCE BY "pæan";
VSN_NUM := 74;
TRANS_LEN := 4;
RESULT := VSNTRANS_TEXT(VSN_NUM, SOURCE, SOURCE_START, DEST,
      DEST_START, TRANS_LEN, CS_LOWTOUPCASEV);
WRITE(OUT,<"RESULT = ",J4>,RESULT);
IF RESULT EQL CS_DATAOKV THEN
      WRITE(OUT,<"DEST = ",A7>,DEST);
CLOSE(OUT,LOCK);
END.
```

Output

```
RESULT = 1  
DEST = PÆAN
```

In this call, the parameters have the following meanings:

VSN_NUM is an integer passed to the procedure. It contains the number of the ccsversion to be used. The ccsversion contains the rules for translation of text. Refer to the MLS Guide for a list of the ccsversion numbers. The values allowed for VSN_NUM and the meanings of the values are as follows:

Value	Meaning
Greater than or equal to 0	Use the specified ccsversion number.
-2	Use the system default ccsversion. If the system default ccsversion is not available, an error is returned.

SOURCE is an EBCDIC array passed to the procedure. It contains the data to translate.

SOURCE_START is an integer passed to the procedure. It contains the byte offset (0 relative) of SOURCE where translation starts.

DEST is an EBCDIC array returned by the procedure. It contains the translated text. DEST must be at least as large as DEST_START and TRANS_LEN.

DEST_START is an integer returned by the procedure. It indicates the starting offset location where translated text should be placed.

TRANS_LEN is an integer passed to the procedure. It contains the number of characters to translate, beginning at SOURCE_START.

CS_LOWTOUPCASEV represents an integer that requests the translation of all lowercase characters to uppercase. It is one of five options:

Value	Sample Data Name	Meaning
5	CS_NUMTOALTDIGV	Translate numbers 0 through 9 to alternate digits specified in the ccsversion.
6	CS_ALTDIGTONUMV	Translate alternate digits to numbers 0 through 9.
7	CS_LOWTOUPCASEV	Translate all characters from lowercase to uppercase.
8	CS_UPTOLOWCASEV	Translate all characters from uppercase to lowercase.
9	CS_ESCMENTPERCHARV	Translate a character to its escapement value.

A ccsversion is not required to have a definition for each of these tables. Some of the tables are optional. Options 5, 6, 7, and 8 are not required. A result of CS_DATA_NOT_FOUNDV (4002) might be returned if the table has not been defined for the ccsversion. The input text remains the unchanged.

RESULT is an integer returned by the procedure. It indicates whether or not an error occurred during the execution of the procedure. An explanation of the error result values and messages can be found at the end of this section. The values returned by this procedure include:

CS_ARRAY_TOO_SMALLV (3001)	CS_DATAOKV (1)
CS_BAD_ARRAY_DESCRIPTIONV (3000)	CS_FAULTV (1001)
CS_BAD_DATA_LEN (3002)	CS_FILE_ACCESS_ERRORV (1000)
CS_BAD_TYPE_CODEV (3006)	CS_NO_NUM_FOUNDV (3003)
CS_DATA_NOT_FOUNDV (4002)	CS_SOFTERRV (1002)

VSNTRUTHSET

This procedure returns a truth set for the designated ccsversion. The truth set contains the characters in a given data class for the ccsversion. Truth sets are available for the following data classes:

- Alphabetic
- Numeric
- Space
- Presentation (The characters that can be displayed or printed on a presentation device)

The truth set from VSNTRUTHSET can be retained in working memory. This truth set can then be used with the SCAN statement. This improves performance over a call to VSNINSPECT_TEXT. Because the table is stored in working memory, it does not have to be retrieved each time it is to be used. This process can improve performance of an application program that performs a high volume of translations.

Example

This example gets the numeric data class truth set for the CanadaEBCDIC ccsversion. The truth set is returned in the TSET_ARY. The MLS Guide can be used to determine that the ccsversion number for CanadaEBCDIC is 74. This number can also be retrieved by calling VALIDATE_NAME_RETURN_NUM with the name CanadaEBCDIC.

```
BEGIN

$ INCLUDE INTL="SYMBOL/INTL/ALGOL/PROPERTIES." 10000000 - 49999999

FILE OUT(KIND=DISK,UNITS=CHARACTERS,FILEUSE=IO,FILETYPE=0,NEWFILE,
        MAXRECSIZE=80,TITLE="OUT/ALGOL/VSNTRUTH_SET.");
ARRAY TSET_ARY[0:7];
INTEGER VSN_NUM
        ,RESULT;

VSN_NUM := 74;
RESULT := VSNTRUTHSET(VSN_NUM, CS_NUMERICSV, TSET_ARY);
WRITE(OUT,<"RESULT = ",J4>, RESULT);
CLOSE(OUT,LOCK);
END.
```

Output

RESULT = 1

In this call, the parameters have the following meanings:

VSN_NUM is an integer passed to the procedure. It contains the number of the ccsversion from which the truth set is to be retrieved. The ccsversion contains the rules for translation of text. Refer to the MLS Guide for a list of ccsversion numbers. The values allowed for VSN_NUM and the meanings of the values are as follows:

Value	Meaning
Greater than or equal to 0	Use the specified ccsversion number.
-2	Use the system default ccsversion. If the system default ccsversion is not available, an error is returned.

CS_NUMERICSV represents an integer that indicates the type of truth set to be returned. It is one of six options:

Parameter Value	Value Name	Truth Set Returned
12	CS_ALPHAV	Alphabetic characters
13	CS_NUMERICSV	Numeric characters
14	CS_DISPLAYV	Displayable characters
15	CS_SPACESV	Space characters
16	CS_LOWERCASEV	Lowercase characters
17	CS_UPPERCASEV	Uppercase characters

A ccsversion is not required to have a definition for each of these truth sets. Some of the truth sets are optional. Options 16 and 17 are not required. A result of CS_DATA_NOT_FOUNDV (4002) might be returned if the truth set has not been defined for the ccsversion.

TSET_ARY is an array returned from the procedure. It contains the truth set table. It is recommended that the size of this array be 8 words.

RESULT is an integer returned by the procedure. It indicates whether or not an error occurred during the execution of the procedure. An explanation of the error result values and messages can be found at the end of this section. The values returned by this procedure include:

CS_ARRAY_TOO_SMALLV (3001)	CS_FAULTV (1001)
CS_BAD_ARRAY_DESCRIPTIONV (3000)	CS_FILE_ACCESS_ERRORV (1000)
CS_BAD_TYPE_CODEV (3006)	CS_NO_NUM_FOUNDV (3003)
CS_DATA_NOT_FOUNDV (4002)	CS_SOFTERRV (1002)
CS_DATAOKV (1)	

Explanation of Error Values

All of the procedures in the CENTRALSUPPORT library return integer results to indicate the success or failure of the procedure.

The range 1 through 999 indicates that the procedure call completed successfully.

The range 1000 through 1999 consists of error messages caused by a system software error.

Values from 2000 through 2999 contain error messages in which the caller passed invalid data to a procedure, but the CENTRALSUPPORT library was able to return some valid data.

Values from 3000 through 3999 contain error messages in which the caller passed invalid data to a CENTRALSUPPORT procedure, and the CENTRALSUPPORT library was unable to return any valid data.

Values from 4000 through 4999 contain error messages in which the caller passed data for which the CENTRALSUPPORT library could find no return information. The CENTRALSUPPORT library completed the request, but no data was returned.

Table 9–2 lists the possible error results, their values, and the specific description part of the message. For a list of the complete error messages and for information about corrective action to be taken if an error occurs, see the MLS Guide.

Table 9-2. Error Results for Internationalization

Result	Value	Specific Description
CS_ARRAY_TOO_SMALLV	3001	The output array size is smaller than the length of the data it is supposed to contain.
CS_BAD_ALT_FRAC_DIGITSV	3017	The international-fractional-digits value is either missing or out of range.
CS_BAD_AMTV	3009	The amount parameter is not an integer or its value is greater than the maximum double-precision integer.
CS_BAD_ARRAY_DESCRIPTIONV	3000	A parameter was incorrectly specified as less than or equal to 0 (zero).
CS_BAD_CPLV	3028	The characters-per-line value is either missing or it is out of range.
CS_BAD_DATA_LENv	3002	At least one array length is invalid or the offset + length is greater than the size of the array.
CS_BAD_DATEINPUTV	3012	The input date contains illegal characters.
CS_BAD_DATESEPARATORV	3044	The date components are separated by an invalid character.
CS_BAD_DAYV	3048	The day value is outside of the valid range. Acceptable values for months January through December are: 1..31, 1..28 (1..29 in a leap year), 1..31, 1..30, 1..31, 1..30, 1.31, 1..31, 1..30, 1..31, 1..30, and 1..31, respectively.
CS_BAD_DAYOFYEARV	3058	The day of year value is outside the valid range. Acceptable values are in the range 1 through 365 (1 through 366 for a leap year).
CS_BAD_FLAGV	3007	The flag specified is out of acceptable range.
CS_BAD_FRACDIGITSV	3016	The fractional digits value is either missing or out of range.
CS_BAD_HEXCODEV	3042	An invalid character was encountered in a hex value representing a symbol in a monetary or numeric template.

Table 9–2. Error Results for Internationalization

Result	Value	Specific Description
CS_BAD_HOURV	3051	The hour value is outside of the valid range for the 24-hour clock. Acceptable values are in the range 0 through 23.
CS_BAD_INPUTVALV	3035	The input value did not contain digits or an expected symbol was missing.
CS_BAD_LDATETEMPV	3018	The long date template is either missing or contains invalid information.
CS_BAD_LPPV	3027	The lines-per-page value is either missing or it is out of range.
CS_BAD_LTIMETEMPV	3021	The long time template is either missing or it contains invalid information.
CS_BAD_MAXDIGITSV	3015	The maximum digits value is either missing or out of range.
CS_BAD_MINDIGITSV	3032	The mindigits field in a "t" control character in a monetary or numeric template is out of range.
CS_BAD_MINUTEV	3053	The minute value is outside the valid range. Acceptable values are in the range 0 through 59.
CS_BAD_MONTEMPV	3023	The monetary template is either missing or it contains invalid information.
CS_BAD_MONTHV	3047	The month value is outside of the valid range. Acceptable values are in the range 1 through 12.
CS_BAD_NDATETEMPV	3020	The numeric date template is either missing or contains invalid information.
CS_BAD_NTIMETEMPV	3022	The numeric time template is either missing or contains invalid information.
CS_BAD_NUMTEMPV	3024	The numeric template is either missing or it contains invalid information.
CS_BAD_PRECISIONV	3038	The "PRECISION" parameter value is not in the range of 0 through 9.
CS_BAD_PSECONDV	3055	The partial second value contains invalid characters.

Table 9-2. Error Results for Internationalization

Result	Value	Specific Description
CS_BAD_SDATETEMPV	3019	The short date template is either missing or it contains invalid information.
CS_BAD_SECONDSV	3054	The second value is outside of the valid range. Acceptable values are in the range 0 through 59.
CS_BAD_TEMPCHARV	3011	An invalid control character was detected in the template.
CS_BAD_TEMPLENV	3030	An invalid template length value was encountered.
CS_BAD_TEXT_PARAMV	3008	The space for OSVs or total storage allocated in OUTPUT is not big enough for OSVs and/or PSVs.
CS_BADTIMEINPUTV	3013	The input time contains illegal characters.
CS_BAD_TIMESEPARATORV	3050	Time components are separated by an invalid character.
CS_BAD_12HOURV	3052	The hour value is outside of the valid range for the 12-hour clock. Acceptable values are in the range 1 through 12.
CS_BAD_TYPE_CODEV	3006	The type code specified is out of the acceptable range.
CS_BAD_YEARV	3046	A nonzero value is required for the year component.
CS_BAD_YEAR_YYV	3045	The year component exceeds 2 digits.
CS_CNV_EXISTS_ERRV	3014	An attempt was made to add a new convention with the name of an existing convention.
CS_CNVFILE_NOTPRESENTV	3037	A standard convention definition cannot be added, modified, or deleted.
CS_CNV_NOTAVAILV	3036	Specified convention does not exist and cannot be retrieved, modified or deleted.
CS_COMPLEX_TRAN_REQV	4004	Translate tables cannot be used to map from one multibyte CCS to another.

Table 9-2. Error Results for Internationalization

Result	Value	Specific Description
CS_CONVENTION_NOT_FOUNDV	2002	The data is not in the requested convention; it is in MYSELF.CONVENTION or the SYSTEM CONVENTION.
CS_DATA_NOT_FOUNDV	4002	The requested data was not found.
CS_DATAOKV	1	Your request has been processed.
CS_DEL_PERMANENT_CNV_ERRV	3040	The named convention is a standard convention and cannot be modified or deleted.
CS_FALSEV	0	Your request has been processed.
CS_FAULTV	1001	An unexpected fault occurred in CENTRALSUPPORT. Your request cannot be processed at this time.
CS_FIELD_TRUNCATEDV	2003	The date or time component was too long and was truncated.
CS_FILE_ACCESS_ERRORV	1000	An error occurred while accessing the SYSTEM/CCSFILE or the SYSTEM/CONVENTIONS file.
CS_INCOMPLETE_CHARV	2005	The source data ends with an incomplete multibyte character.
CS_INCOMPLETE_DATAV	2004	Only partial data is being returned. There was insufficient space in the output array.
CS_LANGUAGE_NOT_FOUNDV	2001	The data is not in the requested language. It is in MYSELF.LANGUAGE or the SYSTEM LANGUAGE or the first available LANGUAGE.
CS_MISSING_DATE_COMPONENTV	3059	The day of year value cannot be calculated because a date component (year, month, or day) is missing.
CS_MISSING_MONTH_COMPONENTV	3057	The month value is required but missing.
CS_MISSING_RBRACKETV	3033	A right bracket "]" is required to terminate a "t" control character symbol definition list.
CS_MISSING_TCCOLONV	3034	An expected colon ":" is missing from the "t" control character in a monetary or numeric template.

Table 9-2. Error Results for Internationalization

Result	Value	Specific Description
CS_MUTUAL_EXCLUSIVEV	3031	A mutually exclusive combination of control characters has been encountered in a monetary or numeric template.
CS_NO_ALTCURR_DELIMV	3043	The international currency notation is missing a required terminating delimiter.
CS_NO_CNVDNAMEV	3039	A required convention name was not provided.
CS_NO_DATEINPUTV	3049	An input date is required but missing.
CS_NO_HEXCODE_DELIMV	3041	A hexadecimal value representing a symbol in a monetary or numeric template is missing a required delimiter.
CS_NO_MSGNUM_FOUNDV	3005	The requested number was not found.
CS_NO_NAME_FOUNDV	3004	The requested name was not found.
CS_NO_NUM_FOUNDV	3003	The requested number was not found.
CS_NO_TIMEINPUTV	3056	An input time is required but missing.
CS_REQSYMBOLV	3029	A required symbol in either the monetary or the numeric template is missing.
CS_SOFTERRV	1002	A CENTRALSUPPORT software error was detected. Your request cannot be processed at this time.

Appendix A

Run-Time Format-Error Messages

Free-Field Input

The meanings of the format-error numbers pertaining to free-field input are given in the following table.

Number	Error Message
400	An error occurred on free-field input.
416	Evaluation of a list element caused an I/O action on the current file.
420	Input from the <core-to-core file part> required more records than allowed by the <core-to-core blocking part>.
442	The input data corresponding to a single-precision list element consisted of a hexadecimal string of more than 12 significant digits.
443	The input data contained a hexadecimal string containing nonhexadecimal characters.
444	The input data corresponding to a double-precision list element consisted of a hexadecimal string of more than 24 significant digits.
462	The next list element was a pointer, and the corresponding input data consisted of a quoted string that had been only partially assigned.
467	The input data contained a value greater than the maximum value allowed for the corresponding list element.
473	The input data contained a string with no trailing quotation mark character.
484	An expression was used as a list element on input.

Formatted Output

The meanings of the format-error numbers pertaining to formatted output are given in the following table.

Number	Error Message
100	An error occurred on formatted output.
102	The editing phrase letter was V, and the data specified by the list element did not produce an A, C, D, E, F, G, H, I, J, K, L, O, P, R, S, T, U, X, or Z in the appropriate character position.
103	The editing phrase was of the form rV, and the resulting specifier required a <field width>.
104	The editing phrase was of the form rV, and the resulting specifier required a <field width> and <decimal places>.
105	The editing phrase was of the form Fw.d, and d was less than zero.
106	The editing phrase used was Fw.d, and d was less than zero.
107	The editing phrase used was Ew.d or Dw.d, and d was less than zero.
109	The editing phrase used was Zw, and the corresponding list element was not of type INTEGER or BOOLEAN.
110	The type of the input data was not compatible with the type of the corresponding list element.
111	The editing phrase was of the form Zw.d. The phrase chosen to edit the output was Ew.d, but d was less than zero.
113	The editing phrase used was Ew.d or Dw.d, and w was less than or equal to d.
114	The value of a dynamic w or d part was greater than the maximum allowable integer (549,755,813,887).
116	Evaluation of a list element caused an I/O action on the current file (recursive I/O).
117	An attempt was made to write a number of characters greater than the record size.
120	Output to the <core-to-core file part> required more records than allowed by the <core-to-core blocking part>.
131	The value of a dynamic r part was greater than the maximum allowable real value ($4.31359146673 * 10^{**68}$).
132	The value of a dynamic w part was greater than the maximum allowable integer (549,755,813,887).
133	The value of a dynamic d part was greater than the maximum allowable integer (549,755,813,887).

Number	Error Message
163	The record size was not large enough to allow the free-field write specified.
165	An odd-tagged word was encountered by the source pointer when an attempt was made to write a number of characters less than or equal to the record size.

Formatted Input

The meanings of the format-error numbers pertaining to formatted input are given in the following table.

Number	Error Message
200	An error occurred on formatted input.
202	The editing phrase letter was V, and the data specified by the list element did not produce an A, C, D, E, F, G, H, I, J, K, L, O, P, R, S, T, U, X, or Z in the appropriate character position.
203	The editing phrase was of the form rV, and the resulting specifier required a <field width>.
204	The editing phrase was of the form rV, and the resulting specifier required a <field width> and <decimal places>.
205	The editing phrase was of the form rVw, and the resulting specifier required <decimal places>.
206	The editing phrase used was Fw.d, and d was less than zero.
207	The editing phrase used was Ew.d or Dw.d, and d was less than zero.
209	The editing phrase used was Zw, and the corresponding list element was not of type INTEGER or BOOLEAN.
210	The type of a list element was incompatible with the corresponding editing phrase.
213	The editing phrase used was Ew.d or Dw.d, and w was less than or equal to d.
214	The value of a dynamic w or d part was greater than the maximum allowable integer (549,755,813,887).
216	Evaluation of a list element caused an I/O action on the current file (recursive I/O).
217	An attempt was made to read a number of characters greater than the record size.
218	The editing phrase letter was H or K, but the input data contained nonblank, nonhexadecimal characters or nonblank, nonoctal characters, respectively.

Number	Error Message
220	Input from the <core-to-core file part> required more records than allowed by the <core-to-core blocking part>.
231	The value of a dynamic r part was greater than the maximum allowable real value (4.31359146673 * 10**68).
232	The value of a dynamic w part was greater than the maximum allowable integer (549,755,813,887).
233	The value of a dynamic d part was greater than the maximum allowable integer (549,755,813,887).
250	Input was attempted using a U editing phrase.
265	An odd-tagged word was encountered by the destination pointer when an attempt was made to read a number of characters less than or equal to the record size.
271	Input was attempted using a \$ or P format modifier.
281	The input data was invalid for an I editing phrase.
284	An expression was used as a list element on input.
285	The list element was of type REAL, but the corresponding input data contained a value greater than the maximum allowable real value (4.31359146673 * 10**68).
286	The list element was of type INTEGER or BOOLEAN, but the input data contained a value greater than the maximum allowable integer (549,755,813,887).
291	The input data corresponding to a numeric editing phrase contained a nondigit character in the exponent part following at least one valid digit.
292	The input data corresponding to a numeric editing phrase contained more than one sign in the exponent part.
293	The input data corresponding to a numeric editing phrase contained an invalid character after the exponent sign and before the exponent value.
294	The input data corresponding to a numeric editing phrase contained an invalid character after the decimal point.
295	The input data corresponding to a numeric editing phrase contained more than one sign in the mantissa.

Appendix B

Reserved Words

A reserved word in Extended ALGOL has the same syntax as an identifier. The reserved words are divided into three types.

A reserved word of type 1 can never be declared as an identifier; that is, it has a predefined meaning that cannot be changed. For example, because LIST is a type 1 reserved word, the following declaration is flagged with a syntax error:

```
ARRAY LIST[0:999]
```

A reserved word of type 2 can be redeclared as an identifier; it 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 in a program but it is not declared as a variable, then the error message that results is not the expected *UNDECLARED IDENTIFIER*. Instead, it might be *NO STATEMENT CAN START WITH THIS*.

A reserved word of type 3 is context-sensitive. It 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 STATUS are illustrated in the following example.

```
BEGIN
  TASK T;
  REAL STATUS;
  % IN THE NEXT STATEMENT, "STATUS" IS A REAL VARIABLE
  STATUS:= 4.5;
  % IN THE NEXT STATEMENT, "STATUS" IS A TASK ATTRIBUTE
  IF T.STATUS = VALUE(TERMINATED) THEN
    % IN THE NEXT STATEMENT, "STATUS" IS A REAL VARIABLE
    STATUS := 10.0;
END.
```

Reserved Words

Type 3 reserved words include the following:

- File attribute names
- Task attribute names
- Library attribute names
- Direct array attribute names
- Mnemonics for attribute values

All file attributes, direct array attributes, and mnemonics described in the *File Attributes Programming Reference Manual* are type 3 reserved words in ALGOL.

Because attribute mnemonics are type 3 reserved words in ALGOL, situations could arise with existing programs in which a conflict occurs on a new release between a local variable used with a file attribute and the newly introduced mnemonic. The compiler always treats the identifier as a mnemonic if a mnemonic is allowed in that context and it is a valid mnemonic. To make sure that a local variable is treated as such by the compiler, enclose it in parentheses to force its independent evaluation as an arithmetic expression. The effect of parentheses on the interpretation of a valid mnemonic in context is illustrated by the following example:

```
BEGIN
REAL
    INDEXED,
    INDXXED;
FILE F1 (FILEORGANIZATION =
    INDEXED);           % Mnemonic = 2
FILE F2 (FILEORGANIZATION =
    INDXXED);           % Item value = 0
FILE F3 (FILEORGANIZATION =
    (INDEXED));         % Item value = 0
FILE F4 (FILEORGANIZATION =
    (INDXXED));         % Item value = 0
INDEXED := 4;
INDXXED := 3;
F1 (KIND=DISK, FILEORGANIZATION =
    INDEXED);           % Mnemonic = 2
F2 (KIND=DISK, FILEORGANIZATION =
    INDXXED);           % Item value = 3
F3 (KIND=DISK, FILEORGANIZATION =
    (INDEXED));         % Item value = 4
F4 (KIND=DISK, FILEORGANIZATION =
    (INDXXED));         % Item value = 3
END.
```

Reserved Words List

Table B-1 provides an alphabetical list of reserved words for Extended ALGOL. Each reserved word is identified as type 1, 2, or 3.

Table B-1. Reserved Words List

Reserved Word	Type 1	Type 2	Type 3
ABORTTRANSACTION		X	
ABORT			X
ABS		X	
ACCEPT		X	
ACCEPTCLOSE			X
ACCEPTOPEN			X
ACTUALNAME			X
AFTER		X	
ALL		X	
ALPHA	X		
ALPHA7			X
ALPHA8			X
AND		X	
ANYFAULT			X
ANYTYPE			X
APPLYINSERT		X	
APPLYMODIFY		X	
ARCCOS		X	
ARCSIN		X	
ARCTAN		X	
ARCTAN2		X	
ARRAY	X		
ARRAYS			X
ARRAYSEARCH		X	
AS			X
ASCII		X	
ASCIITOEBCDIC			X
ASCIITOHX			X

Reserved Words

Table B-1. Reserved Words List

Reserved Word	Type 1	Type 2	Type 3
ASSOCIATEDDATA			X
ASSOCIATEDDATALENGTH			X
ATANH		X	
ATEND			X
ATTACH		X	
AVAILABLE		X	
AWAITOPEN		X	
BACKUPPREFIX			X
BASE			X
BEFORE		X	
BEGIN	X		
BLOCK			X
BOOLEAN	X		
BY		X	
BYFUNCTION			X
BYTITLE			X
CABS		X	
CALL		X	
CALLING			X
CANCEL		X	
CANCELTRPOINT		X	
CASE		X	
CAT		X	
CAUSE		X	
CAUSEANDRESET		X	
CCOS		X	
CEXP		X	
CHANGEFILE		X	
CHARGECODE			X
CLASS			X
CHECKPOINT		X	
CHECKSUM		X	

Table B-1. Reserved Words List

Reserved Word	Type 1	Type 2	Type 3
CLN		X	
CLOSE		X	
CLOSEDISPOSITION			X
CODE			X
COMMENT	X		
COMPILETIME		X	
COMPILETYPE			X
COMPLEX		X	
CONJUGATE		X	
CONNECTTIMELIMIT			X
CONNECTION		X	
CONTROL			X
CONTINUE	X		
COREESTIMATE			X
COS		X	
COSH		X	
COTAN		X	
CRUNCH			X
CSIN		X	
CSQRT		X	
CURRENT		X	
DABS		X	
DALPHA		X	
DAND		X	
DARCCOS		X	
DARCSIN		X	
DARCTAN		X	
DARCTAN2		X	
DBS			X
DCOS		X	
DCOSH		X	
DEALLOCATE		X	

Reserved Words

Table B-1. Reserved Words List

Reserved Word	Type 1	Type 2	Type 3
DECIMAL		X	
DECIMALPOINTISCOMMA			X
DECLAREDPRIORITY			X
DEFINE		X	
DELINKLIBRARY		X	
DELTA		X	
DEQV		X	
DERF		X	
DERFC		X	
DETACH		X	
DEXP		X	
DGAMMA		X	
DICTIONARY		X	
DIGITS		X	
DIMP		X	
DINTEGER		X	
DIRECT	X		
DISABLE		X	
DISCARD			X
DISK			X
DISKPACK			X
DISPLAY		X	
DIV		X	
DLGAMMA		X	
DLN		X	
DLOG		X	
DONTWAIT		X	
DMABS		X	
DMAVG		X	
DMAX		X	
DMCHR		X	
DMCONTAINS		X	

Table B-1. Reserved Words List

Reserved Word	Type 1	Type 2	Type 3
DMCOUNT		X	
DMEQUIV		X	
DMEXCEPTIONINFO		X	
DMEXCEPTIONMSG		X	
DMEXCLUDES		X	
DMEXITS		X	
DMEXT		X	
DMFUNCTION		X	
DMIN		X	
DMISA		X	
DMLENGTH		X	
DMMATCH		X	
DMMAX		X	
DMMIN		X	
DMNEXTEXCEPTION		X	
DMPOS		X	
DMPRED		X	
DMRECORD		X	
DMROUND		X	
DMRPT		X	
DMSQRT		X	
DMSUCC		X	
DMSUM		X	
DMTRUNC		X	
DMUPDATECOUNT		X	
DNABS		X	
DNORMALIZE		X	
DNOT		X	
DO	X		
DONTWAITFORFILE			X
DOR		X	
DOUBLE	X		

Reserved Words

Table B-1. Reserved Words List

Reserved Word	Type 1	Type 2	Type 3
DROP		X	
DSCALELEFT		X	
DSCALERIGHT		X	
DSCALERIGHTT		X	
DSIN		X	
DSINH		X	
DSQRT		X	
DTAN		X	
DTANH		X	
DUMP		X	
EBCDIC		X	
EBCDICTOASCII			X
EBCDICTOHEX			X
EGI		X	
ELAPSEDTIME			X
ELSE	X		
EMI		X	
EMPTY		X	
EMPTY4		X	
EMPTY7		X	
EMPTY8		X	
ENABLE		X	
END	X		
ENTIER		X	
ENTITY		X	
EPILOG	X		
EQL		X	
EQV		X	
ERF		X	
ERFC		X	
ESI		X	
EVENT	X		

Table B-1. Reserved Words List

Reserved Word	Type 1	Type 2	Type 3
EXCEPTIONEVENT			X
EXCEPTIONTASK			X
EXCHANGE		X	
EXCLUDE		X	
EXCEPTION	X		
EXITS		X	
EXP		X	
EXPONENTOVERFLOW			X
EXPONENTUNDERFLOW			X
EXPORT		X	
EXPORTLIBRARY		X	
EXTERNAL		X	
FALSE	X		
FAMILY			X
FILE	X		
FILECARDS			X
FILES			X
FILL		X	
FIRST		X	
FIRSTONE		X	
FIRSTWORD		X	
FIX		X	
FOR	X		
FORMAL			X
FORMAT	X		
FORWARD		X	
FUNCTIONNAME			X
FREE		X	
FREEZE		X	
GAMMA		X	
GEQ		X	
GO	X		

Reserved Words

Table B-1. Reserved Words List

Reserved Word	Type 1	Type 2	Type 3
GTR		X	
HAPPENED		X	
HEAD		X	
HEX		X	
HEXTOASCII			X
HEXTOEBCDIC			X
HISTORY			X
IF	X		
IMAG		X	
IMP		X	
IMPORTED		X	
IN		X	
INCLUDE		X	
INITIATOR			X
INPUTHEADER		X	
INTEGER	X		
INTEGEROVERFLOW			X
INTEGERT		X	
INTERLOCK		X	
INTERRUPT		X	
INTERRUPTIBLE			X
INTNAME			X
INTERFACENAME			X
INVALIDADDRESS			X
INVALIDINDEX			X
INVALIDOP			X
INVALIDPROGRAMWORD			X
INVERSE		X	
IS		X	
ISNT		X	
ISVALID		X	
ISYS		X	

Table B-1. Reserved Words List

Reserved Word	Type 1	Type 2	Type 3
JOBNUMBER			X
LABEL	X		
LAST			X
LB		X	
LENGTH		X	
LEQ		X	
LIBACCESS			X
LIBERATE		X	
LIBPARAMETER			X
LIBRARIES			X
LIBRARY		X	
LINE		X	
LINENUMBER		X	
LINKLIBRARY		X	
LIST	X		
LISTLOOKUP		X	
LN		X	
LNGAMMA		X	
LOCK		X	
LOCKED			X
LOCKSTATUS		X	
LOG		X	
LONG	X		
LOOP			X
LSS		X	
MASKSEARCH		X	
MAX		X	
MAXIOTIME			X
MAXLINES			X
MAXPROCTIME			X
MEMORYPARITY			X
MEMORYPROTECT			X

Reserved Words

Table B-1. Reserved Words List

Reserved Word	Type 1	Type 2	Type 3
MERGE		X	
MESSAGECOUNT		X	
MESSAGESEARCHER		X	
MIN		X	
MLSACCEPT		X	
MLSDISPLAY		X	
MOD		X	
MODIFY		X	
MONITOR		X	
MUX		X	
MYJOB		X	
MYSELF		X	
NABS		X	
NAME			X
NEQ		X	
NO		X	
NOCR		X	
NOLF		X	
NONE		X	
NORMALIZE		X	
NOT		X	
NULL		X	
NUMERIC		X	
OF		X	
OFFER			X
OFFSET		X	
ON		X	
ONES		X	
OPEN		X	
OPTION			X
OR		X	
ORDER		X	

Table B-1. Reserved Words List

Reserved Word	Type 1	Type 2	Type 3
ORDERLY			X
ORGUNIT			X
OUT			X
OUTPUTHEADER		X	
OUTPUTMESSAGE		X	
OWN	X		
PACK			X
PAGED			X
PARTICIPATE			X
PARTNER			X
PENDING			X
PERMANENT			X
PICTURE		X	
POINTER	X		
POTC		X	
POTH		X	
POTL		X	
PRIVATE		X	
PRIVATELIBRARIES			X
PROCEDURE	X		
PROCESS		X	
PROCESSID		X	
PROCESSTIME			X
PROCESSTIME			X
PROCURE		X	
PROGRAMDUMP		X	
PROGRAMMEDOPERATOR			X
PROLOG		X	
PUBLIC		X	
PURGE		X	
QUERY		X	
RANDOM		X	

Reserved Words

Table B-1. Reserved Words List

Reserved Word	Type 1	Type 2	Type 3
RB		X	
READ		X	
READLOCK		X	
READONLY			X
READWRITE			X
READYCL		X	
REAL	X		
RECEIVE		X	
RECORD		X	
REEL			X
REFERENCE	X		
REJECTCLOSE			X
REJECTOPEN			X
REMAININGCHARS		X	
REMOVEFILE		X	
REPEAT		X	
REPLACE		X	
RESET		X	
RESIZE		X	
RESPOND		X	
RESPONDTYPE			X
RESTART			X
RETAIN			X
RETRIEVE		X	
REWIND		X	
RUN		X	
SAVETRPOINT		X	
SCALELEFT		X	
SCALERIGHT		X	
SCALERIGHTF		X	
SCALERIGHTT		X	
SCAN		X	

Table B-1. Reserved Words List

Reserved Word	Type 1	Type 2	Type 3
SCANPARITY			X
SDIGITS		X	
SECONDDWORD		X	
SEEK		X	
SELECT		X	
SEND		X	
SET		X	
SETACTUALNAME		X	
SETTOCHILD		X	
SETTOPARENT		X	
SIBS			X
SIGN		X	
SIN		X	
SINGLE		X	
SINH		X	
SIZE		X	
SKIP		X	
SOME		X	
SORT		X	
SPACE		X	
SQRT		X	
STACKER		X	
STACKNO			X
STACKSIZE			X
STARTINSERT		X	
STARTMODIFY		X	
STARTTIME			X
STATION		X	
STATUS			X
STEP	X		
STOP		X	
STOPPOINT			X

Reserved Words

Table B-1. Reserved Words List

Reserved Word	Type 1	Type 2	Type 3
STRING		X	
STRINGPROTECT			X
STRING4		X	
STRING7		X	
STRING8		X	
STRUCTURE		X	
SUBFILE		X	
SUBROLE		X	
SWITCH	X		
SYNCHRONIZE			X
TADS			X
TAIL		X	
TAKE		X	
TAN		X	
TANH		X	
TARGETTIME			X
TASK	X		
TASKATTERR			X
TASKFILE			X
TASKVALUE			X
TERMINAL		X	
THEN	X		
THRU		X	
TIME		X	
TIMELIMIT		X	
TIMES		X	
TITLE			X
TO		X	
TRANSITIVE		X	
TRANSLATE		X	
TRANSLATETABLE	X		
TRUE	X		

Table B-1. Reserved Words List

Reserved Word	Type 1	Type 2	Type 3
TRUTHSET	X		
TRY		X	
TYPE		X	
UNLOCK		X	
UNREADYCL			X
UNTIL	X		
URGENT			X
USERCODE			X
USING		X	
VALUE	X		
WAIT		X	
WAITANDRESET		X	
WAITFORFILE			X
WHEN		X	
WHERE		X	
WHILE	X		
WITH		X	
WORDS		X	
WRITE		X	
ZERODIVIDE			X
ZIP	X		

Reserved Words

Appendix C

Data Representation

Field Notation

The notation $[m:n]$ is used in this manual to describe fields within data words. The 48 accessible bits of a data word are considered to be numbered, with the leftmost bit numbered as bit 47 and the rightmost bit numbered as bit 0. In the notation $[m:n]$, m denotes the number of the leftmost bit of the field being described, and n denotes the number of bits in the field. For example, the field indicated by the shaded area in Figure C-1 (bits 29 through 24) is described as $[29:6]$.

47	43	39	35	31	27	23	19	15	11	7	3
46	42	38	34	30	26	22	18	14	10	6	2
45	41	37	33	29	25	21	17	13	9	5	1
44	40	36	32	28	24	20	16	12	8	4	0

Figure C-1. Field Notation, [29:6]

All data words have a field associated with them that is called the tag of the word. The tag identifies the type of the data word; that is, whether the word is an operand, descriptor, and so on. The tag is not accessible to ALGOL programs.

Hexadecimal format is used extensively in this manual to indicate word contents. This format is particularly suited to describe the value of a data word, because each hexadecimal digit indicates the contents of a 4-bit field. Such fields can be visualized as the columns in Figure C-1.

Character Representation

Characters are stored in fields of one, two, three, four, six, or eight bits. In the table below, the Character Type column lists the valid ALGOL character types. The Field Size and Bits Used columns show the field size in bits and the number of bits within that field, respectively, that are used for storing each type of character. The Valid Constructs column shows the character-oriented ALGOL constructs that are used to manipulate each character type.

Character Type	Field Size	Bits Used	Valid Constructs
EBCDIC	8	8	Pointer, character array, string, string literal
ASCII	8	7	Pointer, character array, string, string literal
Hexadecimal (HEX)	4	4	Pointer, character array, string, string literal
Octal	3	3	String literal
Quaternary	2	2	String literal
Binary	1	1	String literal

Figures C-2 through C-4 illustrate how the various character types are stored within a data word.

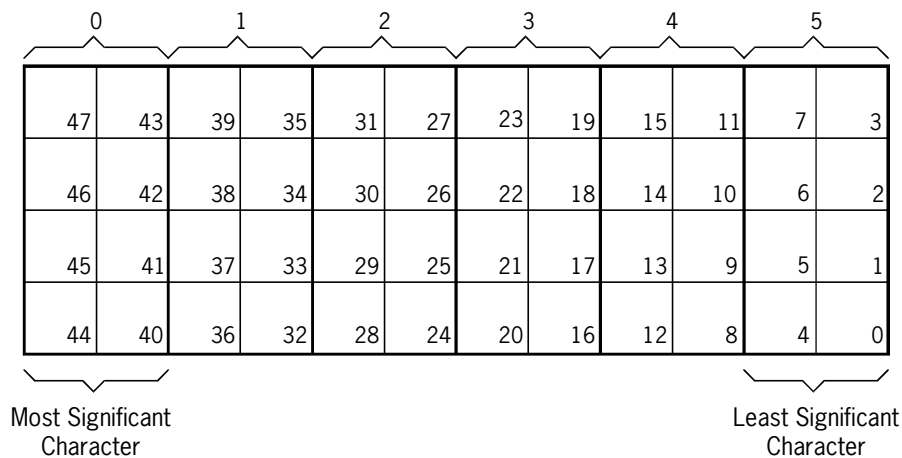


Figure C-2. EBCDIC Characters (8-Bit Fields)

Character Values and Graphics

The following is a list of the 256 EBCDIC values represented in binary, octal, decimal, and hexadecimal formats. The corresponding ASCII and EBCDIC graphics (as they appear when printed using an EBCDIC96 print train) are also shown. In the EBCDIC column, corresponding standard mnemonics appear for some nonprintable EBCDIC values.

Binary	Octal	Decimal	Hexadecimal	ASCII	EBCDIC
00000000	000	0	00	NUL	NUL
00000001	001	1	01	SOH	SOH
00000010	002	2	02	STX	STX
00000011	003	3	03	ETX	ETX
00000100	004	4	04	EOT	
00000101	005	5	05	ENQ	HT
00000110	006	6	06	ACK	
00000111	007	7	07	BEL	DEL
00001000	010	8	08	BS	
00001001	011	9	09	HT	
00001010	012	10	0A	LF	
00001011	013	11	0B	VT	VT
00001100	014	12	0C	FF	FF
00001101	015	13	0D	CR	CR
00001110	016	14	0E	SO	SO
00001111	017	15	0F	SI	SI
00010000	020	16	10	DLE	DLE
00010001	021	17	11	DC1	DC1
00010010	022	18	12	DC2	DC2
00010011	023	19	13	DC3	DC3
00010100	024	20	14	DC4	
00010101	025	21	15	NAK	NL
00010110	026	22	16	SYN	BS
00010111	027	23	17	ETB	
00011000	030	24	18	CAN	CAN
00011001	031	25	19	EM	EM
00011010	032	26	1A	SUB	
00011011	033	27	1B	ESC	

Data Representation

Binary	Octal	Decimal	Hexadecimal	ASCII	EBCDIC
00011100	034	28	1C	FS	FS
00011101	035	29	1D	GS	GS
00011110	036	30	1E	RS	RS
00011111	037	31	1F	US	US
00100000	040	32	20	SP	
00100001	041	33	21	!	
00100010	042	34	22	"	
00100011	043	35	23	#	
00100100	044	36	24	\$	
00100101	045	37	25	%	LF
00100110	046	38	26	&	ETB
00100111	047	39	27	'	ESC
00101000	050	40	28	(
00101001	051	41	29)	
00101010	052	42	2A	*	
00101011	053	43	2B	+	
00101100	054	44	2C	,	
00101101	055	45	2D	-	ENQ
00101110	056	46	2E	.	ACK
00101111	057	47	2F	/	BEL
00110000	060	48	30	0	
00110001	061	49	31	1	
00110010	062	50	32	2	SYN
00110011	063	51	33	3	
00110100	064	52	34	4	
00110101	065	53	35	5	
00110110	066	54	36	6	
00110111	067	55	37	7	EOT
00111000	070	56	38	8	
00111001	071	57	39	9	
00111010	072	58	3A	:	
00111011	073	59	3B	;	
00111100	074	60	3C	<	DC4

Data Representation

Binary	Octal	Decimal	Hexadecimal	ASCII	EBCDIC
00111101	075	61	3D	=	NAK
00111110	076	62	3E	>	
00111111	077	63	3F	?	SUB
01000000	100	64	40	@	SP (blank)
01000001	101	65	41	A	
01000010	102	66	42	B	
01000011	103	67	43	C	
01000100	104	68	44	D	
01000101	105	69	45	E	
01000110	106	70	46	F	
01000111	107	71	47	G	
01001000	110	72	48	H	
01001001	111	73	49	I	
01001010	112	74	4A	J	[
01001011	113	75	4B	K	.
01001100	114	76	4C	L	<
01001101	115	77	4D	M	(
01001110	116	78	4E	N	+
01001111	117	79	4F	O	!
01010000	120	80	50	P	&
01010001	121	81	51	Q	
01010010	122	82	52	R	
01010011	123	83	53	S	
01010100	124	84	54	T	
01010101	125	85	55	U	
01010110	126	86	56	V	
01010111	127	87	57	W	
01011000	130	88	58	X	
01011001	131	89	59	Y	
01011010	132	90	5A	Z]
01011011	133	91	5B	[\$
01011100	134	92	5C	\	*
01011101	135	93	5D])

Binary	Octal	Decimal	Hexadecimal	ASCII	EBCDIC
01011110	136	94	5E	^	;
01011111	137	95	5F	_	^
01100000	140	96	60	`	-
01100001	141	97	61	a	/
01100010	142	98	62	b	
01100011	143	99	63	c	
01100100	144	100	64	d	
01100101	145	101	65	e	
01100110	146	102	66	f	
01100111	147	103	67	g	
01101000	150	104	68	h	
01101001	151	105	69	i	
01101010	152	106	6A	j	l
01101011	153	107	6B	k	,
01101100	154	108	6C	l	%
01101101	155	109	6D	m	_
01101110	156	110	6E	n	>
01101111	157	111	6F	o	?
01110000	160	112	70	p	
01110001	161	113	71	q	
01110010	162	114	72	r	
01110011	163	115	73	s	
01110100	164	116	74	t	
01110101	165	117	75	u	
01110110	166	118	76	v	
01110111	167	119	77	w	
01111000	170	120	78	x	
01111001	171	121	79	y	`
01111010	172	122	7A	z	:
01111011	173	123	7B	{	#
01111100	174	124	7C		@
01111101	175	125	7D	}	'
01111110	176	126	7E	~	=

Data Representation

Binary	Octal	Decimal	Hexadecimal	ASCII	EBCDIC
01111111	177	127	7F	DEL	"
10000000	200	128	80		
10000001	201	129	81		a
10000010	202	130	82		b
10000011	203	131	83		c
10000100	204	132	84		d
10000101	205	133	85		e
10000110	206	134	86		f
10000111	207	135	87		g
10001000	210	136	88		h
10001001	211	137	89		i
10001010	212	138	8A		
10001011	213	139	8B		
10001100	214	140	8C		
10001101	215	141	8D		
10001110	216	142	8E		
10001111	217	143	8F		
10010000	220	144	90		
10010001	221	145	91		j
10010010	222	146	92		k
10010011	223	147	93		l
10010100	224	148	94		m
10010101	225	149	95		n
10010110	226	150	96		o
10010111	227	151	97		p
10011000	230	152	98		q
10011001	231	153	99		r
10011010	232	154	9A		
10011011	233	155	9B		
10011100	234	156	9C		
10011101	235	157	9D		
10011110	236	158	9E		
10011111	237	159	9F		

Binary	Octal	Decimal	Hexadecimal	ASCII	EBCDIC
10100000	240	160	A0		
10100001	241	161	A1		~
10100010	242	162	A2		s
10100011	243	163	A3		t
10100100	244	164	A4		u
10100101	245	165	A5		v
10100110	246	166	A6		w
10100111	247	167	A7		x
10101000	250	168	A8		y
10101001	251	169	A9		z
10101010	252	170	AA		
10101011	253	171	AB		
10101100	254	172	AC		
10101101	255	173	AD		
10101110	256	174	AE		
10101111	257	175	AF		
10110000	260	176	B0		
10110001	261	177	B1		
10110010	262	178	B2		
10110011	263	179	B3		
10110100	264	180	B4		
10110101	265	181	B5		
10110110	266	182	B6		
10110111	267	183	B7		
10111000	270	184	B8		
10111001	271	185	B9		
10111010	272	186	BA		
10111011	273	187	BB		
10111100	274	188	BC		
10111101	275	189	BD		
10111110	276	190	BE		
10111111	277	191	BF		
11000000	300	192	C0		{

Data Representation

Binary	Octal	Decimal	Hexadecimal	ASCII	EBCDIC
1100001	301	193	C1		A
1100010	302	194	C2		B
1100011	303	195	C3		C
1100100	304	196	C4		D
1100101	305	197	C5		E
1100110	306	198	C6		F
1100111	307	199	C7		G
1101000	310	200	C8		H
1101001	311	201	C9		I
1101010	312	202	CA		
1101011	313	203	CB		
1101100	314	204	CC		
1101101	315	205	CD		
1101110	316	206	CE		
1101111	317	207	CF		
11010000	320	208	D0		}
11010001	321	209	D1		J
11010010	322	210	D2		K
11010011	323	211	D3		L
11010100	324	212	D4		M
11010101	325	213	D5		N
11010110	326	214	D6		O
11010111	327	215	D7		P
11011000	330	216	D8		Q
11011001	331	217	D9		R
11011010	332	218	DA		
11011011	333	219	DB		
11011100	334	220	DC		
11011101	335	221	DD		
11011110	336	222	DE		
11011111	337	223	DF		
11100000	340	224	E0		\
11100001	341	225	E1		

Binary	Octal	Decimal	Hexadecimal	ASCII	EBCDIC
11100010	342	226	E2		S
11100011	343	227	E3		T
11100100	344	228	E4		U
11100101	345	229	E5		V
11100110	346	230	E6		W
11100111	347	231	E7		X
11101000	350	232	E8		Y
11101001	351	233	E9		Z
11101010	352	234	EA		
11101011	353	235	EB		
11101100	354	236	EC		
11101101	355	237	ED		
11101110	356	238	EE		
11101111	357	239	EF		
11110000	360	240	F0		0
11110001	361	241	F1		1
11110010	362	242	F2		2
11110011	363	243	F3		3
11110100	364	244	F4		4
11110101	365	245	F5		5
11110110	366	246	F6		6
11110111	367	247	F7		7
11111000	370	248	F8		8
11111001	371	249	F9		9
11111010	372	250	FA		
11111011	373	251	FB		
11111100	374	252	FC		
11111101	375	253	FD		
11111110	376	254	FE		
11111111	377	255	FF		

Default Character Type

The default character type is the character type assumed by pointers, string variables, string literals, and so on when a character type is not explicitly specified. The default character type is also used as the default value of the INTMODE file attribute. For more information on INTMODE, refer to the *File Attributes Programming Reference Manual*.

The ASCII compiler control option affects the default character type. For more information, refer to “ASCII Option” in Section 6 “Compiling Programs.” If the ASCII option is TRUE, the default character type is ASCII. If the ASCII option is not TRUE, the default character type is EBCDIC.

Free-field and formatted I/O generate EBCDIC data. If the default character type is EBCDIC or ASCII, EBCDIC data is generated.

The following example program demonstrates the effect of a default character type.

Example

```
BEGIN
% OPTION RECORD
  FILE F(KIND=DISK,NEWFILE=TRUE);
  ARRAY A[0:9];
  POINTER P;
  STRING S;          % DECLARATION OF S
  OPEN(F);
  P := POINTER(A);
  S := "ABC";        % STRING ASSIGNMENT
END.
```

If this example is compiled and executed as is, the default character type is EBCDIC. Thus, the INTMODE attribute of file F is EBCDIC, the character size of pointer P is eight bits, the string type of string S is EBCDIC, and, after the string assignment is executed, S contains the EBCDIC string *ABC*.

If *% OPTION RECORD* on the third line is replaced by the compiler control record *\$ SET ASCII*, and the program is compiled and executed, the default character type is ASCII. Thus, the INTMODE of file F is ASCII, the character size of pointer P is eight bits, the string type of string S is ASCII, and, after the string assignment is executed, S contains the ASCII string *ABC*.

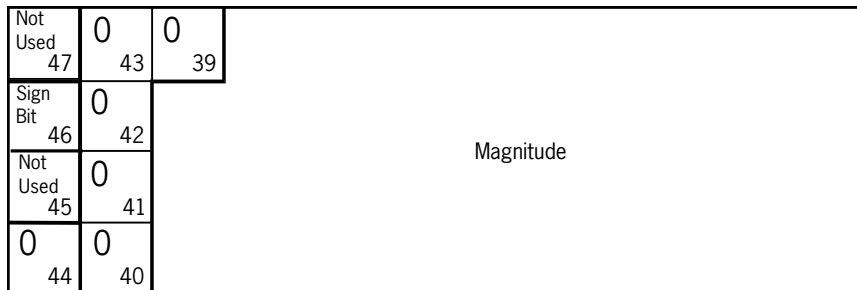
Signs of Numeric Fields

Certain operations in ALGOL require an indication of a numeric sign in character data. The sign of a numeric field is represented as follows:

- 8-bit characters The sign is in the zone field of the least significant character (field [7:4] of the character). A bit configuration of 1"1101" (4"D") indicates a negative number; any other bit configuration indicates a positive number.
- 6-bit characters The sign is in the zone field of the least significant character (field [5:2] of the character). A bit configuration of 1"10" indicates a negative number; any other bit configuration indicates a positive number.
- 4-bit digits The sign is carried as a separate digit, and it is the most significant digit of the field. A bit configuration of 1"1101" (4"D") indicates a negative number; any other bit configuration indicates a positive number.

Integer Operand

The internal structure of an integer operand is illustrated in Figure C–6.



Legend

Field	Use
Tag	0
[47:1]	Not used
[46:1]	Sign: 1 = negative 0 = positive
[45:1]	Not used
[44:6]	0
[38:39]	Magnitude

Figure C–6. Integer Operand

Integer values are represented internally in signed-magnitude notation. The sign of the value is denoted by bit 46 of the data word. A minus sign is denoted by a 1 in bit 46. The magnitude of the value is stored in field [38:39]. The maximum absolute value of an integer operand is $2^{39}-1$. There is an implied radix point to the right of bit zero.

For example, the internal representation of the integer 10 is

4"00000000000A"

The internal representation of –10 is

4"40000000000A"

The internal representation of 99,999,999,999 is

4"00174876E7FF"

The following internal representation represents the decimal value 549,755,813,887 (the maximum value an integer operand can contain). A larger value would have to be stored in a real operand or a double-precision operand.

4"007FFFFFFFFF"

Note that the internal format of an integer operand is the same as the internal format of a real operand with an exponent of zero.

Boolean Operand

The internal structure of a Boolean operand is illustrated in Figure C-7.

V ₄₇	V ₄₃	V ₃₉	V ₃₅	V ₃₁	V ₂₇	V ₂₃	V ₁₉	V ₁₅	V ₁₁	V ₇	V ₃
V ₄₆	V ₄₂	V ₃₈	V ₃₄	V ₃₀	V ₂₆	V ₂₂	V ₁₈	V ₁₄	V ₁₀	V ₆	V ₂
V ₄₅	V ₄₁	V ₃₇	V ₃₃	V ₂₉	V ₂₅	V ₂₁	V ₁₇	V ₁₃	V ₉	V ₅	V ₁
V ₄₄	V ₄₀	V ₃₆	V ₃₂	V ₂₈	V ₂₄	V ₂₀	V ₁₆	V ₁₂	V ₈	V ₄	V ₀

Legend

Field	Use
Tag	0
[47:47]	Any bit or field within this field can be referenced as a Boolean value using the <partial word part> or <concatenation> constructs.
[0:1]	The Boolean value of the operand as a whole.

Figure C-7. Boolean Operand

Boolean operations are performed on a bit-by-bit basis on all 48 bits of a Boolean operand. The one exception is the NOT operation performed on an arithmetic relation, where NOT is performed on the low-order bit (bit zero), but not necessarily on the other 47 bits. However, when a Boolean operand is referenced as a whole, only the low-order bit (bit zero) is significant (0 is FALSE, 1 is TRUE).

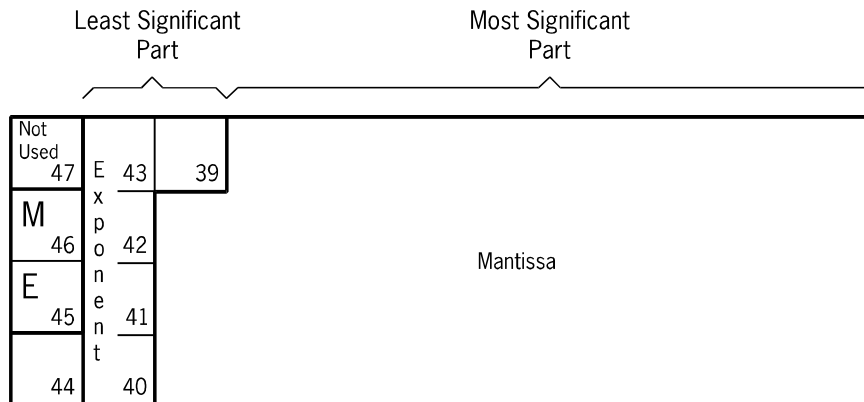
In Figure C-7, the shaded V's indicate that each bit of the entire 48-bit Boolean operand can be used to store an individual Boolean value. The individual bits can be referenced by using the <partial word part> and <concatenation> constructs.

Two-Word Operand

Double-precision and complex operands each require two words of storage.

Double-Precision Operand

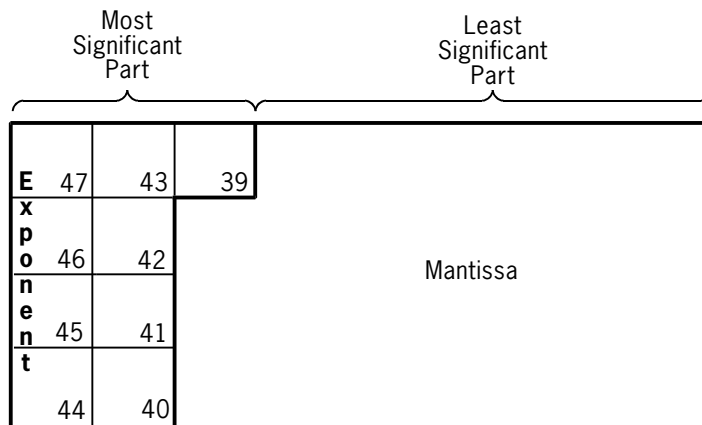
The internal structure of a double-precision operand is illustrated in Figures C-8 and C-9.



Legend

Field	Use
Tag	2
[47:1]	Not used
[46:1]	Sign of mantissa: 1 = negative 0 = positive
[45:1]	Sign of exponent: 1 = negative 0 = positive
[44:6]	Least significant portion of the exponent
[38:39]	Most significant portion of the mantissa

Figure C-8. First Word, Double-Precision Operand



LEGEND

Field	Use
Tag	2
[47:9]	Most significant portion of the exponent
[39:39]	Least significant portion of the mantissa

Figure C-9. Second Word, Double-Precision Operand

Double-precision values are represented internally in signed-magnitude, mantissa-and-exponent notation. The sign of the mantissa is contained in bit 46 of the first data word, and the sign of the exponent is contained in bit 45 of the first data word. A minus sign is denoted by a 1 in the appropriate sign bit.

The magnitude of the exponent of a double-precision operand is contained in a total of 15 bits. The most significant nine of these 15 bits are contained in field [47:9] of the second data word. The least significant six of these 15 bits are contained in field [44:6] of the first data word.

The magnitude of the mantissa of a double-precision operand is represented by a total of 78 bits. The most significant 39 bits are contained in field [38:39] of the first data word. The least significant 39 bits are contained in field [38:39] of the second data word. There is an implied radix point to the right of bit zero of the first data word; that is, between the most significant and least significant parts of the mantissa.

The value represented by a double-precision operand can be obtained by the following formula:

$$(MM + LM * 8^{(-13)}) * 8^{(ME * 2^{*6} + LE)}$$

where

- MM is the most significant part of the mantissa.
- LM is the least significant part of the mantissa.
- ME is the most significant part of the exponent.
- LE is the least significant part of the exponent.

Some operations return a normalized double-precision operand as their result. A normalized double-precision operand is a double-precision operand in which the leftmost octade (3-bit field) of the full 78-bit mantissa is nonzero. For example, the double-precision value 1@@0, in normalized form, is represented internally as

$$4^{26}1000000000, 4^{000000000000}$$

In this example, the mantissa is $1 * 8^{12}$ and the exponent is -12 .

Complex Operand

A complex operand requires two words of storage. The first word contains the real part of the complex value; the second word contains the imaginary part. The internal structure of both the real and imaginary parts of the complex value is identical to the internal structure of a real operand. For more information, refer to "Real Operand" earlier in this appendix.

Type Coercion of One-Word and Two-Word Operands

Because the internal representation of Boolean, complex, double precision, integer, and both normalized and nonnormalized real variables are all different, coercing a variable from one type to another must be done with care. Certain system software, such as intrinsics and operating system procedures, expect input parameters to be in a particular format, and if the expected bit pattern of a variable has been altered through type coercion, this software will not work correctly.

Type coercion can be effected through the following:

- Address equation between Boolean, integer, and real variables.
- Using a type transfer variable on the left side of an arithmetic or Boolean assignment.
- Using a type-transfer function such as the INTEGER function.
- Using a call-by-name parameter (for example, by passing a real variable to a call-by-name integer parameter).

The following is an example of address equation:

```
BEGIN
  BOOLEAN B;
  INTEGER I = B;
  REAL    R = B;
  B := TRUE;      % B = TRUE,    I = 1, R = 1.0
  B := NOT TRUE;  % B = FALSE,   I = Invalid or 0,
                  %              R = (-7.0064 E-46)
  I := 3;         % B = TRUE,    I = 3, R = 3.0
  R := 3;         % B = TRUE,    I = 3, R = 3.0
  R := 3.0;       % B = FALSE,   I = Invalid, 2 E37, or 3, R = 3.0
  I := 4/3;       % B = TRUE,    I = 1, R = 1.0
END.
```

Data Descriptors and Pointer

An unindexed data descriptor is the mechanism used on the enterprise server to represent the contents of an array. An indexed data descriptor is used to reference one element of a word array. A pointer references one character within a word or character array.

Appendix D

Understanding Railroad Diagrams

This appendix explains railroad diagrams, including the following concepts:

- Paths of a railroad diagram
- Constants and variables
- Constraints

The text describes the elements of the diagrams and provides examples.

Railroad Diagram Concepts

Railroad diagrams are diagrams that show you the standards for combining words and symbols into commands and statements. These diagrams consist of a series of paths that show the allowable structures of the command or statement.

Paths

Paths show the order in which the command or statement is constructed and are represented by horizontal and vertical lines. Many commands and statements have a number of options so the railroad diagram has a number of different paths you can take.

The following example has three paths:



The three paths in the previous example show the following three possible commands:

- REMOVE
- REMOVE SOURCE
- REMOVE OBJECT

A railroad diagram is as complex as a command or statement requires. Regardless of the level of complexity, all railroad diagrams are visual representations of commands and statements.

Understanding Railroad Diagrams

Railroad diagrams are intended to show

- Mandatory items
- User-selected items
- Order in which the items must appear
- Number of times an item can be repeated
- Necessary punctuation

Follow the railroad diagrams to understand the correct syntax for commands and statements. The diagrams serve as quick references to the commands and statements.

The following table introduces the elements of a railroad diagram:

Table D-1. Elements of a Railroad Diagram

The diagram element . . .	Indicates an item that . . .
Constant	Must be entered in full or as a specific abbreviation
Variable	Represents data
Constraint	Controls progression through the diagram path

Constants and Variables

A constant is an item that must be entered as it appears in the diagram, either in full or as an allowable abbreviation. If part of a constant appears in boldface, you can abbreviate the constant by

- Entering only the boldfaced letters
- Entering the boldfaced letters plus any of the remaining letters

If no part of the constant appears in boldface, the constant cannot be abbreviated.

Constants 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.

In railroad diagrams, variables are enclosed in angle brackets.

In the following example, BEGIN and END are constants, whereas <statement list> is a variable. The constant BEGIN can be abbreviated, since part of it appears in boldface.

— **BEGIN** —<statement list>— END —————|

Valid abbreviations for BEGIN are

- BE
- BEG
- BEGI

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

- Is used when the railroad diagram is too long to fit on one line and must continue on the next
- Appears at the end of the first line, and again at the beginning of the next line

— SCALERIGHT — (—<arithmetic expression>— , —————→
→<arithmetic expression>—) —————|

Required Item

A required item can be

- A constant
- A variable
- Punctuation

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.

A required item appears on a horizontal line as a single entry or with other items. Required items can also exist on horizontal lines within alternate paths, or nested (lower-level) diagrams.

In the following example, the word EVENT is a required constant and <identifier> is a required variable:

— EVENT —<identifier>—————|

User-Selected Item

A user-selected item can be

- A constant
- A variable
- Punctuation

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) above the other items, none of the choices are required.

In the following railroad diagram, either the plus sign (+) or the 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 a 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.

Some loops include a return character. A return character is a character—often a comma (,) or semicolon (;)—that is required before each repetition of a loop. If no return character is included, the items must be separated by one or more spaces.

— [← ;] —<field value>—————|

Bridge

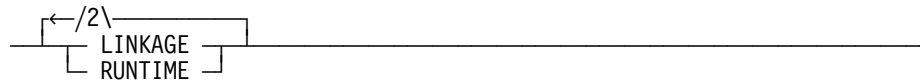
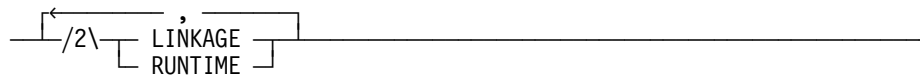
A loop can also include a bridge. A bridge is an integer enclosed in sloping lines (/ \) that

- Shows the maximum number of times the loop can be repeated
- Indicates the number of times you can cross that point in the diagram

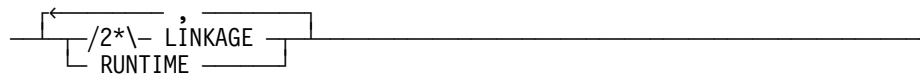
The bridge can precede both the contents of the loop and the return character (if any) on the upper line of the loop.

Not all loops have bridges. Those that do not can be repeated any number of times until all valid entries have been used.

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 cross that point in the diagram at least once. The maximum number of times that you can cross that point is indicated by the number in the bridge.

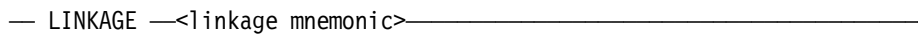


In the previous bridge example, you must enter LINKAGE at least once but no more than twice, and you can enter RUNTIME any number of times.

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; others have several alternate paths that provide choices in the commands or statements.

The following railroad diagram indicates only one path that requires the constant LINKAGE and the variable <linkage mnemonic>:



Alternate paths are provided by

- Loops
- User-selected items
- A combination of loops and user-selected items

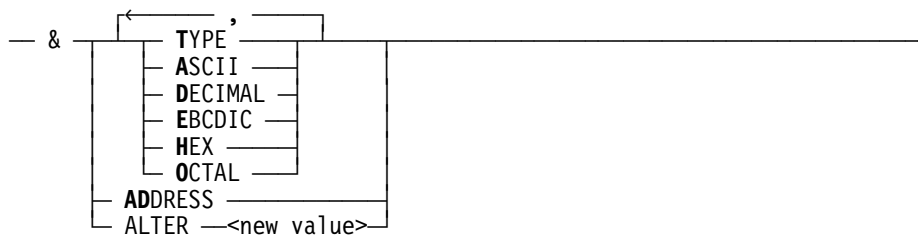
More complex railroad diagrams can consist of many alternate paths, or nested (lower-level) diagrams, that show a further level of detail.

For example, the following railroad diagram consists of a top path and two alternate paths. The top path includes

- An ampersand (&)
- Constants that are user-selected items

These constants are within a loop that can be repeated any number of times until all options have been selected.

The first alternative path requires the ampersand and the required constant ADDRESS. The second alternative path requires the ampersand followed by the required constant ALTER and the required variable <new value>.



Railroad Diagram Examples with Sample Input

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	Explanation
LOCK (FILE4)	LOCK is a constant and cannot be altered. Because no part of the word appears in boldface, the entire word must be entered. The parentheses are required punctuation, and FILE4 is a sample file identifier.

Example 2

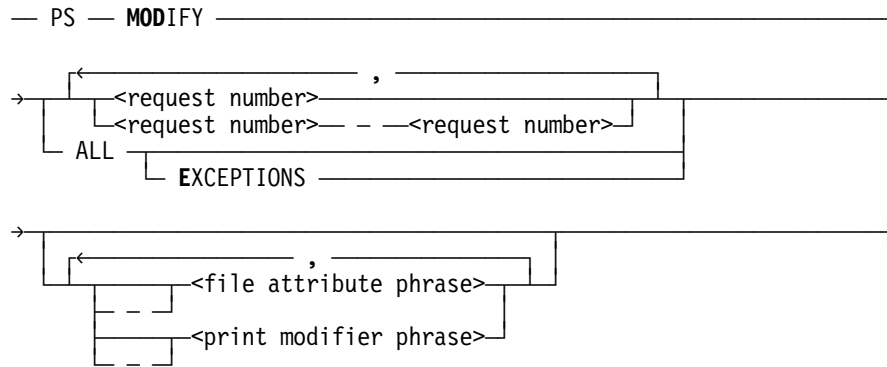
<open statement>

— OPEN — [INQUIRY] [UPDATE] — <database name> —————|

Sample Input	Explanation
OPEN DATABASE1	The constant OPEN is followed by the variable DATABASE1, which is a database name. The railroad diagram shows two user-selected items, INQUIRY and UPDATE. However, because an empty path (solid line) is included, these entries are not required.
OPEN INQUIRY DATABASE1	The constant OPEN is followed by the user-selected constant INQUIRY and the variable DATABASE1.
OPEN UPDATE DATABASE1	The constant OPEN is followed by the user-selected constant UPDATE and the variable DATABASE1.

Example 5

<PS MODIFY command>



Sample Input	Explanation
PS MODIFY 11159	The constants PS and MODIFY are followed by the variable 11159, which is a request number.
PS MODIFY 11159,11160,11163	Because the diagram contains a loop, the variable 11159 can be followed by a comma, the variable 11160, another comma, and the final variable 11163.
PS MOD 11159-11161 DESTINATION = "LP7"	The constants PS and MODIFY are followed by the user-selected variables 11159-11161, which are request numbers, and the user-selected variable DESTINATION = "LP7", which is a file attribute phrase. Note that the constant MODIFY has been abbreviated to its minimum allowable form.
PS MOD ALL EXCEPTIONS	The constants PS and MODIFY are followed by the user-selected constants ALL and EXCEPTIONS.

Appendix E

Related Product Information

The following documents provide information that is directly related to the primary subject of this manual.

MCP/AS ALGOL and MCP Interfaces to POSIX Features Programming Reference Manual (7011 8351)

This manual describes how to access POSIX features in programs that are not written in C language. Two basic interface methods are described: ALGOL include file functions and client library procedures that import objects from the MCPSUPPORT library. The ALGOL include file defines a subset of POSIX functions that can be used only in ALGOL programs. Client library procedures can be used by any programming language that supports the use of libraries. This manual is written for systems programmers.

MCP/AS ALGOL Compiler Messages Support Reference Manual (8600 0031)

This manual lists and describes in alphabetical order the messages that might be produced at compile time by the ALGOL compiler. This manual is written for application programmers and system analysts experienced in developing, maintaining, and reading ALGOL programs.

MCP/AS ALGOL Programming Reference Manual, Volume 2: Product Interfaces (8600 0734)

This manual describes the extensions to the Extended ALGOL language that allow application programs to use the Advanced Data Dictionary System (ADDS), the Communications Management System (COMS), the Data Management System II (DMSII), the Screen Design Facility Plus (SDF Plus), or the Semantic Information Manager (SIM). This manual is written for programmers who are familiar with Extended ALGOL programming language concepts and terms.

MCP/AS ALGOL Test and Debug System (TADS) Programming Guide (8600 2144)

This guide describes the features of ALGOL TADS, an interactive tool used for testing and debugging ALGOL programs and libraries. ALGOL TADS allows the programmer to monitor and control the execution of programs under test and examine the data at any given point during program execution. This guide is written for programmers who are familiar with ALGOL programming language concepts and terms.

MCP/AS Binder Programming Reference Manual (8600 0304)

This manual describes the functions and applications of the Binder, an efficiency tool that reduces the need to recompile an entire program when only a portion of the program has been modified. This manual is written for programmers who are familiar with programming language concepts and terms.

MCP/AS CANDE Operations Reference Manual (8600 1500)

This manual describes how CANDE operates to allow generalized file preparation and updating in an interactive, terminal-oriented environment. This manual is written for a wide range of computer users who work with text and program files.

MCP/AS Menu-Assisted Resource Control (MARC) Operations Guide (8600 0403)

This guide provides an overview of MARC, a description of the menu structure, and information on how to use help text, commands, security features, and Unisys e-@ction Transaction Server windows from MARC. The guide also explains how to run programs from MARC, how to customize MARC to meet user needs, and how to use MARC in a multinational environment. This guide is written for a wide audience, ranging from experienced system administrators to end users with no previous knowledge of MARC.

MCP/AS Message Translation Utility (MSGTRANS) Operations Guide (8600 0106)

This guide describes how to use the Message Translation Utility (MSGTRANS) to translate compiled program messages from any natural language to any other natural language. It provides complete instructions for running and using the screen and batch interfaces of MSGTRANS. This guide is written for programmers and translators who create and translate program messages in a Multilingual System (MLS) environment.

Unisys e-@ction ClearPath Enterprise Servers Distributed Systems Service Operations Guide (8807 6385)

This guide explains how to use most of the TCP/IP services provided by the DSS products. These products include File Transfer Protocol (FTP), Telnet, and Domain Name Services (DNS). The guide also explains how to use the TCP/IP DSS debugging tools. This guide is written for system planners, system programmers, application programmers, and computer operators who use TCP/IP DSSs. For information on the Simple Mail Transfer Protocol (SMTP) used by the Mail System, see the Mail System Operations Guide and the Mail System Administration Guide.

Unisys e-@ction ClearPath Enterprise Servers File Attributes Programming Reference Manual (8600 0064)

This manual contains information about each file attribute and each direct I/O buffer attribute. The manual is written for programmers and operations personnel who need to understand the functionality of a given attribute. The *I/O Subsystem Programming Guide* is a companion manual.

Unisys e-@ction ClearPath Enterprise Servers I/O Subsystem Programming Guide (8600 0056)

This guide contains information about how to program for various types of peripheral files and how to program for interprocess communication, using port files. This guide is written for programmers who need to understand how to describe the characteristics of a file in a program. The *File Attributes Programming Reference Manual* is a companion manual.

Unisys e-@ction ClearPath Enterprise Servers MultiLingual System (MLS) Administration, Operations, and Programming Guide (8600 0288)

This guide describes how to use the MLS environment, which encompasses many Unisys products. The MLS environment includes a collection of operating system features, productivity tools, utilities, and compiler extensions. The guide explains how these products are used to create application systems tailored to meet the needs of users in a multilingual or multicultural business environment. It explains, for example, the procedures for translating system and application output messages, help text, and user interface screens from one natural language to one or more other languages; for instance, from English to French and Spanish. This guide is written for international vendors, branch systems personnel, system managers, programmers, and customers who wish to create customized application systems.

Unisys e-@ction ClearPath Enterprise Servers System Commands Operations Reference Manual (8600 0395)

This manual gives a complete description of the system commands used to control system resources and work flow. This manual is written for systems operators and administrators.

Unisys e-@ction ClearPath Enterprise Servers System Software Utilities Operations Reference Manual (8600 0460)

This manual provides information on the system utilities BARS, CARDLINE, COMPARE, DCAUDITOR, DCSTATUS, DUMPALL, DUMPANALYZER, FILECOPY, FILEDATA, HARDCOPY, INTERACTIVEXREF, ISTUTILITY, LOGANALYZER, LOGGER, PATCH, PRINTCOPY, RLTABLEGEN, SORT, XREFANALYZER, and the V Series conversion utilities. It also provides information on KEYEDIO support, Peripheral Test Driver (PTD), and mathematical functions. This manual is written for applications programmers, system support personnel, and operators.

Unisys e-@ction ClearPath Enterprise Servers Task Attributes Programming Reference Manual (8600 0502)

This manual describes all the available task attributes. It also gives examples of statements for reading and assigning task attributes in various programming languages. The *Task Management Programming Guide* is a companion manual.

Unisys e-@ction ClearPath Enterprise Servers Task Management Programming Guide (8600 0494)

This guide explains how to initiate, monitor, and control processes on an enterprise server. It describes process structures and process family relationships, introduces the uses of many task attributes, and gives an overview of interprocess communication techniques. The *Task Attributes Programming Reference Manual* is a companion manual.

Unisys e-@ction ClearPath Enterprise Servers Work Flow Language Programming Reference Manual (8600 1047)

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

Index

A

- <abs function>, 5-43
- <abstract value mnemonic>, 3-58
- <accept statement>, 4-2
- ACCEPTCLOSE, 4-176
- ACCEPTOPEN, 4-176
- accidental entry (thunk), 3-134
- <action labels or finished event>, 4-129
 - semantics, 4-129
- <actual parameter part>, 4-107
- <actual parameter>, 4-108
- <actual text part>, 3-27
- <actual text>, 3-28
- ACTUALNAME option of LIBRARY
 - declaration, 3-139
- addition, 5-3
- address equation, 3-14
- ALPHA, 3-183
- ALPHA declaration, 3-2
- ALPHA7, 3-183
- ALPHA8, 3-183
- AND, 5-14
- ANYFAULT fault (ON statement example), 4-100
- ANYTYPE parameter, 3-137
- <arccos function>, 5-43
- <arcsin function>, 5-43
- <arctan function>, 5-43
- <arithmetic assignment>, 4-5
- <arithmetic attribute specification>, 3-57
- <arithmetic attribute>, 4-7
- <arithmetic case expression>, 5-7
- <arithmetic concatenation expression>, 5-8
- <arithmetic direct array attribute>, 4-7
- <arithmetic expression>, 5-2
 - in <instuff parameter>, 5-102
 - precision of, 5-3
- <arithmetic file attribute>, 4-7
- <arithmetic function designator>, 5-29
- <arithmetic intrinsic name>, 5-29
- <arithmetic library attribute specification>, 3-100
- <arithmetic operand>, 5-7
- <arithmetic operator>, 5-3
- arithmetic operators, 5-3
 - ** , 5-4
 - +, -, *, /, 5-3
 - DIV, 5-3
 - MOD, 5-3
 - MUX, 5-3
 - precedence of, 5-4
 - TIMES, 5-3
- arithmetic primaries, 5-7
 - strings used as, 5-7
- <arithmetic primary>, 5-7
- <arithmetic relation>, 5-18
- <arithmetic table membership>, 5-21
- <arithmetic task attribute>, 4-7
- <arithmetic type transfer variable>, 4-6
- <arithmetic update assignment>, 4-8
- <arithmetic variable>, 4-5
- arithmetic-valued attributes
 - assigning values
 - arithmetic assignment target
 - <arithmetic attribute>, 4-7
 - FILE declaration, 3-57
 - multiple attribute assignment statement, 4-93
 - interrogating
 - arithmetic operand <arithmetic attribute>, 5-7
 - VALUE function, 5-103
- <arithmetic-valued direct array attribute name>, 4-7
- <arithmetic-valued file attribute name>, 3-58
- <arithmetic-valued library attribute name>, 3-100
- <arithmetic-valued task attribute name>, 4-7
- array
 - allocation, 3-4
 - array class, 3-5
 - array reference, 3-12
 - array reference assignment, 4-9
 - array row, 3-9
 - array row equivalence, 3-8
 - array row read, 4-128
 - array row write, 4-220

- bits per element, 3-5
 - bound pair list, 3-6
 - character array, 3-5
 - declarations
 - ARRAY, 3-3
 - ARRAY REFERENCE, 3-12
 - DIRECT ARRAY, 3-33
 - STRING ARRAY, 3-160
 - VALUE ARRAY, 3-186
 - default type, 3-5
 - descriptor, 3-6
 - dimensions, 3-7
 - direct array, 3-33
 - dumping declared arrays, 4-116
 - element width, 3-5
 - equivalent, 3-8
 - functions
 - ARRAYSEARCH, 5-44
 - CHECKSUM, 5-47
 - LISTLOOKUP, 5-73
 - MASKSEARCH, 5-75
 - SIZE, 5-88
 - in sort procedures, 4-196
 - long (unpaged), 3-3
 - lower and upper bounds, 3-6
 - NOSTACKARRAYS compiler control
 - option, 6-38
 - original, 3-7
 - OWN, 3-4
 - paged (segmented), 3-3
 - referenced (touched), 3-4
 - referred, 3-7
 - resizing referenced unpaged
 - (unsegmented) arrays, 4-170
 - row selector, 3-10
 - statements
 - DEALLOCATE, 4-48
 - FILL, 4-58
 - RESIZE, 4-169
 - string array, 3-160
 - subarray selector, 4-11
 - unpaged (unsegmented), 3-3
 - value array, 3-186
 - word array, 3-5
 - ZIP WITH array, 4-224
- <array class>, 3-5
 - <array declaration>, 3-3
 - <array designator>, 4-10
 - for SIZE function, 5-88
 - <array identifier>, 3-4
 - <array name>, 3-9
 - array parameters, 3-136, 3-141
 - <array reference assignment>, 4-9
 - <array reference declaration>, 3-12
 - <array reference identifier>, 3-12
 - <array reference variable>, 4-10
 - <array row equivalence>, 3-8
 - array row read, 4-128
 - <array row resize parameters>, 4-169
 - array row write, 4-220
 - <array row>, 3-9
 - in <real array row>, 5-102
 - in LIST declaration, 3-105
 - <array specification>, 3-136
 - <array type>, 3-136
 - arrays of strings, 3-160
 - <arraysearch function>, 5-44
 - <ASCII character>, 2-14
 - <ASCII option>, 6-16
 - <ASCII string constant>, 5-35
 - ASSIGNMENT statement, 4-4
 - arithmetic assignment, 4-5
 - array reference assignment, 4-9
 - Boolean assignment, 4-12
 - complex assignment, 4-15
 - mnemonic attribute assignment, 4-16
 - multiple attribute assignment
 - statement, 4-93
 - pointer assignment, 4-17
 - string assignment, 4-20
 - SWAP statement, 4-198
 - task assignment, 4-22
 - <assignment statement>, 4-4
 - <associateddata option>, 4-44, 4-101
 - <atanh function>, 5-45
 - ATEND option, 4-101
 - <attach statement>, 4-23
 - attribute handling
 - assigning arithmetic-valued attributes
 - arithmetic assignment target
 - <arithmetic attribute>, 4-7
 - FILE declaration, 3-57
 - multiple attribute assignment
 - statement, 4-93
 - assigning Boolean-valued attributes
 - Boolean assignment target <Boolean attribute>, 4-13
 - FILE declaration, 3-57
 - multiple attribute assignment
 - statement, 4-93
 - assigning pointer-valued attributes
 - FILE declaration, 3-57
 - multiple attribute assignment
 - statement, 4-93
 - replace family-change statement, 4-163

- replace pointer-valued attribute statement, 4-165
- assigning string-valued attributes
 - LIBRARY declaration, 3-99
 - string assignment target <string-valued library attribute>, 4-20
- assigning task-valued attributes
 - task assignment, 4-22
- assigning the LIBACCESS library attribute
 - LIBRARY declaration, 3-99
 - mnemonic attribute assignment, 4-16
- assigning translate-table-valued attributes
 - FILE declaration, 3-57
 - multiple attribute assignment statement, 4-93
- interrogating arithmetic-valued attributes
 - arithmetic operand <arithmetic attribute>, 5-7
- interrogating Boolean-valued attributes
 - Boolean operand <Boolean attribute>, 5-17
- interrogating event-valued attributes
 - <event designator>, 3-46
- interrogating pointer-valued attributes
 - REPLACE statement source part <pointer-valued attribute>, 4-138, 4-160
- interrogating string-valued attributes
 - stringprimary <string-valued library attribute>, 5-34
- interrogating task-valued attributes
 - <task designator>, 3-176
- VALUE function, 5-103
- <attribute parameter list>, 4-7
- <attribute parameter specification>, 4-7
- <attribute specifications>, 3-57
- <autobind option>, 6-16
- AVAILABLE control option, 4-24
- <available function>, 5-45
- AVAILABLE option, 4-101
- AVAILATEND option, 4-101
- <awaitopen control option>
 - AVAILABLE, 4-24
 - DONTWAIT, 4-24
 - WAIT, 4-24
- <awaitopen options>, 4-24
- <awaitopen statement>, 4-24

B

- bad GO TO, 4-70
- <basic symbol>, 2-2
- BDMSALGOL
 - compile-time facility in, 7-1
- <beginsegment option>, 6-18
- <binary code>, 2-12
- binary read, 4-127
- <binary string>, 2-12
- binary write, 4-219
- <bind option>, 6-19
- <binder command>, 6-12
- <binder option>, 6-19
- <binder_match option>, 6-19
- binding
 - AUTOBIND option, 6-16
 - BIND option, 6-19
 - BINDER option, 6-19
 - BINDER_MATCH option, 6-19
 - DONTBIND option, 6-21
 - DUMPINFO option, 6-21
 - EXTERNAL option, 6-22
 - HOST option, 6-23
 - INITIALIZE option, 6-26
 - INTRINSICS option, 6-27
 - LARGE_LINEINFO option, 6-27
 - LEVEL option, 6-28
 - LIBRARY option, 6-29
 - LOADINFO option, 6-32
 - PURGE option, 6-41
 - STOP option, 6-47
 - USE option, 6-50
- bit manipulation, 5-8
 - concatenation expression, 5-8
 - partial word expression, 5-12
- <bit manipulation expression>, 5-8
- <block>, 1-2
- blocking, 4-71
- <Boolean assignment>, 4-12
- <Boolean attribute specification>, 3-58
- <Boolean attribute>, 4-13
- <Boolean case expression>, 5-17
- <Boolean concatenation expression>, 5-9
- Boolean data type
 - Boolean array declaration, 3-3
 - Boolean array reference declaration, 3-12
 - Boolean assignment, 4-12
 - BOOLEAN declaration, 3-14
 - Boolean expression, 5-13
 - Boolean functions
 - ACCEPT statement, 4-2

- AVAILABLE function, 5-45
 - BOOLEAN function, 5-45
 - CHANGEFILE statement, 4-33
 - CHECKPOINT statement, 4-36
 - FIX statement, 4-60
 - FREE statement, 4-67
 - HAPPENED function, 5-64
 - READ statement, 4-121
 - READLOCK function, 5-80
 - REMOVEFILE statement, 4-135
 - SEEK statement, 4-186
 - SPACE statement, 4-197
 - WAIT statement, 4-206
 - WRITE statement, 4-215
 - Boolean operand internal structure, C-16
 - Boolean procedure declaration, 3-132
 - Boolean value array declaration, 3-186
 - direct Boolean array declaration, 3-33
 - functions with Boolean parameters
 - READLOCK function, 5-80
 - functions with Boolean parameters REAL function, 5-82
 - intrinsic functions returning values of type BOOLEAN, 5-42
 - width of Boolean array elements, 3-5
 - <Boolean declaration>, 3-14
 - <Boolean direct array attribute>, 4-13
 - <Boolean expression>, 5-13
 - <Boolean file attribute>, 4-13
 - <Boolean function designator>, 5-29
 - <Boolean function>, 5-45
 - <Boolean identifier>, 3-14
 - <Boolean intrinsic name>, 5-29
 - alphabetical listing of, 5-42
 - <Boolean library attribute specification>, 3-101
 - Boolean operand internal structure, C-16
 - <Boolean operand>, 5-17
 - <Boolean operator>, 5-13
 - Boolean operators, 5-14
 - precedence of, 5-16
 - <Boolean option>, 6-12
 - Boolean primaries, 5-17
 - <Boolean primary>, 5-17
 - <Boolean task attribute>, 4-13
 - <Boolean type transfer variable>, 4-12
 - <Boolean update assignment>, 4-14
 - <Boolean value>, 5-17
 - <Boolean variable>, 4-12
 - for <fault number>, 4-97
 - Boolean-valued attributes
 - assigning values
 - Boolean assignment target <Boolean attribute>, 4-13
 - FILE declaration, 3-57
 - multiple attribute assignment statement, 4-93
 - interrogating
 - Boolean operand <Boolean attribute>, 5-17
 - VALUE function, 5-103
 - <Boolean-valued direct array attribute name>, 4-13
 - <Boolean-valued file attribute name>, 3-58
 - <Boolean-valued library attribute name>, 3-101
 - <Boolean-valued task attribute name>, 4-13
 - <bound pair list>, 3-6
 - bound pair lists in array declarations, 3-6
 - <bound pair>, 3-6
 - <bounds part>, 3-38
 - <bracket>, 2-2
 - buffering, 4-71
 - by-calling procedure, 3-138
 - BYFUNCTION library access value, 8-9
 - BYINITIATOR library access value, 8-9
 - BYTITLE library access value, 8-9
- ## C
- <cabs function>, 5-46
 - <call statement>, 4-26
 - call-by-reference parameters, 3-134
 - call-by-value parameters, 3-133
 - calling procedures with parameters, 4-108
 - <cancel statement>, 4-28
 - CARD file, 6-3
 - <case body>, 4-29
 - <case head>, 4-29
 - <case statement>, 4-29
 - CAT, 5-36
 - <cause statement>, 4-31
 - <causeandreset statement>, 4-32
 - <ccos function>, 5-46
 - CCS, 9-4
 - CCSINFO procedure, 9-26
 - CCSTOCCS_TRANS_TABLE procedure, 9-31
 - CCSTOCCS_TRANS_TABLE_ALT
 - procedure, 9-33
 - CCSTOCCS_TRANS_TEXT procedure, 9-35
 - CCSTOCCS_TRANS_TEXT_COMPLEX
 - procedure, 9-37
 - ccsversion

- definition of, 9-4
- selecting, 9-4
- <ccsversion name>, 3-110
- CENTRALSTATUS procedure, 9-42
- CENTRALSUPPORT library call, 9-23
- CENTRALSUPPORT library procedures, 9-13
- <cexp function>, 5-46
- <change file statement>, 4-33
- character array, 3-5
- <character array identifier>, 3-4
- <character array name>, 5-32
- <character array part>, 5-31
- <character array row>, 5-31
- character set, 9-1, 9-4
 - designating, 9-5
- <character set>, 3-178
- <character size>, 5-77
- character string manipulation, C-4
 - ASCII collating sequence, C-4
 - ASCII option, 6-16
 - character array, 3-5
 - default character type, C-12
 - DELTA function, 5-53
 - DOUBLE function, 5-58
 - EBCDIC collating sequence, C-4
 - INTEGER function, 5-66
 - internal representation of characters, C-2
 - intrinsic functions returning values of type
 - POINTER, 5-42
 - OFFSET function, 5-76
 - PICTURE declaration, 3-117
 - pointer assignment, 4-17
 - POINTER declaration, 3-128
 - pointer expression, 5-31
 - POINTER function, 5-77
 - pointer relation, 5-19
 - READLOCK function, 5-80
 - REAL function, 5-82
 - REMAININGCHARS function, 5-83
 - REPLACE statement, 4-137
 - SCAN statement, 4-181
 - SIZE function, 5-88
 - string literal, 2-12
 - string relation, 5-19
 - TRANSLATETABLE declaration, 3-178
 - TRUTHSET declaration, 3-182
- <character type>, 3-5
- <check option>, 6-20
- checkpoint, 4-36
 - checkpoint/restart messages, 4-38
 - effect on the PROCESSID function, 5-79
 - with the sort intrinsic, 4-194
- <checkpoint statement>, 4-36
- <checksum function>, 5-47
- <cln function>, 5-47
- <close file part>, 4-42
- <close options>
 - CRUNCH, 4-43
 - DOWNSIZEAREA, 4-43
 - DOWNSIZEAREALOCK, 4-43
 - LOCK, 4-43
 - PURGE, 4-43
 - REEL, 4-43
 - REWIND, 4-43
- <close statement>, 4-42
- close with retention, 4-44
- <closedisposition option>, 4-44
- CNV_ADD procedure, 9-44
- CNV_CONVERTCURRENCY_STAR
 - procedure, 9-47
- CNV_CONVERTDATE_STAR procedure, 9-49
- CNV_CONVERTNUMERIC_STAR
 - procedure, 9-52
- CNV_CONVERTTIME_STAR procedure, 9-54
- CNV_CURRENCYEDIT procedure, 9-57
- CNV_CURRENCYEDIT_DOUBLE
 - procedure, 9-59
- CNV_CURRENCYEDITTMP procedure, 9-61
- CNV_CURRENCYEDITTMP_DOUBLE
 - procedure, 9-63
- CNV_DELETE procedure, 9-65
- CNV_DISPLAYMODEL procedure, 9-66
- CNV_FORMATDATE procedure, 9-68
- CNV_FORMATDATETMP procedure, 9-70
- CNV_FORMATTIME procedure, 9-72
- CNV_FORMATTIMETMP procedure, 9-74
- CNV_FORMSIZE procedure, 9-76
- CNV_INFO procedure, 9-78
- CNV_MODIFY procedure, 9-81
- CNV_NAMES procedure, 9-84
- CNV_SYMBOLS procedure, 9-86
- CNV_SYSTEMDATETIME procedure, 9-91
- CNV_SYSTEMDATETIMETMP
 - procedure, 9-94
- CNV_TEMPLATE procedure, 9-97
- CNV_VALIDATENAME procedure, 9-99
- CODE file, 6-5
- code optimization
 - BEGINSEGMENT option, 6-18
 - ENDSEGMENT option, 6-22
 - OPTIMIZE option, 6-40
- <code option>, 6-20
- coded character set, 9-4
 - selecting, 9-4
- <column width>, 4-126
- <comment characters>, 2-10

- <comment remark>, 2-10
- <commentary>, 4-130
- <compare procedure>, 4-191
- COMPARE_TEXT_USING_ORDER_INFO
 - procedure, 9-100
- comparing text
 - for internationalization, 9-8
- compiler control options, 6-16
 - ASCII option, 6-16
 - AUTOBIND option, 6-16
 - BEGINSEGMENT option, 6-18
 - BIND option, 6-19
 - BINDER option, 6-19
 - BINDER_MATCH option, 6-19
 - CHECK option, 6-20
 - CODE option, 6-20
 - DONTBIND option, 6-21
 - DUMPINFO option, 6-21
 - ENDSEGMENT option, 6-22
 - ERRLIST option, 6-22
 - ERRORLIMIT option, 6-29
 - EXTERNAL option, 6-22
 - FORMAT option, 6-22
 - GO TO option, 6-23
 - HOST option, 6-23
 - INCLNEW option, 6-23
 - INCLSEQ option, 6-24
 - INCLUDE option, 6-24
 - INITIALIZE option, 6-26
 - INSTALLATION option, 6-26
 - INTRINSICS option, 6-27
 - LARGE_LINEINFO option, 6-27
 - LEVEL option, 6-28
 - LIBRARY option, 6-29
 - LIMIT option, 6-29
 - LINEINFO option, 6-29
 - LIST option, 6-30
 - LISTDELETED option, 6-30
 - LISTDOLLAR option, 6-30
 - LISTINCL option, 6-31
 - LISTOMITTED option, 6-31
 - LOADINFO option, 6-32
 - MAKEHOST option, 6-34
 - MCP option, 6-36
 - MERGE option, 6-37
 - NEW option, 6-37
 - NEWSEQERR option, 6-38
 - NOBINDINFO option, 6-38
 - NOSTACKARRAYS option, 6-38
 - NOXREFLIST option, 6-39
 - OMIT option, 6-39
 - OPTIMIZE option, 6-40
 - PAGE option, 6-40
 - PARAMCHECK option, 6-41
 - PURGE option, 6-41
 - SEGDESCABOVE option, 6-41
 - SEGS option, 6-42
 - SEPCOMP option, 6-42
 - SEQ option, 6-44
 - SEQERR option, 6-44
 - SHARING option, 6-45
 - SINGLE option, 6-46
 - STACK option, 6-46
 - STATISTICS option, 6-46
 - STOP option, 6-47
 - TADS option, 6-48
 - TIME option, 6-50
 - USE option, 6-50
 - user option, 6-50
 - VERSION option, 6-51
 - VOID option, 6-52
 - VOIDT option, 6-52
 - WARNSUPR option, 6-53
 - WRITEAFTER option, 6-53
 - XDECS option, 6-54
 - XREF option, 6-54
 - XREFFILES option, 6-56
 - XREFS option, 6-56
- <compiler control record>, 6-10
- compiler input files, 6-3
- compiler output files, 6-5
- compile-time facility
 - <compile-time arithmetic expression>, 7-2
 - <compile-time begin statement>, 7-4
 - <compile-time Boolean expression>, 7-5
 - compile-time compiler control options, 7-8
 - CTLIST option, 7-8
 - CTMON option, 7-8
 - CTTRACE option, 7-8
 - LISTSKIP option, 7-9
 - <compile-time define identifier>, 7-4
 - <compile-time define statement>, 7-4
 - <compile-time for statement>, 7-5
 - <compile-time identifier>, 7-3
 - <compile-time if statement>, 7-5
 - <compile-time invoke statement>, 7-6
 - <compile-time let statement>, 7-6
 - <compile-time statement>, 7-3
 - <compile-time text>, 7-4
 - <compile-time thru statement>, 7-6
 - <compile-time variable declaration>, 7-2
 - <compile-time variable>, 7-2
 - <compile-time while statement>, 7-6
 - <ctlist option>, 7-8
 - <ctmon option>, 7-8
 - <cttrace option>, 7-8

- <listskip option>, 7-9
- <number identifier>, 7-2
- <starting value>, 7-2
- <vector length>, 7-2
- <compiletime function>, 5-48
- <complex assignment>, 4-15
- <complex case expression>, 5-24
- complex data type
 - complex array declaration, 3-3
 - complex array reference declaration, 3-12
 - complex assignment, 4-15
 - COMPLEX declaration, 3-16
 - complex expression, 5-24
 - complex functions
 - CCOS function, 5-46
 - CEXP function, 5-46
 - CLN function, 5-47
 - COMPLEX function, 5-48
 - CONJUGATE function, 5-48
 - CSIN function, 5-49
 - CSQRT function, 5-49
 - complex operand internal structure, C-19
 - complex procedure declaration, 3-132
 - complex relation, 5-18
 - complex value array declaration, 3-186
 - functions with complex parameters
 - CABS function, 5-46
 - functions with complex parameters IMAG function, 5-65
 - functions with complex parameters REAL function, 5-82
 - intrinsic functions returning values of type COMPLEX, 5-42
 - width of complex array elements, 3-5
- <complex declaration>, 3-16
- <complex equality operator>, 5-18
- <complex expression>, 5-24
- <complex function designator>, 5-29
- <complex function>, 5-48
- <complex identifier>, 3-16
- <complex intrinsic name>, 5-29
 - alphabetical listing of, 5-42
- complex operand internal structure, C-19
- <complex operand>, 5-24
- <complex operator>, 5-24
- <complex primary>, 5-24
- <complex relation>, 5-18
- <complex update assignment>, 4-15
- <complex variable>, 4-15
- <compound statement>, 1-3
- <concatenation expression>, 5-8
- <concatenation>, 5-9
- <condition>, 4-139
- <conditional arithmetic expression>, 5-7
- <conditional Boolean expression>, 5-17
- <conditional complex expression>, 5-24
- <conditional designational expression>, 5-27
- <conditional expression>, 5-26
- <conditional pointer expression>, 5-32
- <conjugate function>, 5-48
- <connection block item designator>, 3-20
- <connection block item>, 3-20
- <connection block qualifier>, 3-20
- <connection block reference variable declaration>, 3-18
- <connection block reference variable>, 3-18
- CONNECTION BLOCK TYPE
 - Declaration, 3-19
 - referencing items outside the connection block, 3-20
- <connection block type declaration>, 3-19
- <connection library declaration>, 3-24
- <connection library identifier>, 3-24
- <connection library instance designator>, 3-24
- <connection library specification>, 3-24
- CONNECTION parameter, 3-137
- <connecttimelimit option>, 4-24
- <constant arithmetic expression>, 5-7
- <constant expression>, 3-186
- <constant list>, 3-186
- <constant string expression>, 5-35
- <constant>, 3-186
- contents of printer listing
 - CODE option, 6-20
 - FORMAT option, 6-22
 - LIST option, 6-30
 - LISTDELETED option, 6-30
 - LISTINCL option, 6-31
 - LISTOMITTED option, 6-31
 - PAGE option, 6-40
 - SEGS option, 6-42
 - SINGLE option, 6-46
 - STACK option, 6-46
 - TIME option, 6-50
 - WARNSUPR option, 6-53
- <continue statement>, 4-47
- <control character>, 3-122
- <control part>, 3-38
- CONVENTION task attribute, 9-3
- conventions
 - for localization, establishing, 9-3
- CONVERTDATEANDTIME procedure, 8-20
- <core-to-core blocking part>, 4-124
- <core-to-core blocking>, 4-124
- <core-to-core event>, 4-204

- <core-to-core file part>, 4-124
- <core-to-core part>, 4-124
- <core-to-core record size>, 4-124
- <cos function>, 5-48
- <cosh function>, 5-49
- <cotan function>, 5-49
- <count part>, 4-138
- creating multilingual messages, guidelines for, 9-10
- critical block, 4-112
- cross reference, 6-54
 - NOXREFLIST option, 6-39
 - XDECS option, 6-54
 - XREF option, 6-54
 - XREFFILE file, 6-7
 - XREFFILES option, 6-56
 - XREFS option, 6-56
- CRUNCH <close option>, 4-43
- <csin function>, 5-49
- <csqrt function>, 5-49

- D**
- <dabs function>, 5-49
- <dalpha function>, 5-49
- <dand function>, 5-50
- <darccos function>, 5-50
- <darcsin function>, 5-50
- <darctan function>, 5-50
- <darctan2 function>, 5-51
- data class, definition of, 9-7
- data communications protocols, international, 9-1
- data descriptors
 - internal structure, C-20
- <data error label>, 4-129
- <data exponent part>, 3-68
- data number, 3-68
- <data type>, 3-90
- DCALGOL
 - compile-time facility in, 7-1
- <dcos function>, 5-51
- <dcosh function>, 5-51
- <deallocate statement>, 4-48
- <decimal function>, 5-51
- <decimal places>, 3-64
- <declaration list>, 1-2
- declarations
 - CONNECTION BLOCK REFERENCE VARIABLE, 3-18
 - EPILOG PROCEDURE, 3-42
 - EXCEPTION PROCEDURE, 3-48
 - FORWARD REFERENCE, 3-87
 - PENDING PROCEDURE, 3-115
 - PROLOG PROCEDURE, 3-154
 - STRUCTURE BLOCK ARRAY, 3-161
 - STRUCTURE BLOCK REFERENCE VARIABLE, 3-163
 - STRUCTURE BLOCK TYPE, 3-164
 - STRUCTURE BLOCK VARIABLE, 3-167
- default character type, C-12
- default settings
 - for internationalization, 9-3
- <define declaration>, 3-26
- <define identifier>, 3-26
- <define invocation>, 3-27
- <definition>, 3-26
- <delimiter>, 2-2
- <delinklibrary function>, 5-52
- <delta function>, 5-53
- DEPENDENTSPECS file attribute, 6-3
- <deqv function>, 5-54
- <derf function>, 5-54
- <derfc function>, 5-54
- <designational case expression>, 5-27
- <designational expression>, 5-27
- <destination characters>, 3-178
- <destination>, 4-137
- <detach statement>, 4-49
- <device>, 4-36
- <dexp function>, 5-54
- <dgamma function>, 5-54
- diagnostic tools
 - DUMP declaration, 3-37
 - MONITOR declaration, 3-106
 - PROGRAMDUMP statement, 4-115
 - STATISTICS option, 6-46
 - TADS option, 6-48
- <digit convert part>, 4-138
- <digit>, 2-2
- dimensionality of arrays, 3-7
- <dimp function>, 5-55
- <dinteger function>, 5-56
- <dintegert function>, 5-56
- <direct array declaration>, 3-33
- <direct array identifier>, 3-33, 5-101
- <direct array name>, 3-33
- <direct array reference identifier>, 3-12
- direct array row, 4-206
- <direct array row equivalence>, 3-33
- <direct array row>, 3-33
- <direct file identifier>, 3-57
- direct I/O, 4-72
- direct linkage, 5-69

- <direct switch file identifier>, 3-168
- <directory element>, 4-34
- <disable statement>, 4-50
- <disabling on statement>, 4-99
- <disk size>, 4-193
- <display statement>, 4-51
- <disposition>, 4-36
- DIV, 5-3
- division, 5-3
- <dgamma function>, 5-56
- <dln function>, 5-56
- <dlog function>, 5-56
- DMALGOL
 - compile-time facility in, 7-1
- <dmax function>, 5-57
- <dmin function>, 5-57
- <dnabs function>, 5-57
- <dnormalize function>, 5-57
- <dnot function>, 5-57
- <do statement>, 4-52
- <dontbind option>, 6-21
- DONTWAIT control option, 4-24
- DONTWAIT option, 4-101
- <dor function>, 5-58
- double data type
 - arithmetic assignment, 4-5
 - arithmetic expression, 5-2
 - arithmetic relation, 5-18
 - direct double array declaration, 3-33
 - double array declaration, 3-3
 - double array reference declaration, 3-12
 - DOUBLE declaration, 3-36
 - double functions
 - DABS function, 5-49
 - DAND function, 5-50
 - DARCCOS function, 5-50
 - DARCSIN function, 5-50
 - DARCTAN function, 5-50
 - DARCTAN2 function, 5-51
 - DCOS function, 5-51
 - DCOSH function, 5-51
 - DECIMAL function, 5-51
 - DEQV function, 5-54
 - DERF function, 5-54
 - DERFC function, 5-54
 - DEXP function, 5-54
 - DGAMMA function, 5-54
 - DIMP function, 5-55
 - DINTEGER function, 5-56
 - DLGAMMA function, 5-56
 - DLN function, 5-56
 - DLOG function, 5-56
 - DMAX function, 5-57
 - DMIN function, 5-57
 - DNABS function, 5-57
 - DNORMALIZE function, 5-57
 - DNOT function, 5-57
 - DOR function, 5-58
 - DOUBLE function, 5-58
 - DSCALELEFT function, 5-60
 - DSCALERIGHT function, 5-60
 - DSCALERIGHTT function, 5-60
 - DSIN function, 5-61
 - DSINH function, 5-61
 - DSQRT function, 5-61
 - DTAN function, 5-61
 - DTANH function, 5-61
 - <pot function>, 5-79
 - POTC function, 5-79
 - POTH function, 5-79
 - POTL function, 5-79
 - double procedure declaration, 3-132
 - double value array declaration, 3-186
 - double-precision operand internal
 - structure, C-17
 - width of double array elements, 3-5
- <double declaration>, 3-36
- <double function>, 5-58
- <double identifier>, 3-36
- <double variable>, 4-198
- double-precision operand internal
 - structure, C-17
- DOWNSIZEAREA <close option>, 4-43
- DOWNSIZEAREALOCK <close option>, 4-43
- <drop function>, 5-59
- DS (Discontinue) ODT command, 8-6
- <dscaleleft function>, 5-60
- <dscaleright function>, 5-60
- <dscalerightt function>, 5-60
- <dsin function>, 5-61
- <dsinh function>, 5-61
- <dsqrt function>, 5-61
- <dtan function>, 5-61
- <dtanh function>, 5-61
- dump, 4-115
- <dump declaration>, 3-37
- <dump list>, 3-37
- <dump parameters>, 3-38
- <dumpinfo option>, 6-21
- <dynamic procedure specification>, 3-138

E

- <EBCDIC character>, 2-14
- <EBCDIC string constant>, 5-34
- <EBCDIC string literal>, 3-51
- <editing modifier>, 3-64
- editing phrase
 - \$ editing modifier, 3-86
 - A editing phrase letter, 3-66
 - C editing phrase letter, 3-66
 - D editing phrase letter, 3-68
 - decimal places, 3-65
 - E editing phrase letter, 3-70
 - editing modifiers, 3-86
 - F editing phrase letter, 3-71
 - field width, 3-64
 - G editing phrase letter, 3-71
 - H editing phrase letter, 3-72
 - I editing phrase letter, 3-75
 - J editing phrase letter, 3-76
 - K editing phrase letter, 3-72
 - L editing phrase letter, 3-77
 - multiple editing phrases, 3-61
 - O editing phrase letter, 3-78
 - P editing modifier, 3-86
 - R editing phrase letter, 3-79
 - repetition of, 3-63
 - S editing phrase letter, 3-80
 - simple string literal, 3-62
 - T editing phrase letter, 3-82
 - U editing phrase letter, 3-82
 - V editing phrase letter, 3-83
 - variable editing phrases, 3-65
 - X editing phrase letter, 3-84
 - Z editing phrase letter, 3-85
- <editing phrase>, 3-64
- <editing specifications>, 3-60
- EMPTY, 5-35
- EMPTY6, 5-35
- EMPTY7, 5-35
- EMPTY8, 5-35
- <enable statement>, 4-53
- <enabling on statement>, 4-94
- <end remark>, 2-10
- <ending index>, 5-47
- <end-of-record>, 4-130
- <endsegment option>, 6-22
- <entier function>, 5-62
- entry point (See library)
- <environment>, 6-34
- <eof label>, 4-129
- <epilog procedure declaration>, 3-42
- <equality operator>, 5-19
- <equation part>, 3-14
- equivalent data comparison
 - in internationalization, 9-8
- EQV, 5-14
- ERASE statement, 4-54
- <erf function>, 5-62
- <erfc function>, 5-63
- <errlist option>, 6-22
- error handling for libraries, 8-5
- <error limit>, 6-29
- error messages
 - for formatted input, A-3
 - for formatted output, A-2
 - for free-field input, A-1
- error values
 - for CENTRALSUPPORT procedures, 9-136
- <errorlimit option>, 6-29
- ERRORS file, 6-6
- <escape remark>, 2-10
- <escape text>, 2-10
- <event array declaration>, 3-45
- <event array designator>, 3-47
- <event array identifier>, 3-45
 - for SIZE function, 5-88
- <event declaration>, 3-45
- <event designator>, 3-46, 4-204
- event handling
 - ATTACH statement, 4-23
 - AVAILABLE function, 5-45
 - CAUSE statement, 4-31
 - CAUSEANDRESET statement, 4-32
 - DETACH statement, 4-49
 - EVENT ARRAY declaration, 3-45
 - EVENT declaration, 3-45
 - EVENT_STATUS function, 8-22
 - EXPORT declaration, 3-51
 - FIX statement, 4-60
 - FREE statement, 4-67
 - HAPPENED function, 5-64
 - IMPORTED declaration, 3-90
 - LIBERATE statement, 4-78
 - PROCURE statement, 4-114
 - RESET statement, 4-168
 - SET statement, 4-187
 - UNLOCK statement, 4-203
 - up-level event
 - in ATTACH statement, 4-23
 - in direct I/O, 4-73
 - WAIT statement, 4-204
 - WAITANDRESET statement, 4-209
- <event identifier>, 3-45
- <event list>, 4-204

- <event statement>, 4-55
 - EVENT_STATUS function, 8-22
 - event-valued attributes (*See also* event handling)
 - interrogating
 - <event designator>, 3-46
 - <event-valued file attribute name>, 3-46
 - <event-valued file attribute>, 3-46
 - <event-valued library attribute name>, 3-46
 - <event-valued library attribute>, 3-46
 - <event-valued task attribute name>, 3-46
 - <event-valued task attribute>, 3-46
 - <exception procedure declaration>, 3-48
 - <exchange statement>, 4-56
 - <exp function>, 5-63
 - <explicit delimiter>, 4-130
 - <exponent part>, 2-6
 - exponentiation, 5-4
 - EXPONENTUNDERFLOW fault (ON statement example), 4-99
 - EXPORT declaration
 - event handling, 3-51
 - interlock handling, 3-51
 - READONLY option, 3-51
 - READWRITE option, 3-51
 - <export declaration>, 3-51, 8-1
 - <export object specification>, 3-51
 - <export options>, 3-51
 - <expression>, 5-1
 - expressions
 - try, 5-37
 - <external global connection block type declaration>, 3-19
 - <external global structure block type declaration>, 3-165
 - <external option>, 6-22
- F**
- <family designator>, 4-163
 - family substitution, controlling for library linkage, 8-10
 - <fault action>, 4-98
 - <fault list>, 4-95
 - <fault name>, 4-95
 - <fault number>, 4-97
 - <fault stack history>, 4-97
 - <field delimiter>, 4-130
 - field notation, C-1
 - <field width>, 3-64
 - <field>, 4-130
 - <file declaration>, 3-57
 - <file designator>, 3-168
 - file handling (*See* I/O)
 - <file identifier>, 3-57
 - file parameters, 3-146
 - <file part>, 4-121
 - semantics, 4-121
 - <file specification>, 6-21
 - <file-valued task attribute name>, 4-217
 - <fill statement>, 4-58
 - <first function>, 5-63
 - <firststone function>, 5-63
 - <firstword function>, 5-64
 - <fix statement>, 4-60
 - <for list element>, 4-61
 - <for statement>, 4-61
 - <formal parameter list>, 3-133
 - <formal parameter part>, 3-133
 - <formal parameter specifier>, 3-136
 - <formal parameter>, 3-133
 - <formal symbol part>, 3-27
 - <formal symbol>, 3-27
 - <format and list part>, 4-126
 - semantics for READ statement, 4-126
 - semantics for WRITE statement, 4-218
 - <format declaration>, 3-60
 - <format designator>, 3-170
 - <format identifier>, 3-60
 - <format option>, 6-22
 - <format part>, 3-60
 - format-error messages
 - formatted input, A-3
 - formatted output, A-2
 - free-field input, A-1
 - formatted input format-error messages, A-3
 - formatted output format-error messages, A-2
 - formatted read, 4-126
 - formatted write, 4-219
 - <forward global connection block type declaration>, 3-19
 - <forward global structure block type declaration>, 3-165
 - <forward interrupt declaration>, 3-87
 - <forward procedure declaration>, 3-87
 - <forward reference declaration>, 3-87
 - <forward switch label declaration>, 3-87
 - <free statement>, 4-67
 - free-field data format, 4-130
 - free-field data record, 4-130
 - free-field input format-error messages, A-1
 - <free-field part>, 4-126
 - <freeze statement>, 4-68
 - <function expression>, 5-28

- FUNCTIONNAME library attribute (See library)
- functions, 5-38
 - ABS function, 5-43
 - ACCEPT statement, 4-2
 - ARCCOS function, 5-43
 - ARCSIN function, 5-43
 - ARCTAN function, 5-43
 - ARRAYSEARCH function, 5-44
 - ATANH function, 5-45
 - AVAILABLE function, 5-45
 - BOOLEAN function, 5-45
 - CABS function, 5-46
 - CCOS function, 5-46
 - CEXP function, 5-46
 - CHANGEFILE statement, 4-33
 - CHECKPOINT statement, 4-36
 - CHECKSUM function, 5-47
 - CLN function, 5-47
 - CLOSE statement, 4-42
 - COMPILETIME function, 5-48
 - COMPLEX function, 5-48
 - CONJUGATE function, 5-48
 - COS function, 5-48
 - COSH function, 5-49
 - COTAN function, 5-49
 - CSIN function, 5-49
 - CSQRT function, 5-49
 - DABS function, 5-49
 - DALPHA function, 5-49
 - DAND function, 5-50
 - DARCCOS function, 5-50
 - DARCSIN function, 5-50
 - DARCTAN function, 5-50
 - DARCTAN2 function, 5-51
 - DCOS function, 5-51
 - DCOSH function, 5-51
 - DECIMAL function, 5-51
 - DELINKLIBRARY function, 5-52
 - DELTA function, 5-53
 - DEQV function, 5-54
 - DERF function, 5-54
 - DERFC function, 5-54
 - DEXP function, 5-54
 - DGAMMA function, 5-54
 - DIMP function, 5-55
 - DINTEGER function, 5-56
 - DINTEGERT function, 5-56
 - DLGAMMA function, 5-56
 - DLN function, 5-56
 - DLOG function, 5-56
 - DMAX function, 5-57
 - DMIN function, 5-57
 - DNABS function, 5-57
 - DNORMALIZE function, 5-57
 - DNOT function, 5-57
 - DOR function, 5-58
 - DOUBLE function, 5-58
 - DROP function, 5-59
 - DSCALELEFT function, 5-60
 - DSCALERIGHT function, 5-60
 - DSCALERIGHTT function, 5-60
 - DSIN function, 5-61
 - DSINH function, 5-61
 - DSQRT function, 5-61
 - DTAN function, 5-61
 - DTANH function, 5-61
 - ENTIER function, 5-62
 - ERF function, 5-62
 - ERFC function, 5-63
 - EVENT_STATUS, 8-22
 - EXP function, 5-63
 - FIRST function, 5-63
 - FIRSTONE function, 5-63
 - FIRSTWORD function, 5-64
 - FIX statement, 4-60
 - FREE statement, 4-67
 - GAMMA function, 5-64
 - HAPPENEDfunction, 5-64
 - HEAD function, 5-65
 - IMAG function, 5-65
 - INTEGER function, 5-66
 - INTEGERT function, 5-67
 - intrinsic functions returning values of type BOOLEAN, 5-42
 - intrinsic functions returning values of type COMPLEX., 5-42
 - intrinsic functions returning values of type POINTER, 5-42
 - intrinsic functions returning values of type STRING, 5-42
 - LENGTH function, 5-68
 - LINENUMBER function, 5-68
 - LISTLOOKUP function, 5-73
 - LN function, 5-73
 - LNGAMMA function, 5-73
 - LOCKSTATUS function, 5-74
 - LOG function, 5-75
 - MASKSEARCH function, 5-75
 - MAX function, 5-75
 - MESSAGESEARCHER statement, 4-84
 - MIN function, 5-75
 - NABS function, 5-76
 - NORMALIZE function, 5-76
 - OFFSET function, 5-76
 - ONES function, 5-77

OPEN statement, 4-101
 POINTER function, 5-77
 POTC function, 5-79
 <potfunction>, 5-79
 POTH function, 5-79
 POTL function, 5-79
 PROCESSID function, 5-79
 RANDOM function, 5-80
 READ statement, 4-121
 READLOCK function, 5-80
 READYCL function, 5-81
 REAL function, 5-82
 REMAININGCHARS function, 5-83
 REMOVEFILE statement, 4-135
 REPEAT function, 5-83
 SCALELEFT function, 5-84
 SCALERIGHT function, 5-84
 SCALERIGHTF function, 5-84
 SCALERIGHTT function, 5-85
 SECONDWORD function, 5-85
 SEEK statement, 4-186
 SETACTUALNAME function, 5-86
 SIGN function, 5-87
 SIN function, 5-87
 SINGLE function, 5-88
 SINH function, 5-88
 SIZE function, 5-88
 SPACE statement, 4-197
 SQRT function, 5-89
 STRING function, 5-89
 STRING4 function, 5-89
 STRING7 function, 5-89
 STRING8 function, 5-89
 TAIL function, 5-91
 TAKE function, 5-91
 TAN function, 5-92
 TANH function, 5-92
 THIS, 5-92
 THISCL function, 5-93
 TIME function, 5-93
 TRANSLATE function, 5-100
 UNREADYCL function, 5-100
 USERDATA function, 5-101
 USERDATALOCATOR function, 5-102
 USERDATAREBUILD function, 5-102
 VALUE function, 5-103
 WAIT statement, 4-205
 WAITANDRESET statement, 4-209
 WRITE statement, 4-215

G

<gamma function>, 5-64
 GET_CS_MSG procedure, 9-102
 <global connection block type
 declaration>, 3-19
 global identifiers, 1-6
 <global part>, 1-3
 <global procedure reference array
 declaration>, 3-151
 <global structure block type
 declaration>, 3-164
 <go to option>, 6-23
 <go to statement>, 4-70

H

<happened function>, 5-64
 <head function>, 5-65
 <hex string>, 4-130
 <hexadecimal string constant>, 5-35
 hexadecimal strings in free-field data
 records, 4-132
 HOST file, 6-4
 <host option>, 6-23

I

I/O

ACCEPT statement, 4-2
 DIRECT ARRAY declaration, 3-33
 direct I/O
 array reference variable use with, 3-12
 checkpoint/restart inhibition with, 4-38
 event use with, 3-45
 file declaration for, 3-57
 general information for, 4-72
 READ statement with, 4-129
 WAIT statement with, 4-204
 DISPLAY statement, 4-51
 file handling
 CHANGEFILE statement, 4-33
 CLOSE statement, 4-42
 EXCHANGE statement, 4-56
 FILE declaration, 3-57
 LOCK file statement, 4-79
 MERGE statement, 4-82
 multiple attribute assignment
 statement, 4-93
 OPEN statement, 4-101

- REMOVEFILE statement, 4-135
- REWIND statement, 4-178
- SEEK statement, 4-186
- SPACE statement, 4-197
- SWITCH FILE declaration, 3-168
- formatting
 - FORMAT declaration, 3-60
 - free-field data format, 4-130
 - LIST declaration, 3-104
 - run-time format-error messages, A-1
 - SWITCH FORMAT declaration, 3-170
 - SWITCH LIST declaration, 3-174
- I/O statement, 4-71
- interlock handling
 - LOCK interlock statement, 4-80
- normal I/O, 4-71
- READ statement, 4-121
- serial I/O operation, 4-186
- synchronized output (See synchronized output)
- WRITE statement, 4-215
- WRITEAFTER option, 6-53
- <I/O option or carriage control>, 4-122
- <I/O statement>, 4-71
- <identifier>, 2-5
- <if clause>, 4-74
- <if statement>, 4-74
- <imag function>, 5-65
- <immediate option>, 6-12
- IMP, 5-14
- <imported array specification>, 3-90
- <imported data identifier>, 5-86
- <imported data specification>, 3-90
- IMPORTED declaration
 - event handling, 3-90
 - interlock handling, 3-90
 - READONLY option, 3-90
 - READWRITE option, 3-90
- <inclnew option>, 6-23
- <inclseq option>, 6-24
- INCLUDE compiler control option, 9-23
- INCLUDE files, 6-4
- <include option>, 6-24
- indirect linkage, 5-70
- INFO file, 6-4, 6-7
- <initial part>, 4-61
- <initial value>, 4-58
- <initialize option>, 6-26
- initialized pointer, 3-128
- <in-out part>, 3-60
- <input option>, 4-190
- input parameters
 - for library procedures, 9-24
- <input procedure>, 4-190
- <installation number list>, 6-26
- <installation number>, 6-26
- <installation option>, 6-26
- <instuff parameter>, 5-102
- integer data type
 - arithmetic assignment, 4-5
 - arithmetic expression, 5-2
 - arithmetic relation, 5-18
 - direct integer array declaration, 3-33
 - functions for manipulating integer expressions
 - DINTEGER function, 5-56
 - DOUBLE function, 5-58
 - NORMALIZE function, 5-76
- integer array declaration, 3-3
- integer array reference declaration, 3-12
- INTEGER declaration, 3-92
- integer functions
 - ARRAYSEARCH function, 5-44
 - CLOSE statement, 4-42
 - DELINKLIBRARY function, 5-52
 - DELTA function, 5-53
 - ENTIER function, 5-62
 - FIRSTONE function, 5-63
 - INTEGER function, 5-66
 - INTEGERT function, 5-67
 - LENGTH function, 5-68
 - LINENUMBER function, 5-68
 - LISTLOOKUP function, 5-73
 - MASKSEARCH function, 5-75
 - MESSAGESEARCHER statement, 4-84
 - OFFSET function, 5-76
 - ONES function, 5-77
 - OPEN statement, 4-101
 - PROCESSID function, 5-79
 - REMAININGCHARS function, 5-83
 - SCALELEFT function, 5-84
 - SCALERIGHT function, 5-84
 - SCALERIGHTT function, 5-85
 - SETACTUALNAME function, 5-86
 - SIGN function, 5-87
 - SIZE function, 5-88
 - VALUE function, 5-103
 - WAIT statement, 4-205
 - WAITANDRESET statement, 4-209
- integer operand internal structure, C-15
- integer procedure declaration, 3-132
- integer value array declaration, 3-186
 - width of integer array elements, 3-5
- <integer declaration>, 3-92
- <integer function>, 5-66
- <integer identifier>, 3-92

- integer operand internal structure, C-15
 - <integer variable>, 4-198
 - for<fault number>, 4-97
 - <integer>, 2-7
 - <integert function>, 5-67
 - INTERFACENAME, 3-101
 - INTERFACENAME library attribute, 8-8
 - <interlock designator>, 3-95
 - interlock handling
 - EXPORT declaration, 3-51
 - IMPORTED declaration, 3-90
 - INTERLOCK and INTERLOCK ARRAY declarations, 3-94
 - LOCK interlock statement, 4-80
 - LOCKSTATUS function, 5-74
 - UNLOCK statement, 5-100
 - <internal file name>, 6-21
 - internationalization, 9-1
 - ccsversion selection, 9-4
 - ccsversions, 9-4
 - coded character sets, 9-4
 - comparing text, 9-8
 - data classes, 9-7
 - default settings, changing, 9-3
 - equivalent comparison of data, 9-8
 - hierarchy, 9-3
 - logical data comparison, 9-8
 - message creation in application program, 9-9
 - multilingual message creation, 9-10
 - ordering sequence value (OSV), 9-8
 - output message arrays, 9-9
 - positioning, 9-8
 - priority sequence value (PSV), 9-8
 - translate tables, 9-6
 - accessing, 9-6
 - truth sets, 9-7
 - INTERRUPTIBLE option, 4-80
 - <interrupt declaration>, 3-96
 - interrupt handling
 - ATTACH statement, 4-23
 - DETACH statement, 4-49
 - DISABLE statement, 4-50
 - ENABLE statement, 4-53
 - INTERRUPT declaration, 3-96
 - interrupt statement, 4-76
 - ON statement, 4-94
 - <interrupt identifier>, 3-96
 - <interrupt statement>, 4-76
 - INTNAME library attribute (See library)
 - <intrinsic translate table>, 4-139
 - <intrinsics option>, 6-27
 - <introduction code>, 3-118
 - <introduction>, 3-118
 - INVALIDINDEX fault (ON statement example), 4-99
 - <invocation statement>, 4-77
 - invoking defines, 3-27
 - IS, 5-15
 - ISNT, 5-15
 - <isolated procedure specification>, 3-139
 - ISVALID function, 5-67
 - <iteration clause>, 3-105
 - <iteration part>, 4-61
- J**
- job and task control
 - CALL statement, 4-26
 - CHECKPOINT statement, 4-36
 - CONTINUE statement, 4-47
 - PROCEDURE declaration, 3-132
 - PROCESS statement, 4-112
 - PROCESSID function, 5-79
 - RUN statement, 4-179
 - TASK ARRAY declaration, 3-176
 - task assignment, 4-22
 - TASK declaration, 3-176
 - ZIP statement, 4-224
- L**
- <label counter modulus>, 3-38
 - <label counter>, 3-38
 - <label declaration>, 3-98
 - <label designator>, 5-27
 - <label identifier>, 3-98
 - <labeled statement>, 4-1
 - language
 - run-time, establishing, 9-3
 - <language component>, 2-1
 - <language name>, 3-110
 - <language specification>, 4-83
 - LANGUAGE task attribute, 9-3
 - <large_lineinfo option>, 6-27
 - <left bit from>, 5-9
 - <left bit to>, 5-9
 - <left bit>, 5-12
 - <length function>, 5-68
 - length of string literals, 4-141
 - <letter string>, 2-2
 - <letter>, 2-2
 - <level 2 procedure>, 1-3

- <level option>, 6-28
- <lex level restriction part>, 3-129
- LIBACCESS library attribute (*See also* library)
 - assigning values
 - LIBRARY declaration, 3-99
 - mnemonic attribute assignment, 4-16
 - using family substitution with, 8-10
- <liberate statement>, 4-78
- LIBPARAMETER library attribute (*See* library)
- library
 - ACTUALNAME option of LIBRARY
 - declaration, 3-139
 - assigning values to LIBACCESS library attribute, 3-99, 4-16
 - assigning values to string-valued library attributes, 3-99, 4-20
 - attributes, 8-8
 - defined, 8-9
 - in declaration, 3-102
 - in linkage, 8-4
 - calling programs, 8-2
 - circular chain, 5-71, 8-2
 - creating libraries, 8-6
 - declarations
 - EXPORT, 3-51
 - LIBRARY, 3-99
 - PROCEDURE, 3-132
 - delinking, 8-5
 - description of libraries, 8-1
 - direct linkage, 8-4
 - dumping, 4-116
 - duration, 8-3
 - dynamic linkage, 8-4
 - entry points
 - allowed parameters, 3-53
 - allowed types, 3-53
 - at initiation, 8-3
 - declaring in calling program, 3-139
 - declaring in library program, 3-51
 - defined, 8-1
 - for MCPSUPPORT library
 - (example), 8-22
 - matching types, 8-11
 - passing parameters, 8-12
 - error handling, 8-5
 - examples
 - calling programs, 8-15, 8-19
 - direct linkage, 8-16
 - dynamic linkage, 8-17
 - indirect linkage, 8-17
 - MCPSUPPORT library, 8-22
 - functional description of libraries, 8-2
 - functions
 - DELINKLIBRARY, 5-52
 - EVENT_STATUS, 8-22
 - LINKLIBRARY, 8-7
 - SETACTUALNAME, 5-86, 8-7
 - indirect linkage, 8-4
 - initiation, 8-3
 - interrogating string-valued library attributes, 5-34
 - library directories, 8-2
 - library object, 8-1
 - library programs, 8-2
 - library templates, 8-2
 - linkage provisions, 8-4, 8-10
 - nature of freeze, 8-3
 - parameter passing rules, 8-12
 - permanent specification, 8-3
 - referencing libraries, 8-7
 - restricting use, 8-6
 - SHARING option
 - DONTCARE, 6-45, 8-6
 - PRIVATE, 6-45, 8-6
 - SHARED BY ALL, 6-45, 8-6
 - SHARED BY RUN UNIT, 6-45, 8-6
 - statements
 - CANCEL, 4-28, 8-5
 - FREEZE, 4-68
 - temporary specification, 8-3
- <library attribute specifications>, 3-100
- library call
 - for CENTRALSUPPORT, 9-23
- <library declaration>, 3-99
- <library entry point identifier>, 5-86
- <library entry point specification>, 3-139
- library entry points (*See* library)
- <library identifier>, 3-99
- <library object access mode>, 3-52
- <library object attributes>, 3-90, 3-101
- <library object declaration list>, 3-101
- <library object declaration>, 3-101
- library objects (*See* library)
- <library option>, 6-29
- library procedures
 - CENTRALSUPPORT, 9-13
 - input parameters, 9-24
 - output parameters, 9-25
 - result, 9-25
- <library specification>, 3-99
- <limit option>, 6-29
- LINE file, 6-6
- <lineinfo option>, 6-29
- <linenumber function>, 5-68
- <linewidth>, 6-54
- LINKCLASS, 3-51

<linklibrary function>, 5-69, 8-7
 <list declaration>, 3-104
 <list designator>, 3-174
 <list element>, 3-104
 <list identifier>, 3-104
 <list option>, 6-30
 <list>, 4-126
 <listdeleted option>, 6-30
 LISTDOLLAR option, 6-30
 <listincl option>, 6-31
 <listlookup function>, 5-73
 <listomitted option>, 6-31
 <ln function>, 5-73
 <lngamma function>, 5-73
 <loadinfo option>, 6-32
 <local connection block type
 declaration>, 3-19
 local identifiers, 1-6
 <local procedure reference array
 declaration>, 3-151
 <local structure block type
 declaration>, 3-164
 localization, 9-1
 establishing conventions for, 9-3
 LOCK <close option>, 4-43
 <lock interlock option>, 4-80
 <lock option>, 4-79
 <lock statement>, 4-79
 <lockstatus function>, 5-74
 <log function>, 5-75
 logical data comparison
 in internationalization, 9-8
 <logical operator>, 2-3
 logical operators, 5-15
 AND, 5-14
 EQV, 5-14
 IMP, 5-14
 NOT, 5-14
 OR, 5-14
 precedence of, 5-16
 results of, 5-15
 long (unpaged) arrays, 3-3
 LOOP fault (ON statement example), 4-99
 <lower bound list>, 3-136
 <lower bound>, 3-6
 <lower bounds>, 3-13
 <lower limit>, 3-38

M

<makehost option>, 6-34
 <masksearch function>, 5-75
 <max function>, 5-75
 <MCP option>, 6-36
 MCP_BOUND_LANGUAGES
 procedure, 9-105
 MCPSUPPORT library, 8-22
 <membership expression>, 3-182
 <membership primary>, 3-182
 <memory size>, 4-193
 MEMORYPROTECT fault (ON statement
 example), 4-99
 <merge option>, 6-37
 <merge statement>, 4-82
 <merging option list>, 4-82
 <merging option>, 4-82
 messages, multilingual
 creation for application program, 9-10
 <messagesearcher statement>, 4-83
 <min function>, 5-75
 MLS (See MultiLingual System (MLS))
 <MLSAccept statement>, 4-86
 <MLSdisplay statement>, 4-88
 <mnemonic attribute assignment>, 4-16
 <mnemonic attribute value>, 4-16
 <mnemonic attribute>, 4-16
 <mnemonic file attribute value>, 3-58
 <mnemonic library attribute value>, 3-101
 <mnemonic library attribute>, 4-16
 <mnemonic task attribute value>, 5-103
 MOD, 5-3
 <monitor declaration>, 3-106
 <monitor element>, 3-106
 <multidimensional array designator>, 4-173
 MultiLingual System (MLS), 9-1
 MESSAGESEARCHER statement, 4-83
 OUTPUTMESSAGE ARRAY
 declaration, 3-110
 <multiple attribute assignment
 statement>, 4-93
 multiplication, 5-3
 MUX, 5-3

N

<nabs function>, 5-76
 <name and title>, 6-21
 natural language, 9-1, 9-9
 <new character>, 3-118

- <new option>, 6-37
- <new size>, 4-169
- <newseqerr option>, 6-38
- NEWSOURCE file, 6-5
- NEWTAPE file, 6-5
- <nobindinfo option>, 6-38
- <non-array data identifier>, 3-51
- normal I/O, 4-71
- <normalize function>, 5-76
- normalized form
 - double precision, C-19
 - single precision, C-14
- <nostackarrays option>, 6-38
- NOT, 5-14
- <noxreflist option>, 6-39
- null statement, 4-1
- <null value>, 5-30
- <number list>, 4-29
- <number of bits>, 5-9
- <number of columns>, 4-126
- <number of tapes>, 4-191
- <numbered statement group>, 4-29
- <numbered statement list>, 4-29
- numbers
 - compiler conversion, 2-9
 - exponents, 2-9
 - in free-field data records, 4-131
 - ranges in ALGOL, 2-8
- <numeric convert part>, 4-138
- numeric sign, C-13

O

- <octal character>, 2-13
- <octal code>, 2-13
- <octal string>, 2-13
- OFFER option, 4-101
- <offset function>, 5-76
- <omit option>, 6-39
- <on statement>, 4-94
- <one-dimensional array name>, 3-9
- <one-dimensional direct array name>, 3-33
- <one-dimensional real direct array identifier>, 5-101
- <ones function>, 5-77
- one-word operands
 - type coercion, C-20
- <open control option>, 4-101
- <open file part>, 4-101
- <open options>, 4-101
- <open statement>, 4-101

- operands
 - type coercion of one-word and two-word operands, C-20
- <operator>, 2-3
- <optimize option>, 6-40
- optimizing code
 - BEGINSEGMENT option, 6-18
 - ENDSEGMENT option, 6-22
 - OPTIMIZE option, 6-40
- <option expression>, 6-12
- <option primary>, 6-12
- OR, 5-14
- ordering sequence value (OSV)
 - in internationalization, 9-8
- OSV, 9-8
- <outer level>, 6-28
- <output message array declaration>, 3-110
- <output message array identifier>, 3-110
- <output message array>, 3-110
- <output message case expression>, 3-112
- <output message case part>, 3-112
- <output message number>, 3-111
- <output message parameter number>, 3-112
- <output message parameter value>, 3-112
- <output message parameter>, 3-111
- <output message part>, 3-110
- <output message segment>, 3-111
- <output message>, 3-111
- <output option>, 4-189
- output parameters
 - for library procedures, 9-25
- <output procedure>, 4-189
- <outstuff parameter>, 5-101
- OWN
 - arrays, 3-4
 - pointers, 3-128
 - simple variables, 3-92

P

- <pack size>, 4-193
- PAGE option, 6-40
- paged (segmented) arrays, 3-3
- PARAMCHECK option, 6-41
- <parameter delimiter>, 2-2
- <parameter element>, 4-83, 4-88
- parameters
 - array parameters, 3-136, 3-141
 - array specification, 3-136
 - call-by-reference, 3-134
 - call-by-value, 3-133

- complex call-by-name parameters, 3-134
- event parameters, 3-147
- file parameters, 3-146
- format parameters, 3-147
- label parameters, 3-147
- list parameters, 3-147
- passing parameters to CANDE-initiated procedures, 3-132
- passing parameters to procedures, 4-108
- passing parameters to WFL-initiated procedures, 3-132
- picture parameters, 3-147
- pointer parameters, 3-147
- procedure parameters, 3-144
- procedure reference array parameters, 3-143
- restrictions on call-by-name pointer parameters, 3-131
- simple variable parameters, 3-145
- string parameters, 3-146
- task parameters, 3-147
- <parity error label>, 4-129
- <partial word expression>, 5-12
- <partial word part>, 5-12
- <participate option>, 4-24
- <pending procedure declaration>, 3-115
- <picture character>, 3-123
- <picture declaration>, 3-117
- <picture identifier>, 3-117
- <picture skip>, 3-121
- <picture symbol>, 3-117
- <picture>, 3-117
- pictures
 - character fields affected by picture symbols, 3-121
 - characters used by picture symbols, 3-120
 - control characters, 3-122
 - effects of hardware flip-flops on picture symbols, 3-120
 - in REPLACE statement, 4-156
 - introduction codes, 3-119
 - picture characters, 3-123
 - picture skip characters, 3-121
 - single picture characters, 3-122
 - string literals in pictures, 3-118
- <pointer assignment>, 4-17
- <pointer attribute specification>, 3-58
- <pointer case expression>, 5-31
- <pointer declaration>, 3-128
- <pointer expression>, 5-31
 - in <instuff parameter>, 5-102
 - in <outstuff parameter>, 5-101
 - in USERDATALOCATOR function, 5-102
- <pointer function designator>, 5-30
- <pointer function>, 5-77
- pointer handling (See character string manipulation)
- <pointer identifier>, 3-128
 - for SIZE function, 5-88
- <pointer intrinsic name>, 5-30
 - alphabetical listing of, 5-42
- <pointer part>, 5-19
- <pointer primary>, 5-31
- <pointer relation>, 5-19
- <pointer statement>, 4-104
- <pointer table membership>, 5-21
- <pointer update assignment>, 4-17
- <pointer valued library attribute specification>, 3-100
- <pointer variable>, 4-17
- pointers
 - functions with pointer parameters
 - DELTA function, 5-53
 - DOUBLE function, 5-58
 - INTEGER function, 5-66
 - OFFSET function, 5-76
 - READLOCK function, 5-80
 - REAL function, 5-82
 - REMAININGCHARS function, 5-83
 - SIZE function, 5-88
 - STRING function, 5-89
 - initialized, 3-128
 - internal structure, C-20
 - intrinsic functions returning values of type POINTER, 5-42
 - lexical level restrictions, 3-129
 - OWN, 3-128
 - pointer assignment, 4-17
 - POINTER declaration, 3-128
 - pointer expression, 5-31
 - pointer functions
 - POINTER function, 5-77
 - READLOCK function, 5-80
 - pointer relation, 5-19
 - string relation, 5-19
 - up-level pointer assignment, 3-129
- <pointer-valued attribute>, 4-165
- pointer-valued attributes
 - assigning values
 - FILE declaration, 3-57
 - multiple attribute assignment statement, 4-93
 - replace family-change statement, 4-163
 - replace pointer-valued attribute statement, 4-165
- interrogating

- REPLACE statement source part
 - <pointer-valued attribute>, 4-138, 4-160
 - library attributes, 8-8
 - VALUE function, 5-103
 - <pointer-valued file attribute name>, 3-58
 - <pointer-valued file attribute>, 4-165
 - <pointer-valued library attribute name>, 3-100
 - <pointer-valued task attribute>, 4-165
 - pool array, 4-141
 - pool array pointer, 4-141
 - <port close option>, 4-44
 - <pot function>, 5-79
 - POTC, 5-79
 - POTH, 5-79
 - POTL, 5-79
 - precedence
 - of arithmetic operators, 5-4
 - of Boolean operators, 5-16
 - precision of arithmetic expressions, 5-3
 - primary coroutine, 4-26
 - <primary identifier>, 6-49
 - <printable character>, 2-14
 - priority sequence value (PSV)
 - in internationalization, 9-8
 - procedure
 - as a function, 3-132
 - by-calling, 3-138
 - EXTERNAL, 3-138
 - formal parameters, 3-133
 - forward procedure declaration, 3-87
 - initiated through CANDE or WFL, 3-132
 - library entry points, 3-139
 - parameters
 - array, 3-136, 3-141
 - call-by-reference, 3-134
 - call-by-value, 3-133
 - file, 3-146
 - procedure, 3-144
 - simple variable, 3-145
 - string, 3-146
 - passing parameters to, 4-108
 - PROCEDURE declaration, 3-132
 - selection procedure, 3-139
 - statements
 - CALL, 4-26
 - CONTINUE, 4-47
 - procedure invocation, 4-107
 - PROCESS, 4-112
 - RUN, 4-179
 - <procedure body>, 3-138
 - <procedure declaration>, 3-132
 - <procedure heading>, 3-132
 - <procedure identifier>, 3-133
 - <procedure invocation statement>, 4-107
 - <procedure library attribute specification>, 3-101
 - procedure reference
 - assignment, 4-18
 - declaration, 3-150
 - procedure reference array
 - declaration, 3-151
 - parameters, 3-143
 - specification, 3-137
 - <procedure reference array declaration>, 3-151
 - <procedure reference array designator>, 3-151
 - for SIZE function, 5-88
 - <procedure reference array element>, 3-151
 - <procedure reference array identifier>, 3-137, 3-151
 - <procedure reference array row>, 3-152
 - <procedure reference array specification>, 3-137
 - <procedure reference assignment>, 4-18
 - <procedure reference variable>, 4-18
 - <procedure specification>, 3-136
 - <procedure type>, 3-132
 - <procedure-valued library attribute name>, 3-101
 - <process statement>, 4-112
 - <processid function>, 5-79
 - <procure statement>, 4-114
 - <program unit>, 1-2
 - <programdump destination>, 4-118
 - <programdump option>, 4-115
 - <programdump statement>, 4-115
 - <prolog procedure declaration>, 3-154
 - protected linkage class, 3-51
 - protocols, data communications,
 - international, 9-1
 - PSV, 9-8
 - PURGE <close option>, 4-43
 - <purge option>, 6-41
- Q**
- <quaternary code>, 2-13
 - <quaternary string>, 2-13
 - <quoted string>, 4-130
 - quoted strings in free-field data
 - records, 4-132

R

- railroad diagrams, explanation of, D-1
- <random function>, 5-80
- range of numbers, 2-8
- READ statement
 - array row read, 4-128
 - binary read, 4-127
 - formatted read, 4-126
 - free-field data format, 4-130
 - free-field data record
 - hexadecimal strings, 4-132
 - numbers, 4-131
 - quoted strings, 4-132
 - unquoted strings, 4-131
- <read statement>, 4-121
- <read subfile specification>, 4-123
- READABLE option, 5-68
- <readlock function>, 5-80
- READONLY option
 - EXPORT declaration, 3-51
 - IMPORTED declaration, 3-90
- READWRITE option, 5-68
 - EXPORT declaration, 3-51
 - IMPORTED declaration, 3-90
- ready queue, 4-31
- <readycl function>, 5-81
- <real array identifier>
 - in <outstuff parameter>, 5-101
 - <in real read/write array row>, 5-101
- <real array reference identifier>
 - in <outstuff parameter>, 5-101
 - <in real read/write array row>, 5-101
- <real array row>, 5-102
- real data type
 - arithmetic assignment, 4-5
 - arithmetic expression, 5-2
 - arithmetic relation, 5-18
 - direct real array declaration, 3-33
 - functions for manipulating real expressions
 - DINTEGER function, 5-56
 - DOUBLE function, 5-58
 - ENTIER function, 5-62
 - INTEGER function, 5-66
 - NORMALIZE function, 5-76
 - real array declaration, 3-3
 - real array reference declaration, 3-12
 - REAL declaration, 3-155
 - real functions
 - ABS function, 5-43
 - ARCCOS function, 5-43
 - ARCSIN function, 5-43
 - ARCTAN function, 5-43
 - ATANH function, 5-45
 - CABS function, 5-46
 - CHECKSUM function, 5-47
 - COMPILETIME function, 5-48
 - COS function, 5-48
 - COSH function, 5-49
 - COTAN function, 5-49
 - ERF function, 5-62
 - ERFC function, 5-63
 - EXP function, 5-63
 - FIRST function, 5-63
 - FIRSTWORD function, 5-64
 - GAMMA function, 5-64
 - IMAG function, 5-65
 - LN function, 5-73
 - LNGAMMA function, 5-73
 - LOG function, 5-75
 - MAX function, 5-75
 - MIN function, 5-75
 - NABS function, 5-76
 - NORMALIZE function, 5-76
 - RANDOM function, 5-80
 - READLOCK function, 5-80
 - REAL function, 5-82
 - SCALERIGHTF function, 5-84
 - SECONDWORD function, 5-85
 - SIN function, 5-87
 - SINGLE function, 5-88
 - SINH function, 5-88
 - SQRT function, 5-89
 - TAN function, 5-92
 - TANH function, 5-92
 - TIME function, 5-93
 - real operand internal structure, C-14
 - real procedure declaration, 3-132
 - real value array declaration, 3-186
 - width of real array elements, 3-5
- <real declaration>, 3-155
- <real direct array identifier>, 5-101
- <real function>, 5-82
- <real identifier>, 3-155
- real operand internal structure, C-14
- <real read/write array row>
 - in <outstuff parameter>, 5-101
 - in USERDATAAREBUILD function, 5-103
- <real variable>, 4-198
 - for <fault number>, 4-97
- <record length>, 4-192
- <record number>, 4-186
- REEL <close option>, 4-43
- reference checking, 5-67
- REJECTOPEN, 4-176

- <relational operator>, 5-18
 - <remainingchars function>, 5-83
 - <remark>, 2-10
 - <removefile statement>, 4-135
 - <repeat function>, 5-83
 - <repeat part value>, 3-121
 - <repeat part>, 3-61
 - <replace family-change statement>, 4-163
 - <replace pointer-valued attribute statement>, 4-165
 - REPLACE statement
 - short and long string literals, 4-141
 - string literals interpreted as arithmetic expressions, 4-146
 - <replace statement>, 4-137
 - <reserved word>, B-1
 - reserved words, B-3
 - <reset statement>, 4-168
 - resettable standard Boolean options, 6-14
 - <residual count>, 4-138
 - <resize statement>, 4-169
 - <respond file part>, 4-176
 - <respond options>, 4-176
 - RESPOND statement, 4-176
 - <respond statement>, 4-176
 - <respondtype option>, 4-176
 - <restart specifications>, 4-194
 - result
 - for library procedures, 9-25
 - <result length>, 4-83
 - <result pointer>, 4-83
 - <result>, 4-123
 - REWIND <close option>, 4-43
 - <rewind statement>, 4-178
 - <row number>, 4-56
 - <row selector>, 3-10
 - in <real read/write array row>, 5-101
 - <row/copy numbers>, 4-56
 - <run statement>, 4-179
- S**
- <scale factor>, 3-64
 - <scaleleft function>, 5-84
 - <scalerright function>, 5-84
 - <scalerrightf function>, 5-84
 - <scalerrightt function>, 5-85
 - <scan part>, 4-139
 - <scan statement>, 4-181
 - scope, 1-5
 - global identifiers, 1-6
 - local identifiers, 1-6
 - secondary coroutine, 4-26
 - <secondary identifier>, 6-49
 - <secondword function>, 5-85
 - security of library object, 3-52
 - <seek statement>, 4-186
 - <segdescabove option>, 6-41
 - <segs option>, 6-42
 - <selection procedure identifier>, 3-138
 - <separate procedure>, 1-3
 - SEPCOMP facility
 - MAKEHOST option, 6-34
 - SEPCOMP option, 6-42
 - <sepcomp option>, 6-42
 - <seq option>, 6-44
 - <seqerr option>, 6-44
 - <sequence number>, 6-23
 - serial I/O, 4-186
 - <set statement>, 4-187
 - <setabstractvalue statement>, 4-188
 - <setactualname function>, 5-86
 - <sharing option>, 6-45
 - short and long string literals, 4-141
 - <sign function>, 5-87
 - signs of numeric fields, C-13
 - <simple arithmetic expression>, 5-2
 - <simple Boolean expression>, 5-13
 - <simple complex expression>, 5-24
 - <simple pointer expression>, 5-31
 - <simple source>, 4-163
 - <simple variable declaration>, 3-157
 - simple variable parameters, 3-145
 - <simple variable>, 4-5
 - <sin function>, 5-87
 - <single function>, 5-88
 - <single option>, 6-46
 - <single picture character>, 3-122
 - <single space>, 2-3
 - <single-precision simple variable>, 5-101
 - <sinh function>, 5-88
 - <size function>, 5-88
 - <size specifications>, 4-193
 - <skip>, 5-31
 - <sort statement>, 4-189
 - <source characters>, 3-178
 - SOURCE file, 6-4
 - <source part list>, 4-138
 - <source part>, 4-138
 - <source>, 4-138
 - <space statement>, 4-197
 - <space>, 2-3
 - <special array resize parameters>, 4-172
 - <special destination character>, 3-178

- <special new character>, 3-118
- <specification>, 3-135
- <specified lower bound>, 3-136
- <specifier>, 3-135
- <sqrt function>, 5-89
- <stack option>, 6-46
- start index, 4-206
- <start specification>, 6-24
- <starting index>, 5-47
- <statement list>, 1-3
- <statement>, 4-1
- <statistics option>, 6-46
- <stop option>, 6-47
- <stop specification>, 6-24
- <storage queue event>, 4-204
- <string array declaration>, 3-160
- <string array designator>, 3-160
- <string array identifier>, 3-160
- <string assignment>, 4-20
- <string character set>, 5-65
- string code, 2-15
- string concatenation, 5-36
- <string concatenation operator>, 5-34
- <string constant>, 5-34
- string data type
 - functions with string parameters
 - DECIMAL function, 5-51
 - DROP function, 5-59
 - FIRST function, 5-63
 - HEAD function, 5-65
 - LENGTH function, 5-68
 - REPEAT function, 5-83
 - TAIL function, 5-91
 - TAKE function, 5-91
 - TRANSLATE function, 5-100
 - intrinsic functions returning values of type
 - STRING, 5-42
 - STRING ARRAY declaration, 3-160
 - string assignment, 4-20
 - string concatenation, 5-36
 - STRING declaration, 3-158
 - string expression, 5-34
 - string expression relation, 5-20
 - string functions
 - DROP function, 5-59
 - HEAD function, 5-65
 - REPEAT function, 5-83
 - STRING function, 5-89
 - STRING4 function, 5-89
 - STRING7 function, 5-89
 - STRING8 function, 5-89
 - TAIL function, 5-91
 - TAKE function, 5-91
 - TRANSLATE function, 5-100
 - string parameters in procedures, 3-146
 - STRING PROCEDURE declaration, 3-132
 - <string declaration>, 3-158
 - <string designator>, 4-20
 - string expression
 - in REPLACE statement, 4-161
 - <string expression relation>, 5-20
 - <string expression>, 5-34
 - <string function designator>, 5-30
 - <string function>, 5-89
 - <string identifier>, 3-158
 - <string intrinsic name>, 5-30
 - alphabetical listing of, 5-42
 - string literal
 - as an arithmetic primary, 5-7
 - ASCII string, 2-16
 - binary string, 2-12
 - character size, 2-15
 - default character type, C-12
 - dollar signs in strings, 2-16
 - in editing specifications, 3-60, 3-62
 - in pictures, 3-118
 - in REPLACE statement source parts, 4-141
 - interpreted as arithmetic expression in
 - REPLACE statement, 4-146
 - maximum length of, 2-16
 - octal string, 2-13
 - pool array, 2-16
 - quaternary string, 2-13
 - quotation marks in strings, 2-16
 - string code, 2-15
 - string length, 4-141
 - <string literal>, 2-12, 5-102
 - string manipulation (See character string manipulation)
 - <string or pointer library attribute specification>, 3-100
 - string parameters, 3-146
 - <string primary>, 5-34
 - <string procedure identifier>, 3-133
 - <string relation>, 5-19
 - <string relational operator>, 5-18
 - string type, 3-158
 - <string type>, 3-158
 - <string valued library attribute specification>, 3-100
 - <string variable>, 5-35
 - STRING4, 5-89
 - STRING7, 5-89
 - STRING8, 5-89
 - string-valued attributes
 - assigning values

LIBRARY declaration, 3-99
 string assignment target <string-valued library attribute>, 4-20
 interrogating
 string primary <string-valued library attribute>, 5-34
 library attributes, 8-8
 <string-valued library attribute name>, 3-100
 <string-valued library attribute>, 5-35
 <structure block array declaration>, 3-161
 <structure block item designator>, 3-165
 <structure block item>, 3-166
 <structure block qualifier>, 3-165
 <structure block reference variable declaration>, 3-163
 STRUCTURE BLOCK TYPE declaration, 3-164
 referencing items outside the structure block, 3-165
 <structure block type declaration>, 3-164
 <structure block type identifier>, 3-165
 <structure block variable declaration>, 3-167
 <subarray selector>, 4-10
 <subfile index>, 4-42
 subfile specification
 URGENT clause, 4-217
 <subfile specification>, 4-123
 <subscript>, 3-10
 in <outstuff parameter>, 5-101
 <subscripted string variable>, 5-35
 <subscripted variable>, 4-5
 subtraction, 5-3
 <swap statement>, 4-198
 <switch file declaration>, 3-168
 <switch file identifier>, 3-168
 <switch file list>, 3-168
 <switch format declaration>, 3-170
 <switch format identifier>, 3-170
 <switch format list>, 3-170
 <switch format segment>, 3-170
 <switch label declaration>, 3-172
 <switch label identifier>, 3-172
 <switch label list>, 3-172
 <switch list declaration>, 3-174
 <switch list identifier>, 3-174
 <symbol construct>, 2-1
 synchronized output
 explanation, 4-217
 setting file attribute (example), 3-59, 4-9
 syntax for WRITE statement, 4-122
 using WRITE statement (example), 4-217
 SYSTEM/CCSFILE data file, 9-4

T

<TADS option>, 6-48
 <tail function>, 5-91
 <take function>, 5-91
 <tan function>, 5-92
 <tanh function>, 5-92
 <task array declaration>, 3-176
 <task array designator>, 3-177
 <task array identifier>, 3-176
 <task assignment>, 4-22
 <task declaration>, 3-176
 <task designator>, 3-176
 <task identifier>, 3-176
 TASKSTRING attribute, 6-9
 task-valued attributes (*See also* job and task control)
 assigning values
 task assignment, 4-22
 interrogating
 <task designator>, 3-176
 <task-valued task attribute name>, 3-176
 TCP/IP, 4-218
 <text>, 3-26
 THAW (Thaw Frozen Library) ODT command, 8-6
 THIS function, 5-92
 THISCL function, 5-93
 <thru statement>, 4-200
 thunk, 3-134
 <time function>, 5-93
 <time option>, 6-50
 <time>, 4-204
 TIMES, 5-3
 TIMEZONENAME procedure, 8-21
 TIMEZONEOFFSET procedure, 8-22
 TITLE library attribute (*See* library)
 <title>, 6-21
 touched array, 3-4
 <transfer part>, 4-138
 <translate function>, 5-100
 <translate part>, 4-139
 translate table
 definition of, 9-6
 for internationalization, 9-6
 in REPLACE statement, 4-154
 <translate table declaration>, 3-178
 <translate table element>, 3-178
 <translate table identifier>, 3-178
 <translate table>, 4-139
 <translate-table attribute specification>, 3-58
 translate-table-valued attributes

- assigning values
 - FILE declaration, 3-57
 - multiple attribute assignment statement, 4-93
 - VALUE function, 5-103
- <translate-table-valued file attribute name>, 3-58
- <translation specifier>, 3-178
- <translator's help text>, 3-111
- Transmission Control Protocol/Internet Protocol, 4-218
- truth set
 - definition of, 9-7
- <truth set declaration>, 3-182
- <truth set identifier>, 3-182
- truth set table
 - in Boolean expressions, 5-21
 - in REPLACE statement, 4-158, 4-159
 - in SCAN statement, 4-183, 4-184
- <truth set table>, 4-139
- try arithmetic expression, 5-37
- try Boolean expression, 5-37
- try complex expression, 5-37
- try expression, 5-37
- <try expression>, 5-37
- try pointer expression, 5-37
- TRY statement, 4-201
- <try statement>, 4-201
- two-word operands
 - type coercion, C-20
- type coercion
 - of one-word and two-word operands, C-20
- type declarations
 - BOOLEAN, 3-14
 - COMPLEX, 3-16
 - DOUBLE, 3-36
 - INTEGER, 3-92
 - POINTER, 3-128
 - PROCEDURE REFERENCE ARRAY, 3-151
 - REAL, 3-155
 - STRING, 3-158
- type transfer functions
 - BOOLEAN function, 5-45
 - COMPLEX function, 5-48
 - DECIMAL function, 5-51
 - DINTEGER function, 5-56
 - DOUBLE function, 5-58
 - ENTIER function, 5-62
 - FIRST function, 5-63
 - FIRSTWORD function, 5-64
 - IMAG function, 5-65
 - INTEGER function, 5-66
 - INTEGERT function, 5-67

- REAL function, 5-82
- SECONDWORD function, 5-85
- SINGLE function, 5-88
- STRING function, 5-89
- types resulting from arithmetic operations, 5-6

U

- uninitialized pointer, 3-128
- <unit count>, 4-138
- <unlabeled statement>, 4-1
- UNLOCK statement, 5-100
- unpaged (long) arrays, 3-3
- <unquoted string>, 4-130
- unquoted strings in free-field data records, 4-131
- UNREADYCL function, 5-100
- unsegmented arrays, 3-3
- <up or down>, 4-163
- <update pointer>, 4-137
- <update symbols>, 4-8
- up-level event
 - in ATTACH statement, 4-23
 - in direct I/O, 4-73
- up-level pointer assignment, 3-129
- <upper bound>, 3-6
- <upper limit>, 3-38
- URGENT clause, 4-217
- <use option>, 6-50
- <user option>, 6-50
- USERDATA function, 5-101
- USERDATALOCATOR function, 5-102
- USERDATAREBUILD function, 5-102

V

- VALIDATE_NAME_RETURN_NUM procedure, 9-107
- VALIDATE_NUM_RETURN_NAME procedure, 9-109
- <value array declaration>, 3-186
- <value array identifier>, 3-186
- <value function>, 5-103
- <value list>, 4-58
- <value part>, 3-133
- <variable>, 4-5
- <version option>, 6-51
- <void option>, 6-52
- <voidt option>, 6-52

VSNCOMPARE_TEXT procedure, 9-111
VSNESCAPEMENT procedure, 9-114
VSNGETORDERINGFOR_ONE_TEXT
 procedure, 9-116
VSNINFO procedure, 9-119
VSNINSPECT_TEXT procedure, 9-123
VSNORDERING_INFO procedure, 9-126
VSNTRANS_TEXT procedure, 9-131
VSNTRANSTABLE procedure, 9-129
VSNTRUTHSET procedure, 9-134

W

WAIT control option, 4-24
WAIT option, 4-101
WAIT parameter list, 4-205
<wait parameter list>, 4-204
<wait statement>, 4-204
<waitandreset statement>, 4-209
<waitimport option>, 6-53
<warnsupr option>, 6-53
<when statement>, 4-212
<while statement>, 4-213
width of array elements, 3-5
word array, 3-5
<word array identifier>, 3-4
<word array part>, 5-31
<word array row>, 5-32
<word type>, 3-5
<write file part>, 4-215
 semantics, 4-215
WRITE statement
 array row write, 4-220
 binary write, 4-219
 formatted write, 4-219
 synchronized output (See synchronized
 output)

<write statement>, 4-215
<write subfile specification>, 4-217
<writeafter option>, 6-53

X

<xdecs option>, 6-54
<xref option>, 6-54
XREFFILE file, 6-7
<xreffiles option>, 6-56
<xrefs option>, 6-56

Z

ZERODIVIDE fault (ON statement
 example), 4-99
ZIP
 with array, 4-224
 with file, 4-225
<zip statement>, 4-224

Special Characters

** (exponentiation operator), 5-4
@ (single-precision exponent delimiter), 2-6
@@ (double-precision exponent delimiter), 2-6
| (OR), 5-14
|| (string concatenation operator), 5-36



8600098-505